## Preface

What Is MATLAB? ................................................................. ii
MATLAB Documentation ............................................... iv

## Command Summary

1

General Purpose Commands ............................................. 1-2
Operators and Special Characters .................................... 1-3
Logical Functions ............................................................... 1-4
Language Constructs and Debugging .................................. 1-4
Elementary Matrices and Matrix Manipulation ..................... 1-5
Specialized Matrices .......................................................... 1-7
Elementary Math Functions ................................................ 1-7
Specialized Math Functions ................................................ 1-8
Coordinate System Conversion ......................................... 1-8
Matrix Functions - Numerical Linear Algebra ....................... 1-9
Data Analysis and Fourier Transform Functions ................... 1-10
Polynomial and Interpolation Functions .............................. 1-11
Function Functions - Nonlinear Numerical Methods .......................... 1-12
Sparse Matrix Functions .................................................. 1-12
Sound Processing Functions ............................................. 1-14
Character String Functions ............................................... 1-14
Low-Level File I/O Functions .......................................... 1-15
Bitwise Functions ............................................................ 1-16
Structure Functions ......................................................... 1-17
Object Functions .............................................................. 1-17
Cell Array Functions ......................................................... 1-17
Multidimensional Array Functions ...................................... 1-17

Reference

List of Commands

Function Names ................................................................. A-2
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 2-24
- `doc`: Load hypertext documentation. page 2-199
- `help`: Online help for MATLAB functions and M-files. page 2-350
- `lasterr`: Last error message. page 2-408
- `lookfor`: Keyword search through all help entries. page 2-425
- `pathe`: Control MATLAB’s directory search path. page 2-503
- `profile`: Measure and display M-file execution profiles. page 2-533
- `rmpath`: Remove directories from MATLAB’s search path. page 2-571
- `type`: List file. page 2-684
- `version`: MATLAB version number. page 2-693
- `what`: Directory listing of M-files, MAT-files, and MEX-files. page 2-701
- `whatsnew`: Display README files for MATLAB and toolboxes. page 2-702
- `which`: Locate functions and files. page 2-703

Managing Variables and the Workspace

- `clear`: Remove items from memory. page 2-111
- `disp`: Display text or array. page 2-195
- `length`: Length of vector. page 2-413
- `load`: Retrieve variables from disk. page 2-416
- `pack`: Consolidate workspace memory. page 2-499
- `save`: Save workspace variables on disk. page 2-581
- `size`: Array dimensions. page 2-597
- `who`, `whos`: List directory of variables in memory. page 2-706

Controlling the Command Window

- `echo`: Echo M-files during execution. page 2-202
- `format`: Control the output display format. page 2-278
- `more`: Control paged output for the command window. page 2-454

Working with Files and the Operating Environment

- `acopy`: Copy Macintosh file from one folder to another. page 2-19
- `amove`: Move Macintosh file from one folder to another. page 2-29
- `applescript`: Load a compiled AppleScript from a file and execute it. page 2-34
- `rename`: Rename Macintosh File. page 2-35
- `areveal`: Reveal filename on Macintosh desktop. page 2-36
- `cd`: Change working directory. page 2-88
Operators and Special Characters

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Plus</td>
<td>2-2</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Minus</td>
<td>2-2</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Matrix multiplication</td>
<td>2-2</td>
</tr>
<tr>
<td><code>.*</code></td>
<td>Array multiplication</td>
<td>2-2</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Matrix power</td>
<td>2-2</td>
</tr>
<tr>
<td><code>.^</code></td>
<td>Array power</td>
<td>2-2</td>
</tr>
<tr>
<td><code>kron</code></td>
<td>Kronecker tensor product</td>
<td>2-407</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash or left division</td>
<td>2-2</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Slash or right division</td>
<td>2-2</td>
</tr>
<tr>
<td><code>./</code></td>
<td>Array division, right and left</td>
<td>2-2</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Colon</td>
<td>2-16</td>
</tr>
<tr>
<td><code>()</code></td>
<td>Parentheses</td>
<td>2-13</td>
</tr>
<tr>
<td><code>[ ]</code></td>
<td>Brackets</td>
<td>2-13</td>
</tr>
<tr>
<td><code>{}</code></td>
<td>Continuation</td>
<td>2-13</td>
</tr>
<tr>
<td><code>...</code></td>
<td>. . . . . . .</td>
<td>2-13</td>
</tr>
<tr>
<td><code>;</code></td>
<td>Comma</td>
<td>2-13</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Semicolon</td>
<td>2-13</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Comment</td>
<td>2-13</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Exclamation point</td>
<td>2-13</td>
</tr>
<tr>
<td><code>'</code></td>
<td>Transpose and quote</td>
<td>2-13</td>
</tr>
</tbody>
</table>
Command Summary

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
Logical EXCLUSIVE OR ............................................ page 2-14

Logical Functions

all Test to determine if all elements are nonzero ............... page 2-27
any Test for any nonzeros ....................................... page 2-32
exist Check if a variable or file exists ......................... page 2-231
find Find indices and values of nonzero elements ............... page 2-258
is* Detect state .................................................... page 2-398
*isa Detect an object of a given class ........................... page 2-402
logical Convert numeric values to logical ....................... page 2-421

Language Constructs and Debugging

MATLAB as a Programming Language

builtin Execute builtin function from overloaded method .... page 2-80
eval Interpret strings containing MATLAB expressions .......... page 2-228
feval Function evaluation ........................................ page 2-244
function Function M-files ........................................ page 2-296
global Define global variables .................................... page 2-332
nargchk Check number of input arguments ....................... page 2-457
script Script M-files .............................................. page 2-586

Control Flow

break Break out of flow control structures ..................... page 2-79
case Case switch .................................................. page 2-86
else Conditionally execute statements ............................ page 2-217
elseif Conditionally execute statements ........................ page 2-218
end Terminate for, while, switch, and if statements or indicate last index ................................ page 2-220
error Display error messages ...................................... page 2-225
for Repeat statements a specific number of times .............. page 2-276
Language Constructs and Debugging

**Conditionally execute statements**

Conditionally execute statements

**Default part of switch statement**

Default part of switch statement

**Return to the invoking function**

Return to the invoking function

**Switch among several cases based on expression**

Switch among several cases based on expression

**Display warning message**

Display warning message

**Repeat statements an indefinite number of times**

Repeat statements an indefinite number of times

**Interactive Input**

**Request user input**

Request user input

**Invoke the keyboard in an M-file**

Invoke the keyboard in an M-file

**Generate a menu of choices for user input**

Generate a menu of choices for user input

**Halt execution temporarily**

Halt execution temporarily

**Object-Oriented Programming**

**Create object or return class of object**

Create object or return class of object

**Convert to double precision**

Convert to double precision

**Inferior class relationship**

Inferior class relationship

**Construct an inline object**

Construct an inline object

**Detect an object of a given class**

Detect an object of a given class

**Superior class relationship**

Superior class relationship

**Convert to unsigned 8-bit integer**

Convert to unsigned 8-bit integer

**Debugging**

**Clear breakpoints**

Clear breakpoints

**Resume execution**

Resume execution

**Change local workspace context**

Change local workspace context

**Enable MEX-file debugging**

Enable MEX-file debugging

**Quit debug mode**

Quit debug mode

**Display function call stack**

Display function call stack

**List all breakpoints**

List all breakpoints

**Execute one or more lines from a breakpoint**

Execute one or more lines from a breakpoint

**Set breakpoints in an M-file function**

Set breakpoints in an M-file function

**List M-file with line numbers**

List M-file with line numbers

**Change local workspace context**

Change local workspace context
Elementary Matrices and Matrix Manipulation

Elementary Matrices and Arrays

- `eye`: Identity matrix. ........................................ page 2-238
- `linspace`: Generate linearly spaced vectors .................. page 2-415
- `logspace`: Generate logarithmically spaced vectors .......... page 2-424
- `ones`: Create an array of all ones .......................... page 2-495
- `rand`: Uniformly distributed random numbers and arrays ... page 2-549
- `randn`: Normally distributed random numbers and arrays ... page 2-551
- `zeros`: Create an array of all zeros ........................ page 2-715
- `:` (colon): Regularly spaced vector ......................... page 2-16

Special Variables and Constants

- `ans`: The most recent answer ................................. page 2-31
- `computer`: Identify the computer on which MATLAB is running ... page 2-119
- `eps`: Floating-point relative accuracy ........................ page 2-222
- `flops`: Count floating-point operations ...................... page 2-266
- `i`: Imaginary unit............................................. page 2-353
- `Inf`: Infinity.................................................. page 2-370
- `inputname`: Input argument name ............................ page 2-378
- `j`: Imaginary unit............................................. page 2-405
- `NaN`: Not-a-Number ........................................... page 2-456
- `nargin`, `nargout`: Number of function arguments ........ page 2-458
- `pi`: Ratio of a circle's circumference to its diameter, $\pi$ ... page 2-513
- `realmax`: Largest positive floating-point number .......... page 2-561
- `realmin`: Smallest positive floating-point number .......... page 2-562
- `varargin`, `varargout`: Pass or return variable numbers of arguments .... page 2-690

Time and Dates

- `calendar`: Calendar ............................................ page 2-82
- `clock`: Current time as a date vector ...................... page 2-113
- `cputime`: Elapsed CPU time ................................ page 2-136
- `date`: Current date string ................................ page 2-143
- `datenum`: Serial date number ............................... page 2-144
- `datenum`: Date string format ................................ page 2-145
- `datenum`: Date components ................................ page 2-147
- `eomday`: End of month ....................................... page 2-221
- `etimer`: Elapsed time ......................................... page 2-227
now: Current date and time. ........................................ page 2-470
tic, toc: Stopwatch timer .......................................... page 2-676
weekday: Day of the week ........................................... page 2-700

Matrix Manipulation

cat: Concatenate arrays ........................................... page 2-87
di ag: Diagonal matrices and diagonals of a matrix .......... page 2-190
fli pl r: Flip matrices left-right ................................ page 2-263
fli pud: Flip matrices up-down .................................... page 2-264
repmat: Replicate and tile an array .............................. page 2-565
reshape: Reshape array ............................................. page 2-566
rot 90: Rotate matrix 90 degrees ................................ page 2-574
tril: Lower triangular part of a matrix ............................ page 2-681
tri u: Upper triangular part of a matrix .......................... page 2-682
: (colon): Index into array, rearrange array .................... page 2-16

Specialized Matrices

compan: Companion matrix ......................................... page 2-118
gallery: Test matrices ............................................... page 2-306
hadm ar d: Hadamard matrix ....................................... page 2-344
hankel: H ank el matrix ............................................... page 2-345
hilb: Hilbert matrix .................................................. page 2-352
invhilb: Inverse of the Hilbert matrix ......................... page 2-396
magic: Magic square .................................................. page 2-438
pascal: Pascal matrix ................................................ page 2-502
toeplitz: Toeplitz matrix ............................................ page 2-677
wilkinson: Wilkinson’s eigenvalue test matrix ................. page 2-708

Elementary Math Functions

abs: Absolute value and complex magnitude .................... page 2-18
acos, acosh: Inverse cosine and inverse hyperbolic cosine page 2-20
acot, acoth: Inverse cotangent and inverse hyperbolic cotangent page 2-21
acsc, aCSch: Inverse cosecant and inverse hyperbolic cosecant page 2-22
angle: Phase angle .................................................... page 2-30
asec, asech: Inverse secant and inverse hyperbolic secant page 2-37
asin, asinh: Inverse sine and inverse hyperbolic sine .......... page 2-38
atan, atanh: Inverse tangent and inverse hyperbolic tangent page 2-40
at an2: Four-quadrant inverse tangent .......................... page 2-42
### Command Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>Round toward infinity</td>
<td>2-91</td>
</tr>
<tr>
<td>conj</td>
<td>Complex conjugate</td>
<td>2-124</td>
</tr>
<tr>
<td>cos, cosh</td>
<td>Cosine and hyperbolic cosine</td>
<td>2-131</td>
</tr>
<tr>
<td>cot, coth</td>
<td>Cotangent and hyperbolic cotangent</td>
<td>2-132</td>
</tr>
<tr>
<td>csc, csch</td>
<td>Cosecant and hyperbolic cosecant</td>
<td>2-138</td>
</tr>
<tr>
<td>exp</td>
<td>Exponential</td>
<td>2-233</td>
</tr>
<tr>
<td>fix</td>
<td>Round towards zero</td>
<td>2-261</td>
</tr>
<tr>
<td>floor</td>
<td>Round towards minus infinity</td>
<td>2-265</td>
</tr>
<tr>
<td>gcd</td>
<td>Greatest common divisor</td>
<td>2-328</td>
</tr>
<tr>
<td>imag</td>
<td>Imaginary part of a complex number</td>
<td>2-359</td>
</tr>
<tr>
<td>lcm</td>
<td>Least common multiple</td>
<td>2-410</td>
</tr>
<tr>
<td>log</td>
<td>Natural logarithm</td>
<td>2-418</td>
</tr>
<tr>
<td>log2</td>
<td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td>
<td>2-419</td>
</tr>
<tr>
<td>log10</td>
<td>Common (base 10) logarithm</td>
<td>2-420</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (signed remainder after division)</td>
<td>2-453</td>
</tr>
<tr>
<td>real</td>
<td>Real part of complex number</td>
<td>2-560</td>
</tr>
<tr>
<td>rem</td>
<td>Remainder after division</td>
<td>2-564</td>
</tr>
<tr>
<td>round</td>
<td>Round to nearest integer</td>
<td>2-575</td>
</tr>
<tr>
<td>sec, sech</td>
<td>Secant and hyperbolic secant</td>
<td>2-587</td>
</tr>
<tr>
<td>sign</td>
<td>Signum function</td>
<td>2-594</td>
</tr>
<tr>
<td>sin, sinh</td>
<td>Sine and hyperbolic sine</td>
<td>2-595</td>
</tr>
<tr>
<td>sqrt</td>
<td>Square root</td>
<td>2-630</td>
</tr>
<tr>
<td>tan, tanh</td>
<td>Tangent and hyperbolic tangent</td>
<td>2-672</td>
</tr>
</tbody>
</table>

### Specialized Math Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>airy</td>
<td>Airy functions</td>
<td>2-25</td>
</tr>
<tr>
<td>besselh</td>
<td>Bessel functions of the third kind (Hankel functions)</td>
<td>2-49</td>
</tr>
<tr>
<td>bessel, besselk</td>
<td>Modified Bessel functions</td>
<td>2-51</td>
</tr>
<tr>
<td>bessely</td>
<td>Bessel functions</td>
<td>2-53</td>
</tr>
<tr>
<td>bet, betaic, betai, betal</td>
<td>Beta functions</td>
<td>2-56</td>
</tr>
<tr>
<td>ellipj</td>
<td>Jacobi elliptic functions</td>
<td>2-213</td>
</tr>
<tr>
<td>ellipke</td>
<td>Complete elliptic integrals of the first and second kind</td>
<td>2-215</td>
</tr>
<tr>
<td>erf, erfc, erfcx, erfinv</td>
<td>Error functions</td>
<td>2-223</td>
</tr>
<tr>
<td>expint</td>
<td>Exponential integral</td>
<td>2-234</td>
</tr>
<tr>
<td>gamma, gammainc, gammain</td>
<td>Gamma functions</td>
<td>2-326</td>
</tr>
</tbody>
</table>

1-8
Coordinate System Conversion

Legendre

Associated Legendre functions ........................................... page 2-411

pow2

Base 2 power and scale floating-point numbers ..................... page 2-530

rat, rats

Rational fraction approximation ........................................... page 2-555

Coordinate System Conversion

cart2pol

Transform Cartesian coordinates to polar or cylindrical .......... page 2-83

cart2sph

Transform Cartesian coordinates to spherical ....................... page 2-85

pol2cart

Transform polar or cylindrical coordinates to Cartesian .......... page 2-517

sph2cart

Transform spherical coordinates to Cartesian ........................ page 2-615

Matrix Functions - Numerical Linear Algebra

Matrix Analysis

cond

Condition number with respect to inversion ....................... page 2-121

condeig

Condition number with respect to eigenvalues ..................... page 2-122

det

Matrix determinant ....................................................... page 2-189

norm

Vector and matrix norms ................................................ page 2-468

null

Null space of a matrix ................................................... page 2-471

orth

Range space of a matrix ................................................ page 2-496

rank

Rank of a matrix .......................................................... page 2-554

rcond

Matrix reciprocal condition number estimate ...................... page 2-558

rref, rrefmovie

Reduced row echelon form. ............................................... page 2-576

subspace

Angle between two subspaces ......................................... page 2-659

trace

Sum of diagonal elements ................................................. page 2-678

Linear Equations

\ /

Linear equation solution ................................................ page 2-2

chol

Cholesky factorization .................................................... page 2-103

inv

Matrix inverse ............................................................... page 2-393

lscov

Least squares solution in the presence of known covariance .... page 2-427

lu

LU matrix factorization .................................................... page 2-428

nnls

Nonnegative least squares .............................................. page 2-464

pinv

Moore-Penrose pseudoinverse of a matrix ............................ page 2-514

qr

Orthogonal-triangular decomposition ................................ page 2-539
### Eigenvalues and Singular Values

- **balance**
  - Improve accuracy of computed eigenvalues.
  - Page 2-45
- **cdf2rdf**
  - Convert complex diagonal form to real block diagonal form.
  - Page 2-89
- **eig**
  - Eigenvalues and eigenvectors.
  - Page 2-204
- **hess**
  - Hessenberg form of a matrix.
  - Page 2-348
- **poly**
  - Polynomial with specified roots.
  - Page 2-518
- **qz**
  - QZ factorization for generalized eigenvalues.
  - Page 2-548
- **rsf2csf**
  - Convert real Schur form to complex Schur form.
  - Page 2-578
- **schur**
  - Schur decomposition.
  - Page 2-584
- **svd**
  - Singular value decomposition.
  - Page 2-662

### Matrix Functions

- **expm**
  - Matrix exponential.
  - Page 2-236
- **funm**
  - Evaluate functions of a matrix.
  - Page 2-298
- **logm**
  - Matrix logarithm.
  - Page 2-422
- **sqrtm**
  - Matrix square root.
  - Page 2-631

### Low Level Functions

- **qrdelete**
  - Delete column from QR factorization.
  - Page 2-542
- **qrinsert**
  - Insert column in QR factorization.
  - Page 2-543

### Data Analysis and Fourier Transform Functions

#### Basic Operations

- **convhull**
  - Convex hull.
  - Page 2-128
- **cumprod**
  - Cumulative product.
  - Page 2-139
- **cumsum**
  - Cumulative sum.
  - Page 2-140
- **cumtrapz**
  - Cumulative trapezoidal numerical integration.
  - Page 2-141
- **delaunay**
  - Delaunay triangulation.
  - Page 2-185
- **dsearch**
  - Search for nearest point.
  - Page 2-201
- **factor**
  - Prime factors.
  - Page 2-240
- **inpolygon**
  - Detect points inside a polygonal region.
  - Page 2-376
- **max**
  - Maximum elements of an array.
  - Page 2-443
- **mean**
  - Average or mean value of arrays.
  - Page 2-444
- **median**
  - Median value of arrays.
  - Page 2-445
- **min**
  - Minimum elements of an array.
  - Page 2-452
- **perms**
  - All possible permutations.
  - Page 2-511
- **polyarea**
  - Area of polygon.
  - Page 2-521
- **primes**
  - Generate list of prime numbers.
  - Page 2-531
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prod</code></td>
<td>Product of array elements</td>
<td>2-532</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort elements in ascending order</td>
<td>2-599</td>
</tr>
<tr>
<td><code>sortrows</code></td>
<td>Sort rows in ascending order</td>
<td>2-600</td>
</tr>
<tr>
<td><code>std</code></td>
<td>Standard deviation</td>
<td>2-639</td>
</tr>
<tr>
<td><code>sum</code></td>
<td>Sum of array elements</td>
<td>2-660</td>
</tr>
<tr>
<td><code>trapz</code></td>
<td>Trapezoidal numerical integration</td>
<td>2-679</td>
</tr>
<tr>
<td><code>tsearch</code></td>
<td>Search for enclosing Delaunay triangle</td>
<td>2-683</td>
</tr>
<tr>
<td><code>voronoi</code></td>
<td>Voronoi diagram</td>
<td>2-694</td>
</tr>
</tbody>
</table>

**Finite Differences**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>del2</code></td>
<td>Discrete Laplacian</td>
<td>2-182</td>
</tr>
<tr>
<td><code>diff</code></td>
<td>Differences and approximate derivatives</td>
<td>2-192</td>
</tr>
<tr>
<td><code>gradient</code></td>
<td>Numerical gradient</td>
<td>2-338</td>
</tr>
</tbody>
</table>

**Correlation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>corrcoef</code></td>
<td>Correlation coefficients</td>
<td>2-130</td>
</tr>
<tr>
<td><code>cov</code></td>
<td>Covariance matrix</td>
<td>2-133</td>
</tr>
</tbody>
</table>

**Filtering and Convolution**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conv</code></td>
<td>Convolution and polynomial multiplication</td>
<td>2-125</td>
</tr>
<tr>
<td><code>conv2</code></td>
<td>Two-dimensional convolution</td>
<td>2-126</td>
</tr>
<tr>
<td><code>deconv</code></td>
<td>Deconvolution and polynomial division</td>
<td>2-181</td>
</tr>
<tr>
<td><code>filter</code></td>
<td>Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter</td>
<td>2-255</td>
</tr>
<tr>
<td><code>filter2</code></td>
<td>Two-dimensional digital filtering</td>
<td>2-257</td>
</tr>
</tbody>
</table>

**Fourier Transforms**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Absolute value and complex magnitude</td>
<td>2-18</td>
</tr>
<tr>
<td><code>angle</code></td>
<td>Phase angle</td>
<td>2-30</td>
</tr>
<tr>
<td><code>cplxpair</code></td>
<td>Sort complex numbers into complex conjugate pairs</td>
<td>2-135</td>
</tr>
<tr>
<td><code>fft</code></td>
<td>One-dimensional fast Fourier transform</td>
<td>2-245</td>
</tr>
<tr>
<td><code>fft2</code></td>
<td>Two-dimensional fast Fourier transform</td>
<td>2-248</td>
</tr>
<tr>
<td><code>fftpshift</code></td>
<td>Shift DC component of fast Fourier transform to center of spectrum</td>
<td>2-250</td>
</tr>
<tr>
<td><code>ifft</code></td>
<td>Inverse one-dimensional fast Fourier transform</td>
<td>2-356</td>
</tr>
<tr>
<td><code>ifft2</code></td>
<td>Inverse two-dimensional fast Fourier transform</td>
<td>2-357</td>
</tr>
<tr>
<td><code>nextpow2</code></td>
<td>Next power of two</td>
<td>2-463</td>
</tr>
<tr>
<td><code>unwrap</code></td>
<td>Correct phase angles</td>
<td>2-688</td>
</tr>
</tbody>
</table>
Vector Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cross</code></td>
<td>Vector cross product</td>
<td>2-137</td>
</tr>
<tr>
<td><code>intersect</code></td>
<td>Set intersection of two vectors</td>
<td>2-392</td>
</tr>
<tr>
<td><code>ismember</code></td>
<td>Detect members of a set</td>
<td>2-403</td>
</tr>
<tr>
<td><code>setdiff</code></td>
<td>Return the set difference of two vectors</td>
<td>2-589</td>
</tr>
<tr>
<td><code>setxor</code></td>
<td>Set exclusive-or of two vectors</td>
<td>2-592</td>
</tr>
<tr>
<td><code>union</code></td>
<td>Set union of two vectors</td>
<td>2-686</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>Unique elements of a vector</td>
<td>2-687</td>
</tr>
</tbody>
</table>

Polynomial and Interpolation Functions

Polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conv</code></td>
<td>Convolution and polynomial multiplication</td>
<td>2-125</td>
</tr>
<tr>
<td><code>deconv</code></td>
<td>Deconvolution and polynomial division</td>
<td>2-181</td>
</tr>
<tr>
<td><code>poly</code></td>
<td>Polynomial with specified roots</td>
<td>2-518</td>
</tr>
<tr>
<td><code>polyder</code></td>
<td>Polynomial derivative</td>
<td>2-522</td>
</tr>
<tr>
<td><code>polyeig</code></td>
<td>Polynomial eigenvalue problem</td>
<td>2-523</td>
</tr>
<tr>
<td><code>polyfit</code></td>
<td>Polynomial curve fitting</td>
<td>2-524</td>
</tr>
<tr>
<td><code>polyval</code></td>
<td>Polynomial evaluation</td>
<td>2-527</td>
</tr>
<tr>
<td><code>polyvalm</code></td>
<td>Matrix polynomial evaluation</td>
<td>2-528</td>
</tr>
<tr>
<td><code>residue</code></td>
<td>Convert between partial fraction expansion and polynomial coefficients</td>
<td>2-567</td>
</tr>
<tr>
<td><code>roots</code></td>
<td>Polynomial roots</td>
<td>2-572</td>
</tr>
</tbody>
</table>

Data Interpolation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>griddata</code></td>
<td>Data gridding</td>
<td>2-341</td>
</tr>
<tr>
<td><code>interp1</code></td>
<td>One-dimensional data interpolation (table lookup)</td>
<td>2-380</td>
</tr>
<tr>
<td><code>interp2</code></td>
<td>Two-dimensional data interpolation (table lookup)</td>
<td>2-383</td>
</tr>
<tr>
<td><code>interp3</code></td>
<td>Three-dimensional data interpolation (table lookup)</td>
<td>2-387</td>
</tr>
<tr>
<td><code>interpft</code></td>
<td>One-dimensional interpolation using the FFT method</td>
<td>2-389</td>
</tr>
<tr>
<td><code>interpn</code></td>
<td>Multidimensional data interpolation (table lookup)</td>
<td>2-390</td>
</tr>
<tr>
<td><code>meshgrid</code></td>
<td>Generate X and Y matrices for three-dimensional plots</td>
<td>2-447</td>
</tr>
<tr>
<td><code>ndgrid</code></td>
<td>Generate arrays for multidimensional functions and interpolation</td>
<td>2-461</td>
</tr>
<tr>
<td><code>spline</code></td>
<td>Cubic spline interpolation</td>
<td>2-616</td>
</tr>
</tbody>
</table>

Function Functions - Nonlinear Numerical Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dblquad</code></td>
<td>Numerical double integration</td>
<td>2-152</td>
</tr>
</tbody>
</table>
### Sparse Matrix Functions

#### Elementary Sparse Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spdiags</td>
<td>Extract and create sparse band and diagonal matrices</td>
<td>2-609</td>
</tr>
<tr>
<td>speye</td>
<td>Sparse identity matrix</td>
<td>2-613</td>
</tr>
<tr>
<td>sprand</td>
<td>Sparse uniformly distributed random matrix</td>
<td>2-622</td>
</tr>
<tr>
<td>sprandn</td>
<td>Sparse normally distributed random matrix</td>
<td>2-623</td>
</tr>
<tr>
<td>sprandsym</td>
<td>Sparse symmetric random matrix</td>
<td>2-624</td>
</tr>
</tbody>
</table>

#### Full to Sparse Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>Find indices and values of nonzero elements</td>
<td>2-258</td>
</tr>
<tr>
<td>full</td>
<td>Convert sparse matrix to full matrix</td>
<td>2-294</td>
</tr>
<tr>
<td>sparse</td>
<td>Create sparse matrix</td>
<td>2-605</td>
</tr>
<tr>
<td>spconvert</td>
<td>Import matrix from sparse matrix external format</td>
<td>2-607</td>
</tr>
</tbody>
</table>

#### Working with Nonzero Entries of Sparse Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnz</td>
<td>Number of nonzero matrix elements</td>
<td>2-466</td>
</tr>
<tr>
<td>nonzeros</td>
<td>Nonzero matrix elements</td>
<td>2-467</td>
</tr>
<tr>
<td>nzmax</td>
<td>Amount of storage allocated for nonzero matrix elements</td>
<td>2-474</td>
</tr>
<tr>
<td>spalloc</td>
<td>Allocate space for sparse matrix</td>
<td>2-604</td>
</tr>
<tr>
<td>spfun</td>
<td>Apply function to nonzero sparse matrix elements</td>
<td>2-614</td>
</tr>
<tr>
<td>spones</td>
<td>Replace nonzero sparse matrix elements with ones</td>
<td>2-618</td>
</tr>
</tbody>
</table>

#### Visualizing Sparse Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spy</td>
<td>Visualize sparsity pattern</td>
<td>2-629</td>
</tr>
</tbody>
</table>
Reordering Algorithms

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>colmd</td>
<td>Sparse column minimum degree permutation</td>
<td>2-114</td>
</tr>
<tr>
<td>colperm</td>
<td>Sparse column permutation based on nonzero count</td>
<td>2-117</td>
</tr>
<tr>
<td>dmperm</td>
<td>Dulmage-Mendelsohn decomposition</td>
<td>2-198</td>
</tr>
<tr>
<td>randperm</td>
<td>Random permutation</td>
<td>2-553</td>
</tr>
<tr>
<td>symmd</td>
<td>Sparse symmetric minimum degree ordering</td>
<td>2-668</td>
</tr>
<tr>
<td>symrcm</td>
<td>Sparse reverse Cuthill-McKee ordering</td>
<td>2-670</td>
</tr>
</tbody>
</table>

Norm, Condition Number, and Rank

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>condest</td>
<td>1-norm matrix condition number estimate</td>
<td>2-123</td>
</tr>
<tr>
<td>norrest</td>
<td>2-norm estimate</td>
<td>2-469</td>
</tr>
</tbody>
</table>

Sparse Systems of Linear Equations

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bicg</td>
<td>BiConjugate Gradients method</td>
<td>2-58</td>
</tr>
<tr>
<td>bicgstab</td>
<td>BiConjugate Gradients Stabilized method</td>
<td>2-65</td>
</tr>
<tr>
<td>cgs</td>
<td>Conjugate Gradients Squared method</td>
<td>2-97</td>
</tr>
<tr>
<td>cholinc</td>
<td>Incomplete Cholesky factorizations</td>
<td>2-105</td>
</tr>
<tr>
<td>gmres</td>
<td>Generalized Minimum Residual method (with restarts)</td>
<td>2-334</td>
</tr>
<tr>
<td>lufact</td>
<td>Incomplete LU matrix factorizations</td>
<td>2-431</td>
</tr>
<tr>
<td>pcg</td>
<td>Preconditioned Conjugate Gradients method</td>
<td>2-506</td>
</tr>
<tr>
<td>qmr</td>
<td>Quasi-Minimal Residual method</td>
<td>2-535</td>
</tr>
</tbody>
</table>

Sparse Eigenvalues and Singular Values

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>eig</td>
<td>Find a few eigenvalues and eigenvectors</td>
<td>2-207</td>
</tr>
<tr>
<td>svds</td>
<td>A few singular values</td>
<td>2-664</td>
</tr>
</tbody>
</table>

Miscellaneous

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spparms</td>
<td>Set parameters for sparse matrix routines</td>
<td>2-619</td>
</tr>
</tbody>
</table>

Sound Processing Functions

General Sound Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound</td>
<td>Convert vector into sound</td>
<td>2-601</td>
</tr>
</tbody>
</table>

SPARCstation-specific Sound Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>aured</td>
<td>Read NeXT/SUN (.au) sound file</td>
<td>2-43</td>
</tr>
</tbody>
</table>
auwrite Write NeXT/SUN (. au) sound file. page 2-44

**.WAV Sound Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavread</td>
<td>Read Microsoft WAVE (. wav) sound file</td>
<td>2-697</td>
</tr>
<tr>
<td>wavwrite</td>
<td>Write Microsoft WAVE (. wav) sound file</td>
<td>2-698</td>
</tr>
</tbody>
</table>

**Macintosh Sound Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>readsnd</td>
<td>Read snd resources and files</td>
<td>2-559</td>
</tr>
<tr>
<td>recordsound</td>
<td>Record sound</td>
<td>2-563</td>
</tr>
<tr>
<td>soundcap</td>
<td>Sound capabilities</td>
<td>2-602</td>
</tr>
<tr>
<td>speak</td>
<td>Speak text string</td>
<td>2-612</td>
</tr>
<tr>
<td>writesnd</td>
<td>Write snd resources and files</td>
<td>2-711</td>
</tr>
</tbody>
</table>

**Character String Functions**

### General

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Absolute value and complex magnitude</td>
<td>2-18</td>
</tr>
<tr>
<td>eval</td>
<td>Interpret strings containing MATLAB expressions</td>
<td>2-228</td>
</tr>
<tr>
<td>real</td>
<td>Real part of complex number</td>
<td>2-560</td>
</tr>
<tr>
<td>strings</td>
<td>MATLAB string handling</td>
<td>2-646</td>
</tr>
</tbody>
</table>

### String Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>deblank</td>
<td>Strip trailing blanks from the end of a string</td>
<td>2-177</td>
</tr>
<tr>
<td>findstr</td>
<td>Find one string within another</td>
<td>2-260</td>
</tr>
<tr>
<td>lower</td>
<td>Convert string to lower case</td>
<td>2-426</td>
</tr>
<tr>
<td>strcat</td>
<td>String concatenation</td>
<td>2-642</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compare strings</td>
<td>2-644</td>
</tr>
<tr>
<td>strjust</td>
<td>Justify a character array</td>
<td>2-647</td>
</tr>
<tr>
<td>strmatch</td>
<td>Find possible matches for a string</td>
<td>2-648</td>
</tr>
<tr>
<td>strncmp</td>
<td>Compare the first n characters of two strings</td>
<td>2-649</td>
</tr>
<tr>
<td>strrep</td>
<td>String search and replace</td>
<td>2-650</td>
</tr>
<tr>
<td>strtok</td>
<td>First token in string</td>
<td>2-651</td>
</tr>
<tr>
<td>strvcat</td>
<td>Vertical concatenation of strings</td>
<td>2-654</td>
</tr>
<tr>
<td>upper</td>
<td>Convert string to upper case</td>
<td>2-689</td>
</tr>
</tbody>
</table>

### String to Number Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Create character array (string)</td>
<td>2-101</td>
</tr>
</tbody>
</table>
1 Command Summary

1 int2str
   Integer to string conversion ..................................... page 2-379
2 mat2str
   Convert a matrix into a string ..................................... page 2-440
3 num2str
   Number to string conversion ....................................... page 2-473
4 sprintf
   Write formatted data to a string ................................. page 2-625
5 sscanf
   Read string under format control ................................ page 2-635
6 str2num
   String to number conversion ........................................ page 2-641

Radix Conversion
7 bin2dec
   Binary to decimal number conversion ............................... page 2-69
8 dec2bin
   Decimal to binary number conversion ............................... page 2-179
9 dec2hex
   Decimal to hexadecimal number conversion ....................... page 2-180
10 hex2dec
    IEEE hexadecimal to decimal number conversion ............... page 2-350
11 hex2num
    Hexadecimal to double number conversion ....................... page 2-351

Low-Level File I/O Functions

File Opening and Closing
12 fclose
    Close one or more open files .................................... page 2-241
13 fopen
    Open a file or obtain information about open files ............ page 2-273

Unformatted I/O
14 fread
    Read binary data from file ...................................... page 2-284
15 fwrite
    Write binary data to a file ..................................... page 2-300

Formatted I/O
16 fgetl
    Return the next line of a file as a string without line terminator(s) ........................................ page 2-251
17 fgets
    Return the next line of a file as a string with line terminator(s) ........................................ page 2-252
18 fprintf
    Write formatted data to file ..................................... page 2-279
19 fscanf
    Read formatted data from file ................................... page 2-289

File Positioning
20 feof
    Test for end-of-file ............................................. page 2-242
21 ferror
    Query MATLAB about errors in file input or output ........ page 2-243
22 frewind
    Rewind an open file ............................................. page 2-288
23 fseek
    Set file position indicator ...................................... page 2-292
ftell
Get file position indicator .................................. page 2-293

String Conversion
sprintf
Write formatted data to a string ............................... page 2-625
sscanf
Read string under format control ............................. page 2-635

Specialized File I/O
qttwrite
Write QuickTime movie file to disk .......................... page 2-544
dlmread
Read an ASCII delimited file into a matrix ................. page 2-196
dlmwrite
Write a matrix to an ASCII delimited file ................ page 2-197
iminfo
Return information about a graphics file .................... page 2-360
imread
Read image from graphics file ............................... page 2-363
imwrite
Write an image to a graphics file ............................ page 2-366
wk1read
Read a Lotus123 WK1 spreadsheet file into a matrix ...... page 2-709
wk1write
Write a matrix to a Lotus123 WK1 spreadsheet file ......... page 2-710
xlgetrange
Get range of cells from Microsoft Excel worksheet ........ page 2-712
xlselrange
Set range of cells in Microsoft Excel worksheet ............ page 2-713

Bitwise Functions
bitand
Bit-wise AND .................................................. page 2-70
bitcmp
Complement bits ................................................. page 2-71
bitor
Bit-wise OR .................................................... page 2-74
bitmax
Maximum floating-point integer ............................... page 2-73
bitset
Set bit .................................................................. page 2-75
bitshift
Bit-wise shift ..................................................... page 2-76
bitget
Get bit ................................................................ page 2-72
bitxor
Bit-wise XOR ...................................................... page 2-77

Structure Functions
fielddnames
Field names of a structure ....................................... page 2-253
getfield
Get field of structure array ...................................... page 2-331
rmfield
Remove structure fields .......................................... page 2-570
setfield
Set field of structure array ...................................... page 2-590
struct
Create structure array ........................................... page 2-652
struct2cell
Structure to cell array conversion .............................. page 2-653
## Object Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class</code></td>
<td>Create object or return class of object</td>
<td>2-110</td>
</tr>
<tr>
<td><code>isa</code></td>
<td>Detect an object of a given class</td>
<td>2-402</td>
</tr>
</tbody>
</table>

## Cell Array Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cell</code></td>
<td>Create cell array</td>
<td>2-92</td>
</tr>
<tr>
<td><code>cellstr</code></td>
<td>Create cell array of strings from character array</td>
<td>2-96</td>
</tr>
<tr>
<td><code>cell2struct</code></td>
<td>Cell array to structure array conversion</td>
<td>2-93</td>
</tr>
<tr>
<td><code>celldisp</code></td>
<td>Display cell array contents</td>
<td>2-94</td>
</tr>
<tr>
<td><code>cellplot</code></td>
<td>Graphically display the structure of cell arrays</td>
<td>2-95</td>
</tr>
<tr>
<td><code>num2cell</code></td>
<td>Convert a numeric array into a cell array</td>
<td>2-472</td>
</tr>
</tbody>
</table>

## Multidimensional Array Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat</code></td>
<td>Concatenate arrays</td>
<td>2-87</td>
</tr>
<tr>
<td><code>flipdim</code></td>
<td>Flip array along a specified dimension</td>
<td>2-262</td>
</tr>
<tr>
<td><code>ind2sub</code></td>
<td>Subscripts from linear index</td>
<td>2-369</td>
</tr>
<tr>
<td><code>ipermute</code></td>
<td>Inverse permute the dimensions of a multidimensional array</td>
<td>2-397</td>
</tr>
<tr>
<td><code>ndgrid</code></td>
<td>Generate arrays for multidimensional functions and interpolation</td>
<td>2-461</td>
</tr>
<tr>
<td><code>ndims</code></td>
<td>Number of array dimensions</td>
<td>2-462</td>
</tr>
<tr>
<td><code>permute</code></td>
<td>Rearrange the dimensions of a multidimensional array</td>
<td>2-512</td>
</tr>
<tr>
<td><code>reshape</code></td>
<td>Reshape array</td>
<td>2-566</td>
</tr>
<tr>
<td><code>shiftdim</code></td>
<td>Shift dimensions</td>
<td>2-593</td>
</tr>
<tr>
<td><code>squeeze</code></td>
<td>Remove singleton dimensions</td>
<td>2-634</td>
</tr>
<tr>
<td><code>sub2ind</code></td>
<td>Single index from subscripts</td>
<td>2-655</td>
</tr>
</tbody>
</table>
Reference

This chapter describes all MATLAB operators, commands, and functions in alphabetical order.
Arithmetic Operators + - * / \ ^ '

**Purpose**
Matrix and array arithmetic

**Syntax**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>A+B</td>
</tr>
<tr>
<td>Subtraction</td>
<td>A-B</td>
</tr>
<tr>
<td>Multiplication</td>
<td>A*B</td>
</tr>
<tr>
<td>Division</td>
<td>A/B</td>
</tr>
<tr>
<td>Backslash</td>
<td>A\B</td>
</tr>
<tr>
<td>Power</td>
<td>A^B</td>
</tr>
<tr>
<td>Transpose</td>
<td>A'</td>
</tr>
</tbody>
</table>

**Description**

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

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

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

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

**/**
Matrix left division. 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 power. 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.

**\**
Array right division. A\ B is roughly the same as B* inv(A) . More precisely, A\ B = (B\ A)' . 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 \ B is roughly the same as inv(A) * B, except it is computed in a different way. If A is an n-by-n matrix and B is a column vector with n components, or a matrix with several such columns, then X = A \ B is the solution to the equation AX = B computed by Gaussian elimination (see “Algorithm” for details). A warning message prints if A is badly scaled or nearly singular. If A is an m-by-n matrix with m != n and B is a column vector with m components, or a matrix with several such columns, then X = A \ B is the solution in the least squares sense to the under- or overdetermined system of equations AX = B. The effective rank, k, of A, is determined from the QR decomposition with pivoting (see “Algorithm” for details). A solution X is computed which has at most k nonzero components per column. If k < n, this is usually not the same solution as pinv(A) * B, which is the least squares solution with the smallest norm, ||X||.

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

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

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

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

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

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

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

- **Binary addition**: \( A + B \) \( \text{plus}(A, B) \)
- **Unary plus**: \( +A \) \( \text{uplus}(A) \)
- **Binary subtraction**: \( A - B \) \( \text{minus}(A, B) \)
- **Unary minus**: \( -A \) \( \text{uminus}(A) \)
- **Matrix multiplication**: \( A \times B \) \( \text{mtimes}(A, B) \)
- **Array-wise multiplication**: \( A \odot B \) \( \text{times}(A, B) \)
- **Matrix right division**: \( A / B \) \( \text{mrdivide}(A, B) \)
- **Array-wise right division**: \( A \odot B \) \( \text{rdivide}(A, B) \)
- **Matrix left division**: \( A \backslash B \) \( \text{mldivide}(A, B) \)
- **Array-wise left division**: \( A \oslash B \) \( \text{ldivide}(A, B) \)
- **Matrix power**: \( A^B \) \( \text{mpower}(A, B) \)
- **Array-wise power**: \( A \odot^B \) \( \text{power}(A, B) \)
- **Complex transpose**: \( A' \) \( \text{ctranspose}(A) \)
- **Matrix transpose**: \( A.' \) \( \text{transpose}(A) \)
### Examples

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

<table>
<thead>
<tr>
<th>Matrix Operations</th>
<th>Array Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>x'</td>
<td>y'</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>x+y</td>
<td>x-y</td>
</tr>
<tr>
<td>5</td>
<td>-3</td>
</tr>
<tr>
<td>7</td>
<td>-3</td>
</tr>
<tr>
<td>9</td>
<td>-3</td>
</tr>
<tr>
<td>x + 2</td>
<td>x-2</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>x * y</td>
<td>x.*y</td>
</tr>
<tr>
<td>Error</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>x' * y</td>
<td>x'.*y</td>
</tr>
<tr>
<td>32</td>
<td>Error</td>
</tr>
<tr>
<td>x*y'</td>
<td>x.*y'</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>x*2</td>
<td>x.*2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>x\y</td>
<td>x./y</td>
</tr>
<tr>
<td>16/7</td>
<td>4</td>
</tr>
<tr>
<td>5/2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2\x</td>
<td>2./x</td>
</tr>
<tr>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3/2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2/3</td>
<td></td>
</tr>
</tbody>
</table>
### Algorithm

The specific algorithm used for solving the simultaneous linear equations denoted by $X = A \backslash B$ 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 `symm` 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. Nonnegative

### Arithmetic Operators

<table>
<thead>
<tr>
<th>Matrix Operations</th>
<th>Array Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x/y$</td>
<td>$x./y$</td>
</tr>
<tr>
<td>$x/2$</td>
<td>$x./2$</td>
</tr>
<tr>
<td>$x^y$</td>
<td>$x.^y$</td>
</tr>
<tr>
<td>$x^2$</td>
<td>$x.^2$</td>
</tr>
<tr>
<td>$2^x$</td>
<td>$2.^x$</td>
</tr>
<tr>
<td>$(x+i*y)'$</td>
<td>$1 - 4i$</td>
</tr>
<tr>
<td>$(x+i*y).'$</td>
<td>$1 + 4i$</td>
</tr>
</tbody>
</table>
definite matrices are usually detected almost immediately, so this check also requires little time. If successful, the Cholesky factorization is
\[ A = R' \ast R \]
where \( R \) is upper triangular. The solution \( X \) is computed by solving two triangular systems,
\[ X = R \backslash (R' \ast B) \]

- If \( A \) is square, but not a permutation of a triangular matrix, or is not Hermitian with positive elements, or the Cholesky factorization fails, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see \( lu \)). If \( A \) is sparse, a nonsymmetric minimum degree preordering is applied (see \( col \) and \( spparms \)). This results in
\[ A = L \ast U \]
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.
\[ X = U \backslash (L \backslash B) \]

- If \( A \) is not square and is full, then Householder reflections are used to compute an orthogonal-triangular factorization.
\[ A \ast P = Q \ast R \]
where \( P \) is a permutation, \( Q \) is orthogonal and \( R \) is upper triangular (see \( qr \)). The least squares solution \( X \) is computed with
\[ X = P \ast (R \backslash (Q' \ast B)) \]

- If \( A \) is not square and is sparse, then the augmented matrix is formed by:
\[ S = [c \ast I \ A; \ A' \ 0] \]
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 \( R = B \ast A \ast X \) are computed by
\[ S \ast [(R/c; \ X)] = [B; \ 0] \]
with minimum degree preordering and sparse Gaussian elimination with numerical pivoting.

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

**Diagnostics**

From matrix division, if a square A is singular:

Matrix is singular to working precision.

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

Divide by zero.

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

If the inverse was found, but is not reliable:

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

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

Warning: Rank deficient, rank = xxx tol = xxx

**See Also**

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

**References**

Relational Operators $<$  $>$  $<=$  $=>$  $==$  $~=$

**Purpose**

Relational operations

**Syntax**

A $<$ B  
A $>$ B  
A $<=$ B  
A $=>$ B  
A $==$ B  
A $~=$ B

**Description**

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

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

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

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

**Examples**

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

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

produce the same result:

```matlab
ans =
1 1 1
1 1 0
0 0 0
```
Relational Operators $<$  $>$  $<=$  $=>$  $==$  $==$  $~=$

| See Also | Logical Operators & | ~
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>find</td>
<td>Find indices and values of nonzero elements</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compare strings</td>
</tr>
</tbody>
</table>

2-10
Logical Operators & | ~

**Purpose**
Logical operations

**Syntax**
- $A \& B$
- $A \mid B$
- $\neg A$

**Description**
The symbols $\&$, $\mid$, and $\neg$ 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 $\neg A$ complements the elements of $A$. The function $\text{xor}(A, B)$ implements the exclusive OR operation. Truth tables for these operators and functions follow.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>$A$</th>
<th>$B$</th>
<th>and $A &amp; B$</th>
<th>or $A \mid B$</th>
<th>xor $\text{xor}(A, B)$</th>
<th>not $\neg A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $\neg$ has the highest precedence.
2. and $\&$ and or $\mid$ 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 $\neg A$ 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:

\[
1 \mid 0 \& 0 = 0 \\
0 \& 0 \mid 1 = 1
\]

**See Also**

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

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

Special characters

Syntax

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

Description

[ ] Brackets are used to form vectors and matrices. \[ 6, 9, 9.64, \sqrt{-1} \] is a vector with three elements separated by blanks. \[ 6.9, 9.64, i \] is the same thing. \[ 1+j, 2-j, 3 \] and \[ 1 +j, 2 –j, 3 \] are not the same. The first has three elements, the second has five. \[ 11, 12, 13; 21, 22, 23 \] is a 2-by-3 matrix. The semicolon ends the first row.

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

\( A = [ \] stores an empty matrix in \( A \). \( A(\ m:\ ) = [ \) deletes row \( m \) of \( A \).

\( A(:,\ n) = [ \) deletes column \( n \) of \( A \). \( A(\ n) = [ \) reshapes \( A \) into a column vector and deletes the third element.

\[ \{ A1, A2, A3, \ldots \} = \text{function} \] assigns function output to multiple variables.

For the use of [ and ] on the left of an “=” in multiple assignment statements, see \( \text{lu}, \text{eig}, \text{svd} \), and so on.

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

\( A(2,1) = \{[1\ 2\ 3; 4\ 5\ 6]\}, \) or \( A\{2,2\} = (\text{str}) \). See \text{help paren} for more information about \{ \}. 
Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If \( X \) and \( V \) are vectors, then \( X(V) = [X(V(1)), X(V(2)), \ldots, X(V(n))] \). 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 the `help paren` for more information about parentheses.

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 \( X \). \( ' \) 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. \( A.(\text{field}) \) and \( A(\text{i}).\text{field} \), when \( A \) is a structure, access the contents of \( \text{field} \).

\( .. \) Parent directory. See \( \text{cd} \).

\( ... \) 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

\[
\begin{align*}
\{A, B, C\ldots\} & \quad \text{horzcat} (A, B, C\ldots) \\
\{A; B; C\ldots\} & \quad \text{vertcat} (A, B, C\ldots)
\end{align*}
\]

Vertical concatenation

Subscript reference

\[
\begin{align*}
A(i,j,k\ldots) & \quad \text{subsref} (A, S) \text{. See help subsref.} \\
A(i,j,k\ldots) = B & \quad \text{subsasgn} (A, S, B) \text{. See help subsasgn.}
\end{align*}
\]

Subscript assignment

See Also Arithmetic, relational, and logical operators.
Colon :

**Purpose**
Create vectors, array subscripting, and for iterations

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

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

\[ j:k \]

is the same as \([j, j+1, \ldots, k]\)

\[ j:k \]

is empty if \(j > k\)

\[ j:i:k \]

is the same as \([j, j+i, j+2i, \ldots, k]\)

\[ j:i:k \]

is empty if \(i > 0\) and \(j > k\) or \(i < 0\) 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 \(A(j), A(j+1), \ldots, A(k)\)

\[ A(:,j:k) \]

is \(A(:,j), A(:,j+1), \ldots, A(:,k)\)

\[ A(:,:,k) \]

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 \(A(i,j,k,1), A(i,j,k,2), A(i,j,k,3), \ldots\).

\[ A(:) \]

is all the elements of \(A\), regarded as a single column. On the left side of an assignment statement, \(A(\cdot)\) 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

\[ A(:,:,2) = \text{pascal}(3) \]

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

\[ A(:,:,1) = \\
0     0     0 \\
0     0     0 \\
0     0     0 \]

\[ A(:,:,2) = \\
1     1     1 \\
1     2     3 \\
1     3     6 \]

See Also

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

Absolute value and complex magnitude

Syntax

Y = abs(X)

Description

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

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

\[ \text{abs}(X) = \sqrt{\text{real}(X)^2 + \text{imag}(X)^2} \]

Examples

\[
\begin{align*}
\text{abs}(-5) &= 5 \\
\text{abs}(3+4i) &= 5
\end{align*}
\]

See Also

angle Phase angle
sign Signum function
unwrap Correct phase angles
Purpose

Copy Macintosh file from one folder to another

Syntax

acopy(filename, foldername)

Description

acopy(filename, foldername) copies the file filename to the folder foldername. Both filename and foldername can be full or partial path names.

See Also

amove Move Macintosh file from one folder to another
applescript Load a compiled AppleScript from a file and execute it
arename Rename Macintosh File
areveal Reveal filename on Macintosh desktop
acos, acosh

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 \([-1, 1]\), \(\text{acos}(X)\) is real and in the range \([0, \pi]\). For real elements of X outside the domain \([-1, 1]\), \(\text{acos}(X)\) 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 \(1 \leq x \leq \pi\).

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

Algorithm
\[
\cos^{-1}(z) = -i \log\left[z + i \left(1 - z^2\right)^{1/2}\right]
\]
\[
\cosh^{-1}(z) = \log\left[z + (z^2 - 1)^{1/2}\right]
\]

See Also
cos, cosh
Cosine and hyperbolic cosine

2-20
Purpose

Inverse cotangent and inverse hyperbolic cotangent

Syntax

\[ Y = \text{acot}(X) \]
\[ Y = \text{acoth}(X) \]

Description

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

\[ Y = \text{acot}(X) \] returns the inverse cotangent (arccotangent) for each element of \( X \).

\[ Y = \text{acoth}(X) \] returns the inverse hyperbolic cotangent for each element of \( X \).

Examples

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 \(1 < x \leq 30\).

\[ x1 = -2\pi : \pi / 30: -0.1; \ x2 = 0.1: \pi / 30: 2\pi; \]
\[ \text{plot}(x1, \text{acot}(x1), x2, \text{acot}(x2)) \]
\[ x1 = -30: 0.1: -1.1; \ x2 = 1.1: 0.1: 30; \]
\[ \text{plot}(x1, \text{acoth}(x1), x2, \text{acoth}(x2)) \]

Algorithm

\[ \cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right) \]
\[ \coth^{-1}(z) = \tanh^{-1}\left(\frac{1}{z}\right) \]

See Also

`cot`, `coth`  
Cotangent and hyperbolic cotangent
**acsc, acsch**

**Purpose**
Inverse cosecant and inverse hyperbolic cosecant

**Syntax**
Y = acsc(X)
Y = acsch(X)

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

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

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

**Examples**
Graph the inverse cosecant over the domains \(-10 \leq x < -1\) and \(1 < x \leq 10\), and the inverse hyperbolic cosecant over the domains \(-20 \leq x < -1\) and \(1 \leq x \leq 20\).

```matlab
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**

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

\[ \csch^{-1}(z) = \sinh^{-1}\left(\frac{1}{z}\right) \]
See Also

csc, csch

Cosecant and hyperbolic cosecant
addpath

Purpose
Add directories to MATLAB's search path

Syntax
addpath('directory')
addpath('dir1','dir2','dir3',...)
addpath(...,'–flag')

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

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

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

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

Examples
path

MATLABPATH
C:\MATLAB\toolbox\general
C:\MATLAB\toolbox\ops
C:\MATLAB\toolbox\strfun

addpath('C:\MATLAB\myfiles')

path

MATLABPATH
C:\MATLAB\myfiles
C:\MATLAB\toolbox\general
C:\MATLAB\toolbox\ops
C:\MATLAB\toolbox\strfun

See Also
path
rmpath

Control MATLAB's directory search path
Remove directories from MATLAB's search path
**Purpose**
Airy functions

**Syntax**
- \( W = \text{airy}(Z) \)
- \( W = \text{airy}(k, Z) \)
- \( [W, ierr] = \text{airy}(k, Z) \)

**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:

\[
\begin{align*}
\text{Ai}(Z) &= \left[\frac{1}{\pi} \sqrt{\frac{1}{Z}}\right] K_{1/3}(\zeta) \\
\text{Bi}(Z) &= \sqrt{\frac{2}{3}} Z^{3/2} [I_{-1/3}(\zeta) + I_{1/3}(\zeta)]
\end{align*}
\]

where,

\[
\zeta = \frac{2}{3} Z^{3/2}
\]

**Description**
- \( W = \text{airy}(Z) \) returns the Airy function, \( \text{Ai}(Z) \), for each element of the complex array \( Z \).
- \( W = \text{airy}(k, Z) \) returns different results depending on the value of \( k \):

<table>
<thead>
<tr>
<th>( k )</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The same result as ( \text{airy}(Z) )</td>
</tr>
<tr>
<td>1</td>
<td>The derivative, ( \text{Ai}'(Z) )</td>
</tr>
<tr>
<td>2</td>
<td>The Airy function of the second kind, ( \text{Bi}(Z) )</td>
</tr>
<tr>
<td>3</td>
<td>The derivative, ( \text{Bi}'(Z) )</td>
</tr>
</tbody>
</table>
airy

\[
[\text{W ierr}] = \text{airy}(k, Z) \text{ also returns an array of error flags.}
\]

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

See Also
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

Purpose
Test to determine if all elements are nonzero

Syntax
\[
\begin{align*}
B &= \text{all}(A) \\
B &= \text{all}(A, \text{dim})
\end{align*}
\]

Description
\( \text{B} = \text{all}(A) \) tests whether all the elements along various dimensions of an array are nonzero or logical true (1).

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

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

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

\[
\begin{align*}
B &= \text{all}(A, \text{dim}) \text{ tests along the dimension of } A \\
&\text{specified by scalar } \text{dim}
\end{align*}
\]

Examples
Given,
\[
A = \begin{bmatrix}
0.53 & 0.67 & 0.01 & 0.38 & 0.07 & 0.42 & 0.69
\end{bmatrix}
\]
then \( B = (A < 0.5) \) returns logical true (1) only where \( A \) is less than one half:
\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]
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,
\[
\begin{align*}
&\text{if } \text{all}(A < 0.5) \\
&\quad \text{do something}
\end{align*}
\]
end
where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

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

```matlab
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`
| **Purpose** | Move Macintosh file from one folder to another |
| **Syntax** | `amove(filename, foldername)` |
| **Description** | `amove(filename, foldername)` moves the file `<filename>` to the folder `<foldername>`. Both `<filename>` and `<foldername>` can be full or partial path names. |
| **See Also** | `acopy` Copy Macintosh file from one folder to another  
`applescript` Load a compiled AppleScript from a file and execute it  
`arename` Rename Macintosh File  
`areveal` Reveal filename on Macintosh desktop |
**Purpose**
Phase angle

**Syntax**
\[ P = \text{angle}(Z) \]

**Description**
\[ P = \text{angle}(Z) \]
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
\[
R = \text{abs}(Z) \quad \text{% magnitude}\n\]
\[
\theta = \text{angle}(Z) \quad \text{% phase angle}\n\]
and the statement
\[ Z = R \times \exp(i \times \theta) \]
converts back to the original complex \( Z \).

**Examples**
\[
Z = \\
1.0000 - 1.0000i \quad 2.0000 + 1.0000i \quad 3.0000 - 1.0000i \quad 4.0000 + 1.0000i \\
1.0000 + 2.0000i \quad 2.0000 - 2.0000i \quad 3.0000 + 2.0000i \quad 4.0000 - 2.0000i \\
1.0000 - 3.0000i \quad 2.0000 + 3.0000i \quad 3.0000 - 3.0000i \quad 4.0000 + 3.0000i \\
1.0000 + 4.0000i \quad 2.0000 - 4.0000i \quad 3.0000 + 4.0000i \quad 4.0000 - 4.0000i \\
\]
\[
P = \text{angle}(Z) \\
P = \\
-0.7854 \quad 0.4636 \quad -0.3218 \quad 0.2450 \\
1.1071 \quad -0.7854 \quad 0.5880 \quad -0.4636 \\
-1.2490 \quad 0.9828 \quad -0.7854 \quad 0.6435 \\
1.3258 \quad -1.1071 \quad 0.9273 \quad -0.7854 \\
\]

**Algorithm**
\[ \text{angle}(z) = \text{imag}(\log(z)) = \text{atan2}(\text{imag}(z), \text{real}(z)) \]

**See Also**
- abs  
  Absolute value and complex magnitude
- unwrap  
  Correct phase angles
**Purpose**  
The most recent answer

**Syntax**  
ans

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

**Examples**  
The statement

\[ 2+2 \]

is the same as

\[ \text{ans} \ = \ 2+2 \]
any

Purpose
Test for any nonzeros

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

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

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

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

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

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

Examples
Given,
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
then B = (A < 0.5) returns logical true (1) only where A is less than one half:
0 0 1 1 1 1 0

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

This makes any particularly useful in if statements,
if any(A < 0.5)
dosomething
end
where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

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

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

**See Also**

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

`all` Test to determine if all elements are nonzero

Other functions that collapse an array’s dimensions include:

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

**Purpose**
Load a compiled AppleScript from a file and execute it

**Syntax**
```
applescript(filename)
applescript(filename, '-useEnglish')
result = applescript(filename)
applescript(filename, 'VarName1', 'VarValue1', ...)
```

**Description**
- `applescript(filename)` loads a compiled AppleScript from the file `filename` and executes it. If `filename` is not a full path name, then `applescript` searches for `filename` along the MATLAB path.
- `applescript(filename, '-useEnglish')` forces `applescript` to use the English AppleScript dialect when compiling both the script in `filename` and any AppleScript variables passed to the script. By default, `applescript` uses the current system AppleScript dialect, which can be set with (for example) the Script Editor application.
- `result = applescript(filename)` returns in `result` the value that the AppleScript returns, converted to a string.
- `applescript(filename, 'VarName1', 'VarValue1', ...)` sets the value of the AppleScript's property or variable whose name is specified in `VarName` to the value specified in `VarValue`.

**Remarks**
`applescript` is available on the Macintosh only.

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

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

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

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

Syntax
rename(oldfilename, newname)

Description
rename(oldfilename, newname) renames the file oldfilename to have the
name newname. oldfilename can be a full or partial path name.

See Also
acopy Copy Macintosh file from one folder to another
amove Move Macintosh file from one folder to another
areveal Reveal filename on Macintosh desktop
applescript Load a compiled AppleScript from a file and execute it
### Purpose
Reveal filename on Macintosh desktop

### Syntax
```
areveal(filename)
```

### Description
`areveal(filenam)` opens the window of the folder containing `filename` on the Macintosh desktop. `filename` can be a full or partial path name.

### See Also
- `acopy` Copy Macintosh file from one folder to another
- `amove` Move Macintosh file from one folder to another
- `applescript` Load a compiled AppleScript from a file and execute it
- `arename` Rename Macintosh File
Purpose
Inverse secant and inverse hyperbolic secant

Syntax
Y = asec(X)
Y = asech(X)

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

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

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

Examples
Graph the inverse secant over the domains 1 ≤ x ≤ 5 and -5 ≤ x ≤ -1, and the inverse hyperbolic secant over the domain 0 < x ≤ 1.

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

Algorithm
\[ \sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right) \]
\[ \sech^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right) \]

See Also
sec, sech
Secant and hyperbolic secant
asinh

Purpose
Inverse sine and inverse hyperbolic sine

Syntax
Y = asin(X)
Y = asinh(X)

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

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

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

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

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

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

See Also
sin, sinh
Sine and hyperbolic sine
Purpose  
Assign value to variable in workspace

Syntax  
assignin(ws,'name',v)

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

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

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

See Also  
evalin  Evaluate expression in workspace.
atan, atanh

Purpose
Inverse tangent and inverse hyperbolic tangent

Syntax
Y = atan(X)
Y = atanh(X)

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

Y = atan(X) returns the inverse tangent (arctangent) for each element of X.
For real elements of X, atan(X) is in the range \([-\pi/2, \pi/2]\).

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

Examples
Graph the inverse tangent function over the domain \(-20 \leq x \leq 20\), 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
\[
\tan^{-1}(z) = \frac{i}{2} \log \left( \frac{i + z}{1 - z} \right)
\]
\[
\tanh^{-1}(z) = \frac{1}{2} \log \left( \frac{1 + z}{1 - z} \right)
\]

See Also
at2an          Four-quadrant inverse tangent
atan, atanh

tan, tanh
Tangent and hyperbolic tangent
Purpose
Four-quadrant inverse tangent

Syntax
\[ P = \text{atan2}(Y, X) \]

Description
\( P = \text{atan2}(Y, X) \) 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 half-open interval \([\pi, 0] \). The specific quadrant is determined by \( \text{sign}(Y) \) and \( \text{sign}(X) \):

\[
\begin{array}{c}
\pi/2 \\
\pi \\
–\pi/2 \\
0
\end{array}
\begin{array}{c}
y \\
\pi/2 \\
\pi \\
–\pi/2
\end{array}
\begin{array}{c}
x
\end{array}
\]

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.

Examples
Any complex number \( z = x + iy \) is converted to polar coordinates with

\[
\begin{align*}
r &= \text{abs}(z) \\
\theta &= \text{atan2}(\text{imag}(z), \text{real}(z))
\end{align*}
\]

To convert back to the original complex number:

\[
z = r \times \exp(i \times \theta)
\]

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

See Also
\( \text{atan} \), \( \text{atanh} \)  
Inverse tangent and inverse hyperbolic tangent  
\( \text{tan} \), \( \text{tanh} \)  
Tangent and hyperbolic tangent
Purpose
Read NeXT/SUN (.au) sound file

Syntax
\[
y = \text{auread}(\text{aufile})
\]
\[
[y, Fs, bits] = \text{auread}(\text{aufile})
\]
\[
[\ldots] = \text{auread}(\text{aufile}, N)
\]
\[
[\ldots] = \text{auread}(\text{aufile}, [N1, N2])
\]
\[
siz = \text{auread}(\text{aufile}, 'size')
\]

Description
Supports multi-channel data in the following formats:

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

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

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

\[
[\ldots] = \text{auread}(\text{aufile}, N)\]
returns only the first N samples from each channel in the file.

\[
[\ldots] = \text{auread}(\text{aufile}, [N1, N2])\]
returns only samples N1 through N2 from each channel in the file.

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

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

Purpose
Write NeXT/SUN (.au) sound file

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

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

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

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

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

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

See Also
auread Read NeXT/SUN (.au) sound file
wavwrite Write Microsoft WAVE (.wav) sound file
Purpose
Improve accuracy of computed eigenvalues

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

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

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

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

$$\text{cond}( V ) = \text{norm}( V ) * \text{norm}( \text{inv}( V ) )$$

where

[ V, D ] = eig( A )

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

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

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

\[
[D, B] = \text{balance}(A)
\]

\[
D = \\
1.0e+03 * \\
2.0480 \ 0 \ 0 \\
0 \ 0.0320 \ 0 \\
0 \ 0 \ 0.0003
\]

\[
B = \\
1.0000 \ 1.5625 \ 1.2207 \\
0.6400 \ 1.0000 \ 0.7812 \\
0.8192 \ 1.2800 \ 1.0000
\]

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

\[
[V, E] = \text{eig}(A); V
\]

\[
V = \\
-1.0000 \ 0.9999 \ -1.0000 \\
0.0050 \ 0.0100 \ 0.0034 \\
0.0000 \ 0.0001 \ 0.0001
\]

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.

\[
[V, E] = \text{eig}(B); V
\]

\[
V = \\
-0.8873 \ 0.6933 \ 0.8919 \\
0.2839 \ 0.4437 \ -0.3264 \\
0.3634 \ 0.5679 \ -0.3129
\]
Now the eigenvectors are well behaved and cond(V) is 31.9814. The ill conditioning is concentrated in the scaling matrix; cond(D) is 8192.

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

**Algorithm**

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

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

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

**Limitations**

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

**Diagnostics**

If A is not a square matrix:

Matrix must be square.

**See Also**

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

**References**

**Purpose**  
Base to decimal number conversion

**Syntax**  
\[ d = \text{base2dec}(\text{strn}, \text{base}) \]

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

**Examples**  
The expression \( \text{base2dec('212', 3)} \) converts \( 212_3 \) to decimal, returning 23.

**See Also**  
\text{dec2base}
Bessel functions of the third kind (Hankel functions)

Syntax

\[ H = \text{besselh}(\nu, K, Z) \]
\[ H = \text{besselh}(\nu, Z) \]
\[ H = \text{besselh}(\nu, 1, Z, 1) \]
\[ H = \text{besselh}(\nu, 2, Z, 1) \]
\[ [H, ierr] = \text{besselh}(...) \]

Definitions

The differential equation

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

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

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

The relationship between the Hankel and Bessel functions is:

\[
H_1(\nu, z) = J_\nu(z) + i Y_\nu(z)
\]
\[
H_2(\nu, z) = J_\nu(z) - i Y_\nu(z)
\]

Description

\( H = \text{besselh}(\nu, K, Z) \) for \( K = 1 \) or 2 computes the Hankel functions \( H_1(\nu, z) \) or \( H_2(\nu, z) \) for each element of the complex array \( Z \). If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

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

\[ H = \text{besselh}(\nu, 1, Z, 1) \] scales \( H_1(\nu, z) \) by \( \exp(-i \times z) \).

\[ H = \text{besselh}(\nu, 2, Z, 1) \] scales \( H_2(\nu, z) \) by \( \exp(i \times z) \).
\[ [H, ierr] = \text{besselh}(...) \] also returns an array of error flags:

- \texttt{ierr} = 1 \quad \text{Illegal arguments.}
- \texttt{ierr} = 2 \quad \text{Overflow. Return Inf.}
- \texttt{ierr} = 3 \quad \text{Some loss of accuracy in argument reduction.}
- \texttt{ierr} = 4 \quad \text{Unacceptable loss of accuracy, Z or nu too large.}
- \texttt{ierr} = 5 \quad \text{No convergence. Return NaN.}
**Purpose**

Modified Bessel functions

**Syntax**

\[
I = \text{besseli}(n, Z) \quad \text{ Modified Bessel function of the 1st kind}
K = \text{besselk}(n, Z) \quad \text{ Modified Bessel function of the 3rd kind}
E = \text{besseli}(n, Z, 1)
K = \text{besselk}(n, Z, 1)
[I, ierr] = \text{besseli}(\ldots)
[K, ierr] = \text{besselk}(\ldots)
\]

**Definitions**

The differential equation

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

where \( \nu \) is a nonnegative constant, is called the modified Bessel's equation, and its solutions are known as modified Bessel functions.

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

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

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

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

**Description**

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

If \( n \) 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**

\[ K = \text{besselk}(\nu, Z) \]  
computes modified Bessel functions of the second kind, \( K_\nu(z) \), for each element of the complex array \( Z \).

\[ E = \text{besseli}(\nu, Z, 1) \]  
computes \( \text{besseli}(\nu, Z) \cdot \exp(-Z) \).

\[ K = \text{besselk}(\nu, Z, 1) \]  
computes \( \text{besselk}(\nu, Z) \cdot \exp(-Z) \).

\([l, ierr] = \text{besseli}(\ldots)\) and \([K, ierr] = \text{besselk}(\ldots)\) also return an array of error flags.

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

**Algorithm**
The \text{besseli} and \text{besselk} functions use a Fortran MEX-file to call a library developed by D. E. Amos [3] [4].

**See Also**
- airy  
  Airy functions
- \text{besselj, bessely}  
  Bessel functions

**References**
Purpose: Bessel functions

Syntax:

- \( J = \text{besselj}(\nu, Z) \)  
  Bessel function of the 1st kind
- \( Y = \text{bessely}(\nu, Z) \)  
  Bessel function of the 2nd kind
- \([J, i\text{err}] = \text{besselj}(\nu, Z)\)  
- \([Y, i\text{err}] = \text{bessely}(\nu, Z)\)

Definition:
The differential equation

\[
\frac{d^2}{dz^2} J_\nu(z) + \frac{1}{z} \frac{d}{dz} J_\nu(z) - (\nu^2 + \nu) J_\nu(z) = 0
\]

where \( \nu \) is a nonnegative constant, is called Bessel's equation, and its solutions are known as Bessel functions.

- \( J_\nu(z) \)
- \( J_{-\nu}(z) \)

form a fundamental set of solutions of Bessel's equation for noninteger \( \nu \).

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

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

Description:

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

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

- \( Y = \text{bessely}(\nu, Z) \) computes Bessel functions of the second kind, \( Y_{\nu}(z) \), for real, nonnegative order \( \nu \) and argument \( Z \).
\[ [J, \text{ ierr}] = \text{besselj}(n, Z) \quad \text{and} \quad [Y, \text{ ierr}] = \text{bessely}(n, Z) \quad \text{also return an array of error flags.} \]

\begin{align*}
\text{ierr} = 1 & \quad \text{Illegal arguments.} \\
\text{ierr} = 2 & \quad \text{Overflow. Return Inf.} \\
\text{ierr} = 3 & \quad \text{Some loss of accuracy in argument reduction.} \\
\text{ierr} = 4 & \quad \text{Unacceptable loss of accuracy, Z or nu too large.} \\
\text{ierr} = 5 & \quad \text{No convergence. Return NaN.} \\
\end{align*}

**Remarks**

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

\[
\begin{align*}
H^{(1)}_\nu(z) &= J_\nu(z) + i Y_\nu(z) \\
H^{(2)}_\nu(z) &= J_\nu(z) - i Y_\nu(z)
\end{align*}
\]

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

**Examples**

\text{besselj}(3:9, (0:.2:10)') generates the entire table on page 398 of Abramowitz and Stegun, Handbook of Mathematical Functions."

**Algorithm**

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

**See Also**

airy, \text{Airy functions} \\
bessel i, \text{bessel k, Modified Bessel functions}

**References**


**Purpose**  
Beta functions

**Syntax**

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

**Definition**

The beta function is:

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

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

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

**Description**

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

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

- **L = betaln(Z, W)** computes the natural logarithm of the beta function, \( \log(\text{beta}(Z, W)) \), 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.
Examples

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

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

In this case, with integer arguments,

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

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

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

Algorithm

beta(z,w) = exp(gammaln(z)+gammaln(w)–gammaln(z+w))
betaln(z,w) = gammaln(z)+gammaln(w)–gammaln(z+w)
Purpose
BiConjugate Gradients method

Syntax
[x, fl ag] = bicg(A, b, tol, maxi t, M1, M2)  
[x, fl ag, rel res, iter] = bicg(A, b, tol, maxi t, M1, M2, x0) 
[x, fl ag, rel res, iter, resvec] = bicg(A, b, tol, maxi t, M1, M2, x0)

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

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

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

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

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

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

x = bicg(A, b, tol, maxi t, Ml, M2, x0) returns a solution x. If bicg converged, a message to that effect is displayed. If bicg failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual norm(b−A*x)/norm(b) and the iteration number at which the method stopped or failed.

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bicg converged to the desired tolerance tol within maxi t iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>bicg iterated maxi t times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>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).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during bicg became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x, flag, relres] = bicg(A, b, tol, maxi t, Ml, M2, x0) also returns the relative residual norm(b−A*x)/norm(b). If flag is 0, then relres ≤ tol.

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

[x, flag, relres, iter, resvec] = bi cg(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 bi cg.

[x, flag, relres, iter, resvec] = bi cg(A, b)
flag =
1
relres =
1
iter =
0

The value of flag indicates that bi cg iterated the default 20 times without
converging. The value of iter shows that the method behaved so badly that the
initial all zero guess was better than all the subsequent iterates. The value of
relres supports this: relres = norm(b-A*x) / norm(b) = norm(b) / norm(b) = 1.
The plot `semilogy(0:20, resvec/norm(b), 'o')` below confirms that the unpreconditioned method oscillated rather wildly.

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

```matlab
[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 \( U_1 \). Thus it is singular. When we try to use it as a preconditioner:

\[
[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{bicg}(A, b, 1e^{-6}, 20, L_1, U_1)
\]

<table>
<thead>
<tr>
<th>flag</th>
<th>relres</th>
<th>iter</th>
<th>resvec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7.0557e+005</td>
</tr>
</tbody>
</table>

the method fails in the very first iteration when it tries to solve a system of equations involving the singular \( U_1 \) with backslash. It is forced to return the initial estimate since no other iterates were produced.

Try again with a slightly less sparse preconditioner:

\[
[L_2, U_2] = \text{luinc}(A, 1e^{-6})
\]

<table>
<thead>
<tr>
<th>nnz(L2)</th>
<th>nnz(U2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6231</td>
<td>4559</td>
</tr>
</tbody>
</table>

This time there is no warning message. All entries on the main diagonal of \( U_2 \) are nonzero.

\[
[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{bicg}(A, b, 1e^{-15}, 10, L_2, U_2)
\]

<table>
<thead>
<tr>
<th>flag</th>
<th>relres</th>
<th>iter</th>
<th>resvec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.8664e-16</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

and \text{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

\[
i\text{nv}(U)\ast i\text{nv}(L)\ast L\ast U\ast x = i\text{nv}(U)\ast i\text{nv}(L)\ast b,
\]

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 \texttt{bicg} using six different incomplete LU factors as preconditioners. Each line in the graph is labelled with the drop tolerance of the preconditioner used in \texttt{bicg}.

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

See Also

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

References

Purpose
BiConjugate Gradients Stabilized method

Syntax
x = bicgstab(A, b)
bicgstab(A, b, tol)
bicgstab(A, b, tol, maxit)
bicgstab(A, b, tol, maxit, M)
bicgstab(A, b, tol, maxit, M1, M2)
bicgstab(A, b, tol, maxit, M1, M2, x0)

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

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

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

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

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

\begin{verbatim}
  [M1, M2] = lu(M);
bicgstab(A, b, tol, maxit, M1, M2).
\end{verbatim}

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

\texttt{x = bicgstab}(A, b, tol, maxit, M1, M2, x0) returns a solution \texttt{x}. If \texttt{bicgstab} converged, a message to that effect is displayed. If \texttt{bicgstab} failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual \( \frac{\text{norm}(b - A \cdot x)}{\text{norm}(b)} \) and the iteration number at which the method stopped or failed.

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{bicgstab} converged to the desired tolerance \texttt{tol} within \texttt{maxit} iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>\texttt{bicgstab} iterated \texttt{maxit} times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>One of the systems of equations of the form ( M \cdot y = r ) involving the preconditioner was ill-conditioned and did not return a usable result when solved by ( \backslash ) (backslash).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during \texttt{bicgstab} became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever \texttt{flag} is not 0, the solution \texttt{x} returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the \texttt{flag} output is specified.
bicgstab

[x, flag, relres] = bicgstab(A, b, tol, maxit, M1, M2, x0) also returns the relative residual \( \frac{\|b-Ax\|}{\|b\|} \). If flag is 0, then relres \( \leq \) 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 \( \leq \) iter \( \leq \) 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) = \( \|b-Ax_0\| \). If flag is 0, resvec is of length 2*iter + 1, whether iter is an integer or not. In this case, resvec(end) \( \leq \) tol * \( \|b\| \).

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) = \( \|b\| \) and resvec2(end) = \( \|b-Ax_2\| \). You may follow the progress of bicgstab by plotting the relative residuals at the half way point and end of

2-67
bicgstab

each iteration starting from the initial estimate (iterate number 0) with

```
semilogy(0:0.5:iter2, resvec2/norm(b), '–o')
```

**See Also**

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

**References**


Purpose
Binary to decimal number conversion

Syntax
bin2dec(binarystr)

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

Examples
bin2dec('010111') returns 23.

See Also
dec2bin
Purpose

Bit-wise AND

Syntax

C = bitand(A, B)

Description

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

Examples

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

C = bitand(13, 27)

C =

9

See Also

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

Syntax
\[ C = \text{bitcmp}(A, n) \]

Description
\( C = \text{bitcmp}(A, n) \) returns the bit-wise complement of \( A \) as an \( n \)-bit floating-point integer (flint).

Example
With eight-bit arithmetic, the ones’ complement of 01100011 (99, decimal) is 10011100 (156, decimal).
\[
C = \text{bitcmp}(99, 8)
\]
\[
C = 156
\]

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

Get bit

Syntax

\[
C = \text{bitget}(A, \text{bit})
\]

Description

\[
C = \text{bitget}(A, \text{bit})
\]
returns the value of the bit at position \text{bit} in \(A\).Operand \(A\) must be a nonnegative integer, and \text{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 \text{ceil}, \text{fix}, \text{floor}, and \text{round} functions.

Example

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

\begin{verbatim}
   disp(dec2bin(13))
1101
   C = bitget(13, 4: -1: 1)

   C =
1 1 0 1
\end{verbatim}

See Also

- \text{bitand} Bit-wise AND
- \text{bitcmp} Complement bits
- \text{bitmax} Maximum floating-point integer
- \text{bitor} Bit-wise OR
- \text{bitset} Set bit
- \text{bitshift} Bit-wise shift
- \text{bitxor} Bit-wise XOR
Purpose
Maximum floating-point integer

Syntax
bitmax

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

See Also
- bit and: Bit-wise AND
- bit cmp: Complement bits
- bit get: Get bit
- bit or: Bit-wise OR
- bit set: Set bit
- bit shift: Bit-wise shift
- bit xor: Bit-wise XOR
Purpose
Bit-wise OR

Syntax
C = bitor(A, B)

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

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

C = bitor(13, 27)

C =

31

See Also
bitand - Bit-wise AND
bitor - Bit-wise OR
bitorcmp - Complement bits
bitget - Get bit
bitmax - Maximum floating-point integer
bitset - Set bit
bitshift - Bit-wise shift
bitxor - Bit-wise XOR
**Purpose**
Set bit

**Syntax**

\[ C = \text{bitset}(A, \text{bit}) \]
\[ C = \text{bitset}(A, \text{bit}, v) \]

**Description**

\[ C = \text{bitset}(A, \text{bit}) \] sets bit position \( \text{bit} \) in \( A \) to 1 (on). \( A \) must be a nonnegative integer and \( \text{bit} \) 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 \( \text{ceil} \), \( \text{fix} \), \( \text{floor} \), and \( \text{round} \) functions.

\[ C = \text{bitset}(A, \text{bit}, v) \] sets the bit at position \( \text{bit} \) 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.

\[ C = \text{bitset}(9, 5) \]

\[ C = 25 \]

**See Also**

- \texttt{bitand}  
  Bit-wise AND
- \texttt{bitcmp}  
  Complement bits
- \texttt{bitget}  
  Get bit
- \texttt{bitmax}  
  Maximum floating-point integer
- \texttt{bitor}  
  Bit-wise OR
- \texttt{bitshift}  
  Bit-wise shift
- \texttt{bitxor}  
  Bit-wise XOR
**Purpose**
Bit-wise shift

**Syntax**

\[ C = \text{bitshift}(A, n) \]

**Description**

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

**Examples**

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

\[ C = \text{bitshift}(12, 2) \]

\[ C = 48 \]

**See Also**

- \text{bitand}
- \text{bitcmp}
- \text{bitget}
- \text{bitmax}
- \text{bitor}
- \text{bitset}
- \text{bitxor}

Bit-wise AND
Complement bits
Get bit
Maximum floating-point integer
Bit-wise OR
Set bit
Bit-wise XOR
**Purpose**  
Bit-wise XOR

**Syntax**  
\[ C = \text{bitxor}(A, B) \]

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

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

\[ C = \text{bitxor}(13, 27) \]

\[
C = 22
\]

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

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

Syntax
break

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

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

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

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

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

Purpose
Calendar

Syntax
\[
c = \text{calendar}
\]
\[
c = \text{calendar}(d)
\]
\[
c = \text{calendar}(y,m)
\]
\[
c = \text{calendar}(\ldots)
\]

Description
\[
c = \text{calendar}
\text{ returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.}
\]
\[
c = \text{calendar}(d), \text{ where } d \text{ is a serial date number or a date string, returns a calendar for the specified month.}
\]
\[
c = \text{calendar}(y,m), \text{ where } y \text{ and } m \text{ are integers, returns a calendar for the specified month of the specified year.}
\]
\[
c = \text{calendar}(\ldots) \text{ displays the calendar on the screen.}
\]

Examples
The command:
\[
\text{calendar}(1957,10)
\]
reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

<table>
<thead>
<tr>
<th>Oct 1957</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

See Also
\[
datenum
\]
Serial date number

2-82
Purpose
Transform Cartesian coordinates to polar or cylindrical

Syntax

\[
[\text{THETA, RHO, Z}] = \text{cart2pol}(X, Y, Z) \\
[\text{THETA, RHO}] = \text{cart2pol}(X, Y)
\]

Description

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

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

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

Two-Dimensional Mapping
\[
\begin{align*}
\text{THETA} &= \text{atan2}(y, x) \\
\text{RHO} &= \sqrt{x^2 + y^2}
\end{align*}
\]

Three-Dimensional Mapping
\[
\begin{align*}
\text{THETA} &= \text{atan2}(y, x) \\
\text{RHO} &= \sqrt{x^2 + y^2} \\
\text{Z} &= z
\end{align*}
\]
cart2pol

See Also

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cart2sph</td>
<td>Transform Cartesian coordinates to spherical</td>
</tr>
<tr>
<td>pol2cart</td>
<td>Transform polar or cylindrical coordinates to Cartesian</td>
</tr>
<tr>
<td>sph2cart</td>
<td>Transform spherical coordinates to Cartesian</td>
</tr>
</tbody>
</table>
**Purpose**
Transform Cartesian coordinates to spherical

**Syntax**

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

**Description**

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

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

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

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

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

**See Also**

- cart2pol
  Transform Cartesian coordinates to polar or cylindrical
- pol2cart
  Transform polar or cylindrical coordinates to Cartesian
- sph2cart
  Transform spherical coordinates to Cartesian
**Purpose**

Case switch

**Description**

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

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

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

**Examples**

The general form of the `switch` statement is:

```plaintext
switch switch_expr
  case case_expr
      statement,...,statement
  case {case_expr1, case_expr2, case_expr3,...}
      statement,...,statement
  ...
  otherwise
      statement,...,statement
end
```

See `switch` for more details.

**See Also**

`switch`  
Switch among several cases based on expression
Purpose
Concatenate arrays

Syntax
\[
C = \text{cat}(\text{dim}, A, B)
\]
\[
C = \text{cat}(\text{dim}, A1, A2, A3, A4, \ldots)
\]

Description
\[
C = \text{cat}(\text{dim}, A, B)
\]
concatenates the arrays \(A\) and \(B\) along \(\text{dim}\).
\[
C = \text{cat}(\text{dim}, A1, A2, A3, A4, \ldots)
\]
concatenates all the input arrays \((A1, A2, A3, A4, \text{and so on})\) along \(\text{dim}\).
\[
cat(2, A, B)\] is the same as \(\begin{bmatrix} A \end{bmatrix}\) and \(\text{cat}(1, A, B)\) is the same as \(\begin{bmatrix} A; B\end{bmatrix}\).

Remarks
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.

Examples
Given,
\[
A = \begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6 \\
7 & 8
\end{bmatrix}
\]
\[
B = \begin{bmatrix}
5 & 6 \\
7 & 8 \\
1 & 2 \\
3 & 4
\end{bmatrix}
\]
concatenating along different dimensions produces:
\[
C = \text{cat}(1, A, B) \\
C = \text{cat}(2, A, B) \\
C = \text{cat}(3, A, B)
\]
The commands
\[
A = \text{magic}(3); \quad B = \text{pascal}(3); \\
C = \text{cat}(4, A, B);
\]
produce a 3-by-3-by-1-by-2 array.

See Also
\(\) (Special characters) Build arrays
e\(num2cell\) Convert a numeric array into a cell array
**cd**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Change working directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>cd</td>
</tr>
<tr>
<td></td>
<td>cd directory</td>
</tr>
<tr>
<td></td>
<td>cd ..</td>
</tr>
<tr>
<td>Description</td>
<td>cd, by itself, prints out the current directory.</td>
</tr>
<tr>
<td></td>
<td>cd directory sets the current directory to the one specified. On UNIX platforms, the character ~ is interpreted as the user’s root directory.</td>
</tr>
<tr>
<td></td>
<td>cd .. changes to the directory above the current one.</td>
</tr>
<tr>
<td>Examples</td>
<td>UNIX: cd /usr/local/matlab/toolbox/demos</td>
</tr>
<tr>
<td></td>
<td>DOS: cd C:MATLAB\DEMOS</td>
</tr>
<tr>
<td></td>
<td>VMS: cd DISK1: [MATLAB.DEMOS]</td>
</tr>
<tr>
<td></td>
<td>Macintosh: cd Toolbox:Demos</td>
</tr>
<tr>
<td></td>
<td>To specify a Macintosh directory name that includes spaces, enclose the name in single quotation marks, as in 'Toolbox:New M Files'.</td>
</tr>
<tr>
<td>See Also</td>
<td>dir Directory listing</td>
</tr>
<tr>
<td></td>
<td>path Control MATLAB’s directory search path</td>
</tr>
<tr>
<td></td>
<td>what Directory listing of M-files, MAT-files, and MEX-files</td>
</tr>
</tbody>
</table>
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, \text{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
\[ X = VDV' \]
continues to hold. The individual columns of \(V\) are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in \(D\) spans the corresponding invariant vectors.

Examples
The matrix
\[
X =
\begin{bmatrix}
1 & 2 & 3 \\
0 & 4 & 5 \\
0 & -5 & 4 \\
\end{bmatrix}
\]
has a pair of complex eigenvalues.
\[
[ V, D ] = \text{eig}(X)
\]

\[
V =
\begin{bmatrix}
1.0000 & 0.4002 - 0.0191i & 0.4002 + 0.0191i \\
0 & 0.6479 & 0.6479 \\
0 & 0 + 0.6479i & 0 - 0.6479i \\
\end{bmatrix}
\]

\[
D =
\begin{bmatrix}
1.0000 & 0 & 0 \\
0 & 4.0000 + 5.0000i & 0 \\
0 & 0 & 4.0000 - 5.0000i \\
\end{bmatrix}
\]
Converting this to real block diagonal form produces

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

\[
V =
\begin{bmatrix}
1.0000 & 0.4002 & -0.0191 \\
0 & 0.6479 & 0 \\
0 & 0 & 0.6479
\end{bmatrix}
\]

\[
D =
\begin{bmatrix}
1 & 0 & 0 \\
0 & 4 & 5 \\
0 & -5 & 4
\end{bmatrix}
\]

**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
**Purpose**  
Round toward infinity

**Syntax**  
\[ B = \text{ceil}(A) \]

**Description**  
\[ B = \text{ceil}(A) \] rounds the elements of \( A \) to the nearest integers greater than or equal to \( A \). For complex \( A \), the imaginary and real parts are rounded independently.

**Examples**  
\[
a = \\
\begin{array}{cccc}
-1.9000 & -0.2000 & 3.4000 & 5.6000 \\
7.0000 & 2.4000 + 3.6000i
\end{array}
\]

\[
\text{ceil}(a)
\]
\[
\begin{array}{cccc}
-1.0000 & 0 & 4.0000 & 6.0000 \\
7.0000 & 3.0000 + 4.0000i
\end{array}
\]

**See Also**  
- `fix`  
  Round toward zero  
- `floor`  
  Round toward minus infinity  
- `round`  
  Round to nearest integer
cell

Purpose
Create cell array

Syntax
\[
c = \text{cell}(n)
c = \text{cell}(m,n)
c = \text{cell}([m n])
c = \text{cell}(m,n,p,...)
c = \text{cell}([m n p ...])
c = \text{cell}(\text{size}(A))
\]

Description
\[
c = \text{cell}(n) \text{ creates an } n\text{-by-}n \text{ cell array of empty matrices. An error message appears if } n \text{ is not a scalar.}
\]
\[
c = \text{cell}(m,n) \text{ or } c = \text{cell}([m n]) \text{ creates an } m\text{-by-}n \text{ cell array of empty matrices. Arguments } m \text{ and } n \text{ must be scalars.}
\]
\[
c = \text{cell}(m,n,p,...) \text{ or } c = \text{cell}([m n p ...]) \text{ creates an } m\text{-by-}n\text{-by-}p\text{-... cell array of empty matrices. Arguments } m, n, p, ... \text{ must be scalars.}
\]
\[
c = \text{cell}([\text{size}(A)]) \text{ creates a cell array the same size as } A \text{ containing all empty matrices.}
\]

Examples
\[
A = \text{ones}(2,2)
\]
\[
A =
\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
\]
\[
c = \text{cell}([\text{size}(A)])
\]
\[
c =
\begin{bmatrix}
[ ] & [ ] \\
[ ] & [ ]
\end{bmatrix}
\]

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

2-92
Purpose

Cell array to structure array conversion

Syntax

\[ s = \text{cell2struct}(c, \text{fields}, \text{dim}) \]

Description

\[ s = \text{cell2struct}(c, \text{fields}, \text{dim}) \]

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

Examples

\[ c = \{\text{tree'}, 37.4, \text{birch'}\}; \]
\[ \text{f} = \{\text{category'}, \text{height'}, \text{name'}\}; \]
\[ s = \text{cell2struct}(c, f, 2) \]

\[ s = \]

\[
\begin{align*}
\text{category: 'tree'} \\
\text{height: 37.4000} \\
\text{name: 'birch'}
\end{align*}
\]

See Also

\[ \text{fieldnames} \quad \text{Field names of a structure} \]
\[ \text{struct2cell} \quad \text{Structure to cell array conversion} \]
 celldisp

Purpose
Display cell array contents.

Syntax
celldisp(C)

description
Celldisp(c) recursively displays the contents of a cell array.

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

\[
C = \begin{bmatrix}
    [1 2] & 'Tony' & 3+4i \\
    [1 2; 3 4] & -5 & 'abc'
\end{bmatrix}
\]

celldisp(C)

\[
C{1,1} = \\
1 \quad 2
\]

\[
C{2,1} = \\
1 \quad 2 \\
3 \quad 4
\]

\[
C{1,2} = \\
Tony
\]

\[
C{2,2} = \\
-5
\]

\[
C{1,3} = \\
3.0000 + 4.0000i
\]

\[
C{2,3} = \\
abc
\]

See Also
Celldisp

Graphically display the structure of cell arrays
Purpose

Graphically display the structure of cell arrays

Syntax

\begin{verbatim}
    cellplot(c)
    cellplot(c,'legend')
    handles = cellplot(...)
\end{verbatim}

Description

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

\texttt{cellplot(c, 'legend')} also puts a legend next to the plot.

\texttt{handles = cellplot(c)} displays a figure window and returns a vector of surface handles.

Limitations

The \texttt{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:

\begin{verbatim}
    c{1,1} = '2-by-2';
    c{1,2} = 'eigenvalues of eye(2)';
    c{2,1} = eye(2);
    c{2,2} = eig(eye(2));
\end{verbatim}

The command \texttt{cellplot(c)} produces:

![Cellplot diagram]
Purpose  
Create cell array of strings from character array

Syntax  
c = cellstr(S)

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

Examples  
Given the string matrix

\[
S = \\
abc \\
defg \\
hi
\]

The command \( c = \text{cellstr}(S) \) returns the 3-by-1 cell array:

\[
c = \\
'abc' \\
'defg' \\
'hi'
\]

See Also  
iscellstr  
True for cell array of strings

strings  
MATLAB string handling
Purpose  
Conjugate Gradients Squared method

Syntax

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

Description

\[ x = \text{cgs}(A, b) \] attempts to solve the system of linear equations \( A^*x = b \) for \( x \). The coefficient matrix \( A \) must be square and the right hand side (column) vector \( b \) must have length \( n \), where \( A \) is \( n \)-by-\( n \). \text{cgs} will start iterating from an initial estimate that by default is an all zero vector of length \( n \). Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate \( x \) has relative residual \( \| b - A^*x \|_2 / \| b \|_2 \) less than or equal to the tolerance of the method. The default tolerance is \( 1 \times 10^{-6} \). The default maximum number of iterations is the minimum of \( n \) and 20. No preconditioning is used.

\[ \text{cgs}(A, b, \text{tol}) \] specifies the tolerance of the method, \( \text{tol} \).

\[ \text{cgs}(A, b, \text{tol}, \text{maxit}) \] additionally specifies the maximum number of iterations, \( \text{maxit} \).

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

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

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cgs converged to the desired tolerance tol within maxi t iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>cgs iterated maxi t times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>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).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during cgs became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

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] = cgs(A, b, tol, maxi t, M1, M2, x0) also returns the relative residual norm(b–A*x)/norm(b). If flag is 0, then rel res ≤ tol.

[x, flag, rel res, iter] = cgs(A, b, tol, maxi t, M1, M2, x0) also returns the iteration number at which x was computed. This always satisfies 0 ≤ iter ≤ maxi t.
[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 \( \text{resvec}(1) = \text{norm}(b - A \times x_0) \). If \( \text{flag} \) is 0, \( \text{resvec} \) is of length \( \text{iter} + 1 \) and \( \text{resvec}(\text{end}) \leq \text{tol} \times \text{norm}(b) \).

Examples

\begin{verbatim}
load west0479
A = west0479
b = sum(A, 2)
[x, flag] = cgs(A, b)
\end{verbatim}

\( \text{flag} \) is 1 since \( \text{cgs} \) will not converge to the default tolerance \( 1 \times 10^{-6} \) within the default 20 iterations.

\begin{verbatim}
[L1, U1] = luinc(A, 1e-5)
[x1, flag1] = cgs(A, b, 1e-6, 20, L1, U1)
\end{verbatim}

\( \text{flag1} \) is 2 since the upper triangular \( \text{U1} \) has a zero on its diagonal so \( \text{cgs} \) fails in the first iteration when it tries to solve a system such as \( \text{U1} \times y = r \) for \( y \) with backslash.

\begin{verbatim}
[L2, U2] = luinc(A, 1e-6)
[x2, flag2, relres2, iter2, resvec2] = cgs(A, b, 1e-15, 10, L2, U2)
\end{verbatim}

\( \text{flag2} \) is 0 since \( \text{cgs} \) will converge to the tolerance of \( 7 \times 10^{-16} \) (the value of \( \text{relres2} \)) at the fifth iteration (the value of \( \text{iter2} \)) when preconditioned by the incomplete LU factorization with a drop tolerance of \( 1 \times 10^{-6} \). \( \text{resvec2}(1) = \text{norm}(b) \) and \( \text{resvec2}(6) = \text{norm}(b - A \times x2) \). You may follow the progress of \( \text{cgs} \).
by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with `semilogy(0:iter2,res2/norm(b),’–o’).

See Also

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

References


Purpose
Create character array (string)

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

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

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

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

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

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

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

See Also
g et, s et, and t ext in the online M ATLAB Function Reference, and:

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

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

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

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

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

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

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

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

R = chol (X)
R =
1 1 1 1 1
0 1 2 3 4
0 0 1 3 6
0 0 0 1 4
0 0 0 0 1
Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

\[
X(n, n) = X(n, n) - 1
\]

\[
X =
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 \\
1 & 3 & 6 & 10 & 15 \\
1 & 4 & 10 & 20 & 35 \\
1 & 5 & 15 & 35 & 69 \\
\end{pmatrix}
\]

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**

cholinc

**Purpose**
Incomplete Cholesky factorizations

**Syntax**
cholinc(X, '0')
R = cholinc(X, '0')
[R, p] = cholinc(X, '0')
R = cholinc(X, dr opt ol)
R = cholinc(X, opt ions)

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

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

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

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

The off-diagonal entries R(i, j) which are smaller in magnitude than the local drop tolerance, which is given by dr opt ol * norm(X(:,j))/R(i,i), are dropped from the factor. The diagonal entries are preserved even if they are too small in an attempt to avoid a singular factor.
R = cholinc(X, options) specifies a structure with up to three fields which may be used in any combination: droptol, michol, rdiag. Additional fields are ignored.

droptol is the drop tolerance of the incomplete factorization.

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

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

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

**Remarks**

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. A single 0 on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the rdiag option to replace a zero diagonal only gets rid of the symptoms of the problem, but it does not solve it. The preconditioner may not be singular, but it probably is not useful, and a warning message is printed.

**Examples**

Start with a symmetric positive definite matrix, S.

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

S is the two-dimensional, five-point discrete negative Lapacian on the grid generated by numgrid('C', 15).
Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make \( S \) singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level 0.

\[
C = \text{chol}(S);
\]
\[
R_0 = \text{chol}(S, '0');
\]
\[
S_2 = S; \quad S_2(101, 101) = 0;
\]
\[
[R, p] = \text{cholinc}(S_2, '0');
\]

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 \( S_2 \) stopped at row \( p = 101 \) resulting in a 100-by-139 partial factor.

\[
D_1 = (R_0' * R_0) \cdot \text{spones}(S) - S;
\]
\[
D_2 = (R' * R) \cdot \text{spones}(S_2) - S_2;
\]

\( D_1 \) has elements of the order of \( \text{eps} \), showing that \( R_0' * R_0 \) agrees with \( S \) over its sparsity pattern. \( D_2 \) has elements of the order of \( \text{eps} \) over its first 100 rows and first 100 columns, \( D_2(1:100, :) \) and \( D_2(:, 1:100) \).

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 $\frac{\text{norm}(R' \cdot R - S)}{\text{norm}(S)}$ in the next figure.
cholinc

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

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

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

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

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

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

**Purpose**
Create object or return class of object

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

**Description**
- `str = class(object)` returns a string specifying the class of `object`.
- The possible object classes are:
  - `cell` Multidimensional cell array
  - `double` Multidimensional double precision array
  - `sparse` Two-dimensional real (or complex) sparse array
  - `char` Array of alphanumeric characters
  - `struct` Structure
  - `'class_name'` User-defined object class

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

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

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

**See Also**
- `inferiorto` Inferior class relationship
- `isa` Detect an object of a given class
- `superiorto` Superior class relationship
clear

**Purpose**
Remove items from memory

**Syntax**
clear
clear name
clear name1 name2 name3...
clear global name

clear keyword where keyword is one of:
- functions
- variables
- mex
- global
- all

**Description**
clear, by itself, clears all variables from the workspace.
clear name removes just the M-file or MEX-file function or variable name from the workspace. If name is global, it is removed from the current workspace, but left accessible to any functions declaring it global.
clear name1 name2 name3 removes name1, name2, and name3 from the workspace.
clear global name removes the global variable name.
clear keyword clears the indicated items:
- clear functions Clears all the currently compiled M-functions from memory.
- clear variables Clears all variables from the workspace.
- clear mex Clears all MEX-files from memory.
- clear global Clears all global variables.
- clear all Removes all variables, functions, and MEX-files from memory, leaving the workspace empty.

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

Limitations

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

See Also

pack

Consolidate workspace memory
Purpose
Current time as a date vector

Syntax
\[ c = \text{clock} \]

Description
\text{clock} returns a 6-element date vector containing the current date and time in decimal form:

\[ c = [\text{year} \ \text{month} \ \text{day} \ \text{hour} \ \text{minute} \ \text{seconds}] \]

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

See Also
\begin{tabular}{ll}
\text{cputime} & CPU time in seconds \\
\text{datenum} & Serial date number \\
\text{datenum} & Date components \\
\text{etime} & Elapsed time \\
\text{tic} & Start a stopwatch timer \\
\text{toc} & Read the stopwatch timer \\
\end{tabular}
Purpose
Sparse column minimum degree permutation

Syntax
\[ p = \text{colmmd}(S) \]

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

The \text{colmmd} permutation is automatically used by \( \backslash \) and \( / \) for the solution of nonsymmetric and symmetric indefinite sparse linear systems.

Use \text{spparms} to change some options and parameters associated with heuristics in the algorithm.

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

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

Examples
The Harwell-Boeing collection of sparse matrices includes a test matrix ABB313. It is a rectangular matrix, of order 313-by-176, associated with least squares adjustments of geodesic data in the Sudan. Since this is a least squares problem, form the augmented matrix (see \text{spaugment} ), which is square and of order 489. The \text{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 \text{colmmd} ordering scrambles this...
structure. (Note that this example requires the Harwell-Boeing collection of software.)

```matlab
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.

\begin{verbatim}
spym(lu(S))
spym(lu(S(:,p)))
\end{verbatim}

See Also

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

References


Purpose  
Sparse column permutation based on nonzero count

Syntax  
j = colperm(S)

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

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

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

Examples  
The n-by-n arrowhead matrix

\[
A = [\text{ones}(1, n); \text{ones}(n-1, 1) \text{ speye}(n-1, n-1)]
\]

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

\[j = \text{colperm}(A)\]

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

On the other hand, the Bucky ball example, \(B = \text{bucky}\),

has exactly three nonzero elements in each row and column, so

\[j = \text{colperm}(B)\] is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

See Also  

<table>
<thead>
<tr>
<th>chol</th>
<th>Cholesky factorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>colmmd</td>
<td>Sparse minimum degree ordering</td>
</tr>
<tr>
<td>lu</td>
<td>LU matrix factorization</td>
</tr>
<tr>
<td>symrcm</td>
<td>Sparse reverse Cuthill-McKee ordering</td>
</tr>
</tbody>
</table>
Companion matrix

**Syntax**

\[ A = \text{compan}(u) \]

**Description**

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

**Examples**

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

\[
\begin{bmatrix}
0 & 7 & -6 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

The eigenvalues are the polynomial roots:

\[
\text{eig}\left(\text{compan}(u)\right)
\]

\[
\begin{bmatrix}
-3.0000 \\
2.0000 \\
1.0000
\end{bmatrix}
\]

This is also \(\text{roots}(u)\).

**See Also**

- \(\text{eig}\) Eigenvalues and eigenvectors
- \(\text{poly}\) Polynomial with specified roots
- \(\text{polyval}\) Polynomial evaluation
- \(\text{roots}\) Polynomial roots
computer

**Purpose**
Identify the computer on which MATLAB is running

**Syntax**
str = computer
[str, maxsize] = computer

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

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

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

<table>
<thead>
<tr>
<th>String</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN4</td>
<td>Sun4 SPARC workstation</td>
</tr>
<tr>
<td>SOL2</td>
<td>Solaris 2 SPARC workstation</td>
</tr>
<tr>
<td>PCWIN</td>
<td>MS-Windows</td>
</tr>
<tr>
<td>MAC2</td>
<td>All Macintosh</td>
</tr>
<tr>
<td>HP700</td>
<td>HP 9000/700</td>
</tr>
<tr>
<td>ALPHA</td>
<td>DEC Alpha</td>
</tr>
<tr>
<td>AXP_VMSG</td>
<td>Alpha VMS G_float</td>
</tr>
<tr>
<td>AXP_VMSI_EEE</td>
<td>Alpha VMS IEEE</td>
</tr>
<tr>
<td>VAX_VM6D</td>
<td>VAX/VMS D_float</td>
</tr>
</tbody>
</table>
### See Also

- **isieee**
- **isunix**
- **isvms**

<table>
<thead>
<tr>
<th>String</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX_VMSG</td>
<td>VAX/VMS G_float</td>
</tr>
<tr>
<td>LNX86</td>
<td>Linux Intel</td>
</tr>
<tr>
<td>SGI</td>
<td>Silicon Graphics (R4000)</td>
</tr>
<tr>
<td>SGI 64</td>
<td>Silicon Graphics (R8000)</td>
</tr>
<tr>
<td>IBM_RS</td>
<td>IBM RS6000 workstation</td>
</tr>
</tbody>
</table>
Purpose
Condition number with respect to inversion

Syntax
\[
c = \text{cond}(X) \\
c = \text{cond}(X, p)
\]

Description
The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of \(\text{cond}(X)\) 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:
\[
\text{norm}(X, p) \times \text{norm}(\text{inv}(X), p)
\]

<table>
<thead>
<tr>
<th>If (p) is...</th>
<th>Then (\text{cond}(X, p)) returns the...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-norm condition number</td>
</tr>
<tr>
<td>2</td>
<td>2-norm condition number</td>
</tr>
<tr>
<td>'fro'</td>
<td>Frobenius norm condition number</td>
</tr>
<tr>
<td>inf</td>
<td>Infinity norm condition number</td>
</tr>
</tbody>
</table>

Algorithm
The algorithm for \(\text{cond}\) (when \(p = 2\)) uses the singular value decomposition, \(\text{svd}\).

See Also
- \(\text{condeig}\)  Condition number with respect to eigenvalues
- \(\text{condeig}\)  1-norm matrix condition number estimate
- \(\text{nrm}\)  Vector and matrix norms
- \(\text{rank}\)  Rank of a matrix
- \(\text{svd}\)  Singular value decomposition

References
**condeig**

**Purpose**
Condition number with respect to eigenvalues

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

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

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

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

**See Also**
- `balance` Improve accuracy of computed eigenvalues
- `cond` Condition number with respect to inversion
- `eig` Eigenvalues and eigenvectors
Purpose

1-norm matrix condition number estimate

Syntax

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

Description

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

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

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

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

See Also

cond
nor est

Condition number with respect to inversion
2-norm estimate

Reference

Purpose: Complex conjugate

Syntax: \( ZC = \text{conj}(Z) \)

Description: \( ZC = \text{conj}(Z) \) returns the complex conjugate of the elements of \( Z \).

Algorithm: If \( Z \) is a complex array:
\[
\text{conj}(Z) = \text{real}(Z) - i \cdot \text{imag}(Z)
\]

See Also: 
- \( i \), \( j \) (Imaginary unit \((\sqrt{-1})\))
- \( \text{imag} \) (Imaginary part of a complex number)
- \( \text{real} \) (Real part of a complex number)
Purpose
Convolution and polynomial multiplication

Syntax
w = conv(u, v)

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

Definition
Let \( m = \text{length}(u) \) 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 \( j = \max(1, k+1-n) : m \leq n \leq k \). When \( m = n \), this gives

\[
  w(1) = u(1) \ast v(1)
  w(2) = u(1) \ast v(2) + u(2) \ast v(1)
  w(3) = u(1) \ast v(3) + u(2) \ast v(2) + u(3) \ast v(1)
  \ldots
  w(n) = u(1) \ast v(n) + u(2) \ast v(n-1) + \ldots + u(n) \ast v(1)
  \ldots
  w(2n-1) = u(n) \ast v(n)
\]

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

\[
  X = \text{fft}([x \text{ zeros}(1,1 \text{ length}(y)-1)]) \text{ and } Y = \text{fft}([y \text{ zeros}(1,1 \text{ length}(x)-1)])
\]

then \( \text{conv}(x, y) = \text{ifft}(X \ast Y) \)

See Also
convmtx, xconv2, xcorr, in the Signal Processing Toolbox, and:
decov
Deconvolution and polynomial division
filter
Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
Purpose  
Two-dimensional convolution

Syntax  
\[
C = \text{conv2}(A, B)  \\
C = \text{conv2}(hcol, hrow, A)  \\
C = \text{conv2}(\ldots, '\text{shape}')
\]

Description  
\(C = \text{conv2}(A, B)\) 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 \([ma+mb-1, na+nb-1]\).

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

\(C = \text{conv2}(\ldots, '\text{shape}')\) returns a subsection of the two-dimensional convolution, as specified by the \textit{shape} parameter:

- \texttt{full}  
Returns the full two-dimensional convolution (default).

- \texttt{same}  
Returns the central part of the convolution of the same size as \(A\).

- \texttt{valid}  
Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, \(C\) has size \([ma-nb+1, na-nb+1]\) when \(\text{size}(A) > \text{size}(B)\).

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

\[
s = \begin{bmatrix}
1 & 2 & 1 \\
0 & 0 & 0 \\
-1 & -2 & -1
\end{bmatrix}
\]

These commands extract the horizontal edges from a raised pedestal:

\[
A = \text{zeros}(10);  \\
A(3:7, 3:7) = \text{ones}(5);  \\
H = \text{conv2}(A, s);  \\
\text{mesh}(H)
\]
These commands display first the vertical edges of \( A \), then both horizontal and vertical edges.

\[
V = \text{conv2}(A, s^\prime);
\]
\[
\text{mesh}(V)
\]
\[
\text{mesh}(\sqrt{H.^2+V.^2})
\]

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)
convhull

Purpose
Convex hull

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

Description
K = convhull(x, y) returns indices into the x and y vectors of the points on the convex hull.
K = convhull(x, y, TRI) uses the triangulation (as obtained from delaunay) instead of computing it each time.

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

See Also
delaunay Delauney triangulation
polyarea Area of polygon
voronoi Voronoi diagram
Purpose
N-dimensional convolution

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

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

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

- 'full' returns the full N-dimensional convolution (default).
- 'same' returns the central part of the result that is the same size as A.
- 'valid' returns only those parts of the convolution that can be computed without assuming that the array A is zero-padded. The size of the result is
  \[ \max(\text{size}(A) - \text{size}(B) + 1, 0). \]

See Also
conv Convolutions and polynomial multiplication
conv2 Two-dimensional convolution
Purpose
Correlation coefficients

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

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

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

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

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

See Also
xcov, xcorr in the Signal Processing Toolbox, and:
cov Covariance matrix
mean Average or mean value of arrays
std 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 \(-5 \leq x \leq 5\).

\[ x = -\pi:0.01:\pi; \quad \text{plot}(x, \cos(x)) \]
\[ x = -5:0.01:5; \quad \text{plot}(x, \cosh(x)) \]

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
\[ \cos(x + iy) = \cos(x) \cosh(y) - i \sin(x) \sinh(y) \]
\[ \cos(z) = \frac{e^{iz} + e^{-iz}}{2} \]
\[ \cosh(z) = \frac{e^z + e^{-z}}{2} \]

See Also
acos, acosh  
Inverse cosine and inverse hyperbolic cosine
cot, coth

Purpose
Cotangent and hyperbolic cotangent

Syntax
Y = cot(X)
Y = coth(X)

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

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

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

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

Algorithm
\[
\cot(z) = \frac{1}{\tan(z)}
\]
\[
\coth(z) = \frac{1}{\tanh(z)}
\]

See Also
acot, acoth
Inverse cotangent and inverse hyperbolic cotangent
Purpose

Covariance matrix

Syntax

\[
C = \text{cov}(X)
\]
\[
C = \text{cov}(x, y)
\]

Description

\[ C = \text{cov}(x) \] 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 \( \text{cov}([x \ y]) \).

Remarks

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

The covariance function is defined as

\[
\text{cov}(x_1, x_2) = E[(x_1 - \mu_1)(x_2 - \mu_2)]
\]

where \( E \) is the mathematical expectation and \( \mu_i = \text{Ex}_i \).

Examples

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

\[
v = \text{diag}(\text{cov}(A))'
\]

\[
v =
\begin{bmatrix}
10.3333 \\
2.3333 \\
1.0000
\end{bmatrix}
\]

Compare vector \( v \) with covariance matrix \( C \):

\[
C =
\begin{bmatrix}
10.3333 & -4.1667 & 3.0000 \\
-4.1667 & 2.3333 & -1.5000 \\
3.0000 & -1.5000 & 1.0000
\end{bmatrix}
\]

The diagonal elements \( C(i,i) \) represent the variances for the columns of \( A \). The off-diagonal elements \( C(i,j) \) represent the covariances of columns \( i \) and \( j \).

See Also

\texttt{xcov}, \texttt{xcorr}, \texttt{corrcoef}, \texttt{mean}, \texttt{std} in the Signal Processing Toolbox, and:

\texttt{corrcoef} Correlation coefficients
\texttt{mean} Average or mean value of arrays
\texttt{std} Standard deviation
**cplxpair**

**Purpose**
Sort complex numbers into complex conjugate pairs

**Syntax**

```matlab
B = cplxpair(A)
B = cplxpair(A, tol)
B = cplxpair(A, [], dim)
B = cplxpair(A, tol, dim)
```

**Description**

The `cplxpair` function 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 $100 \times \text{eps}$ relative to $\text{abs}(A(i))$ determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message:

```
Complex numbers can't be paired.
```
### cputime

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Elapsed CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>cputime</code></td>
</tr>
<tr>
<td>Description</td>
<td><code>cputime</code> 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.</td>
</tr>
<tr>
<td>Examples</td>
<td>For example</td>
</tr>
<tr>
<td></td>
<td><code>t = cputime; surf(peaks(40)); e = cputime-t</code></td>
</tr>
<tr>
<td></td>
<td><code>e =</code></td>
</tr>
<tr>
<td></td>
<td><code>0.4667</code></td>
</tr>
<tr>
<td></td>
<td>returns the CPU time used to run <code>surf(peaks(40))</code>.</td>
</tr>
<tr>
<td>See Also</td>
<td><code>clock</code></td>
</tr>
<tr>
<td></td>
<td><code>Current time as a date vector</code></td>
</tr>
<tr>
<td></td>
<td><code>etime</code></td>
</tr>
<tr>
<td></td>
<td><code>Elapsed time</code></td>
</tr>
<tr>
<td></td>
<td><code>tic,toc</code></td>
</tr>
<tr>
<td></td>
<td><code>Stopwatch timer</code></td>
</tr>
</tbody>
</table>
**cross**

**Purpose**
Vector cross product

**Syntax**

- \( W = \text{cross}(U, V) \)
- \( W = \text{cross}(U, V, \text{dim}) \)

**Description**

- \( W = \text{cross}(U, V) \) 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, \( \text{cross} \) 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.

**Remarks**
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, \( c = a.' * b \).

**Examples**
The cross and dot products of two vectors are calculated as shown:

- \( a = [1 2 3]; b = [4 5 6] \);
- \( c = \text{cross}(a, b) \)

\[
\begin{align*}
c &= \\
&= \begin{bmatrix}
-3 & 6 & -3
\end{bmatrix}
\end{align*}
\]

- \( d = \text{sum}(a .* b) \)

\[
\begin{align*}
d &= \\
&= 32
\end{align*}
\]
Purpose
Cosecant and hyperbolic cosecant

Syntax
Y = csc(x)
Y = csch(x)

Description
The csc and csch functions operate element-wise on arrays. The functions’
domains and ranges include complex values. All angles are in radians.

Y = csc(x) returns the cosecant for each element of x.

Y = csch(x) returns the hyperbolic cosecant for each element of x.

Examples
Graph the cosecant and hyperbolic cosecant over the domains –π < x < 0 and
0 < x < π.

x1 = -pi +0.01:0.01:-0.01; x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2))
plot(x1,csch(x1),x2,csch(x2))

Algorithm
\[ \csc(z) = \frac{1}{\sin(z)} \]
\[ \csch(z) = \frac{1}{\sinh(z)} \]

See Also
acsc, acsch Inverse cosecant and inverse hyperbolic cosecant
Purpose
Cumulative product

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

Description
B = cumprod(A) returns the cumulative product along different dimensions of an array.

If A is a vector, cumprod(A) returns a vector containing the cumulative product of the elements of A.

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

If A is a multidimensional array, cumprod(A) works on the first nonsingleton dimension.

B = cumprod(A, dim) returns the cumulative product of the elements along the dimension of A specified by scalar dim. For example, cumprod(A, 1) increments the first (row) index, thus working along the rows of A.

Examples
cumprod(1:5) = [1 2 6 24 120]

A = [1 2 3; 4 5 6];

disp(cumprod(A))

1  2  3
4 10 18

disp(cumprod(A, 2))

1  2  6
4 20 120

See Also
cumsum
prod
sum
Cumulative sum
Product of array elements
Sum of array elements
Purpose
Cumulative sum

Syntax
B = cumsum(A)
B = cumsum(A, dim)

Description
B = cumsum(A) returns the cumulative sum along different dimensions of an array.
If A is a vector, cumsum(A) returns a vector containing the cumulative sum of the elements of A.
If A is a matrix, cumsum(A) returns a matrix the same size as A containing the cumulative sums for each column of A.
If A is a multidimensional array, cumsum(A) works on the first nonsingleton dimension.
B = cumsum(A, dim) returns the cumulative sum of the elements along the dimension of A specified by scalar dim. For example, cumsum(A, 1) works across the first dimension (the rows).

Examples
cumsum(1:5) = [1 3 6 10 15]
A = [1 2 3; 4 5 6];
disp( cumsum(A) )
  1  2  3
  5  7  9
disp( cumsum(A, 2) )
  1  3  6
  4  9 15

See Also
sum Sum of array elements
prod Product of array elements
cumprod Cumulative product of elements
Purpose
Cumulative trapezoidal numerical integration

Syntax
Z = cumtrapz(Y)
Z = cumtrapz(X, Y)
Z = cumtrapz(... dim)

Description
Z = cumtrapz(Y) computes an approximation of the cumulative integral of Y via the trapezoidal method with unit spacing. (This is similar to cumsum(Y), except that trapezoidal approximation is used.) To compute the integral with other than unit spacing, multiply z by the spacing increment.

For vectors, cumtrapz(Y) is the cumulative integral of Y.

For matrices, cumtrapz(Y) is a row vector with the cumulative integral over each column.

For multidimensional arrays, cumtrapz(Y) works across the first nonsingleton dimension.

Z = cumtrapz(X, Y) computes the cumulative integral of Y with respect to X using trapezoidal integration. X and Y must be vectors of the same length, or X must be a column vector and Y an array.

If X is a column vector and Y an array whose first nonsingleton dimension is length(X), cumtrapz(X, Y) operates across this dimension.

Z = cumtrapz(... dim) integrates across the dimension of Y specified by scalar dim. The length of X must be the same as size(Y, dim).

Example
Example: If Y = [0 1 2; 3 4 5]

cumtrapz(Y, 1)
ans =
   0   1.0000   2.0000
   1.5000   2.5000   3.5000
and

cumtrapz(Y, 2)
ans =
   0   0.5000   2.0000
   3.0000   3.5000   8.0000
See Also

- `cumsum`  
  Cumulative sum

- `trapz`  
  Trapezoidal numerical integration
cumtrapz
Purpose

Current date string

Syntax

str = date

Description

str = date returns a string containing the date in dd- mmm yyyy format.

See Also

clock

datenum

now

Current time as a date vector

Serial date number

Current date and time
datenum

Purpose
Serial date number

Syntax
N = datenum(str)
N = datenum(Y, M, D)
N = datenum(Y, M, D, H, M, S)

Description
The datenum function converts date strings and date vectors into serial date numbers. Date numbers are serial days elapsed from some reference date. By default, the serial day 1 corresponds to 1-Jan-0000.

N = datenum(str) converts the date string str into a serial date number.

NOTE The string str must be in one of the formats 0, 1, 2, 6, 13, 14, 15, or 16 as defined by datestr.

N = datenum(Y, M, D) returns the serial date number for corresponding elements of the Y, M, and D (year, month, day) arrays. Y, M, and D must be arrays of the same size (or any can be a scalar). Values outside the normal range of each array are automatically “carried” to the next unit.

N = datenum(Y, M, D, H, M, S) returns the serial date number for corresponding elements of the Y, M, D, H, M, and S (year, month, hour, minute, and second) array values. Y, M, D, H, M, and S must be arrays of the same size (or any can be a scalar).

Examples
n = datenum('19-May-1995') returns n = 728798.
n = datenum(1994, 12, 19) returns n = 728647.
n = datenum(1994, 12, 19, 18, 0, 0) returns n = 7.2865e+05.

See Also
datestr 
Datetstring format
datvec 
Date components
now 
Current date and time

2-144
**Purpose**  
Date string format

**Syntax**  
\[ str = 	ext{datestr}(D, 	ext{dateform}) \]

**Description**  
\[ str = 	ext{datestr}(D, 	ext{dateform}) \] converts each element of the array of serial date numbers \((D)\) to a string. Optional argument \text{dateform} specifies the date format of the result, where \text{dateform} can be either a number or a string:

<table>
<thead>
<tr>
<th>dat ef or m,(number)</th>
<th>dat ef or m,(string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'dd- mmm yyyy'</td>
<td>01- Mar-1995</td>
</tr>
<tr>
<td></td>
<td>'HH: MM SS'</td>
<td>03: 45</td>
</tr>
<tr>
<td>1</td>
<td>'dd- mmm yyyy'</td>
<td>01- Mar-1995</td>
</tr>
<tr>
<td>2</td>
<td>'mm/dd/ yy'</td>
<td>03/01/95</td>
</tr>
<tr>
<td>3</td>
<td>'mmm'</td>
<td>Mar</td>
</tr>
<tr>
<td>4</td>
<td>'m'</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>'mm'</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>'mm/dd'</td>
<td>03/01</td>
</tr>
<tr>
<td>7</td>
<td>'dd'</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>'ddd'</td>
<td>Wed</td>
</tr>
<tr>
<td>9</td>
<td>'d'</td>
<td>W</td>
</tr>
<tr>
<td>10</td>
<td>'yyyy'</td>
<td>1995</td>
</tr>
<tr>
<td>11</td>
<td>'yy'</td>
<td>95</td>
</tr>
<tr>
<td>12</td>
<td>'mmmy'</td>
<td>Mar 95</td>
</tr>
<tr>
<td>13</td>
<td>'HH: MM SS'</td>
<td>15: 45: 17</td>
</tr>
</tbody>
</table>
NOTE  dat ef or mnumbers 0, 1, 2, 6, 13, 14, 15, and 16 produce a string suitable for input to dat enum or dat evec. 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 dat ef or m 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
dat e  Current date string
dat enum  Serial date number
dat evec  Date components
Purpose

Date components

C = datvec(A)

[Y, M, D, H, MI, S] = datvec(A)

Description

C = datvec(A) splits its input into an n-by-6 array with each row containing the vector [Y, M, D, H, MI, S]. The first five date vector elements are integers. Input A can either consist of strings of the sort produced by the datestr function, or scalars of the sort produced by the datenum and now functions.

[Y, M, D, H, MI, S] = datvec(A) returns the components of the date vector as individual variables.

When creating your own date vector, you need not make the components integers. Any components that lie outside their conventional ranges affect the next higher component (so that, for instance, the anomalous June 31 becomes July 1). A zeroth month, with zero days, is allowed.

Examples

Let

d = '12/24/1984'
t = '725000.00',

Then datvec(d) and datvec(t) generate [1984 12 24 0 0 0].

See Also

clock

datenum

datestr

Current time as date vector
Serial date number
Date string format
dbclear

Purpose
Clear breakpoints

Syntax

dbclear


dbclear at line no in function

dbclear all in function

dbclear all


dbclear in filename


dbclear if keyword


Description

The at, in, and if keywords, familiar to users of the UNIX debugger dbx, are optional.

dbclear, by itself, clears the breakpoint(s) set by a corresponding dbstop command.

dbclear at line no in function clears the breakpoint set at the specified line in the specified M-file. function must be the name of an M-file function or a MATLABPATH relative partial pathname.

dbclear all in function clears all breakpoints in the specified M-file.

dbclear all clears all breakpoints in all M-file functions, except for errors and warning breakpoints.

dbclear in function clears the breakpoint set at the first executable line in the specified M-file.

dbclear if keyword clears the indicated statement or breakpoint:


dbclear if error

Clears the dbstop error statement, if set. If a runtime error occurs after this command, MATLAB terminates the current operation and returns to the base workspace.


dbclear if nani nf

Clears the dbstop nani nf statement, if set.
**dbclear**

`dbclear if infnan` clears the `dbstop infnan` statement, if set.

`dbclear if warning` clears warning breakpoints.

**See Also**

- `dbcont` - 2-Resume execution
- `dbdown` - 2-Change local workspace context (down)
- `dbquit` - 2-Quit debug mode
- `dbstack` - 2-Display function call stack
- `dbstatus` - 2-List all breakpoints
- `dbstep` - 2-Execute one or more lines from a breakpoint
- `dbstop` - 2-Set breakpoints in an M-file function
- `dbtype` - 2-List M-file with line numbers
- `dbup` - 2-Change local workspace context (up)

See also 2-partialpath.
**Purpose**
Resume execution

**Syntax**
dbcont

**Description**
dbcont resumes execution of an M-file from a breakpoint. Execution continues until either another breakpoint is encountered, an error occurs, or MATLAB returns to the base workspace prompt.

**See Also**
- dbclear
- dbdown
- dbquit
- dbstack
- dbstatus
- dbstep
- dbstop
- dbtype
- dbup

2-Clear breakpoints
2-Change local workspace context (down)
2-Quit debug mode
2-Display function call stack
2-List all breakpoints
2-Execute one or more lines from a breakpoint
2-Set breakpoints in an M-file function
2-List M-file with line numbers
2-Change local workspace context (up)
### dbdown

**Purpose**  
Change local workspace context

**Syntax**  
dbdown

**Description**  
dbdown changes the current workspace context to the workspace of the called M-file when a breakpoint is encountered. You must have issued the dbup command at least once before you issue this command. dbdown is the opposite of dbup.

Multiple dbdown commands change the workspace context to each successively executed M-file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

**See Also**  
- dbclear 2-Clear breakpoints
- dbcont 2-Resume execution
- dbquit 2-Quit debug mode
- dbstack 2-Display function call stack
- dbstatus 2-List all breakpoints
- dbstep 2-Execute one or more lines from a breakpoint
- dbstop 2-Set breakpoints in an M-file function
- dbtype 2-List M-file with line numbers
- dbup 2-Change local workspace context (up)
**Purpose**
Numerical double integration

**Syntax**
```
result = dblquad('fun', inmin, inmax, outmin, outmax)
result = dblquad('fun', inmin, inmax, outmin, outmax, tol, trace)
result = dblquad('fun', inmin, inmax, outmin, outmax, tol, trace, order)
```

**Description**
result = dblquad('fun', inmin, inmax, outmin, outmax) evaluates the double integral \( \int_{\text{inmin}}^{\text{inmax}} \int_{\text{outmin}}^{\text{outmax}} \text{fun}(\text{inner}, \text{outer}) \, d\text{inner} \, d\text{outer} \) using the quad quadrature function. inner is the inner variable, ranging from inmin to inmax, and outer is the outer variable, ranging from outmin to outmax. The first argument 'fun' is a string representing the integrand function. This function must be a function of two variables of the form \( \text{fout} = \text{fun}(\text{inner}, \text{outer}) \). The function must take a vector inner and a scalar outer and return a vector fout that is the function evaluated at outer and each value of inner.

result = dblquad('fun', inmin, inmax, outmin, outmax, tol, trace) passes tol and trace to the quad function. See the help entry for quad for a description of the tol and trace parameters.

result = dblquad('fun', inmin, inmax, outmin, outmax, tol, trace, order) passes tol and trace to the quad or quad8 function depending on the value of the string order. Valid values for order are 'quad' and 'quad8' or the name of any user-defined quadrature method with the same calling and return arguments as quad and quad8.

**Example**
result = dblquad('integrnd', pi, 2*pi, 0, pi) integrates the function \( y \sin(x) + x \cos(y) \), where x ranges from \( \pi \) to 2\( \pi \), and y ranges from 0 to \( \pi \), assuming:

- x is the inner variable in the integration.
- y is the outer variable.
- the M-file integrnd.m is defined as:

```matlab
function out = integrnd(x, y)
    out = y*sin(x)+x*cos(y);
end
```

Note that integrnd.m is valid when x is a vector and y is a scalar. Also, x must be the first argument to integrnd.m since it is the inner variable.
See Also

2-quad, quad8
2-Numerical evaluation of integrals
**Purpose**  
Enable MEX-file debugging

**Syntax**

- `dbmex on`
- `dbmex off`
- `dbmex stop`
- `dbmex print`

**Description**

`dbmex on` enables MEX-file debugging. To use this option, first start MATLAB from within a debugger by typing `matlab –Ddebugger`, where `debugger` is the name of the debugger.

`dbmex off` disables MEX-file debugging.

`dbmex stop` returns to the debugger prompt.

`dbmex print` displays MEX-file debugging information.

`dbmex` is not available on the Macintosh or the PC.

**See Also**

- `dbstop` 2-Set breakpoints in an M-file function
- `dbclear` 2-Clear breakpoints
- `dbcont` 2-Resume execution
- `dbdown` 2-Change local workspace context (down)
- `dbquit` 2-Quit debug mode
- `dbstack` 2-Display function call stack
- `dbstat` 2-List all breakpoints
- `dbstep` 2-Execute one or more lines from a breakpoint
- `dbtype` 2-List M-file with line numbers
- `dbup` 2-Change local workspace context (up)
**dbquit**

**Purpose**
Quit debug mode

**Syntax**
dbquit

**Description**
dbquit immediately terminates the debugger and returns control to the base workspace prompt. The M-file being processed is not completed and no results are returned.

All breakpoints remain in effect.

**See Also**
dbclear 2-Clear breakpoints
dbcont 2-Resume execution
dbdown 2-Change local workspace context (down)
dbstack 2-Display function call stack
dbstatus 2-List all breakpoints
dbstep 2-Execute one or more lines from a breakpoint
dbstop 2-Set breakpoints in an M-file function
dbtype 2-List M-file with line numbers
dbup 2-Change local workspace context (up)
Purpose
Display function call stack

Syntax
dbstack
[ST, I] = dbstack

Description
dbstack displays the line numbers and M-file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. In other words, the line number of the most recently executed function call (at which the current breakpoint occurred) is listed first, followed by its calling function, which is followed by its calling function, and so on, until the topmost M-file function is reached.

[ST, I] = dbstack returns the stack trace information in an m-by-1 structure ST with the fields:

- name function name
- line function line number

The current workspace index is returned in I.

Examples
>> dbstack
> In /usr/local/matlab/toolbox/matlab/cond.m at line 13
   In test1.m at line 2
   In test.m at line 3

See Also
dbclear 2-Clear breakpoints
dbcont 2-Resume execution
dbdown 2-Change local workspace context (down)
dbquit Quit debug mode
dbstatus 2-List all breakpoints
dbstep 2-Execute one or more lines from a breakpoint
dbstop 2-Set breakpoints in an M-file function
dbtype 2-List M-file with line numbers
dbup 2-Change local workspace context (up)
Purpose  
List all breakpoints

Syntax  
dbstatus
   dbstatus function
   s = dbstatus(...)

Description  
dbstatus, by itself, lists all breakpoints in effect including error, warning, and nainf.

   dbstatus function  displays a list of the line numbers for which breakpoints are set in the specified M-file.

   s = dbstatus(...)  returns the breakpoint information in an n-by-1 structure with the fields:

      name  function name
      line  vector of breakpoint line numbers
      cond  condition string (error, warning, or nainf)

Use dbstatus class/function or dbstatus private/function or dbstatus class/private/function to determine the status for methods, private functions, or private methods (for a class named class). In all of these forms you can further qualify the function name with a subfunction name as in dbstatus function/subfunction.

See Also  
dbclear  2-Clear breakpoints
dbcont  2-Resume execution
dbdown  2-Change local workspace context (down)
dbquit  2-Quit debug mode
dbstack  2-Display function call stack
dbstep  2-Execute one or more lines from a breakpoint
dbstop  2-Set breakpoints in an M-file function
dbtype  2-List M-file with line numbers
dbup  2-Change local workspace context (up)
dbstep

Purpose
Execute one or more lines from a breakpoint

Syntax
dbstep
dbstep nlines
dbstep in

Description
This command allows you to debug an M-file by following its execution from the current breakpoint. At a breakpoint, the dbstep command steps through execution of the current M-file one line at a time or at the rate specified by nlines.

dbstep, by itself, executes the next executable line of the current M-file. dbstep steps over the current line, skipping any breakpoints set in functions called by that line.

dbstep nlines executes the specified number of executable lines.

dbstep in steps to the next executable line. If that line contains a call to another M-file, execution resumes with the first executable line of the called file. If there is no call to an M-file on that line, dbstep in is the same as dbstep.

See Also
dbclear 2-Clear breakpoints
dbcont 2-Resume execution
dbdwown 2-Change local workspace context (down)
dbquit 2-Quit debug mode
dbstack 2-Display function call stack
dbstatus List all breakpoints
dbstop 2-Set breakpoints in an M-file function
dbtype 2-List M-file with line numbers
dbup 2-Change local workspace context (up)
Purpose
Set breakpoints in an M-file function

Syntax

dbstop at lineno in function

dbstop in function


dbstop if keyword

where keyword is one of:

<table>
<thead>
<tr>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>nani nf</td>
</tr>
<tr>
<td>inf nan</td>
</tr>
<tr>
<td>warning</td>
</tr>
</tbody>
</table>

Description
The dbstop command sets up MATLAB's debugging mode. dbstop sets a breakpoint at a specified location in an M-file function or causes a break in case an error or warning occurs during execution. When the specified dbstop condition is met, the MATLAB prompt is displayed and you can issue any valid MATLAB command.

dbstop at lineno in function stops execution just prior to execution of that line of the specified M-file function. function must be the name of an M-file function or a MATLABPATH relative partial pathname.

dbstop in function stops execution before the first executable line in the M-file function when it is called.

dbstop if keyword stops execution under the specified conditions:

<table>
<thead>
<tr>
<th>dbstop if error</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbstop if nani nf</td>
<td>Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).</td>
</tr>
<tr>
<td>dbstop if inf nan</td>
<td>Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).</td>
</tr>
<tr>
<td>dbstop if warning</td>
<td>Stops execution if a runtime warning occurs in any M-file function.</td>
</tr>
</tbody>
</table>

Regardless of the form of the dbstop command, when a stop occurs, the line or error condition that caused the stop is displayed. To resume M-file function
execution, issue a \texttt{dbcont} command or step to another line in the file with the \texttt{dbstep} command.

Any breakpoints set by the first two forms of the \texttt{dbstop} command are cleared if the M-file function is edited or cleared.

The \texttt{at, in, and if} keywords, familiar to users of the UNIX debugger dbx, are optional.

\textbf{Examples}

Here is a short example, printed with the \texttt{dbtype} command to produce line numbers.

\begin{verbatim}
   dbtype buggy
   function z = buggy(x)
   n = length(x);
   z = (1:n)./x;
\end{verbatim}

The statement

\begin{verbatim}
   dbstop in buggy
\end{verbatim}

causes execution to stop at line 2, the first executable line. The command

\begin{verbatim}
   dbstep
\end{verbatim}

then advances to line 3 and allows the value of \texttt{n} to be examined.

The example function only works on vectors; it produces an error if the input \texttt{x} is a full matrix. So the statements

\begin{verbatim}
   dbstop if error
   buggy(magic(3))
\end{verbatim}

produce

\begin{verbatim}
   Error using ==> /
   Matrix dimensions must agree.
   Error in ==> buggy.m
   On line 3 ==> z = (1:n)./x;
\end{verbatim}

Finally, if any of the elements of the input \texttt{x} are zero, a division by zero occurs. For example, consider

\begin{verbatim}
   dbstop if nani nf
   buggy(0:2)
\end{verbatim}
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` 2-Clear breakpoints
- `dbcont` 2-Resume execution
- `dbdown` 2-Change local workspace context (down)
- `dbquit` 2-Quit debug mode
- `dbstack` 2-Display function call stack
- `dbstatus` List all breakpoints
- `dbstep` 2-Execute one or more lines from a breakpoint
- `dbtype` 2-List M-file with line numbers
- `dbup` 2-Change local workspace context (up)

See also 2-partialpath.
dbtype

Purpose
List M-file with line numbers

Syntax

dbtype function

dbtype function start:end

Description
dbtype function displays the contents of the specified M-file function with line numbers preceding each line. function must be the name of an M-file function or a MATLABPATH relative partial pathname.

dbtype function start:end displays the portion of the file specified by a range of line numbers.

See Also
dbclear 2-Clear breakpoints
dbcont 2-Resume execution
dbdwn 2-Change local workspace context (down)

See also 2-partialpath.
Purpose
Change local workspace context

Syntax
```
dbup
```

Description
This command allows you to examine the calling M-file by using any other MATLAB command. In this way, you determine what led to the arguments being passed to the called function.

```
dbup  changes the current workspace context (at a breakpoint) to the workspace of the calling M-file.

Multiple dbup commands change the workspace context to each previous calling M-file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)
```

See Also
```
dbclear  2-Clear breakpoints
dbcont  2-Resume execution
dbdown  2-Change local workspace context (down)
dbquit  2-Quit debug mode
dbstack  2-Display function call stack
dbstatus  List all breakpoints
dbstep  2-Execute one or more lines from a breakpoint
dbstop  2-Set breakpoints in an M-file function
dbtype  2-List M-file with line numbers
```
**Purpose**
Set up advisory link

**Syntax**

\[
rc = \text{ddeadv}(\text{channel}, \{'i\text{tem}', 'c\text{allback}'\})
\]

```matlab
rc = \text{ddeadv}(\text{channel}, \{'i\text{tem}', 'c\text{allback}', 'u\text{pmtx}'\})
```

```matlab
rc = \text{ddeadv}(\text{channel}, \{'i\text{tem}', 'c\text{allback}', 'u\text{pmtx}', 'f\text{ormat}'\})
```

```matlab
rc = \text{ddeadv}(\text{channel}, \{'i\text{tem}', 'c\text{allback}', 'u\text{pmtx}', 'f\text{ormat}', 't\text{imeout}'\})
```

**Description**

ddeadv sets up an advisory link between MATLAB and a server application. When the data identified by the `item` argument changes, the string specified by the `callback` argument is passed to the `eval` function and evaluated. If the advisory link is a hot link, DDE modifies `upmtx`, the update matrix, to reflect the data in `item`.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

**Arguments**

- `rc`  
  Return code: 0 indicates failure, 1 indicates success.

- `channel`  
  Conversation channel from `ddeinit`.

- `item`  
  String specifying the DDE `item` name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.

- `callback`  
  String specifying the callback that is evaluated on update notification. Changing the data identified by `item` at the server causes `callback` to get passed to the `eval` function to be evaluated.

- `upmtx` (optional)  
  String specifying the name of a matrix that holds data sent with an update notification. If `upmtx` is included, changing `item` at the server causes `upmtx` to be updated with the revised data. Specifying `upmtx` creates a hot link. Omitting `upmtx` or specifying it as an empty string creates a warm link. If `upmtx` exists in the workspace, its contents are overwritten. If `upmtx` does not exist, it is created.
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:

\[
rc = \text{ddeadv}(\text{channel}, 'r1c1:r5c5', 'disp(x)', 'x');
\]

Communication with Excel must have been established previously with a \texttt{ddeinit} command.

See Also

- \texttt{ddeexec}: Send string for execution
- \texttt{ddeinit}: Initiate DDE conversation
- \texttt{ddepoke}: Send data to application
- \texttt{ddereq}: Request data from application
- \texttt{ddeterr}: Terminate DDE conversation
- \texttt{ddeunadv}: Release advisory link

\texttt{format} (optional)

Two-element array specifying the format of the data to be sent on update. The first element specifies the Windows clipboard format to use for the data. The only currently supported format is \texttt{cf_text}, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are \texttt{numeric} (the default, which corresponds to a value of 0) and \texttt{string} (which corresponds to a value of 1). The default format array is \([1, 0]\).

\texttt{timeout} (optional)

Scalar specifying the time-out limit for this operation. \texttt{timeout} is specified in milliseconds. (1000 milliseconds = 1 second). If an advisory link is not established within \texttt{timeout} milliseconds, the function fails. The default value of \texttt{timeout} is three seconds.
**Purpose**  
Send string for execution

**Syntax**

```
rc = ddeexec(channel,'command')
rc = ddeexec(channel,'command','item')
rc = ddeexec(channel,'command','item',timeout)
```

**Description**

ddeexec sends a string for execution to another application via an established DDE conversation. Specify the string as the `command` argument.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

**Arguments**

- `rc`  
  Return code: 0 indicates failure, 1 indicates success.

- `channel`  
  Conversation channel from `ddeinit`.

- `command`  
  String specifying the command to be executed.

- `item` (optional)  
  String specifying the DDE item name for execution. This argument is not used for many applications. If your application requires this argument, it provides additional information for `command`. Consult your server documentation for more information.

- `timeout` (optional)  
  Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

**Examples**

Given the channel assigned to a conversation, send a command to Excel:

```
rc = ddeexec(channel,'[formula.goto("r1c1")]
```

Communication with Excel must have been established previously with a `ddeinit` command.

**See Also**

- `ddeadv`  
  Set up advisory link

- `ddeinit`  
  Initiate DDE conversation

- `ddepoke`  
  Send data to application

- `dder eq`  
  Request data from application

- `ddet erm`  
  Terminate DDE conversation

- `ddeunadv`  
  Release advisory link

2-166
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Initiate DDE conversation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>channel = ddeinit('service', 'topic')</code></td>
</tr>
<tr>
<td>Description</td>
<td><code>channel = ddeinit('service', 'topic')</code> returns a channel handle assigned to the conversation, which is used with other MATLAB DDE functions. 'service' is a string specifying the service or application name for the conversation. 'topic' is a string specifying the topic for the conversation.</td>
</tr>
<tr>
<td>Examples</td>
<td>To initiate a conversation with Excel for the spreadsheet 'stocks.xls':</td>
</tr>
<tr>
<td></td>
<td><code>channel = ddeinit('excel', 'stocks.xls')</code></td>
</tr>
<tr>
<td></td>
<td><code>channel = 0.00</code></td>
</tr>
<tr>
<td>See Also</td>
<td>ddeadv Set up advisory link</td>
</tr>
<tr>
<td></td>
<td>ddeexec Send string for execution</td>
</tr>
<tr>
<td></td>
<td>ddepoke Send data to application</td>
</tr>
<tr>
<td></td>
<td>ddereq Request data from application</td>
</tr>
<tr>
<td></td>
<td>ddeterm Terminate DDE conversation</td>
</tr>
<tr>
<td></td>
<td>ddeunadv Release advisory link</td>
</tr>
</tbody>
</table>
**Purpose**
Send data to application

**Syntax**
\[ rc = ddepoke( channel, 'item', data ) \]
\[ rc = ddepoke( channel, 'item', data, format ) \]
\[ rc = ddepoke( channel, 'item', data, format, timeout ) \]

**Description**
ddepoke sends data to an application via an established DDE conversation. ddepoke formats the data matrix as follows before sending it to the server application:

- String matrices are converted, element by element, to characters and the resulting character buffer is sent.

- Numeric matrices are sent as tab-delimited columns and carriage-return, line-feed delimited rows of numbers. Only the real part of nonsparse matrices are sent.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

**Arguments**

- **rc**
  Return code: 0 indicates failure, 1 indicates success.

- **channel**
  Conversation channel from ddeinit.

- **item**
  String specifying the DDE item for the data sent. Item is the server data entity that is to contain the data sent in the data argument.

- **data**
  Matrix containing the data to send.

- **format** (optional)
  Scalar specifying the format of the data requested. The value indicates the Windows clipboard format to use for the data transfer. The only format currently supported is cf_text, which corresponds to a value of 1.

- **timeout** (optional)
  Scalar specifying the time-out limit for this operation. timeout is specified in milliseconds. (1000 milliseconds = 1 second). The default value of timeout is three seconds.
Examples

Assume that a conversation channel with Excel has previously been established with `ddeinit`. To send a 5-by-5 identity matrix to Excel, placing the data in Row 1, Column 1 through Row 5, Column 5:

```matlab
c = ddepoke(channel, 'r1c1:r5c5', eye(5));
```

See Also

- `ddeadv`: Set up advisory link
- `ddeexec`: Send string for execution
- `ddeinit`: Initiate DDE conversation
- `ddereq`: Request data from application
- `ddeterm`: Terminate DDE conversation
- `ddeunadv`: Release advisory link
ddereq

Purpose
Request data from application

Syntax
\[
data = \	ext{ddereq}(\text{channel},'\text{item}') \\
data = \	ext{ddereq}(\text{channel},'\text{item}',\text{format}) \\
data = \	ext{ddereq}(\text{channel},'\text{item}',\text{format},\text{timeout})
\]

Description
\text{ddereq} requests data from a server application via an established DDE conversation. \text{ddereq} returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

Arguments
- \text{data} Matrix containing requested data, empty if function fails.
- \text{channel} Conversation channel from \text{ddeinit}.
- \text{item} String specifying the server application's DDE item name for the data requested.
- \text{format} (optional) Two-element array specifying the format of the data requested. The first element specifies the Windows clipboard format to use. The only currently supported format is \text{cf_text}, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are \text{numeric} (the default, which corresponds to 0) and \text{string} (which corresponds to a value of 1). The default format array is \[1 \ 0\].
- \text{timeout} (optional) Scalar specifying the time-out limit for this operation. \text{timeout} is specified in milliseconds. (1000 milliseconds = 1 second). The default value of \text{timeout} is three seconds.

Examples
Assume that we have an Excel spreadsheet \text{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:

\[
\text{channel} = \text{ddeinit}('\text{excel}', '\text{stocks.xls}')
\]

DDE functions require the \text{rcy} reference style for Excel worksheets. In Excel terminology the prices are in \text{r3c1: r3c3} and the shares in \text{r6c2: r8c2}.
To request the prices from Excel:

```matlab
prices = ddereq(channel, 'r3c1:r3c3')
```

```plaintext
prices =
  42.50  15.00  78.88
```

To request the number of shares of each stock:

```matlab
shares = ddereq(channel, 'r6c2:r8c2')
```

```plaintext
shares =
  100.00
  500.00
  300.00
```

**See Also**
- `ddeadv` Set up advisory link
- `ddeexec` Send string for execution
- `ddeinit` Initiate DDE conversation
- `ddepoke` Send data to application
- `ddeterm` Terminate DDE conversation
- `ddeunadv` Release advisory link
**Purpose**
Terminate DDE conversation

**Syntax**
```plaintext
rc = ddeterm(channel)
```

**Description**
`rc = ddeterm(channel)` accepts a channel handle returned by a previous call to `ddeinit` that established the DDE conversation. `ddeterm` terminates this conversation. `rc` is a return code where 0 indicates failure and 1 indicates success.

**Examples**
To close a conversation channel previously opened with `ddeinit`:
```plaintext
rc = ddeterm(channel)

rc =
1.00
```

**See Also**
- ddeadv: Set up advisory link
- ddeexec: Send string for execution
- ddeinit: Initiate DDE conversation
- ddepoke: Send data to application
- ddereq: Request data from application
- ddeunadv: Release advisory link
**Purpose**
Release advisory link

**Syntax**

```matlab
rc = ddeunadv(channel, 'item')
rc = ddeunadv(channel, 'item', format)
rc = ddeunadv(channel, 'item', format, timeout)
```

**Description**
`ddeunadv` releases the advisory link between MATLAB and the server application established by an earlier `ddeadv` call. The `channel`, `item` and `format` must be the same as those specified in the call to `ddeadv` that initiated the link. If you include the `timeout` argument but accept the default `format`, you must specify `format` as an empty matrix.

**Arguments**
- `rc` : Return code: 0 indicates failure, 1 indicates success.
- `channel` : Conversation channel from `ddeinit`.
- `item` : String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
- `format` (optional) : Two-element array. This must be the same as the `format` argument for the corresponding `ddeadv` call.
- `timeout` (optional) : Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

**Example**
To release an advisory link established previously with `ddeadv`:

```matlab
rc = ddeadv(channel, 'r1c1:r5c5')
rc =
1.00
```

**See Also**
- `ddeadv` : Set up advisory link
- `ddeexec` : Send string for execution
- `ddeinit` : Initiate DDE conversation
- `ddepoke` : Send data to application
- `dder eq` : Request data from application
- `ddesterm` : Release advisory link
**Purpose**
Deal inputs to outputs

**Syntax**

\[
\begin{align*}
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(X) \\
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(X1, X2, X3, \ldots)
\end{align*}
\]

**Description**

\[
\begin{align*}
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(X) \quad \text{copies the single input to all the requested outputs. It is the same as } Y_1 = X, Y_2 = X, Y_3 = X, \ldots
\end{align*}
\]

\[
\begin{align*}
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(X1, X2, X3, \ldots) \quad \text{is the same as } Y_1 = X1; Y_2 = X2; Y_3 = X3; \ldots
\end{align*}
\]

**Remarks**
deal is most useful when used with cell arrays and structures via comma separated list expansion. Here are some useful constructions:

\[
\begin{align*}
[S.\text{field}] &= \text{deal}(X) \quad \text{sets all the fields with the name field in the structure array S to the value X. If S doesn't exist, use } [S(1:m).\text{field}] = \text{deal}(X).
\end{align*}
\]

\[
\begin{align*}
[X{:}] &= \text{deal}(A.\text{field}) \quad \text{copies the values of the field with name field to the cell array X. If X doesn't exist, use } [X{:}] = \text{deal}(A.\text{field}).
\end{align*}
\]

\[
\begin{align*}
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(X{:}) \quad \text{copies the contents of the cell array X to the separate variables Y1, Y2, Y3, ...}
\end{align*}
\]

\[
\begin{align*}
[Y_1, Y_2, Y_3, \ldots] &= \text{deal}(S.\text{field}) \quad \text{copies the contents of the fields with the name field to separate variables Y1, Y2, Y3, ...}
\end{align*}
\]

2-174
Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```matlab
C = {rand(3) ones(3, 1) eye(3) zeros(3, 1)};
[a, b, c, d] = deal(C{:})
```

```matlab
a =
    0.9501   0.4860   0.4565
    0.2311   0.8913   0.0185
    0.6068   0.7621   0.8214

b =
    1
    1
    1

c =
    1   0   0
    0   1   0
    0   0   1

d =
    0
    0
    0
```
Use `deal` to obtain the contents of all the name fields in a structure array:

```matlab
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} = 'SIMULINK    ';  
A{2,1} = 'Toolboxes    ';  
A{2,2} = 'The MathWorks    ';  
A =
  'MATLAB'    'SIMULINK'    
  'Toolboxes'    'The MathWorks'    

debank(A)
ans =
  'MATLAB'    'SIMULINK'    
  'Toolboxes'    'The MathWorks'
dec2base

**Purpose**
Decimal number to base conversion

**Syntax**
str = dec2base(d, base)
str = dec2base(d, base, n)

**Description**
str = dec2base(d, base) converts the nonnegative integer d to the specified base. d must be a nonnegative integer smaller than $2^{52}$, and base must be an integer between 2 and 36. The returned argument str is a string.

str = dec2base(d, base, n) produces a representation with at least n digits.

**Examples**
The expression dec2base(23, 2) converts $23_{10}$ to base 2, returning the string '10111'.

**See Also**
base2dec
### Purpose
Decimal to binary number conversion

### Syntax
- `str = dec2bin(d)`
- `str = dec2bin(d, n)`

### Description
- `str = dec2bin(d)` returns the binary representation of `d` as a string. `d` must be a nonnegative integer smaller than $2^{32}$.
- `str = dec2bin(d, n)` produces a binary representation with at least `n` bits.

### Examples
- `dec2bin(23)` returns `'10111'`.

### See Also
- `bi2dec` 2-Binary to decimal number conversion
- `dec2hex` 2-Decimal to hexadecimal number conversion
dec2hex

Purpose
Decimal to hexadecimal number conversion

Syntax
str = dec2hex(d)
str = dec2hex(d, n)

Description
str = dec2hex(d) converts the decimal integer d to its hexadecimal representation stored in a MATLAB string. d must be a nonnegative integer smaller than $2^{52}$.

str = dec2hex(d, n) produces a hexadecimal representation with at least n digits.

Examples
dec2hex(1023) is the string '3ff'.

See Also
dec2bin 2-Decimal to binary number conversion
format 2-Control the output display format
hex2dec 2-IEEE hexadecimal to decimal number conversion
hex2num 2-Hexadecimal to double number conversion
**Purpose**

Deconvolution and polynomial division

**Syntax**

\[ [q, r] = \text{deconv}(v, u) \]

**Description**

\[ [q, r] = \text{deconv}(v, u) \]

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 \( r \).

**Examples**

If

\[
\begin{align*}
u &= [1 \ 2 \ 3 \ 4] \\
v &= [10 \ 20 \ 30]
\end{align*}
\]

the convolution is

\[
\begin{align*}
c &= \text{conv}(u, v) \\
c &= 10 \ 40 \ 100 \ 160 \ 170 \ 120
\end{align*}
\]

Use deconvolution to recover \( u \):

\[
\begin{align*}
[q, r] &= \text{deconv}(c, u) \\
q &= 10 \ 20 \ 30 \\
r &= 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0
\end{align*}
\]

This gives a quotient equal to \( v \) and a zero remainder.

**Algorithm**

\text{deconv} uses the \text{filter} primitive.

**See Also**

\text{conv} \text{tx}, \text{conv}2, and \text{filter} in the Signal Processing Toolbox, and:

\text{conv} 2-Convolution and polynomial multiplication

\text{residue} 2-Convert between partial fraction expansion and polynomial coefficients
**Purpose**
Discrete Laplacian

**Syntax**
- \( L = \text{del2}(U) \)
- \( L = \text{del2}(U, h) \)
- \( L = \text{del2}(U, hx, hy) \)
- \( L = \text{del2}(U, hx, hy, hz, \ldots) \)

**Definition**
If the matrix \( U \) is regarded as a function \( u(x,y) \) evaluated at the point on a square grid, then \( 4 \times \text{del2}(U) \) is a finite difference approximation of Laplace's differential operator applied to \( u \), that is:

\[
\begin{align*}
L &= \nabla^2 u = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right) \\
\end{align*}
\]

where:

\[
l_{ij} = \frac{1}{4} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - u_{i,j})
\]

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables \( u(x,y,z,\ldots) \), \( \text{del2}(U) \) is an approximation,

\[
L = \frac{\nabla^2 u}{2N} = \frac{1}{2N} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \ldots \right)
\]

where \( N \) is the number of variables in \( u \).

**Description**
\( L = \text{del2}(U) \) where \( U \) is a rectangular array is a discrete approximation of

\[
L = \frac{\nabla^2 u}{4} = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)
\]

The matrix \( L \) is the same size as \( U \) with each element equal to the difference between an element of \( U \) and the average of its four neighbors.
$L = \text{del2}(U)$ when $U$ is an multidimensional array, returns an approximation of

$$\nabla^2 u$$

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.

$L = \text{del2}(U, hx, hy, , \ldots)$ where $U$ is multidimensional uses the spacing given by $hx, hy, hz, \ldots$

**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

\[ V = 4 \times \text{del}^2(U) \]

\[
\begin{array}{cccccccccccc}
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\end{array}
\]

See Also

diff 2-Differences and approximate derivatives
gradient 2-Numerical gradient
Purpose

Delaunay triangulation

Syntax

\[
\text{TRI} = \text{delaunay}(x, y)
\]

\[
\text{TRI} = \text{delaunay}(x, y, 'sorted')
\]

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.

\[
\begin{array}{c}
\text{Delaunay triangle} \\
\text{Voronoi polygon}
\end{array}
\]

Description

\[
\text{TRI} = \text{delaunay}(x, y)
\]

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.

\[
\text{TRI} = \text{delaunay}(x, y, 'sorted')
\]

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: \text{griddata} (to interpolate scattered data), \text{convhull}, \text{voronoi} (to compute the Voronoi diagram), and is useful by itself to create a triangular grid for scattered data points.

The functions \text{dsearch} and \text{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.

```matlab
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:

```matlab
[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` 2-Convex hull

2-186
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delaunay</td>
<td>2-Delaunay triangulation</td>
</tr>
<tr>
<td>dsearch</td>
<td>2-Search for nearest point</td>
</tr>
<tr>
<td>griddata</td>
<td>2-Data gridding</td>
</tr>
<tr>
<td>tsearch</td>
<td>2-Search for enclosing Delaunay triangle</td>
</tr>
<tr>
<td>voronoi</td>
<td>2-Voronoi diagram</td>
</tr>
</tbody>
</table>
delete

Purpose
Delete files and graphics objects

Syntax
delete filename
delete(h)

Description
delete filename deletes the named file. Wildcards may be used.

delete(h) deletes the graphics object with handle h. The function deletes the object without requesting verification even if the object is a window.

Use the functional form of delete, such as delete('filename'), when the filename is stored in a string.

See Also
! Operating system command
dir 2-Directory listing
type 2-List file
**Purpose**
Matrix determinant

**Syntax**
d = det(X)

**Description**
d = det(X) returns the determinant of the square matrix X. If X contains only integer entries, the result d is also an integer.

**Remarks**
Using det(X) == 0 as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using abs(det(X)) <= tolerance is not recommended as it is difficult to choose the correct tolerance. The function cond(X) can check for singular and nearly singular matrices.

**Algorithm**
The determinant is computed from the triangular factors obtained by Gaussian elimination

```
[ L, U] = lu(A)
s = det(L)        % This is always +1 or -1
det(A) = s*prod(diag(U))
```

**Examples**
The statement A = [ 1 2 3; 4 5 6; 7 8 9] produces

```
A =
   1   2   3
   4   5   6
   7   8   9
```

This happens to be a singular matrix, so d = det(A) produces d = 0. Changing A(3,3) with A(3,3) = 0 turns A into a nonsingular matrix. Now d = det(A) produces d = 27.

**See Also**
- \, Matrix left division (backslash)
- /, Matrix right division (slash)
- cond, 2-Condition number with respect to inversion
- condest, 2-1-norm matrix condition number estimate
- inv, 2-Matrix inverse
- lu, 2-LU matrix factorization
- rref, 2-Reduced row echelon form
Purpose
Diagonal matrices and diagonals of a matrix

Syntax
\[ X = \text{diag}(v, k) \]
\[ X = \text{diag}(v) \]
\[ v = \text{diag}(X, k) \]
\[ v = \text{diag}(X) \]

Description
\( X = \text{diag}(v, k) \) 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 \( k = 0 \).

Examples
\( \text{diag}(\text{diag}(X)) \) is a diagonal matrix.

\( \text{sum}(\text{diag}(X)) \) is the trace of \( X \).

The statement
\[ \text{diag}(-m:n) + \text{diag}([2:m; 1], 1) + \text{diag}([2*m; 1], -1) \]
produces a tridiagonal matrix of order \( 2*m+1 \).

See Also
spdiags, tril, triu

2-190
Purpose
Save session in a disk file

Syntax
diary

diary filename

diary off

diary on

Description
The diary command creates a log of keyboard input and system responses. The output of diary is an ASCII file, suitable for printing or for inclusion in reports and other documents.

diary, by itself, toggles diary mode on and off.

diary filename writes a copy of all subsequent keyboard input and most of the resulting output (but not graphs) to the named file. If the file already exists, output is appended to the end of the file.

diary off suspends the diary.

diary on resumes diary mode using the current filename, or the default filename diary if none has yet been specified.

Remarks
The function form of the syntax, diary('filename'), is also permitted.

Limitations
You cannot put a diary into the files named off and on.
**Purpose**
Differences and approximate derivatives

**Syntax**

\[ Y = \text{diff}(X) \]
\[ Y = \text{diff}(X, n) \]
\[ Y = \text{diff}(X, n, \text{dim}) \]

**Description**

\[ Y = \text{diff}(X) \] 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:

\[ \left[ X(2) - X(1) \ X(3) - X(2) \ldots \ X(n) - X(n-1) \right] \]

If \( X \) is a matrix, then \( \text{diff}(X) \) returns a matrix of column differences:

\[ \left[ X(2: m:) - X(1: m1, :) \right] \]

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 \( \text{diff} \left( \text{diff}(X) \right) \).

\[ Y = \text{diff}(X, n, \text{dim}) \] 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.

**Remarks**

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.

```matlab
x = [1 2 3 4 5];
y = diff(x)
y =
    1   1   1   1

z = diff(x, 2)
z =
    0   0   0
```

Given,

```matlab
A = rand(1,3,2,4);
```

\( \text{diff}(A) \) is the first-order difference along dimension 2.

\( \text{diff}(A, 3, 4) \) is the third-order difference along dimension 4.

See Also

- `gradient` Approximate gradient.
- `int` Integrate (see Symbolic Toolbox).
- `prod` 2-Product of array elements
- `sum` 2-Sum of array elements
Purpose
Directory listing

Syntax
```
dir
dir dirname
names = dir
names = dir('dirname')
```

Description
`dir`, by itself, lists the files in the current directory.

`dir dirname` lists the files in the specified directory. Use pathnames, wildcards, and any options available in your operating system.

```
names = dir('dirname') or names = dir returns the results in an m-by-1 structure with the fields:
name    Filename
date    Modification date
bytes   Number of bytes allocated to the file
isdir   1 if name is a directory; 0 if not
```

Examples
```
cd /Matlab/Toolbox/Local; dir

Contents.m matlabrc.m siteid.m userpath.m

names = dir

names =

4x1 struct array with fields:
    name
    date
    bytes
    isdir
```

See Also
`!`, `cd`, `delete`, `type`, `what`
Purpose
Display text or array

Syntax
disp(X)

Description
disp(X) displays an array, without printing the array name. If X contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading "X = " which is not always desirable.

Examples
One use of disp in an M-file is to display a matrix with column labels:

disp('     Corn     Oats     Hay')
disp(rand(5,3))

which results in

<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>

See Also
format  2-Control the output display format
int2str  2-Integer to string conversion
num2str  2-Number to string conversion
rats     2-Rational fraction approximation
sprintf  2-Write formatted data to a string
dlmread

Purpose
Read an ASCII delimited file into a matrix

Syntax

\[
M = \text{dlmread}(\text{filename}, \text{delimiter})
\]

\[
M = \text{dlmread}(\text{filename}, \text{delimiter}, r, c)
\]

\[
M = \text{dlmread}(\text{filename}, \text{delimiter}, r, c, \text{range})
\]

Description

\[M = \text{dlmread}(\text{filename}, \text{delimiter})\] reads data from the ASCII delimited format \text{filename}, using the delimiter \text{delimiter}. Use `\t` to specify a tab.

\[M = \text{dlmread}(\text{filename}, \text{delimiter}, r, c)\] reads data from the ASCII delimited format \text{filename}, using the delimiter \text{delimiter}, starting at file offset \text{r} and \text{c}. \text{r} and \text{c} are zero based so that \text{r}=0, \text{c}=0 specifies the first value in the file.

\[M = \text{dlmread}(\text{filename}, \text{delimiter}, r, c, \text{range})\] imports an indexed or named range of ASCII-delimited data. To use the cell range, specify \text{range} by:

\[
\text{range} = [\text{UpperLeftRow} \ \text{UpperLeftColumn} \ \text{LowerRightRow} \ \text{LowerRightColumn}]
\]

Arguments

- \text{delimiter}  
  The character separating individual matrix elements in the ASCII-format spreadsheet file. A comma (,) is the default delimiter.

- \text{r, c}  
  The spreadsheet cell from which the upper-left-most matrix element is taken.

- \text{range}  
  A vector specifying a range of spreadsheet cells.

See Also

- \text{dlmwrite}  
  2-Write a matrix to an ASCII delimited file

- \text{wk1read}  
  2-Read a Lotus123 WK1 spreadsheet file into a matrix

- \text{wk1write}  
  2-Write a matrix to a Lotus123 WK1 spreadsheet file
Purpose

Write a matrix to an ASCII delimited file

Syntax

dlmwrite(filename, A, delimiter)
dlmwrite(filename, A, delimiter, r, c)

Description

The dlmwrite command converts a MATLAB matrix into an ASCII-format file readable by spreadsheet programs.

dlmwrite(filename, A, delimiter) writes matrix A into the upper left-most cell of the ASCII-format spreadsheet file filename, and uses the delimiter to separate matrix elements. Specify \"\t\" to produce tab-delimited files. Any elements whose value is 0 will be omitted. For example, the array \[1 0 2\] will appear in a file as \'1,,2\' when the delimiter is a comma.

dlmwrite(filename, A, delimiter, r, c) writes A into filename, starting at spreadsheet cell r and c, with delimiter used to separate matrix elements.

Arguments

delimiter The character separating individual matrix elements in the ASCII-format spreadsheet file. A comma (,) is the default delimiter.

r, c The spreadsheet cell into which the upper-left-most matrix element is written.

See Also

dlread 2-Read an ASCII delimited file into a matrix
wk1read 2-Read a Lotus123 WK1 spreadsheet file into a matrix
wk1write 2-Write a matrix to a Lotus123 WK1 spreadsheet file
dmperm

Purpose
Dulmage-Mendelsohn decomposition

Syntax

\[ p = \text{dmperm}(A) \]

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

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

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 \( A \).
| **Purpose** | Load hypertext documentation |
| **Syntax** | \texttt{doc}  
\texttt{doc command} |
| **Description** | \texttt{doc}, by itself, loads hypertext-based reference documentation. You'll be presented with an index of MATLAB's main categories of functions.  
\texttt{doc command} loads documentation about a specific command or function. |
| **See Also** | \texttt{help}  
\texttt{type}  
Online help for MATLAB functions and M-files  
2-List file |
**double**

**Purpose**
Convert to double precision

**Syntax**
`double(X)`

**Description**
`double(x)` returns the double precision value for `x`. If `x` is already a double precision array, `double` has no effect.

**Remarks**
`double` is called for the expressions in for, if, and while loops if the expression isn't already double precision. `double` should be overloaded for any object when it makes sense to convert it to a double precision value.
**Purpose**

Search for nearest point

**Syntax**

```plaintext
K = dsearch(x, y, TRI, xi, yi)
K = dsearch(x, y, TRI, xi, yi, S)
```

**Description**

`K = dsearch(x, y, TRI, xi, yi)` returns the index of the nearest (x,y) point to the point (xi,yi). `dsearch` requires a triangulation `TRI` of the points x,y obtained from `delaunay`.

`K = dsearch(x, y, TRI, xi, yi, S)` uses the sparse matrix `S` instead of computing it each time:

```plaintext
S = sparse(TRI(:,[1 1 2 2 3 3]), TRI(:,[2 3 1 3 1 2]), 1, nxy, nxy)
```

where `nxy = prod(size(x))`.

**See Also**

- `delaunay`: Delaunay triangulation
- `tsearch`: 2-Search for enclosing Delaunay triangle
- `voronoi`: 2-Voronoi diagram
**Purpose**

Echo M-files during execution

**Syntax**

```
echo on
echo off
```

**Description**

The `echo` command controls the echoing of M-files during execution. Normally, the commands in M-files do not display on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either on or off, in which case any script used is affected:

- `echo on` Turns on the echoing of commands in all script files.
- `echo off` Turns off the echoing of commands in all script files.
- `echo` Toggles the echo state.

With function files, the use of `echo` is more complicated. If `echo` is enabled on a function file, the file is interpreted, rather than compiled. Each input line is then displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

- `echo fcnname on` Turns on echoing of the named function file.
- `echo fcnname off` Turns off echoing of the named function file.
- `echo fcnname` Toggles the echo state of the named function file.
- `echo on all` Set echoing on for all function files.
- `echo off all` Set echoing off for all function files.

**See Also**

function

2-202
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Edit an M-file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>edit</code></td>
</tr>
<tr>
<td></td>
<td><code>edit fun</code></td>
</tr>
<tr>
<td></td>
<td><code>edit file.ext</code></td>
</tr>
<tr>
<td></td>
<td><code>edit class/fun</code></td>
</tr>
<tr>
<td></td>
<td><code>edit private/fun</code></td>
</tr>
<tr>
<td></td>
<td><code>edit class/private/fun</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>edit</code> opens a new editor window.</td>
</tr>
<tr>
<td></td>
<td><code>edit fun</code> opens the M-file <code>fun.m</code> in a text editor.</td>
</tr>
<tr>
<td></td>
<td><code>edit file.ext</code> opens the specified text file.</td>
</tr>
<tr>
<td></td>
<td><code>edit class/fun</code>, <code>edit private/fun</code>, or <code>edit class/private/fun</code> can be used to edit a method, private function, or private method (for the class named <code>class</code>.)</td>
</tr>
</tbody>
</table>
Purpose

Eigenvalues and eigenvectors

Syntax

\[ d = \text{eig}(A) \]
\[ [V, D] = \text{eig}(A) \]
\[ [V, D] = \text{eig}(A, 'nobalance') \]
\[ d = \text{eig}(A, B) \]
\[ [V, D] = \text{eig}(A, B) \]

Description

\[ d = \text{eig}(A) \] returns a vector of the eigenvalues of matrix \( A \).

\[ [V, D] = \text{eig}(A) \] produces matrices of eigenvalues (\( D \)) and eigenvectors (\( V \)) of matrix \( A \), so that \( A \cdot V = V \cdot D \). Matrix \( D \) is the canonical form of \( A \)—a diagonal matrix with \( A \)'s eigenvalues on the main diagonal. Matrix \( V \) is the modal matrix—its columns are the eigenvectors of \( A \).

The eigenvectors are scaled so that the norm of each is 1.0. Use \[ [W, D] = \text{eig}(A') \]; \( W = W' \) to compute the left eigenvectors, which satisfy \( W \cdot A = D \cdot W \).

\[ [V, D] = \text{eig}(A, 'nobalance') \] finds eigenvalues and eigenvectors without a preliminary balancing step. Ordinarily, balancing improves the conditioning of the input matrix, enabling more accurate computation of the eigenvectors and eigenvalues. However, if a matrix contains small elements that are really due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix, leading to incorrect eigenvectors. Use the nobalance option in this event. See the balance function for more details.

\[ d = \text{eig}(A, B) \] returns a vector containing the generalized eigenvalues, if \( A \) and \( B \) are square matrices.

\[ [V, D] = \text{eig}(A, B) \] produces a diagonal matrix \( D \) of generalized eigenvalues and a full matrix \( V \) whose columns are the corresponding eigenvectors so that \( A \cdot V = B \cdot V \cdot D \). The eigenvectors are scaled so that the norm of each is 1.0.

Remarks

The eigenvalue problem is to determine the nontrivial solutions of the equation:

\[ A \cdot x = \lambda \cdot x \]
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

$$Ax = \lambda B x$$

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

$$B = \begin{bmatrix} 3 & -2 & -9 & 2 \times \text{eps}; -2 & 4 & -1 & -\text{eps}; -\text{eps}/4 & \text{eps}/2 & -1 & 0; -0.5 & -0.5 & 0.1 & 1 \end{bmatrix};$$

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

```matlab
[V, D] = eig(B)
B*V - V*D
[VN, DN] = eig(B, 'nobalance')
B*VN - VN*DN
```
Algorithm

For real matrices, \( \text{eig}(X) \) 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 \( \text{eig} \) is used with two input arguments, the EISPACK routines QZHES, QZT, QZVAL, and QZVEC solve for the generalized eigenvalues via the QZ algorithm. Modifications handle the complex case.

When \( \text{eig} \) is used with one complex argument, the solution is computed using the QZ algorithm as \( \text{eig}(X, \text{eye}(X)) \). Modifications to the QZ routines handle the special case \( B=I \).

For detailed descriptions of these algorithms, see the EISPACK Guide.

Diagnostics

If the limit of 30n iterations is exhausted while seeking an eigenvalue:

Solution will not converge.

See Also

- balance: Improve accuracy of computed eigenvalues
- condeig: Condition number with respect to eigenvalues
- hess: Hessenberg form of a matrix
- qz: QZ factorization for generalized eigenvalues
- schur: Schur decomposition

References


Purpose

Find a few eigenvalues and eigenvectors

Syntax

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

Description

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

eigs(A) or eigs('Afun', n) solves the eigenvalue problem where the first input argument is either a square matrix (which can be full or sparse, symmetric or nonsymmetric, real or complex), or a string containing the name of an M-file which applies a linear operator to the columns of a given matrix. In the latter case, the second input argument must be n, the order of the problem. For example, eigs('fft', ...) is much faster than eigs(F, ...), where F is the explicit FFT matrix.

With one output argument, d is a vector containing k eigenvalues. With two output arguments, V is a matrix with k columns and D is a k-by-k diagonal matrix so that $A*V = V*D$ or $A*V = B*V*D$. 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.
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, \sigma^2)` where \( \sigma^2 \) is close to, but not equal to, \( \sigma \).

The options structure specifies certain parameters in the algorithm.
**Remarks**

\[ d = \text{eigs}(A,k) \text{ is not a substitute for} \]
\[ d = \text{eig(full(A))} \]
\[ d = \text{sort}(d) \]
\[ d = d(\text{end}-k+1:\text{end}) \]

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

```matlab
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.
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. \texttt{eig} computes all 632 eigenvalues. \texttt{eigs} computes the six largest and smallest magnitude eigenvalues of A successfully with:

\[
\begin{align*}
\texttt{d} &= \texttt{eig(full(A))} \\
\texttt{dlm} &= \texttt{eigs(A)} \\
\texttt{dsm} &= \texttt{eigs(A,'sm')}
\end{align*}
\]

However, the repeated eigenvalue at 4 must be handled more carefully. The call \texttt{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, \texttt{eigs} fails. We must use \( \sigma \) near but not equal to 4 to find those 18 eigenvalues.

\[
\begin{align*}
\sigma &= 4 - 1e-6 \\
[ V, D ] &= \texttt{eigs(A,18,\sigma)}
\end{align*}
\]
The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`.

See Also
- `eig` — Eigenvalues and eigenvectors
- `svds` — A few singular values

References


Jacobi elliptic functions

Syntax

\[
[SN, CN, DN] = \text{ellipj}(U, M)
\]

\[
[SN, CN, DN] = \text{ellipj}(U, M, \text{tol})
\]

Definition

The Jacobi elliptic functions are defined in terms of the integral:

\[
u = \int_{0}^{\phi} \frac{d\theta}{(1 - m \sin^2 \theta)^{\frac{1}{2}}}\]

Then

\[
\text{sn}(u) = \sin \phi, \quad \text{cn}(u) = \cos \phi, \quad \text{dn}(u) = \left(1 - \sin^2 \phi\right)^{\frac{1}{2}}, \quad \text{am}(u) = \phi
\]

Some definitions of the elliptic functions use the modulus \(k\) instead of the parameter \(m\). They are related by:

\[
k^2 = m = \sin^2 \alpha
\]

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

Description

\[
[SN, CN, DN] = \text{ellipj}(U, M)
\]

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).

\[
[SN, CN, DN] = \text{ellipj}(U, M, \text{tol})
\]

computes the Jacobi elliptic functions to accuracy \(\text{tol}\). The default is \(\text{eps}\); increase this for a less accurate but more quickly computed answer.

Algorithm

\text{ellipj} computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:

\[
a_0 = 1, b_0 = (1 - m)^{\frac{1}{2}}, c_0 = (m)^{\frac{1}{2}}
\]
ellipj computes successive iterates with:

\[ a_i = \frac{1}{2}(a_{i-1} + b_{i-1}) \]
\[ b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}} \]
\[ c_i = \frac{1}{2}(a_{i-1} - b_{i-1}) \]

Next, it calculates the amplitudes in radians using:

\[ \sin(2\phi_n - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n) \]

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply:

\[ sn(u) = \sin\phi_0 \]
\[ cn(u) = \cos\phi_0 \]
\[ dn(u) = (1 - m \cdot sn(u)^2)^{\frac{1}{2}} \]

**Limitations**

The ellipj function is limited to the input domain \( 0 \leq m \leq 1 \). Map other values of \( M \) to 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**

Purpose

Complete elliptic integrals of the first and second kind

Syntax

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

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:

\[
K(m) = \int_0^1 \frac{dt}{(1 - t^2)(1 - mt^2)} = \int_0^{\pi/2} \frac{d\theta}{(1 - \sin^2 \theta)^{3/2}}
\]

The complete elliptic integral of the second kind,

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

is:

\[
E(m) = \int_0^1 \frac{(1 - t^2)^{-1/2}}{(1 - mt^2)^{1/2}} dt = \int_0^{\pi/2} \frac{d\theta}{(1 - \sin^2 \theta)^{1/2}}
\]

Some definitions of \( K \) and \( E \) use the modulus \( k \) instead of the parameter \( m \). They are related by:

\[ k^2 = m = \sin^2 \alpha \]
ellipke

Description

K = ellipke(M) returns the complete elliptic integral of the first kind for the elements of M.

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

[K, E] = ellipke(M, tol) computes the Jacobian elliptic functions to accuracy tol. The default is eps; increase this for a less accurate but more quickly computed answer.

Algorithm

ellipke computes the complete elliptic integral using the method of the arithmetic-geometric mean described in [1], section 17.6. It starts with the triplet of numbers:

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

ellipke computes successive iterations of \( a_i, b_i, \) and \( c_i \) with:

\[ a_i = \frac{1}{2}(a_{i-1} + b_{i-1}) \]
\[ b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}} \]
\[ c_i = \frac{1}{2}(a_{i-1} - b_{i-1}) \]

stopping at iteration \( n \) when \( c_n = 0 \), 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

Purpose

Conditionally execute statements

Syntax

if expression
    statements
else
    statements
end

Description

The else command is used to delineate an alternate block of statements.

if expression
    statements
else
    statements
end

The second set of statements is executed if the expression has any zero elements. The expression is usually the result of

expression rop expression

where rop is $$\equiv$$, $$<$$, $$\le$$, $$>$$, $$\ge$$, or $$\ne$$.

See Also

break Break out of flow control structures
elseif Conditionally execute statements
end Terminate for, while, and if statements and indicate the last index
for Repeat statements a specific number of times
if Conditionally execute statements
return Return to the invoking function
switch Switch among several cases based on expression
while Repeat statements an indefinite number of times
elseif

**Purpose**
Conditionally execute statements

**Syntax**
```
if expression
    statements
elseif expression
    statements
end
```

**Description**
The `elseif` command conditionally executes statements.

```
if expression
    statements
elseif expression
    statements
end
```

The second block of statements executes if the first `expression` has any zero elements and the second `expression` has all nonzero elements. The expression is usually the result of
```
expression rop expression
```
where `rop` is `==`, `<`, `>`, `<=`, `>=`, or `~=`.

`elseif`, with a space between the `else` and the `if`, differs from `elseif`, with no space. The former introduces a new, nested, `if`, which must have a matching `end`. The latter is used in a linear sequence of conditional statements with only one terminating `end`. 
The two segments

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

produce identical results. Exactly one of the four assignments to x is executed, depending upon the values of the three logical expressions, A, B, and C.

**See Also**

- `break`: Break out of flow control structures
- `else`: Conditionally execute statements
- `end`: Terminate `for`, `while`, and `if` statements and indicate the last index
- `for`: Repeat statements a specific number of times
- `if`: Conditionally execute statements
- `return`: Return to the invoking function
- `switch`: Switch among several cases based on expression
- `while`: Repeat statements an indefinite number of times
end

Purpose

Terminate for, while, switch, and if statements or indicate last index

Syntax

while expression\% (or if or for)

    statements

end

B = A(index: end, index)

Description

end is used to terminate for, while, switch, and if statements. Without an end statement, for, while, switch, and if wait for further input. Each end is paired with the closest previous unpaired for, while, switch, or if and serves to delimit its scope.

The end command also serves as the last index in an indexing expression. In that context, end = (size(x, k)) when used as part of the kth index.

Examples

This example shows end used with for and if. Indentation provides easier readability.

    for i = 1:n
    if a(i) == 0
        a(i) = a(i) + 2;
    end
    end

Here, end is used in an indexing expression:

    A = rand(5, 4)
    B = A(end, 2:end)

In this example, B is a 1-by-3 vector equal to [A(5, 2) A(5, 3) A(5, 4)].

See Also

break        Break out of flow control structures
for          Repeat statements a specific number of times
if           Conditionally execute statements
return       Return to the invoking function
switch       Switch among several cases based on expression
while        Repeat statements an indefinite number of times
**Purpose**  
End of month

**Syntax**  
\[ E = 	ext{eomday}(Y, M) \]

**Description**  
\[ E = 	ext{eomday}(Y, M) \] returns the last day of the year and month given by corresponding elements of arrays \( Y \) and \( M \).

**Examples**  
Because 1996 is a leap year, the statement `eomday(1996, 2)` returns 29. To show all the leap years in this century, try:

```
y = 1900:1999;
E = eomday(y, 2*ones(length(y), 1)');
y(find(E==29))'
```

```
ans =
      Columns 1 through 6
            1904  1908  1912  1916  1920  1924
      Columns 7 through 12
            1928  1932  1936  1940  1944  1948
      Columns 13 through 18
      Columns 19 through 24
```

**See Also**  
datenum  
Serial date number  
datevec  
Date components  
weekday  
Day of the week
Purpose
Floating-point relative accuracy

Syntax
eps

Description
eps returns the distance from 1.0 to the next largest floating-point number.
The value eps is a default tolerance for pinv and rank, as well as several other MATLAB functions. On machines with IEEE floating-point arithmetic, eps = 2^-52, which is roughly 2.22e-16.

See Also
realmax Largest positive floating-point number
realmin Smallest positive floating-point number
**Purpose**
Error functions

**Syntax**

- \[ Y = \text{erf}(X) \] \hspace{1cm} \text{Error function}
- \[ Y = \text{erfc}(X) \] \hspace{1cm} \text{Complementary error function}
- \[ Y = \text{erfcx}(X) \] \hspace{1cm} \text{Scaled complementary error function}
- \[ X = \text{erfinv}(Y) \] \hspace{1cm} \text{Inverse of the error function}

**Definition**
The error function \( \text{erf}(X) \) is defined as the integral of the Gaussian distribution function from 0 to \( x \):

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

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 \( \left( \frac{1}{\sqrt{\pi}} \right) \frac{1}{X} \).

**Description**

- \( Y = \text{erf}(X) \) 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 \(-1 < Y < 1\).

**Examples**
- \( \text{erfinv}(1) \) is \( \text{Inf} \)
- \( \text{erfinv}(-1) \) is \( -\text{Inf} \).
- For \( \text{abs}(Y) > 1 \), \( \text{erfinv}(Y) \) is \( \text{NaN} \).
erf, erfc, erfcx, erfinv

Remarks
The relationship between the error function and the standard normal probability distribution is:

\[ x = -5: 0.1: 5; \]
\[ \text{standard_normal_cdf} = \left(1 + \text{erf}(x \ast \sqrt{2})\right) / 2; \]

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
Purpose
Display error messages

Syntax
error('error_message')

Description
ter('error_message') displays an error message and returns control to the keyboard. The error message contains the input string error_message.

The error command has no effect if error_message is a null string.

Examples
The error command provides an error return from M-files.

function foo(x,y)
    if nargin ~= 2
        error('Wrong number of input arguments')
    end

The returned error message looks like:

» foo(pi)
??? Error using ==> foo
Wrong number of input arguments

See Also
dbstop Set breakpoints in an M-file function
disp Display text or array
lasterr Last error message
warning Display warning message
**Purpose**
Continue execution after errors during testing

**Syntax**
errortrap on
errortrap off

**Description**
errortrap on continues execution after errors when they occur. Execution continues with the next statement in a top level script.

errortrap off (the default) stops execution when an error occurs.
etime

**Purpose**
Elapsed time

**Syntax**
\[ e = etime(t2, t1) \]

**Description**
e = etime(t2, t1) returns the time in seconds between vectors \( t1 \) and \( t2 \). The two vectors must be six elements long, in the format returned by \( \text{clock} \):

\[ T = [\text{Year} \quad \text{Month} \quad \text{Day} \quad \text{Hour} \quad \text{Minute} \quad \text{Second}] \]

**Examples**
Calculate how long a 2048-point real FFT takes.

\[
x = \text{rand}(2048, 1);
t = \text{clock}; \quad \text{fft}(x); \quad \text{etime(clock, t)}
\]
\[
\text{ans} = \quad 0.4167
\]

**Limitations**
As currently implemented, the \text{etime} function fails across month and year boundaries. Since \text{etime} is an M-file, you can modify the code to work across these boundaries if needed.

**See Also**
clock \quad Current time as a date vector
cputime \quad Elapsed CPU time
tic, toc \quad Stopwatch timer
#### eval

**Purpose**
Interpret strings containing MATLAB expressions

**Syntax**
```matlab
a = eval('expression')
[a1, a2, a3...] = eval('expression')
eval(string, catchstring)
```

**Description**
- `a = eval('expression')` returns the value of `expression`, a MATLAB expression, enclosed in single quotation marks. Create `expression` by concatenating substrings and variables inside square brackets.
- `[a1, a2, a3...] = eval('expression')` evaluates and returns the results in separate variables. Use of this syntax is recommended over:
  ```matlab
eval(['a1, a2, a3...] = expression')
```
  which hides information from the MATLAB parser and can produce unexpected behavior.
- `eval(string, catchstring)` provides the ability to catch errors. It executes `string` and returns if the operation was successful. If the operation generates an error, `catchstring` is evaluated before returning. Use `lasterr` to obtain the error string produced by `string`.

**Examples**
```matlab
A = '1+4'; eval(A)
an = 5

P = 'pwd'; eval(P)
an = /home/myname
```

The loop
```matlab
for n = 1:12
    eval(['M, int2str(n), = magic(n)'])
end
```

generates a sequence of 12 matrices named `M1` through `M12`. 

2-228
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.

```matlab
D = ['odedemo ' 'quaddemo' 'zerodemo ' 'fitdemo '];
n = input('Select a demo number: ');
eval(D(n,:))
```

### See Also

- `feval` Function evaluation
- `lasterr` Last error message.
evalin

Purpose
Evaluate expression in workspace.

Syntax

```matlab
evalin(ws,'expression')
[X,Y,Z,...] = evalin(ws,'expression')
evalin(ws,'try','catch')
```

Description
`evalin(ws,'expression')` evaluates `expression` in the context of the workspace `ws`. `ws` can be either 'caller' or 'base'.

`[X,Y,Z,...] = evalin(ws,'expression')` returns output arguments from the expression.

`evalin(ws,'try','catch')` tries to evaluate the `try` expression and if that fails it evaluates the `catch` expression in the specified workspace.

`evalin` is useful for getting values from another workspace while `assignin` is useful for depositing values into another workspace.

Limitation
`evalin` may not be used recursively to evaluate an expression, i.e., a sequence of the form `evalin('caller','evalin('caller','expression')')` doesn't work.

See Also
assignin Assign variable in workspace.
eval Interpret strings containing MATLAB expressions
**Purpose**

Check if a variable or file exists

**Syntax**

```matlab
a = exist('item')
ident = exist('item', kind)
```

**Description**

- `a = exist('item')` returns the status of the variable or file `item`:
  - 0: If `item` does not exist.
  - 1: If the variable `item` exists in the workspace.
  - 2: If `item` is an M-file or a file of unknown type.
  - 3: If `item` is a MEX-file.
  - 4: If `item` is a MDL-file.
  - 5: If `item` is a built-in MATLAB function.
  - 6: If `item` is a P-file.
  - 7: If `item` is a directory.

- `exist('item')` or `exist('item ext')` returns 2 if `item` is on the MATLAB search path but the filename extension (`ext`) is not `mdl`, `p`, or `mex`. `item` may be a `MATLABPATH` relative partial pathname.

- `ident = exist('item', 'kind')` returns logical true (1) if an item of the specified `kind` is found, and returns 0 otherwise. `kind` may be:
  - `'var'`: Checks only for variables.
  - `'builtin'`: Checks only for built-in functions.
  - `'file'`: Checks only for files.
  - `'dir'`: Checks only for directories.
exist

Examples

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

```matlab
ident = exist('plot')
ident =
   5
plot is a built-in function.
```

See Also

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

See also partialpath.
**Purpose**

Exponential

**Syntax**

\[ Y = \exp(X) \]

**Description**

The \( \exp \) function is an elementary function that operates element-wise on arrays. Its domain includes complex numbers.

\[ Y = \exp(X) \] returns the exponential for each element of \( X \). For complex \( z = x + iy \), it returns the complex exponential: \( e^z = e^x(\cos(y) + i\sin(y)) \).

**Remark**

Use \( \expm \) for matrix exponentials.

**See Also**

- \( \expm \) Matrix exponential
- \( \log \) Natural logarithm
- \( \log10 \) Common (base 10) logarithm
- \( \expint \) Exponential integral
Purpose
Exponential integral

Syntax
Y = expint(X)

Definitions
The exponential integral is defined as:
\[ \int_{x}^{\infty} \frac{e^t}{t} \, dt \]

Another common definition of the exponential integral function is the Cauchy principal value integral:
\[ E_i(x) = \int_{-\infty}^{x} e^{-t} \, dt \]

which, for real positive x, is related to expint as follows:
\[
\text{expint}(-x+i*0) = -Ei(x) - i \pi \\
\text{Ei}(x) = \text{real}(-\text{expint}(-x))
\]

Description
Y = expint(X) evaluates the exponential integral for each element of X.

Algorithm
For elements of x in the domain [-38, 2], expint uses a series expansion representation (equation 5.1.11 in [1]):
\[
E_i(x) = -\gamma - \ln x - \sum_{n=1}^{\infty} \frac{(-1)^n x^n}{n \cdot n!}
\]

For all other elements of X, expint uses a continued fraction representation (equation 5.1.22 in [1]):

2-234
\[ E_n(z) = e^{-z} \left( \frac{1}{z+1} \frac{n}{z+1} \frac{n+1}{z+1} \frac{2}{z+1} \ldots \right), |\text{angle}(z)| < \pi \]

References

expm

Purpose
Matrix exponential

Syntax
Y = expm(X)

Description
Y = expm(X) raises the constant $e$ to the matrix power $X$. Complex results are produced if $X$ has nonpositive eigenvalues.

Use exp for the element-by-element exponential.

Algorithm
The expm function is built-in, but it uses the Padé approximation with scaling and squaring algorithm expressed in the file expm1.m.

A second method of calculating the matrix exponential uses a Taylor series approximation. This method is demonstrated in the file expm2.m. The Taylor series approximation is not recommended as a general-purpose method. It is often slow and inaccurate.

A third way of calculating the matrix exponential, found in the file expm3.m, is to diagonalize the matrix, apply the function to the individual eigenvalues, and then transform back. This method fails if the input matrix does not have a full set of linearly independent eigenvectors.

References [1] and [2] describe and compare many algorithms for computing $\exp(X)$. The built-in method, expm, is essentially method 3 of [2].

Examples
Suppose $A$ is the 3-by-3 matrix

\[
\begin{bmatrix}
1 & 1 & 0 \\
0 & 0 & 2 \\
0 & 0 & -1
\end{bmatrix}
\]

then $\exp(A)$ is

\[
\begin{bmatrix}
2.7183 & 1.7183 & 1.0862 \\
0 & 1.0000 & 1.2642 \\
0 & 0 & 0.3679
\end{bmatrix}
\]

while $\exp(A)$ is

\[
\begin{bmatrix}
2.7183 & 2.7183 & 1.0000 \\
1.0000 & 1.0000 & 7.3891 \\
1.0000 & 1.0000 & 0.3679
\end{bmatrix}
\]
Notice that the diagonal elements of the two results are equal; this would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

**See Also**

- **exp**  
  Exponential
- **funm**  
  Evaluate functions of a matrix
- **logm**  
  Matrix logarithm
- **sqrtm**  
  Matrix square root

**References**


Purpose
Identity matrix

Syntax
\[
Y = \text{eye}(n) \\
Y = \text{eye}(m,n) \\
Y = \text{eye(size(A))}
\]

Description
\[Y = \text{eye}(n)\] 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.

\[Y = \text{eye(size(A))}\] returns an identity matrix the same size as \(A\).

Limitations
The identity matrix is not defined for higher-dimensional arrays. The assignment \[y = \text{eye([2,3,4])}\] results in an error.

See Also
- \text{ones} Create an array of all ones
- \text{rand} Uniformly distributed random numbers and arrays
- \text{randn} Normally distributed random numbers and arrays
- \text{zeros} Create an array of all zeros
### Purpose
Prime factors

### Syntax
- `f = factor(n)`
- `f = factor(symb)`

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

### Examples
```matlab
f = factor(123)
f =
    3   41
```

### See Also
- `isprime` True for prime numbers
- `primes` Generate list of prime numbers
Purpose
Close one or more open files

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

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

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

See Also
ferror Query MATLAB about errors in file input or output
fopen Open a file or obtain information about open files
fprintf Write formatted data to file
fwrite Read binary data from file
fscanf Read formatted data from file
fseek Set file position indicator
ftell Get file position indicator
fwrite Write binary data from a MATLAB matrix to a file
Purpose

Test for end-of-file

Syntax

eofstat = feof(fid)

Description

eofstat = feof(fid) tests whether the end-of-file indicator is set for the file with identifier fid. It returns 1 if the end-of-file indicator is set, or 0 if it is not. (See fopen for a complete description of fid.)

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

See Also

fopen

Open a file or obtain information about open files
Purpose

Query MATLAB about errors in file input or output

Syntax

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

Description

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

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

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

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

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

See Also

fclose    Close one or more open files
fopen    Open a file or obtain information about open files
fprintf    Write formatted data to file
fread    Read binary data from file
fscanf    Read formatted data from file
fseek    Set file position indicator
ftell    Get file position indicator
fwrite    Write binary data from a MATLAB matrix to a file
Purpose

Function evaluation

Syntax

\[ [y_1, y_2, \ldots] = \text{feval} (\text{function}, x_1, \ldots, x_n) \]

Description

\[ [y_1, y_2, \ldots] = \text{feval} (\text{function}, x_1, \ldots, x_n) \]

If \text{function} is a string containing the name of a function (usually defined by an M-file), then \text{feval} (\text{function}, x_1, \ldots, x_n) evaluates that function at the given arguments.

Examples

The statements:

\[
[V, D] = \text{feval ('eig', A)}
\]

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

are equivalent. \text{feval} is useful in functions that accept string arguments specifying function names. For example, the function:

\[
\begin{align*}
\text{function} & \quad \text{plotf(fun, x)} \\
y & = \text{feval(fun, x)}; \\
\text{plot}(x, y)
\end{align*}
\]

can be used to graph other functions.

See Also

\begin{align*}
\text{assignin} & \quad \text{Assign value to variable in workspace} \\
\text{builtin} & \quad \text{Execute builtin function from overloaded method} \\
\text{eval} & \quad \text{Interpret strings containing MATLAB expressions} \\
\text{evalin} & \quad \text{Evaluate expression in workspace.}
\end{align*}
Purpose
One-dimensional fast Fourier transform

Syntax
Y = fft(X)
Y = fft(X, n)
Y = fft(X, [], di m)
Y = fft(X, n, di m)

Definition
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:

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

where

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

is an \( n \)th root of unity.

Description
\( \text{Y} = \text{fft}(X) \) returns the discrete Fourier transform of vector \( X \), computed with a fast Fourier transform (FFT) algorithm.

If \( X \) is a matrix, \( \text{fft} \) returns the Fourier transform of each column of the matrix.

If \( X \) is a multidimensional array, \( \text{fft} \) operates on the first nonsingleton dimension.

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

\( \text{Y} = \text{fft}(X, [], \text{di m}) \) and \( \text{Y} = \text{fft}(X, n, \text{di m}) \) apply the FFT operation across the dimension \( \text{di m} \).
Remarks

The `fft` function employs a radix-2 fast Fourier transform algorithm if the length of the sequence is a power of two, and a slower mixed-radix algorithm if it is not. See “Algorithm.”

Examples

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

```matlab
  t = 0:0.001:0.6;  % Time vector
  x = sin(2*pi*50*t) + sin(2*pi*120*t);  % Signal components
  y = x + 2*randn(size(t));  % Noisy signal
  plot(y(1:50));  % Original signal
```

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

```matlab
  Y = fft(y,512);  % FFT of the noisy signal
```

The power spectral density, a measurement of the energy at various frequencies, is

```matlab
  Pyy = Y .* conj(Y) / 512;  % Power spectral density
```

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

```matlab
  f = 1000*(0:256)/512;  % Frequency vector
  plot(f, Pyy(1:257));  % Graph of power spectral density
```

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

Algorithm

When the sequence length is a power of two, a high-speed radix-2 fast Fourier transform algorithm is employed. The radix-2 FFT routine is optimized to perform a real FFT if the input sequence is purely real, otherwise it computes the complex FFT. This causes a real power-of-two FFT to be about 40% faster than a complex FFT of the same length.
When the sequence length is not an exact power of two, an alternate algorithm 
finds the prime factors of the sequence length and computes the mixed-radix 
discrete Fourier transforms of the shorter sequences.

The time it takes to compute an FFT varies greatly depending upon the 
sequence length. The FFT of sequences whose lengths have many prime factors 
is computed quickly; the FFT of those that have few is not. Sequences whose 
lengths are prime numbers are reduced to the raw (and slow) discrete Fourier 
transform (DFT) algorithm. For this reason it is generally better to stay with 
power-of-two FFTs unless other circumstances dictate that this cannot be done. 
For example, on one machine a 4096-point real FFT takes 2.1 seconds and a 
complex FFT of the same length takes 3.7 seconds. The FFTs of neighboring 
sequences of length 4095 and 4097, however, take 7 seconds and 58 seconds, 
respectively.

See Also
dftmtx, filter, freqz, specplot, and spectrum in the Signal Processing 
Toolbox, and:

fft2 Two-dimensional fast Fourier transform
fftsift Rearrange the outputs of fft and fft2
ifft Inverse one-dimensional fast Fourier transform
**Purpose**
Two-dimensional fast Fourier transform

**Syntax**

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

**Description**

Y = fft2(X) performs the two-dimensional FFT. The result Y is the same size as X.

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

**Algorithm**

fft2(X) can be simply computed as

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

This computes the one-dimensional FFT of each column X, then of each row of the result. The time required to compute fft2(X) depends strongly on the number of prime factors in [m n] = size(X). It is fastest when m and n are powers of 2.

**See Also**

fft
fftshift
ifft2

One-dimensional fast Fourier transform
Rearrange the outputs of fft and fft2
Inverse two-dimensional fast Fourier transform
Purpose
Multidimensional fast Fourier transform

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

Description
Y = fftn(X) performs the N-dimensional fast Fourier transform. The result Y is the same size as X.

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

Algorithm
fftn(X) is equivalent to

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

This computes in-place the one-dimensional fast Fourier transform along each dimension of X. The time required to compute fftn(X) depends strongly on the number of prime factors of the dimensions of X. It is fastest when all of the dimensions are powers of 2.

See Also
fft  One-dimensional fast Fourier transform
fftn Two-dimensional fast Fourier transform
ifftn Inverse multidimensional fast Fourier transform
**Purpose**  
Shift DC component of fast Fourier transform to center of spectrum

**Syntax**  
\[ Y = \text{fftshift}(X) \]

**Description**  
\( Y = \text{fftshift}(X) \) rearranges the outputs of \( \text{fft}, \text{fft2}, \) and \( \text{fftn} \) by moving the zero frequency component to the center of the array.

For vectors, \( \text{fftshift}(X) \) swaps the left and right halves of \( X \). For matrices, \( \text{fftshift}(X) \) swaps quadrants one and three of \( X \) with quadrants two and four. For higher-dimensional arrays, \( \text{fftshift}(X) \) swaps “half-spaces” of \( X \) along each dimension.

**Examples**  
For any matrix \( X \)
\[ Y = \text{fft2}(X) \]

has \( Y(1,1) = \sum \sum (X) \); the DC component of the signal is in the upper-left corner of the two-dimensional FFT. For
\[ Z = \text{fftshift}(Y) \]

this DC component is near the center of the matrix.

**See Also**
- \( \text{fft} \)  
  One-dimensional fast Fourier transform
- \( \text{fft2} \)  
  Two-dimensional fast Fourier transform
- \( \text{fftn} \)  
  Multidimensional fast Fourier transform
**Purpose**  
Return the next line of a file as a string without line terminator(s)

**Syntax**  
\[ \text{line} = \text{fgetl}(\text{fid}) \]

**Description**  
\[ \text{line} = \text{fgetl}(\text{fid}) \] returns the next line of the file with identifier \( \text{fid} \). If \( \text{fgetl} \) encounters the end of a file, it returns \(-1\). (See \text{fopen} for a complete description of \( \text{fid} \).)

The returned string \( \text{line} \) does not include the line terminator(s) with the text line (to obtain the line terminator(s), use \text{fgets}).

**See Also**  
\text{fgets}  
Return the next line of a file as a string with line terminator(s)
fgets

**Purpose**

Return the next line of a file as a string with line terminator(s).

**Syntax**

```matlab
line = fgets(fid)
line = fgets(fid, nchar)
```

**Description**

`line = fgets(fid)` returns the next line for the file with identifier `fid`. If `fgets` encounters the end of a file, it returns -1. (See `fopen` for a complete description of `fid`.)

The returned string `line` includes the line terminator(s) associated with the text line (to obtain the string without the line terminator(s), use `fgetl`).

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

**See Also**

`fgetl` Return the next line of a file as a string without line terminator(s).
Field names of a structure

names = fieldnames(s)

returns a cell array of strings containing the structure field names associated with the structure s.

Examples

Given the structure:

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

Then the command n = fieldnames(mystr) yields

n =

' name'
' ID'

See Also

getfield Get field of structure array
setfield Set field of structure array
fileparts

Purpose
Filename parts

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

Description
[path, name, ext, ver] = fileparts(file) returns the path, filename, extension, and version for the specified file. ver will be nonempty only on VMS systems. fileparts is platform dependent.

You can reconstruct the file from the parts using fullfile(path,[name ext ver]).

See Also
fullfile Build full filename from parts
**Purpose**
Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter

**Syntax**

```matlab
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[...] = filter(b,a,X,[],dim)
```

**Description**
The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a direct form II transposed implementation of the standard difference equation (see “Algorithm”).

```matlab
y = filter(b,a,X) filters the data in vector X with the filter described by numerator coefficient vector b and denominator coefficient vector a. If a(1) is not equal to 1, filter normalizes the filter coefficients by a(1). If a(1) equals 0, filter returns an error.
```

If X is a matrix, `filter` operates on the columns of X. If X is a multidimensional array, `filter` operates on the first nonsingleton dimension.

```matlab
[y,zf] = filter(b,a,X) returns the final conditions, zf, of the filter delays. Output zf is a vector of max(size(a), size(b)) or an array of such vectors, one for each column of X.
```

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

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

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

\[
x(m) \quad \cdots \quad b(n) \quad z^{-1} \quad \cdots \quad b(3) \quad z^{-1} \quad b(2) \quad z^{-1} \quad b(1)
\]

\[
\sum_{a(n)} z_{n-1}(m) \quad \cdots \quad -a(3) \quad Z_2(m) \quad -a(2) \quad Z_1(m) \quad y(m)
\]

or

\[
y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(nb+1)x(n-nb) - a(2)y(n-1) - \ldots - a(na+1)y(n-na)
\]

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

\[
y(m) = b(1)x(m) + z_1(m-1)
\]

\[
z_1(m) = b(2)x(m) + z_2(m-1) - a(2)y(m)
\]

\[
\vdots
\]

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

\[
z_{n-1}(m) = b(n)x(m) - a(n)y(m)
\]

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

\[
Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \ldots + a(na+1)z^{-na}}X(z)
\]

See Also
- `filtfilt` in the Signal Processing Toolbox, and:
- `filter2` Two-dimensional digital filtering

References

Purpose
Two-dimensional digital filtering

Syntax
Y = filter2(h, X)
Y = filter2(h, X, shape)

Description
Y = filter2(h, X) filters the data in X with the two-dimensional FIR filter in the matrix h. It computes the result, y, using two-dimensional correlation, and returns the central part of the correlation that is the same size as X.

Y = filter2(h, X, shape) returns the part of Y specified by the shape parameter. shape is a string with one of these values:

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

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

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

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

See Also
conv2 Two-dimensional convolution
filter Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
**Purpose**

Find indices and values of nonzero elements

**Syntax**

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

**Description**

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

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

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

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

**Examples**

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

Some operations on a vector

```matlab
x = [11 0 33 0 55];
find(x)
```

```matlab
ans =

1
3
5
```

```matlab
find(x == 0)
```

```matlab
ans =

2
4
```
find(0 < x & x < 10*pi)

ans =

1

And on a matrix

M = magic(3)

M =

8  1  6
3  5  7
4  9  2

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

i =

j =

m =

1  1  1
2  2  1
3  3  1

See Also

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

nonzeros Nonzero matrix elements
sparse Create sparse matrix
findstr

Purpose

Find one string within another

Syntax

\[ k = \text{findstr}(\text{str1}, \text{str2}) \]

Description

\[ k = \text{findstr}(\text{str1}, \text{str2}) \] finds the starting indices of any occurrences of the shorter string within the longer.

Examples

\[
\text{str1} = '\text{Find the starting indices of the shorter string.}' ; \\
\text{str2} = '\text{the}' ; \\
\text{findstr(str1, str2)}
\]

\[
\text{ans} = \\
\quad 6 \quad 30
\]

See Also

\begin{align*}
\text{strcmp} & \quad \text{Compare strings} \\
\text{strmatch} & \quad \text{Find possible matches for a string} \\
\text{strncmp} & \quad \text{Compare the first } n \text{ characters of two strings}
\end{align*}
Purpose
Round towards zero

Syntax
B = fix(A)

Description
B = fix(A) rounds the elements of A toward zero, resulting in an array of integers. For complex A, the imaginary and real parts are rounded independently.

Examples

```
a =
  Columns 1 through 4
  -1.9000    -0.2000    3.4000    5.6000
  Columns 5 through 6
       7.0000    2.4000 + 3.6000i

fix(a)
ans =
  Columns 1 through 4
       -1.0000         0    3.0000    5.0000
  Columns 5 through 6
       7.0000    2.0000 + 3.0000i
```

See Also
ceil     Round toward infinity
floor    Round towards minus infinity
round    Round to nearest integer
flipdim

Purpose
Flip array along a specified dimension

Syntax
B = flipdim(A, dim)

Description
B = flipdim(A, dim) returns A with dimension dim flipped.
When the value of dim is 1, the array is flipped row-wise down. When dim is 2, the array is flipped columnwise left to right. flipdim(A, 1) is the same as flipud(A), and flipdim(A, 2) is the same as fliplr(A).

Examples
flipdim(A, 1) where
A =

1  4
2  5
3  6

produces

3  6
2  5
1  4

See Also
fliplr Flip matrices left-right
flipud Flip matrices up-down
permute Rearrange the dimensions of a multidimensional array
rot90 Rotate matrix 90°
Purpose

Flip matrices left-right

Syntax

B = fliplr(A)

Description

B = fliplr(A) returns A with columns flipped in the left-right direction, that is, about a vertical axis.

Examples

A =
1 4
2 5
3 6

produces

4 1
5 2
6 3

Limitations

Array A must be two dimensional.

See Also

flipdim Flip array along a specified dimension
flipud Flip matrices up-down
rot90 Rotate matrix 90°
**Purpose**
Flip matrices up-down

**Syntax**

\[ B = \text{flipud}(A) \]

**Description**

\[ B = \text{flipud}(A) \] returns \( A \) with rows flipped in the up-down direction, that is, about a horizontal axis.

**Examples**

\[
A = \\
\begin{bmatrix}
1 & 4 \\
2 & 5 \\
3 & 6 \\
\end{bmatrix}
\]

produces

\[
\begin{bmatrix}
3 & 6 \\
2 & 5 \\
1 & 4 \\
\end{bmatrix}
\]

**Limitations**

Array \( A \) must be two dimensional.

**See Also**

- `flipdim` — Flip array along a specified dimension
- `fliplr` — Flip matrices left-right
- `rot90` — Rotate matrix 90°
**Purpose**
Round towards minus infinity

**Syntax**

\[
B = \text{floor}(A)
\]

**Description**

\[
B = \text{floor}(A)
\]
rounds the elements of \(A\) to the nearest integers less than or equal to \(A\). For complex \(A\), the imaginary and real parts are rounded independently.

**Examples**

\[
a =
\begin{array}{cccc}
-1.9000 & -0.2000 & 3.4000 & 5.6000 \\
7.0000 & 2.4000 + 3.6000i
\end{array}
\]

\[
floor(a)
\]

\[
a =
\begin{array}{cccc}
-2.0000 & -1.0000 & 3.0000 & 5.0000 \\
7.0000 & 2.0000 + 3.0000i
\end{array}
\]

**See Also**

ceil \hspace{1cm} \text{Round toward infinity}
fix \hspace{1cm} \text{Round towards zero}
round \hspace{1cm} \text{Round to nearest integer}
**flops**

**Purpose**

Count floating-point operations

**Syntax**

\[
f = \text{flops}
\]

\[
flops(0)
\]

**Description**

\[
f = \text{flops}
\]

returns the cumulative number of floating-point operations.

\[
flops(0)
\]

resets the count to zero.

**Examples**

If \( A \) and \( B \) are real \( n \)-by-\( n \) matrices, some typical flop counts for different operations are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flop Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A+B )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>( A \cdot B )</td>
<td>( 2n^3 )</td>
</tr>
<tr>
<td>( A^100 )</td>
<td>( 99n^3 + 2n^3 )</td>
</tr>
<tr>
<td>\text{lu}(A)</td>
<td>( (2/3) n^3 )</td>
</tr>
</tbody>
</table>

MATLAB's version of the LINPACK benchmark is:

\[
\begin{align*}
\text{n} &= 100; \\
\text{A} &= \text{rand}(\text{n}, \text{n}); \\
\text{b} &= \text{rand}(\text{n}, 1); \\
\text{flops}(0) \\
\text{tic;} \\
\text{x} &= \text{A}\backslash\text{b}; \\
\text{t} &= \text{toc} \\
\text{megaflops} &= \text{flops}/\text{t}/1.0e6
\end{align*}
\]

**Algorithm**

It is not feasible to count all the floating-point operations, but most of the important ones are counted. Additions and subtractions are each one flop if real and two if complex. Multiplications and divisions count one flop each if the result is real and six flops if it is complex. Elementary functions count one if real and more if complex.
Purpose
Minimize a function of one variable

Syntax
\[ x = \text{fmin}('\text{fun}', x1, x2) \]
\[ x = \text{fmin}('\text{fun}', x1, x2, \text{options}) \]
\[ x = \text{fmin}('\text{fun}', x1, x2, \text{options}, P1, P2, ...) \]
\[ [x, \text{options}] = \text{fmin}(...) \]

Description
\[ x = \text{fmin}('\text{fun}', x1, x2) \] returns a value of \( x \) which is a local minimizer of \( \text{fun}(x) \) in the interval \( x_1 < x < x_2 \).

\[ x = \text{fmin}('\text{fun}', x1, x2, \text{options}) \] does the same as the above, but uses \text{options} control parameters.

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

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

Arguments
\( x1, x2 \) Interval over which \text{function} is minimized.
\( P1, P2 \ldots \) Arguments to be passed to \text{function}.
\text{fun} A string containing the name of the function to be minimized.
\text{options} A vector of control parameters. Only three of the 18 components of \text{options} are referenced by \text{fmin}; Optimization Toolbox functions use the others. The three control \text{options} used by \text{fmin} are:

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

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

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

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

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

Then invoke `fmin` with

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

The result is

```matlab
x =
    0.8165
```

The value of the function at the minimum is

```matlab
y = f(x)
```

```matlab
y =
    -6.0887
```

**Algorithm**

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

**See Also**

- `fmins` Minimize a function of several variables
- `fzero` Zero of a function of one variable
- `foptions` in the Optimization Toolbox (or type `help foptions`).

**References**

**Purpose**
Minimize a function of several variables

**Syntax**

\[ x = \text{fmins}('fun', x0) \]
\[ x = \text{fmins}('fun', x0, \text{options}) \]
\[ x = \text{fmins}('fun', x0, \text{options}, [], P1, P2, \ldots) \]
\[ [x, \text{options}] = \text{fmins}(\ldots) \]

**Description**

\[ x = \text{fmins}('fun', x0) \text{ returns a vector } x \text{ which is a local minimizer of } \text{fun}(x) \text{ near } x_0. \]
\[ x = \text{fmins}('fun', x0, \text{options}) \text{ does the same as the above, but uses options control parameters. } \]
\[ x = \text{fmins}('fun', x0, \text{options}, [], P1, P2, \ldots) \text{ does the same as above, but passes arguments to the objective function, } \text{fun}(x, P1, P2, \ldots). \text{ Pass an empty matrix for options to use the default value. } \]
\[ [x, \text{options}] = \text{fmins}(\ldots) \text{ returns, in options(10), a count of the number of steps taken. } \]

**Arguments**

\[ x0 \quad \text{Starting vector.} \]
\[ P1, P2, \ldots \quad \text{Arguments to be passed to } \text{fun}. \]
\[ [\] \quad \text{Argument needed to provide compatibility with fminu in the Optimization Toolbox.} \]
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.

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

The statements

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

\[
x =
\]

\[
1.0000 \quad 1.0000
\]

\[
\text{ans} =
\]

\[
165
\]

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

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

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

Then the statement

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

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

**Algorithm**

The algorithm is the Nelder-Mead simplex search described in the two references. It is a direct search method that does not require gradients or other derivative information. If \(n\) is the length of \(x\), a simplex in \(n\)-dimensional space is characterized by the \(n+1\) distinct vectors which are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid.

At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

**See Also**

fmin  
Minimize a function of one variable

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


Purpose
Open a file or obtain information about open files

Syntax

\[
fid = \text{fopen}(\text{filename, permission})
\]

\[
[fid, \text{message}] = \text{fopen}(\text{filename, permission, format})
\]

\[
fids = \text{fopen('all')}
\]

\[
[f\text{ilename, permission, format}] = \text{fopen}(f\text{id})
\]

Description

If \text{fopen} successfully opens a file, it returns a file identifier \text{fid}, and the value of \text{message} is empty. The file identifier can be used as the first argument to other file input/output routines. If \text{fopen} does not successfully open the file, it returns a -1 value for \text{fid}. In that case, the value of \text{message} is a string that helps you determine the type of error that occurred.

Two \text{fids} are predefined and cannot be explicitly opened or closed:

- 1 — Standard output, which is always open for appending (\text{permission} set to 'a'), and
- 2 — Standard error, which is always open for appending (\text{permission} set to 'a').

\[
fid = \text{fopen}(\text{filename, permission})
\]

opens the file \text{filename} in the mode specified by \text{permission} and returns \text{fid}, the file identifier. \text{filename} may a MATLABPATH relative partial pathname. If the file is opened for reading and it is not found in the current working directory, \text{fopen} searches down MATLAB's search path.

\text{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.
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).

```matlab
[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.64'` or `'a'`  IEEE floating point with little-endian byte ordering and 64-bit long data type
- `'ieee-be.64'` 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.
[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.
for

**Purpose**
Repeat statements a specific number of times

**Syntax**

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

**Description**
The general format is

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

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

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

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

**Examples**

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

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

Steps with increments of \(-0.1\)

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

Successively set \( e \) to the unit \( n \)-vectors:

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

The line

```
for V = A,..., end
```
has the same effect as

\[
\text{for } j = 1:n, \ V = A(:,j); \ldots, \ \text{end}
\]

except \( j \) is also set here.

**See Also**

- **break** Break out of flow control structures
- **end** Terminate \texttt{for, while, switch, and if} statements and indicate the last index
- **if** Conditionally execute statements
- **return** Return to the invoking function
- **switch** Switch among several cases based on expression
- **while** Repeat statements an indefinite number of times
**Purpose**

Control the output display format

**Syntax**

MATLAB performs all computations in double precision. The `format` command described below switches among different display formats.

**Description**

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format</code></td>
<td>Default. Same as <code>short</code></td>
<td></td>
</tr>
<tr>
<td><code>format short</code></td>
<td>5 digit scaled fixed point</td>
<td>3. 1416</td>
</tr>
<tr>
<td><code>format long</code></td>
<td>15 digit scaled fixed point</td>
<td>3. 14159265358979</td>
</tr>
<tr>
<td><code>format short e</code></td>
<td>5 digit floating-point</td>
<td>3. 1416e+00</td>
</tr>
<tr>
<td><code>format long e</code></td>
<td>15 digit floating-point</td>
<td>3. 141592653589793e+00</td>
</tr>
<tr>
<td><code>format short g</code></td>
<td>Best of 5 digit fixed or floating</td>
<td>3. 1416</td>
</tr>
<tr>
<td><code>format long g</code></td>
<td>Best of 15 digit fixed or floating</td>
<td>3. 14159265358979</td>
</tr>
<tr>
<td><code>format hex</code></td>
<td>Hexadecimal</td>
<td>400921fb54442d18</td>
</tr>
<tr>
<td><code>format bank</code></td>
<td>Fixed dollars and cents</td>
<td>3. 14</td>
</tr>
<tr>
<td><code>format rat</code></td>
<td>Ratio of small integers</td>
<td>355/113</td>
</tr>
<tr>
<td><code>format +</code></td>
<td>+, -, blank</td>
<td>+</td>
</tr>
<tr>
<td><code>format compact</code></td>
<td>Suppresses excess line feeds.</td>
<td></td>
</tr>
<tr>
<td><code>format loose</code></td>
<td>Add line feeds.</td>
<td></td>
</tr>
</tbody>
</table>

**Algorithms**

The command `format +` displays +, -, and blank characters for positive, negative, and zero elements. `format hex` displays the hexadecimal representation of a binary double-precision number. `format rat` uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See `rat.m` for the complete code.

**See Also**

`fprintf`, `num2str`, `rat`, `sprintf`, `spy`
**Purpose**

Write formatted data to file

**Syntax**

\[
\text{count} = \text{fprintf}(\text{fid, format, A, \ldots}) \\
\text{fprintf}(\text{format, A, \ldots})
\]

**Description**

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

Argument \(\text{fid}\) is an integer file identifier obtained from \text{fopen}. (It may also be 1 for standard output (the screen) or 2 for standard error. See \text{fopen} for more information.) Omitting \(\text{fid}\) from \text{fprintf} ’s argument list causes output to appear on the screen, and is the same as writing to standard output (\(\text{fid} = 1\))

\[
\text{fprintf}(\text{format, A, \ldots}) \text{ writes to standard output—the screen.}
\]

The \text{format} string specifies notation, alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters; along with escape characters, conversion specifiers, and other characters, organized as shown below:

```
% -12.5e
```

- **Initial % character**
- **Flag**
- **Field width and precision**
- **Conversion character**

For more information see “Tables” and “References”.

2-279
Remarks

The `fprintf` function behaves like its ANSI C language `fprintf()` namesake with certain exceptions and extensions. These include:

1. The following non-standard subtype specifiers are supported for conversion specifiers `%o`, `%u`, `%x`, and `%X`.
   - The underlying C data type is a float rather than an unsigned integer.
   - The underlying C data type is a double rather than an unsigned integer.

   For example, to print a double-precision value in hexadecimal, use a format like `%bx`.

2. The `fprintf` function is vectorized for the case when input matrix A is non-scalar. The format string is cycled through the elements of A (columnwise) until all the elements are used up. It is then cycled in a similar manner, without reinitializing, through any additional matrix arguments.

Tables

The following tables describe the non-alphanumeric characters found in format specification strings.

<table>
<thead>
<tr>
<th>Escape Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Character</strong></td>
</tr>
<tr>
<td>\n</td>
</tr>
<tr>
<td>\t</td>
</tr>
<tr>
<td>\b</td>
</tr>
<tr>
<td>\r</td>
</tr>
<tr>
<td>\f</td>
</tr>
<tr>
<td>&quot; or &quot;</td>
</tr>
<tr>
<td>%</td>
</tr>
</tbody>
</table>
Conversion characters specify the notation of the output.

**Conversion Specifiers**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td>%d</td>
<td>Decimal notation (signed)</td>
</tr>
<tr>
<td>%e</td>
<td>Exponential notation (using a lowercase e as in 3. 1415e+00)</td>
</tr>
<tr>
<td>%E</td>
<td>Exponential notation (using an uppercase E as in 3. 1415E+00)</td>
</tr>
<tr>
<td>%f</td>
<td>Fixed-point notation</td>
</tr>
<tr>
<td>%g</td>
<td>The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.</td>
</tr>
<tr>
<td>%G</td>
<td>Same as %g, but using an uppercase E</td>
</tr>
<tr>
<td>%o</td>
<td>Octal notation (unsigned)</td>
</tr>
<tr>
<td>%s</td>
<td>String of characters</td>
</tr>
<tr>
<td>%u</td>
<td>Decimal notation (unsigned)</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal notation (using lowercase letters a–f)</td>
</tr>
<tr>
<td>%X</td>
<td>Hexadecimal notation (using uppercase letters A–F)</td>
</tr>
</tbody>
</table>

Other characters can be inserted into the conversion specifier between the % and the conversion character.
Other Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A minus sign (–)</td>
<td>Left-justifies the converted argument in its field.</td>
<td>%–5.2d</td>
</tr>
<tr>
<td>A plus sign (+)</td>
<td>Always prints a sign character (+ or –).</td>
<td>%+5.2d</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Pad with zeros rather than spaces.</td>
<td>%05.2d</td>
</tr>
<tr>
<td>Digits (field width)</td>
<td>A digit string specifying the minimum number of digits to be printed.</td>
<td>%6f</td>
</tr>
<tr>
<td>Digits (precision)</td>
<td>A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.</td>
<td>%6.2f</td>
</tr>
</tbody>
</table>

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
',y);
close(fid)
```
create a text file called exp.txt containing a short table of the exponential function:
```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.00000000</td>
</tr>
<tr>
<td>0.1</td>
<td>1.10517092</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>2.71828183</td>
</tr>
</tbody>
</table>
```
The command
```
fprintf('A unit circle has circumference %g.\n',2*pi)
```
displays a line on the screen:
```
A unit circle has circumference 6.283186.
```
To insert a single quotation mark in a string, use two single quotation marks together. For example,

```matlab
fprintf(1,'It''s Friday.
')
```
displays on the screen:

```
It's Friday.
```

The commands

```matlab
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:

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

```
6
A
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


**Purpose**

Read binary data from file

**Syntax**

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

**Description**

`[A, count] = fread(fid, size, precision)` reads binary data from the specified file and writes it into matrix `A`. Optional output argument `count` returns the number of elements successfully read. `fid` is an integer file identifier obtained from `fopen`. `size` is an optional argument that determines how much data is read. If `size` is not specified, `fread` reads to the end of the file. Valid options are:

- `n` Reads `n` elements into a column vector.
- `inf` Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
- `[m n]` Reads enough elements to fill an `m`-by-`n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix.

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

`precision` is a string representing the numeric precision of the values read, `precision` controls the number of bits read for each value and the interpretation of those bits as an integer, a floating-point value, or a character. The `precision` string may contain a positive integer repetition factor of the form `'n*'` which prepends one of the strings above, like `'40*uchar'`. If `precision` is not specified, the default is `'uchar'` (8-bit unsigned character) is assumed. See “Remarks” for more information.

`[A, count] = fread(fid, size, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip after each read. This is useful for extracting data in noncontiguous fields from fixed length records. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits.
Remarks

Numeric precisions can differ depending on how numbers are represented in your computer’s architecture, as well as by the type of compiler used to produce executable code for your computer.

The tables below give C-compliant, platform-independent numeric precision string formats that you should use whenever you want your code to be portable.

For convenience, MATLAB accepts some C and Fortran data type equivalents for the MATLAB precisions listed. If you are a C or Fortran programmer, you may find it more convenient to use the names of the data types in the language with which you are most familiar.

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>C or Fortran</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'char'</td>
<td>'char *1'</td>
<td>Character; 8 bits</td>
</tr>
<tr>
<td>'schar'</td>
<td>'signed char'</td>
<td>Signed character; 8 bits</td>
</tr>
<tr>
<td>'uchar'</td>
<td>'unsigned char'</td>
<td>Unsigned character; 8 bits</td>
</tr>
<tr>
<td>'int 8'</td>
<td>'integer *1'</td>
<td>Integer; 8 bits</td>
</tr>
<tr>
<td>'int 16'</td>
<td>'integer *2'</td>
<td>Integer; 16 bits</td>
</tr>
<tr>
<td>'int 32'</td>
<td>'integer *4'</td>
<td>Integer; 32 bits</td>
</tr>
<tr>
<td>'int 64'</td>
<td>'integer *8'</td>
<td>Integer; 64 bits</td>
</tr>
<tr>
<td>'uint 8'</td>
<td>'integer *1'</td>
<td>Unsigned integer; 8 bits</td>
</tr>
<tr>
<td>'uint 16'</td>
<td>'integer *2'</td>
<td>Unsigned integer; 16 bits</td>
</tr>
<tr>
<td>'uint 32'</td>
<td>'integer *4'</td>
<td>Unsigned integer; 32 bits</td>
</tr>
<tr>
<td>'uint 64'</td>
<td>'integer *8'</td>
<td>Unsigned integer; 64 bits</td>
</tr>
<tr>
<td>'float 32'</td>
<td>'real *4'</td>
<td>Floating-point; 32 bits</td>
</tr>
<tr>
<td>'float 64'</td>
<td>'real *8'</td>
<td>Floating-point; 64 bits</td>
</tr>
</tbody>
</table>
fread

If you always work on the same platform and don’t care about portability, these platform-dependent numeric precision string formats are also available:

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>C or Fortran</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'short'</td>
<td>'short'</td>
<td>Integer; 16 bits</td>
</tr>
<tr>
<td>'int'</td>
<td>'int'</td>
<td>Integer; 32 bits</td>
</tr>
<tr>
<td>'long'</td>
<td>'long'</td>
<td>Integer; 32 or 64 bits</td>
</tr>
<tr>
<td>'ushort'</td>
<td>'unsigned short'</td>
<td>Unsigned integer; 16 bits</td>
</tr>
<tr>
<td>'uint'</td>
<td>'unsigned int'</td>
<td>Unsigned integer; 32 bits</td>
</tr>
<tr>
<td>'ulong'</td>
<td>'unsigned long'</td>
<td>Unsigned integer; 32 or 64 bits</td>
</tr>
<tr>
<td>'float'</td>
<td>'float'</td>
<td>Floating-point; 32 bits</td>
</tr>
<tr>
<td>'double'</td>
<td>'double'</td>
<td>Floating-point; 64 bits</td>
</tr>
</tbody>
</table>

Two formats map to an input stream of bits rather than bytes:

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>C or Fortran</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bitN'</td>
<td></td>
<td>Signed integer; N bits (1 ≤ N ≤ 64)</td>
</tr>
<tr>
<td>'ubitN'</td>
<td></td>
<td>Unsigned integer; N bits (1 ≤ N ≤ 64)</td>
</tr>
</tbody>
</table>

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
Purpose
Determine frequency spacing for frequency response

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

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

[f1, f2] = freqspace(n) returns the two-dimensional frequency vectors f1 and f2 for an n-by-n matrix.
For n odd, both f1 and f2 are [-n+1: 2: n-1]/n.
For n even, both f1 and f2 are [-n: 2: n-2]/n.

[f1, f2] = freqspace([m n]) returns the two-dimensional frequency vectors f1 and f2 for an m-by-n matrix.
[x1, y1] = freqspace(..., 'meshgrid') is equivalent to
[f1, f2] = freqspace(...);
[x1, y1] = meshgrid(f1, f2);

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

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

See Also
meshgrid Generate X and Y matrices for three-dimensional plots
frewind

**Purpose**
Rewind an open file

**Syntax**
`frewind(fid)`

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

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

**See Also**
- `fclose`: Close one or more open files
- `ferror`: Query MATLAB about errors in file input or output
- `fopen`: Open a file or obtain information about open files
- `fprintf`: Write formatted data to file
- `fread`: Read binary data from file
- `fscanf`: Read formatted data from file
- `fseek`: Set file position indicator
- `ftell`: Get file position indicator
- `fwrite`: Write binary data from a MATLAB matrix to a file
fscanf

Purpose
Read formatted data from file

Syntax
\[ A = \text{fscanf} \left( f i d, \text{format} \right) \]
\[ [ A, \text{count} ] = \text{fscanf} \left( f i d, \text{format}, \text{size} \right) \]

Description
\( A = \text{fscanf} \left( f i d, \text{format} \right) \) reads all the data from the file specified by \( f i d \), converts it according to the specified \( \text{format} \) string, and returns it in matrix \( A \).

Argument \( f i d \) is an integer file identifier obtained from \( \text{fopen} \). \( \text{format} \) is a string specifying the format of the data to be read. See “Remarks” for details.

\( [ A, \text{count} ] = \text{fscanf} \left( f i d, \text{format}, \text{size} \right) \) reads the amount of data specified by \( \text{size} \), converts it according to the specified \( \text{format} \) string, and returns it along with a \( \text{count} \) of elements successfully read. \( \text{size} \) is an argument that determines how much data is read. Valid options are:

- \( n \) Read \( n \) elements into a column vector.
- \( \text{inf} \) Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
- \( [ m \, n ] \) Read enough elements to fill an \( m \times n \) matrix, filling the matrix in column order. \( n \) can be \( \text{inf} \), but not \( m \).

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

Remarks
When MATLAB reads a specified file, it attempts to match the data in the file to the \text{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 \text{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 %g for a double floating-point number.

Valid conversion characters are:

- %: 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 % is used, an element read may use several MATLAB matrix elements, each holding one character. Use %c to read space characters; the format % 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:

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

**See Also**

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

**Syntax**

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

**Description**

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

**Arguments**

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

**See Also**

- `fopen` Open a file or obtain information about open files
- `ftell` Get file position indicator
Purpose

Get file position indicator

Syntax

position = ftell(fid)

Description

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

See Also

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

Purpose
Convert sparse matrix to full matrix

Syntax
A = full(S)

Description
A = full(S) converts a sparse matrix S to full storage organization. If S is a full matrix, it is left unchanged. If A is full, issparse(A) is 0.

Remarks
Let X be an m by n matrix with nz = nnz(X) nonzero entries. Then full(X) requires space to store mn real numbers while sparse(X) requires space to store nz real numbers and (nz+n) integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, sparse(X) requires less storage than full(X) if the density, nnz/ prod(size(X)), is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

Examples
Here is an example of a sparse matrix with a density of about two-thirds. sparse(S) and full(S) require about the same number of bytes of storage.

S = sparse(rand(200,200) < 2/3);
A = full(S);
whos
Name      Size      Bytes     Class           
A    200X200    320000     double array (logical)
S    200X200    318432     sparse array (logical)

See Also
sparse
Create sparse matrix
**Purpose**
Build full filename from parts

**Syntax**
```
fullfile(dir1, dir2, ..., filename)
```

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

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

except that care is taken to handle the cases when the directories begin or end with a directory separator. Specify the filename as `'` to build a pathname from parts. On VMS, care is taken to handle the cases involving `[` or `]`.

**Example**
```
fullfile(matlabroot,'toolbox/matlab/general/Contents.m') and
fullfile(matlabroot,'toolbox','matlab','general','Contents.m')
```
produce the same result on UNIX, but only the second one works on all platforms.
**Purpose**

Function M-files

**Description**

You add new functions to MATLAB's vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an M-file.

M-files can be either scripts or functions. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character, and has a filename extension of .m. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the .m extension. For example, the existence of a file on disk called stat.m with

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

defines a new function called stat that calculates the mean and standard deviation of a vector. The variables within the body of the function are all local variables.

A subfunction, visible only to the other functions in the same file, is created by defining a new function with the `function` keyword after the body of the preceding function or subfunction. For example, `avg` is a subfunction within the file `stat.m`:

```matlab
function mean = avg(x, n)
    mean = sum(x)/n;
```

```matlab
function [mean, stdev] = stat(x)
    n = length(x);
    mean = avg(x, n);
    stdev = sqrt(sum((x-avg(x,n)).^2)/n));
```
Subfunctions are not visible outside the file where they are defined. Functions normally return when the end of the function is reached. Use a return statement to force an early return.

When MATLAB does not recognize a function by name, it searches for a file of the same name on disk. If the function is found, MATLAB compiles it into memory for subsequent use. In general, if you input the name of something to MATLAB, the MATLAB interpreter:

1. Checks to see if the name is a variable.
2. Checks to see if the name is an internal function (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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nargin</td>
<td>Number of function arguments (input)</td>
</tr>
<tr>
<td>nargout</td>
<td>Number of function arguments (output)</td>
</tr>
<tr>
<td>pcode</td>
<td>Create preparsed pseudocode file (P-file)</td>
</tr>
<tr>
<td>varargin</td>
<td>Pass or return variable numbers of arguments (input)</td>
</tr>
<tr>
<td>varargout</td>
<td>Pass or return variable numbers of arguments (output)</td>
</tr>
<tr>
<td>what</td>
<td>Directory listing of M-files, MAT-files, and MEX-files</td>
</tr>
</tbody>
</table>
funm

Purpose
Evaluate functions of a matrix

Syntax
Y = funm(X,'function')
[Y, esterr] = funm(X,'function')

Description
Y = funm(X,'function') evaluates function using Parlett's method [1]. X
must be a square matrix, and function any element-wise function.

The commands funm(X,'sqrt') and funm(X,'log') are equivalent to the
commands sqrtm(X) and logm(X). The commands funm(X,'exp') and expm(X)
compute the same function, but by different algorithms. expm(X) is preferred.

[Y, esterr] = funm(X,'function') does not print any message, but returns
a very rough estimate of the relative error in the computer result. If X is
symmetric or Hermitian, then its Schur form is diagonal, and funm is able to
produce an accurate result.

Examples
The statements
S = funm(X,'sin');
C = funm(X,'cos');
produce the same results to within roundoff error as
E = expm(i*X);
C = real(E);
S = imag(E);

In either case, the results satisfy S*S+C*C = I, where I = eye(size(X)).

Algorithm
The matrix functions are evaluated using Parlett's algorithm, which is
described in [1]. The algorithm uses the Schur factorization of the matrix and
may give poor results or break down completely when the matrix has repeated
eigenvalues. A warning message is printed when the results may be inaccu-
rate.

See Also
expm  Matrix exponential
logm  Matrix logarithm
sqrtm  Matrix square root
References


fwrite

Purpose
Write binary data to a file

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

Description
count = fwrite(fid, A, precision) writes the elements of matrix A to the
specified file, translating MATLAB values to the specified numeric precision.
(See “Remarks” for more information.)

The data are written to the file in column order, and a count is kept of the
number of elements written successfully. Argument fid is an integer file iden-
tifier obtained from fopen.

count = fwrite(fid, A, precision, skip) includes an optional skip argument that specifies the number of bytes to skip before each write. This is useful
for inserting data into noncontiguous fields in fixed-length records. If
precision is a bit format like 'bitN' or 'ubitN', skip is specified in bits.

Remarks
Numeric precisions can differ depending on how numbers are represented in
your computer’s architecture, as well as by the type of compiler used to produce
executable code for your computer.

The tables below give C-compliant, platform-independent numeric precision
string formats that you should use whenever you want your code to be portable.

For convenience, MATLAB accepts some C and Fortran data type equivalents
for the MATLAB precisions listed. If you are a C or Fortran programmer, you
may find it more convenient to use the names of the data types in the language
with which you are most familiar.
If you always work on the same platform and don’t care about portability, these platform-dependent numeric precision string formats are also available:

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>C or Fortran</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'char'</td>
<td>'char *1'</td>
<td>Character; 8 bits</td>
</tr>
<tr>
<td>'schar'</td>
<td>'signed char'</td>
<td>Signed character; 8 bits</td>
</tr>
<tr>
<td>'uchar'</td>
<td>'unsigned char'</td>
<td>Unsigned character; 8 bits</td>
</tr>
<tr>
<td>'int 8'</td>
<td>'integer *1'</td>
<td>Integer; 8 bits</td>
</tr>
<tr>
<td>'int 16'</td>
<td>'integer *2'</td>
<td>Integer; 16 bits</td>
</tr>
<tr>
<td>'int 32'</td>
<td>'integer *4'</td>
<td>Integer; 32 bits</td>
</tr>
<tr>
<td>'int 64'</td>
<td>'integer *8'</td>
<td>Integer; 64 bits</td>
</tr>
<tr>
<td>'uint 8'</td>
<td>'integer *1'</td>
<td>Unsigned integer; 8 bits</td>
</tr>
<tr>
<td>'uint 16'</td>
<td>'integer *2'</td>
<td>Unsigned integer; 16 bits</td>
</tr>
<tr>
<td>'uint 32'</td>
<td>'integer *4'</td>
<td>Unsigned integer; 32 bits</td>
</tr>
<tr>
<td>'uint 64'</td>
<td>'integer *8'</td>
<td>Unsigned integer; 64 bits</td>
</tr>
<tr>
<td>'float 32'</td>
<td>'real *4'</td>
<td>Floating-point; 32 bits</td>
</tr>
<tr>
<td>'float 64'</td>
<td>'real *8'</td>
<td>Floating-point; 64 bits</td>
</tr>
</tbody>
</table>
### Examples

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

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

### See Also

- `fclose`: Close one or more open files
- `ferror`: Query MATLAB about errors in file input or output
- `fopen`: Open a file or obtain information about open files
- `fprintf`: Write formatted data to file
- `fread`: Read binary data from file
- `fscanf`: Read formatted data from file
- `fseek`: Set file position indicator
- `ftell`: Get file position indicator
**Purpose**
Zero of a function of one variable

**Syntax**

\[
z = \text{fzero('fun', x)} \\
z = \text{fzero('fun', x, tol)} \\
z = \text{fzero('fun', x, tol, trace)} \\
z = \text{fzero('fun', x, tol, trace, P1, P2, \ldots)}
\]

**Description**

- `fzero('fun', x)` finds a zero of `fun`. `fun` is a string containing the name of a real-valued function of a single real variable. The value returned is near a point where `fun` changes sign, or `NaN` if the search fails.
- `fzero('fun', x)` where `x` is a vector of length 2, assumes `x` is an interval where the sign of \( f(x(1)) \) differs from the sign of \( f(x(2)) \). An error occurs if this is not true. Calling `fzero` with an interval guarantees `fzero` will return a value near a point where `fun` changes sign.
- `fzero('fun', x)` where `x` is a scalar value, uses `x` as a starting point. `fzero` looks for a sign change for `fun` and containing `x`. If no such interval is found, `NaN` is returned. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.
- `fzero('fun', x, tol)` returns an answer accurate to within a relative error of `tol`.
- `z = \text{fzero('fun', x, tol, trace)}` displays information at each iteration.
- `z = \text{fzero('fun', x, tol, trace, P1, P2, \ldots)}` provides for additional arguments passed to the function `fun(x, P1, P2, \ldots)`. Pass an empty matrix for `tol` or `trace` to use the default value, for example: `fzero('fun', x, [], [], P1)`

For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the x-axis.

**Arguments**

- `fun` A string containing the name of a file in which an arbitrary function of one variable is defined.
- `x` Your initial estimate of the x-coordinate of a zero of the function or an interval in which you think a zero is found.
- `tol` The relative error tolerance. By default, `tol` is `eps`.

2-303
fzero

trace A nonzero value that causes the fzero command to display information at each iteration of its calculations.

P1, P2 Additional arguments passed to the function

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

\[ x = \text{fzero('sin', 3)} \]
\[ x = 3.1416 \]

To find the zero of cosine between 1 and 2:

\[ x = \text{fzero('cos', [1 2])} \]
\[ x = 1.5708 \]

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

\[ z = \text{fzero('f', 2)} \]
\[ z = 2.0946 \]

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 \texttt{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 \texttt{fzero} M-file is based, is in [2].

Limitations
The \texttt{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, \texttt{fzero} executes until \texttt{Inf}, \texttt{NaN}, or a complex value is detected.

See Also
- \texttt{eps} Floating-point relative accuracy
- \texttt{fmin} Minimize a function of one variable
- \texttt{roots} Polynomial roots

References

### gallery

**Purpose**

Test matrices

**Syntax**

\[
[A, B, C, \ldots] = \text{gallery}('tmfun', P1, P2, \ldots)
\]

- gallery(3) a badly conditioned 3-by-3 matrix
- gallery(5) an interesting eigenvalue problem

**Description**

\[
[A, B, C, \ldots] = \text{gallery}('tmfun', P1, P2, \ldots)
\]

returns the test matrices specified by string *tmfun*. *tmfun* is the name of a matrix family selected from the table below. *P1, P2, \ldots* are input parameters required by the individual matrix family. The number of optional parameters *P1, P2, \ldots* 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.
cauchy matrix

C = gallery('cauchy', x, y) returns an n-by-n matrix, C(i,j) = 1/(x(i)+y(j)). Arguments x and y are vectors of length n. If you pass in scalars for x and y, they are interpreted as vectors 1:x and 1:y.

C = gallery('cauchy', x) returns the same as above with y = x. That is, the command returns C(i,j) = 1/(x(i)+x(j)).

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant det(C) is nonzero if x and y both have distinct elements. C is totally positive if 0 < x(1) < ... < x(n) and 0 < y(1) < ... < y(n).
chebspec—Chebyshev spectral differentiation matrix

\[
C = \text{gallery('chebspec',n,switch)}
\]
returns a Chebyshev spectral differentiation matrix of order \( n \). Argument \texttt{switch} is a variable that determines the character of the output matrix. By default, \texttt{switch} = \( 0 \).

For \texttt{switch} = \( 0 \) ("no boundary conditions"), \( C \) is nilpotent (\( C^n = 0 \)) and has the null vector \( \text{ones}(n,1) \). The matrix \( C \) is similar to a Jordan block of size \( n \) with eigenvalue zero.

For \texttt{switch} = \( 1 \), \( C \) is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix \( V \) of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand—Vandermonde-like matrix for the Chebyshev polynomials

\[
C = \text{gallery('chebvand',p)}
\]
produces the (primal) Chebyshev Vandermonde matrix based on the vector of points \( p \), which define where the Chebyshev polynomial is calculated.

\[
C = \text{gallery('chebvand',m,p)}
\]
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

\[
A = \text{gallery('chow',n,al,ta)}
\]
returns \( A \) such that \( A = H(\texttt{al}) + \texttt{ta}*\text{eye}(n) \), where \( H_{i,j}(\alpha) = \alpha^{i-j} \). Argument \( n \) is the order of the Chow matrix, while \texttt{al} and \texttt{ta} are scalars with default values 1 and 0, respectively.

\( H(\alpha) \) has \( p = \lceil n/2 \rceil \) eigenvalues that are equal to zero. The rest of the eigenvalues are equal to \( 4*\alpha*cos(k*pi/(n+2))^2 \), \( k=1:n-p \).
circul—Circulant matrix

\[ C = \text{gallery}'(\text{circul}', v) \] 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 \( C = \text{gallery}'(\text{circul}', 1:v) \).

The eigensystem of \( C \) (n-by-n) is known explicitly: If \( t \) is an \( n \)th root of unity, then the inner product of \( v \) with \( w = [1 \ t \ t^2 \ldots \ t^{n-1}] \) is an eigenvalue of \( C \) and \( w \) \( n-1:1 \) is an eigenvector.

clement—Tridiagonal matrix with zero diagonal entries

\[ A = \text{gallery}'(\text{clement}', n, \text{sym}) \] 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 \( n-1, n-3, n-5, \ldots \), as well as (for odd \( n \)) a final eigenvalue of 1 or 0.

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

compar—Comparison matrices

\[ A = \text{gallery}'(\text{compar}', A, 1) \] 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 \( \text{compar}(A, 1) \) is too.

\[ \text{gallery}'(\text{compar}', A) = \text{diag}(B) - \text{tril}(B, -1) - \text{triu}(B, 1) \], where \( B = \text{abs}(A) \).

\( \text{compar}(A) \) is often denoted by \( M(A) \) in the literature.

\[ \text{gallery}'(\text{compar}', A, 0) \] is the same as \( \text{compar}(A) \).
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:

\[
\begin{array}{ccc}
k & 1 & 2 & 3 & 4 \\
4\text{-by-4} & 3\text{-by-3} & \text{arbitrary} & n \geq 4 \\
\text{LINPACK (rcond)} & \text{LINPACK (rcond)} & \text{LINPACK (rcond) (independent of \( \theta \))} & \text{SONEST (Higham 1988) (default)}
\end{array}
\]

If \( n \) is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order \( n \).

cycol—Matrix whose columns repeat cyclically

A = gallery('cycol', [m n], k) returns an \( m \)-by-\( n \) matrix with cyclically repeating columns, where one “cycle” consists of \text{randn}(m,k). Thus, the rank of matrix \( A \) cannot exceed \( k \). \( k \) must be a scalar.

Argument \( k \) defaults to \text{round}(n/4), and need not evenly divide \( n \).

A = gallery('cycol', n, k), where \( n \) is a scalar, is the same as gallery('cycol', [n n], k).

dorr—Diagonally dominant, ill-conditioned, tridiagonal matrix

\([c, d, e] = \text{gallery('dorr', n, theta})\) 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 \( \| \text{inv}(A) \|_F \) is relatively large, although not necessarily maximal. An anti-Hadamard matrix A is a matrix with elements 0 or 1 for which \( \mu(A) \) is maximal.

n and k must both be scalars. Argument k determines the character of the output matrix:

- **k = 1** Default. A is Toeplitz, with \( \text{abs}(\text{det}(A)) = 1 \), and \( \mu(A) > c(1.75)^n \), where c is a constant. The inverse of A has integer entries.
- **k = 2** A is upper triangular and Toeplitz. The inverse of A has integer entries.
- **k = 3** A has maximal determinant among lower Hessenberg (0,1) matrices. \( \text{det}(A) = \text{the } n\text{th Fibonacci number} \). 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 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, alpha, lambda) returns the n-by-n matrix equal to the Jordan block with eigenvalue lambda, excepting that A(n, 1) = alpha. The default values of scalars alpha and lambda are sqrt(eps) and 0, respectively.

The characteristic polynomial of A is given by:

$$\det(A - \lambda I) = (\lambda - t)^n - alpha \cdot (-1)^n.$$ 

frank—Matrix with ill-conditioned eigenvalues

F = gallery('frank', n, k) returns the Frank matrix of order n. It is upper Hessenberg with determinant 1. If k = 1, the elements are reflected about the anti-diagonal (1, n) — (n, 1). The eigenvalues of F may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if n is odd, 1 is an eigenvalue. F has floor(n/2) ill-conditioned eigenvalues—the smaller ones.

gearmat—Gear matrix

A = gallery('gearmat', n, i, j) returns the n-by-n matrix with ones on the sub- and super-diagonals, sign(i) in the (1, abs(i)) position, sign(j) in the (n, n+1-abs(j)) position, and zeros everywhere else. Arguments i and j default to n and -n, respectively.

Matrix A is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form $2 \cos(a)$ and the eigenvectors are of the form $\{\sin(wa), \sin(wa+2a), \ldots, \sin(wn-a)\}$, where a and w are given in Gear, C. W., “A Simple Set of Test Matrices for Eigenvalue Programs”, Math. Comp., Vol. 23 (1969), pp. 119–125.

grcar—Toeplitz matrix with sensitive eigenvalues

A = gallery('grcar', n, k) returns an n-by-n Toeplitz matrix with -1s on the subdiagonal, 1s on the diagonal, and k superdiagonals of 1s. The default is k = 3. The eigenvalues are sensitive.
hanowa—Matrix whose eigenvalues lie on a vertical line in the complex plane

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

\[
\begin{bmatrix}
d \cdot \text{eye}(m) & -\text{diag}(1:m) \\
\text{diag}(1:m) & d \cdot \text{eye}(m)
\end{bmatrix}
\]

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

A Householder matrix \( H \) satisfies the relationship

\[ H \cdot x = \text{sign}(x(1)) \cdot \| x \| \cdot e_1 \]

where \( e_1 \) is the first column of \( \text{eye}(n, n) \). Note that if x is complex, then \( \text{sign}(x) = \exp(i \cdot \text{arg}(x)) \) (which equals \( x / \| x \| \) 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 \( \text{ones}(n, 1) \cdot x' \) and whose strict upper triangle agrees with that of \( [1 y] \cdot \text{ones}(1, n) \).

The matrix is nonsingular if \( x(1) \neq 0 \) and \( x(i+1) = y(i) \) for all i, and its inverse is an upper Hessenberg matrix. Argument y defaults to \(-x(1:n-1)\).

If x is a scalar, \( \text{invhess}(x) \) is the same as \( \text{invhess}(1:x) \).


\textbf{invol— Involutory matrix}

\[ A = \text{gallery}(\text{'invol'}, n) \] returns an \( n \times n \) involutory \( A^2 = \text{eye}(n) \) and ill-conditioned matrix. It is a diagonally scaled version of \( \text{hilb}(n) \).

\[ B = (\text{eye}(n) - A) / 2 \] and \[ B = (\text{eye}(n) + A) / 2 \] are idempotent \( (B^2 = B) \).

\textbf{ipjfact— Hankel matrix with factorial elements}

\[ [A, d] = \text{gallery}(\text{'ipjfact'}, n, k) \] returns \( A \), an \( n \times n \) 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.

\textbf{jordbloc— Jordan block}

\[ A = \text{gallery}(\text{'jordbloc'}, n, \text{lambda}) \] returns the \( n \times n \) Jordan block with eigenvalue \( \text{lambda} \). The default value for \( \text{lambda} \) is 1.

\textbf{kahan— Upper trapezoidal matrix}

\[ A = \text{gallery}(\text{'kahan'}, n, \text{theta}, \text{pert}) \] 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 \( n(1) \times n(2) \); otherwise, \( A \) is \( n \times n \). The useful range of \( \text{theta} \) is \( 0 < \text{theta} < \pi \), 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 \( \text{pert} \times \text{eps} \times \text{diag}([n:-1:1]) \). The default \( \text{pert} \) is 25, which ensures no interchanges for \( \text{gallery}(\text{'kahan'}, n) \) up to at least \( n = 90 \) in IEEE arithmetic.

\textbf{kms— Kac-Murdock-Szego Toeplitz matrix}

\[ A = \text{gallery}(\text{'kms'}, n, \text{rho}) \] returns the \( n \times n \) Kac-Murdock-Szego Toeplitz matrix such that \( A(i,j) = \text{rho}^{|i-j|} \), for real \( \text{rho} \).

For complex \( \text{rho} \), the same formula holds except that elements below the diagonal are conjugated. \( \text{rho} \) defaults to 0.5.
The KMS matrix \( \Lambda \) has these properties:

- An LDL' factorization with 
  \[ L = \text{inv}(\text{triu}(n, -\rho, 1))', \]
  and 
  \[ D(i, i) = (1 - \text{abs}(\rho)^2) \times \text{eye}(n), \]
  except \( D(1, 1) = 1 \).

- Positive definite if and only if \( 0 < \text{abs}(\rho) < 1 \).

- The inverse \( \text{inv}(\Lambda) \) is tridiagonal.

**krylov— Krylov matrix**

\[ B = \text{gallery}(\text{'krylov'}, \Lambda, x, j) \]
returns the Krylov matrix

\[ [x, \ A x, \ A^2 x, \ldots, \ A^{(j-1)} x] \]
where \( \Lambda \) is an \( n \)-by-\( n \) matrix and \( x \) is a length \( n \) vector. The defaults are 
\( x = \text{ones}(n, 1) \), and \( j = n \).

\[ B = \text{gallery}(\text{'krylov'}, n) \]
is the same as \( \text{gallery}(\text{'krylov'}, \text{randn}(n)) \).

**lauchli— Rectangular matrix**

\[ A = \text{gallery}(\text{'lauchli'}, n, m) \]
returns the \( (n+1) \)-by-\( n \) matrix

\[ \left[ \text{ones}(1, n); \ m \times \text{eye}(n) \right] \]
The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming \( A' \times A \). Argument \( m \) defaults to \( \sqrt{\text{eps}} \).

**lehmer— Symmetric positive definite matrix**

\[ A = \text{gallery}(\text{'lehmer'}, n) \]
returns the symmetric positive definite \( n \)-by-\( n \) matrix such that \( A(i, j) = i \times j \) for \( j \geq i \).

The Lehmer matrix \( \Lambda \) has these properties:

- \( \Lambda \) is totally nonnegative.
- The inverse \( \text{inv}(\Lambda) \) is tridiagonal and explicitly known.
- The order \( n \) \( \leq \text{cond}(\Lambda) \leq 4 \times n \times n \).
gallery

lesp— Tridiagonal matrix with real, sensitive eigenvalues

\[ A = \text{gallery}('lesp', n) \]
returns an \( n \times n \) matrix whose eigenvalues are real
and smoothly distributed in the interval approximately \([-2^n \times 3.5, -4.5]\).

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

lotkin— Lotkin matrix

\[ A = \text{gallery}('lotkin', n) \]
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

\[ A = \text{gallery}('minij', n) \]
returns the \( n \times n \) symmetric positive definite
matrix with \( A(i,j) = \min(i,j) \).

The \text{minij} matrix has these properties:

- The inverse \( \text{inv}(A) \) is tridiagonal and equal to \(-1\) times the second difference
matrix, except its \((n, n)\) element is 1.
- Givens' matrix, \( 2*A\)-ones\((\text{size}(A)) \), has tridiagonal inverse and eigenvalues
\( 0.5 \times \sec((2r-1) \times \pi / (4n))^2 \), where \( r=1:n \).
- \((n+1)\)-ones\((\text{size}(A)) \)-A has elements that are \( \max(i, j) \) and a tridiagonal
inverse.

moler— Symmetric positive definite matrix

\[ A = \text{gallery}('moler', n, \alpha) \]
returns the symmetric positive definite
\( n \times n \) matrix \( U' \times U \), 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 kth type of matrix of order n, where
k > 0 selects exactly orthogonal matrices, and k < 0 selects diagonal scalings
of orthogonal matrices. Available types are:

k = 1  Q(i, j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))
Symmetric eigenvector matrix for second difference matrix. This
is the default.

k = 2  Q(i, j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))
Symmetric.

k = 3  Q(r, s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)
Unitary, the Fourier matrix. Q^4 is the identity. This is
essentially the same matrix as fft(eye(n))/sqrt(n)!

k = 4  Helmert matrix: a permutation of a lower Hessenberg matrix,
whose first row is ones(1:n)/sqrt(n).

k = 5  Q(i, j) = sin(2*pi*(i-1)*(j-1)/n) +
cos(2*pi*(i-1)*(j-1)/n)
Symmetric matrix arising in the Hartley transform.

k = -1  Q(i, j) = cos((i-1)*(j-1)*pi / (n-1))
Chebyshev Vandermonde-like matrix, based on extrema of
T(n-1).

k = -2  Q(i, j) = cos((i-1)*(j-1/2)*pi / n))
Chebyshev Vandermonde-like matrix, based on zeros of T(n).
parter—Toeplitz matrix with singular values near pi

\[ C = \text{gallery}(\text{'parter'}, n) \]
returns the matrix \( C \) such that
\[ C(i,j) = \frac{1}{i-j+0.5}. \]

\( C \) is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of \( C \) are very close to \( \pi \).

pei—Pei matrix

\[ A = \text{gallery}(\text{'pei'}, n, \alpha) \]
where \( \alpha \) is a scalar, returns the symmetric matrix \( \alpha \times \text{eye}(n) + \text{ones}(n) \). 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)

\[ A = \text{gallery}(\text{'poisson'}, n) \]
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

\[ A = \text{gallery}(\text{'prolate'}, n, w) \]
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**

\[ H = \text{gallery('randhess', n)} \] returns an \( n \)-by-\( n \) real, random, orthogonal upper Hessenberg matrix.

\[ H = \text{gallery('randhess', x)} \] 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**

\[ A = \text{gallery('rando', n, k)} \] 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**

\[ A = \text{gallery('randsvd', n, kappa, mode, kl, ku)} \] 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)
- 4 Arithmetically distributed singular values
Condition number kappa defaults to $\sqrt{1/\text{eps}}$. 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 $\text{mode}$. Arguments $k_l$ and $k_u$, if present, are ignored.

**redheff— Redheffer’s matrix of 1s and 0s**

$A = \text{gallery('redheff', n)}$ 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:
- $(n - \text{floor}(\log_2(n))) - 1$ eigenvalues equal to 1
- A real eigenvalue (the spectral radius) approximately $\sqrt{n}$
- A negative eigenvalue approximately $-\sqrt{n}$
- The remaining eigenvalues are provably “small.”
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{(1/2 + \epsilon)})$ for every $\epsilon > 0$.

Barrett and Jarvis conjecture that “the small eigenvalues all lie inside the unit circle $\text{abs}(Z) = 1$,” 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**

$A = \text{gallery('riemann', n)}$ 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 $\epsilon > 0$. 

One large singular value

Random singular values with uniformly distributed logarithm

< 0 If mode is -1, -2, -3, -4, or -5, then randsvd treats mode as $\text{abs}(\text{mode})$, 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.

Random singular values with uniformly distributed logarithm
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 \( \text{abs}(e(i)) \leq m/\sqrt{m} \) 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**

\( A = \text{gallery('ris', n)} \) 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**

\( A = \text{rosser} \) 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**

\( A = \text{gallery('smoke', n)} \) 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.
gallery

\[ A = \text{gallery('smoke', n, 1)} \] returns the same except that element \( A(n, 1) \) is zero.

The eigenvalues of \( \text{smoke}(n, 1) \) are the \( n \)th roots of unity; those of \( \text{smoke}(n) \) are the \( n \)th roots of unity times \( 2^{\frac{1}{n}} \).

toeppd—Symmetric positive definite Toeplitz matrix

\[ A = \text{gallery('toeppd', n, m, w, \theta)} \] returns an \( n \times n \) 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,

\[ T = \sum_{k=1}^{m} w(k) \cdot T(\theta(k)) \]

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

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

toeppen—Pentadiagonal Toeplitz matrix (sparse)

\[ P = \text{gallery('toeppen', n, a, b, c, d, e)} \] returns the \( n \times n \) sparse, pentadiagonal Toeplitz matrix with the diagonals:

\[ P(3, 1) = a, \ P(2, 1) = b, \ P(1, 1) = c, \ P(1, 2) = d, \ \text{and} \ P(1, 3) = e, \]

where \( a, b, c, d, \) and \( e \) are scalars.

By default, \((a, b, c, d, e) \equiv (1, -10, 0, 10, 1)\), yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment

\[ 2 \cos(2t) + 20i \sin(n(t)). \]

tridiag—Tridiagonal matrix (sparse)

\[ A = \text{gallery('tridiag', c, d, e)} \] returns the tridiagonal matrix with subdiagonal \( c \), diagonal \( d \), and superdiagonal \( e \). Vectors \( c \) and \( e \) must have length \( d - 1 \).

\[ A = \text{gallery('tridiag', n, c, d, e)} \] returns the Toeplitz tridiagonal matrix of order \( n \) with subdiagonal elements \( c \), diagonal elements \( d \), and superdiagonal elements \( e \). This matrix has eigenvalues

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

where \( k = 1: n \). (see [1].)
A = gallery(’tridiag’, n) is the same as
A = gallery(’tridiag’, n, -1, 2, -1), which is a symmetric positive definite
M-matrix (the negative of the second difference matrix).

triw—Upper triangular matrix discussed by Wilkinson and others

A = gallery(’triw’, n, alpha, k) returns the upper triangular matrix with
ones on the diagonal and alpha on the first k >= 0 superdiagonals.

Order n may be a 2-vector, in which case the matrix is n(1) -by- n(2) and upper
trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices, J. Reine
Angew. Math., 1954] shows that

\[ \text{cond}(\text{gallery}(’\text{triw}', n, 2)) = \cot\left(\frac{\pi}{4n}\right)^2, \]

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 jth 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 defi-
nite 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
\[ 0.25 \leq \text{eig}(\text{inv}(D) \cdot A) \leq 4.5 \]
where \( D = \text{diag}(\text{diag}(A)) \) for any positive integers \( nx \) and \( ny \) and any densities \( \rho(x, y) \).

**wilk—Various matrices devised or discussed by Wilkinson**

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

<table>
<thead>
<tr>
<th>( n )</th>
<th><strong>MATLAB Code</strong></th>
<th><strong>Result</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 3 )</td>
<td>([A, b] = \text{gallery}('wilk', 3))</td>
<td>Upper triangular system ( Ux=b ) illustrating inaccurate solution.</td>
</tr>
<tr>
<td>( n = 4 )</td>
<td>([A, b] = \text{gallery}('wilk', 4))</td>
<td>Lower triangular system ( Lx=b ), ill-conditioned.</td>
</tr>
<tr>
<td>( n = 5 )</td>
<td>( A = \text{gallery}('wilk', 5))</td>
<td>( \text{hilb}(6)(1:5, 2:6) \times 1.8144 ). A symmetric positive definite matrix.</td>
</tr>
<tr>
<td>( n = 21 )</td>
<td>( A = \text{gallery}('wilk', 21))</td>
<td>( W21+ ), tridiagonal matrix. Eigenvalue problem.</td>
</tr>
</tbody>
</table>
See Also

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hadamard</td>
<td>Hadamard matrix</td>
</tr>
<tr>
<td>hilb</td>
<td>Hilbert matrix</td>
</tr>
<tr>
<td>invhilb</td>
<td>Inverse of the Hilbert matrix</td>
</tr>
<tr>
<td>magic</td>
<td>Magic square</td>
</tr>
<tr>
<td>wilkinson</td>
<td>Wilkinson's eigenvalue test matrix</td>
</tr>
</tbody>
</table>

References

The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Additional detail on these matrices is documented in The Test Matrix Toolbox for MATLAB (Version 3.0) by N. J. Higham, September, 1995. To obtain this report in pdf format, enter the doc command at the MATLAB prompt and select the item Related Papers > Test Matrix Toolbox under the Full Documentation Set entry on the Help Desk main screen. This report is also available via anonymous ftp from The MathWorks at /pub/contrib/linalg/testmatrix/testmatrix.ps or World Wide Web (ftp://ftp.na.man.ac.uk/pub/narep or http://www.ma.man.ac.uk/MCCM/MCCM.html). Further background may be found in the book Accuracy and Stability of Numerical Algorithms, Nicholas J. Higham, SIAM, 1996.
gamma, gammainc, gammaln

**Purpose**
Gamma functions

**Syntax**
- \( Y = \text{gamma}(A) \)  
  Gamma function
- \( Y = \text{gammainc}(X,A) \)  
  Incomplete gamma function
- \( Y = \text{gammaln}(A) \)  
  Logarithm of gamma function

**Definition**
The gamma function is defined by the integral:

\[
\Gamma(a) = \int_0^\infty e^{-t}t^{a-1}dt
\]

The gamma function interpolates the factorial function. For integer \( n \):

\[
\text{gamma}(n+1) = n! = \text{prod}(1:n)
\]

The incomplete gamma function is:

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

**Description**
- \( Y = \text{gamma}(A) \) returns the gamma function at the elements of \( A \). \( A \) must be real.
- \( Y = \text{gammainc}(X,A) \) returns the incomplete gamma function of corresponding elements of \( X \) and \( A \). Arguments \( X \) and \( A \) must be real and the same size (or either can be scalar).
- \( Y = \text{gammaln}(A) \) returns the logarithm of the gamma function, \( \text{gammaln}(A) = \log(\text{gamma}(A)) \). The \( \text{gammaln} \) command avoids the underflow and overflow that may occur if it is computed directly using \( \log(\text{gamma}(A)) \).

**Algorithm**
The computations of \( \text{gamma} \) and \( \text{gammaln} \) are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending on

2-326
upon the value of $A$. Computation of the incomplete gamma function is based on the algorithm in [2].
References


Purpose
Greatest common divisor

Syntax
G = gcd(A,B)
[ G, C, D ] = gcd(A,B)

Description
G = gcd(A,B) returns an array containing the greatest common divisors of the corresponding elements of integer arrays A and B. By convention, gcd(0, 0) returns a value of 0; all other inputs return positive integers for G.

[ G, C, D ] = gcd(A,B) returns both the greatest common divisor array G, and the arrays C and D, which satisfy the equation: A(i).*C(i) + B(i).*D(i) = G(i). These are useful for solving Diophantine equations and computing elementary Hermite transformations.

Examples
The first example involves elementary Hermite transformations.
For any two integers a and b there is a 2-by-2 matrix E with integer entries and determinant = 1 (a unimodular matrix) such that:
E * [ a; b ] = [ g, 0 ],
where g is the greatest common divisor of a and b as returned by the command
[ g, c, d ] = gcd(a,b).
The matrix E equals:
c       d
-b/g   a/g

In the case where a = 2 and b = 4:
[ g, c, d ] = gcd(2, 4)
g =
   2
c =
   1
d =
   0
So that:

\[
\begin{array}{cc}
1 & 0 \\
-2 & 1 \\
\end{array}
\]

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

\[
\lfloor g, c, d \rfloor = \gcd(30, 56) \\
g = 2 \\
c = -13 \\
d = 7
\]

By the definition, for scalars \(c\) and \(d\):

\[
30(-13) + 56(7) = 2,
\]

Multiplying through by 8/2:

\[
30(-13*4) + 56(7*4) = 8
\]

Comparing this to the original equation, a solution can be read by inspection:

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

**See Also**  
lcm  
Least common multiple

**References**  
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Macintosh gestalt function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>gestaltbits = gestalt('selector')</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>gestaltbits = gestalt('selector')</code> passes the four-character string <code>selector</code> to the Macintosh Operating System function <code>gestalt</code>. For details about <code>gestalt</code>, refer to Chapter 1 of Inside Macintosh: Operating System Utilities. The result, a 32-bit integer, is stored bitwise in <code>gestaltbits</code>. Thus, the least significant bit of the result is <code>gestaltbits(32)</code>, while the most significant bit is <code>gestaltbits(1)</code>.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>After executing:</td>
</tr>
<tr>
<td></td>
<td><code>gestaltbits = gestalt('sysa')</code></td>
</tr>
<tr>
<td></td>
<td><code>gestaltbits(32)</code> will be 1 if run from a 680x0-based Macintosh, while <code>gestaltbits(31)</code> will be 1 if run from a PowerPC-based Macintosh.</td>
</tr>
</tbody>
</table>
getfield

**Purpose**
Get field of structure array

**Syntax**

```
f = getfield(s,'field')
f = getfield(s,{i,j},'field',{k})
```

**Description**

\( f = \text{getfield}(s,'field') \), where \( s \) is a 1-by-1 structure, returns the contents of the specified field. This is equivalent to the syntax \( f = s.field \).

\( f = \text{getfield}(s,{i,j},'field',{k}) \) returns the contents of the specified field. This is equivalent to the syntax \( f = s(i,j).field(k) \). All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to \{i,j\} and \{k\} above). Pass field references as strings.

**Examples**
Given the structure:

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

Then the command \( f = \text{getfield}(mystr,{2,1},'name') \) yields

```
f =

gertrude
```

To list the contents of all name (or other) fields, embed `getfield` in a loop:

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

```
name =

'alice'    'gertrude'
```

**See Also**

- `fields` — Field names of a structure
- `setfield` — Set field of structure array
**Purpose**
Define global variables

**Syntax**
global X Y Z

**Description**
global X Y Z defines X, Y, and Z as global in scope.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace and nonfunction scripts. However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global. If the global variable does not exist the first time you issue the global statement, it is 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:

• in the current workspace, a variable with the same name exists.
• in an M-file, it has been referenced previously.

**Remarks**
Use clear global variable to clear a global variable from the global workspace. Use clear variable to clear the global link from the current workspace without affecting the value of the global.

To use a global within a callback, declare the global, use it, then clear the global link from the workspace. This avoids declaring the global after it has been referenced. For example:

```matlab
disp(MY_GLOBAL), MY_GLOBAL = MY_GLOBAL+1, clear MY_GLOBAL,...
' string','count')
```

**Examples**
Here is the code for the functions tic and toc (some comments abridged), which manipulate a stopwatch-like timer. The global variable TICTOC is shared.
by the two functions, but it is invisible in the base workspace or in any other functions that do not declare it.

```matlab
function tic
    % TIC Start a stopwatch timer.
    % TIC; any stuff; TOC
    % prints the time required.
    % See also: TOC, CLOCK.
    global TICTOC
    TICTOC = clock;
end

function t = toc
    % TOC Read the stopwatch timer.
    % TOC prints the elapsed time since TIC was used.
    % t = TOC; saves elapsed time in t, does not print.
    % See also: TIC, ETIME.
    global TICTOC
    if nargout < 1
        elapsed_time = etime(clock, TICTOC)
    else
        t = etime(clock, TICTOC);
    end
end
```

See Also

clear, isglobal, who
Purpose

Generalized Minimum Residual method (with restarts)

Syntax

$x = \text{gmres}(A, b, \text{restart})$

$\text{gmres}(A, b, \text{restart}, \text{tol})$

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit})$

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M)$

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2)$

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

$x = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

$[x, \text{flag}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

$[x, \text{flag}, \text{relres}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

$[x, \text{flag}, \text{relres}, \text{iter}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

$[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0)$

Description

$x = \text{gmres}(A, b, \text{restart})$ 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-$n$. $\text{gmres}$ will start iterating from an initial estimate that by default is an all zero vector of length $n$. $\text{gmres}$ will restart itself every $\text{restart}$ 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 $n / \text{restart}$ and $10$. No preconditioning is used.

$\text{gmres}(A, b, \text{restart}, \text{tol})$ specifies the tolerance of the method, $\text{tol}$.

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit})$ additionally specifies the maximum number of iterations, $\text{maxit}$.

$\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M)$ and $\text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2)$ use left preconditioner $M$ or $M = M_1 * M_2$ and effectively solve the system $i \text{nv}(M_1 * \text{A}^* x) = i \text{nv}(M_1 * b)$ for $x$. If $M_1$ or $M_2$ 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 $\text{gmres}$, it is wise to factor precondi-


GMRES

torers into their LU factors first. For example, replace

\[ \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M) \]

with:

\[ [M1, M2] = \text{lu}(M); \]
\[ \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M1, M2). \]

\[ \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M1, M2, x0) \]

specifies the first initial estimate \( x_0 \). If \( x_0 \) is given as the empty matrix ([]), the default all zero vector is used.

\[ x = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M1, M2, x0) \]

returns a solution \( x \). If \text{gmres} converged, a message to that effect is displayed. If \text{gmres} failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual

\[ \frac{\|b - A x\|}{\|b\|} \]

and the iteration number at which the method stopped or failed.

\[ [x, \text{flag}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M1, M2, x0) \]

returns a solution \( x \) and a flag which describes the convergence of \text{gmres}:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{gmres} converged to the desired tolerance \text{tol} within \text{maxit} iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>\text{gmres} iterated \text{maxit} times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>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 ) (backslash).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
</tbody>
</table>

Whenever \text{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 \text{flag} output is specified.
\[ [x, \text{flag}, \text{relres}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0) \] also returns the relative residual \( \frac{\|b - A*x\|}{\|b\|} \). If flag is 0, then \( \text{relres} \leq \text{tol} \).

\[ [x, \text{flag}, \text{relres}, \text{iter}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0) \] also returns both the outer and inner iteration numbers at which \( x \) was computed. The outer iteration number \( \text{iter}(1) \) is an integer between 0 and \( \text{maxit} \). The inner iteration number \( \text{iter}(2) \) is an integer between 0 and \( \text{restart} \).

\[ [x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{gmres}(A, b, \text{restart}, \text{tol}, \text{maxit}, M_1, M_2, x_0) \] also returns a vector of the residual norms at each inner iteration, starting from \( \text{resvec}(1) = \|b - A*x_0\| \). If flag is 0 and \( \text{iter} = [i j] \), \( \text{resvec} \) is of length \( (i-1) \times \text{restart} + j + 1 \) and \( \text{resvec}(\text{end}) \leq \text{tol} \times \|b\| \).

**Examples**

```matlab
data west0479
A = west0479
b = sum(A,2)
[x, flag] = gmres(A,b,5)
flag is 1 since \( \text{gmres}(5) \) 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 \( U_1 \) has a zero on its diagonal so \( \text{gmres}(5) \) fails in the first iteration when it tries to solve a system such as \( U_1*y = r \) for \( y \) with backslash.

[L2, U2] = luinc(A,1e-6);
tol = 1e-15;
[x4, flag4, relres4, iter4, resvec4] = gmres(A,b,4,tol,5,L2,U2);
[x6, flag6, relres6, iter6, resvec6] = gmres(A,b,6,tol,3,L2,U2);
[x8, flag8, relres8, iter8, resvec8] = gmres(A,b,8,tol,3,L2,U2);
flag4, flag6, and flag8 are all 0 since \( \text{gmres} \) converged when restarted at iterations 4, 6, and 8 while preconditioned by the incomplete LU factorization with a drop tolerance of \( 1e^{-6} \). This is verified by the plots of outer iteration number against relative residual. A combined plot of all three clearly shows the restarting at iterations 4 and 6. The total number of iterations computed may
be more for lower values of restart, but the number of length n vectors stored is fewer, and the amount of work done in the method decreases proportionally.

See Also

- bi cg: BiConjugate Gradients method
- bicgstab: BiConjugate Gradients Stabilized method
- cgs: Conjugate Gradients Squared method
- luinc: Incomplete LU matrix factorizations
- pcg: Preconditioned Conjugate Gradients method
- qmr: Quasi-Minimal Residual method
- \: Matrix left division

References


Purpose
Numerical gradient

Syntax
FX = gradient(F)
[FX, FY] = gradient(F)
[FX, FY, Fz, ...] = gradient(F)
[... ] = gradient(F, h)
[... ] = gradient(F, h1, h2, ...)

Definition
The gradient of a function of two variables, \( F(x,y) \), is defined as:
\[
\nabla F = \frac{\partial F}{\partial x} \hat{i} + \frac{\partial F}{\partial y} \hat{j}
\]
and can be thought of as a collection of vectors pointing in the direction of increasing values of \( F \). In MATLAB, numerical gradients (differences) can be computed for functions with any number of variables. For a function of \( N \) variables, \( F(x,y,z,...) \),
\[
\nabla F = \frac{\partial F}{\partial x} \hat{i} + \frac{\partial F}{\partial y} \hat{j} + \frac{\partial F}{\partial z} \hat{k} + ...
\]

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

[FX, FY] = gradient(F) where \( F \) is a matrix returns the x and y components of the two-dimensional numerical gradient. FX corresponds to \( \frac{\partial F}{\partial x} \), the differences in the x (column) direction. FY corresponds to \( \frac{\partial F}{\partial y} \), the differences in the y (row) direction. The spacing between points in each direction is assumed to be one.

[FX, FY, FZ, ...] = gradient(F) where \( F \) has \( N \) dimensions returns the \( N \) components of the gradient of \( F \).

There are two ways to control the spacing between values in \( F \):

A single spacing value, \( h \), specifies the spacing between points in every direction.

\( N \) spacing values \( (h1, h2, ...) \) specify the spacing for each dimension of \( F \). Scalar spacing parameters specify a constant spacing for each dimension. Vector
parameters specify the coordinates of the values along corresponding dimensions of $F$. In this case, the length of the vector must match the size of the corresponding dimension.

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

\[
\text{[...]} = \text{gradient}(F, h_1, h_2, \ldots) \quad \text{with N spacing parameters specifies the spacing for each dimension of } F.
\]

**Examples**

The statements

```matlab
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

![Graph](image-url)
Given,

\[ F(\cdot,\cdot,1) = \text{magic}(3); \quad F(\cdot,\cdot,2) = \text{pascal}(3); \]
\[ \text{gradient}(F) \text{ takes } dx = dy = dz = 1. \]
\[ [PX, PY, PZ] = \text{gradient}(F, 0.2, 0.1, 0.2) \text{ takes } dx = 0.2, \quad dy = 0.1, \quad \text{and} \]
\[ dz = 0.2. \]

**See Also**

del2  Discrete Laplacian
diff   Differences and approximate derivatives
Purpose
Data gridding

Syntax
\[
ZI = \text{griddata} (x, y, z, XI, YI)
\]
\[
[XI, YI, ZI] = \text{griddata} (x, y, z, xi, yi)
\]
\[
[...] = \text{griddata} (...,
\text{method})
\]

Description
\(ZI = \text{griddata} (x, y, z, XI, YI)\) fits a surface of the form \(z = f(x, y)\) to the data in the (usually) nonuniformly spaced vectors \((x, y, z)\). \text{griddata} interpolates this surface at the points specified by \((XI, YI)\) to produce \(ZI\). The surface always passes through the data points. \(XI\) and \(YI\) usually form a uniform grid (as produced by \text{meshgrid}).

\(XI\) can be a row vector, in which case it specifies a matrix with constant columns. Similarly, \(YI\) can be a column vector, and it specifies a matrix with constant rows.

\([XI, YI, ZI] = \text{griddata} (x, y, z, xi, yi)\) returns the interpolated matrix \(ZI\) as above, and also returns the matrices \(XI\) and \(YI\) formed from row vector \(xi\) and column vector \(yi\). These latter are the same as the matrices returned by \text{meshgrid}.

\([...] = \text{griddata} (...,
\text{method})\) uses the specified interpolation method:

- **'linear'**: Triangle-based linear interpolation (default)
- **'cubic'**: Triangle-based cubic interpolation
- **'nearest'**: Nearest neighbor interpolation
- **'v4'**: MATLAB 4 \text{griddata} method

The \text{method} defines the type of surface fit to the data. The 'cubic' and 'v4' methods produce smooth surfaces while 'linear' and 'nearest' have discontinuities in the first and zero'th derivatives, respectively. All the methods except 'v4' are based on a Delaunay triangulation of the data.

Remarks
\(XI\) and \(YI\) can be matrices, in which case \text{griddata} returns the values for the corresponding points \((XI(i,j), YI(i,j))\). Alternatively, you can pass in the row and column vectors \(xi\) and \(yi\), respectively. In this case, \text{griddata} inter-
prets these vectors as if they were matrices produced by the command `meshgrid(xi, yi)`.

**Algorithm**

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

**Examples**

Sample a function at 100 random points between ±2. 0:

```matlab
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:

```matlab
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:

```matlab
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```

![Graph of gridded data](image)

---

2-343
griddata

See Also  
delaunay, interp2, meshgrid

References  

Purpose

Hadamard matrix

Syntax

\[ H = \text{hadamard}(n) \]

Description

\[ H = \text{hadamard}(n) \]  returns the Hadamard matrix of order \( n \).

Definition

Hadamard matrices are matrices of 1's and -1's whose columns are orthogonal,

\[ H' * H = n * I \]

where \( [n \ n] = \text{size}(H) \) 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 \( n/12 \), or \( n/20 \) is a power of 2.

Examples

The command \( \text{hadamard}(4) \) produces the 4-by-4 matrix:

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1
\end{bmatrix}
\]

See Also

companion  Companion matrix
hankel       Hankel matrix
toeplitz     Toeplitz matrix

References


**Purpose**

Hankel matrix

**Syntax**

\[
H = \text{hankel} \left( c \right) \\
H = \text{hankel} \left( c, r \right)
\]

**Description**

\[H = \text{hankel} \left( c \right)\] returns the square Hankel matrix whose first column is \(c\) and whose elements are zero below the first anti-diagonal.

\[H = \text{hankel} \left( c, r \right)\] returns a Hankel matrix whose first column is \(c\) and whose last row is \(r\). If the last element of \(c\) differs from the first element of \(r\), the last element of \(c\) prevails.

**Definition**

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements \(h(i, j) = p(i+j-1)\), where vector \(p = \left[ c \right.\left. r \left( 2: \text{end} \right) \right]\) completely determines the Hankel matrix.

**Examples**

A Hankel matrix with anti-diagonal disagreement is

\[
c = 1:3; \ r = 7:10;
\]
\[
h = \text{hankel} \left( c, r \right)
\]
\[
h = \\
1 2 3 8 \\
2 3 8 9 \\
3 8 9 10
\]

\[p = \left[ \begin{array}{c}
1 \\
2 \\
3 \\
8 \\
9 \\
10
\end{array} \right]\]

**See Also**

hadamard Hadamard matrix
toeplitz Toeplitz matrix
Purpose

Online help for MATLAB functions and M-files

Syntax

help
help topic

Description

help, by itself, lists all primary help topics. Each main help topic corresponds to a directory name on MATLAB’s search path.

help topic gives help on the specified topic. The topic can be a function name, a directory name, or a MATLABPATH relative partial pathname. If it is a function name, help displays information on that function. If it is a directory name, help displays the contents file for the specified directory. It is not necessary to give the full pathname of the directory; the last component, or the last several components, is sufficient.

It’s possible to write help text for your own M-files and toolboxes; see Remarks.

Remarks

MATLAB’s Help system, like MATLAB itself, is highly extensible. This allows you to write help descriptions for your own M-files and toolboxes — using the same self-documenting method that MATLAB’s M-files and toolboxes use.

The command help, by itself, lists all help topics by displaying the first line (the H1 line) of the contents files in each directory on MATLAB’s search path. The contents files are the M-files named Contents.m within each directory.

The command help topic, where topic is a directory name, displays the comment lines in the Contents.m file located in that directory. If a contents file does not exist, help displays the H1 lines of all the files in the directory.

The command help topic, where topic is a function name, displays help on the function by listing the first contiguous comment lines in the M-file topic.m

Creating Online Help for Your Own M-Files

Create self-documenting online help for your own M-files by entering text on one or more contiguous comment lines, beginning with the second line of the file (first line if it is a script). (See Applying MATLAB for information about creating...
M-files.) For example, an abridged version of the M-file angle.m provided with MATLAB could contain:

```
function p = angle(h)
    % ANGLE Polar angle.
    % ANGLE(H) returns the phase angles, in radians, of a matrix
    % with complex elements. Use ABS for the magnitudes.
    p = atan2(imag(h),real(h));
```

When you execute `help angle`, lines 2, 3, and 4 display. These lines are the first block of contiguous comment lines. The help system ignores comment lines that appear later in an M-file, after any executable statements, or after a blank line.

The first comment line in any M-file (the H1 line) is special. It should contain the function name and a brief description of the function. The `lookfor` command searches and displays this line, and `help` displays these lines in directories that do not contain a `Contents.m` file.

**Creating Contents Files for Your Own M-File Directories**

A `Contents.m` file is provided for each M-file directory included with the MATLAB software. If you create directories in which to store your own M-files, you should create `Contents.m` files for them too. To do so, simply follow the format used in an existing `Contents.m` file.

**Examples**

The command

```
help datafun
```

gives help on the `datafun` directory.

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`; then enter the `help` command.

**See Also**

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

See also `partialpath`. 
**Purpose**
Hessenberg form of a matrix

**Syntax**

- `[P, H] = hess(A)`
- `H = hess(A)`

**Description**

- `H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.
- `[P, H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P * H * P'` and `P' * P = eye(size(A))`.

**Definition**
A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

**Examples**

H is a 3-by-3 eigenvalue test matrix:

```plaintext
H =
-149  -50  -154
 537   180   546
 -27   -9  -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```plaintext
hess(H) =
-149.0000  42.2037 -156.3165
-537.6783  152.5511 -554.9272
 0      0.0728   2.4489
```

**Algorithm**
For real matrices, `hess` uses the EISPACK routines `ORTRAN` and `ORTHES`. `ORTHES` converts a real general matrix to Hessenberg form using orthogonal similarity transformations. `ORTRAN` accumulates the transformations used by `ORTHES`.

When `hess` is used with a complex argument, the solution is computed using the QZ algorithm by the EISPACK routines `QZHEZ`. It has been modified for complex problems and to handle the special case `B = I`.

For detailed write-ups on these algorithms, see the EISPACK Guide.

**See Also**
- `eig` Eigenvalues and eigenvectors
- `qz` QZ factorization for generalized eigenvalues
- `schur` Schur decomposition
References


Purpose
IEEE hexadecimal to decimal number conversion

Syntax
\[ d = \text{hex2dec}('\text{hex\_value}') \]

Description
\[ d = \text{hex2dec('hex\_value')} \] converts \text{hex\_value} to its floating-point integer representation. The argument \text{hex\_value} is a hexadecimal integer stored in a MATLAB string. If \text{hex\_value} is a character array, each row is interpreted as a hexadecimal string.

Examples
\[ \text{hex2dec('3ff')} \text{ is 1023.} \]
For a character array \text{S}
\[
\begin{align*}
\text{S} &= \\
0FF &= \\
2DE &= \\
123 &= \\
\text{hex2dec}(\text{S}) &= \\
\text{ans} &= \\
255 &= \\
734 &= \\
291 &= \\
\end{align*}
\]

See Also
\begin{tabular}{ll}
\text{dec2hex} & Decimal to hexadecimal number conversion \\
\text{format} & Control the output display format \\
\text{hex2num} & Hexadecimal to double number conversion \\
\text{sprintf} & Write formatted data to a string
\end{tabular}
**hex2num**

**Purpose**
Hexadecimal to double number conversion

**Syntax**
f = hex2num('hex_value')

**Description**
f = hex2num('hex_value') converts hex_value to the IEEE double precision floating-point number it represents. NaN, inf, and denormalized numbers are all handled correctly. Fewer than 16 characters are padded on the right with zeros.

**Examples**
f = hex2num('400921fb54442d18')
f =
    3.14159265358979

**Limitations**
hex2num only works for IEEE numbers; it does not work for the floating-point representation of the VAX or other non-IEEE computers.

**See Also**
for mat
hex2dec
sprintf
Control the output display format
IEEE hexadecimal to decimal number conversion
Write formatted data to a string
Purpose
Hilbert matrix

Syntax
H = hilb(n)

Description
H = hilb(n) returns the Hilbert matrix of order n.

Definition
The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i, j) = \frac{1}{i + j - 1}$.

Examples
Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
1.5514e+04
```

Algorithm
See the M-file for a good example of efficient MATLAB programming where conventional for loops are replaced by vectorized statements.

See Also
invhilb Inverse of the Hilbert matrix

References
**Purpose**
Imaginary unit

**Syntax**
- i
- a+bi
- x+i*y

**Description**
As the basic imaginary unit \( \sqrt{-1} \), \( i \) is used to enter complex numbers. Since \( i \) is a function, it can be overridden and used as a variable. This permits you to use \( i \) as an index in for loops, etc.

If desired, use the character \( i \) without a multiplication sign as a suffix in forming a complex numerical constant.

You can also use the character \( j \) as the imaginary unit.

**Examples**
- \( Z = 2+3i \)
- \( Z = x+i*y \)
- \( Z = r*exp(i*theta) \)

**See Also**
- conj: Complex conjugate
- imag: Imaginary part of a complex number
- j: Imaginary unit
- real: Real part of complex number
Purpose
Conditionally execute statements

Syntax
if expression
  statements
end
if expression1
  statements
elseif expression2
  statements
else
  statements
end

Description
if conditionally executes statements.
The simple form is:
if expression
  statements
end

More complicated forms use elseif. Each if must be paired with a matching end.

Arguments
expression A MATLAB expression, usually consisting of smaller expressions or variables joined by relational operators (==, <, >, <=, or >=). Two examples are: count < limit and (height - offset) >= 0. Expressions may also include logical functions, as in: isreal(A).

statements One or more MATLAB statements to be executed only if the expression is true (or nonzero). See Examples for information about how nonscalar variables are evaluated.
Examples

Here is an example showing if, else, and elseif:

```plaintext
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:

```plaintext
A = B =
    1 0 1 1
    2 3 3 4
```

The expression:

- \(A < B\) evaluates as false \(\text{Since } A(1,1) \text{ is not less than } B(1,1).\)
- \(A < (B+1)\) evaluates as true \(\text{Since no element of } A \text{ is greater than the corresponding element of } B.\)
- \(A \& B\) evaluates as false \(\text{Since } A(1,2) \mid B(1,2) \text{ is false.}\)
- \(5 > B\) evaluates as true \(\text{Since every element of } B \text{ is less than } 5.\)

See Also

- break \(\text{Break out of flow control structures}\)
- else \(\text{Conditionally execute statements}\)
- end \(\text{Terminate for, while, switch, and if statements or indicate last index}\)
- for \(\text{Repeat statements a specific number of times}\)
- return \(\text{Return to the invoking function}\)
- switch \(\text{Switch among several cases based on expression}\)
- while \(\text{Repeat statements an indefinite number of times}\)
Purpose
Inverse one-dimensional fast Fourier transform

Syntax
\[
\begin{align*}
y &= \text{ifft}(X) \\
y &= \text{ifft}(X, n) \\
y &= \text{ifft}(X, [], \text{dim}) \\
y &= \text{ifft}(X, n, \text{dim})
\end{align*}
\]

Description
\( y = \text{ifft}(X) \) returns the inverse fast Fourier transform of vector \( X \).

If \( X \) is a matrix, \text{ifft} returns the inverse Fourier transform of each column of the matrix.

If \( X \) is a multidimensional array, \text{ifft} operates on the first non-singleton dimension.

\( y = \text{ifft}(X, n) \) returns the \( n \)-point inverse fast Fourier transform of vector \( X \).

\( y = \text{ifft}(X, [], \text{dim}) \) and \( y = \text{ifft}(X, n, \text{dim}) \) return the inverse discrete Fourier transform of \( X \) across the dimension \( \text{dim} \).

Examples
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.

Algorithm
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.

See Also
\text{dftmtx}, \text{freqz}, \text{specplot}, and \text{spectrum} in the Signal Processing Toolbox, and:
\text{fft} 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
**Purpose**  
Inverse two-dimensional fast Fourier transform

**Syntax**  

\[
Y = \text{ifft2}(X)  \\
Y = \text{ifft2}(X, m, n)
\]

**Description**  
\(Y = \text{ifft2}(X)\) 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 \(X\).

**Examples**  
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.

**Algorithm**  
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.

**See Also**  
\text{dftmtx}, \text{freqz}, \text{specplot}, and \text{spectrum} in the Signal Processing Toolbox, and:\n
- \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
Purpose
Inverse multidimensional fast Fourier transform

Syntax
Y = ifftn(X)
Y = ifftn(X, siz)

Description
Y = ifftn(X) performs the N-dimensional inverse fast Fourier transform. The result Y is the same size as X.

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

Remarks
For any X, ifftn(fftn(X)) equals X within roundoff error. If X is real, ifftn(fftn(X)) may have small imaginary parts.

Algorithm
ifftn(X) is equivalent to

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

This computes in-place the one-dimensional inverse fast Fourier transform along each dimension of X. The time required to compute ifftn(X) depends strongly on the number of prime factors of the dimensions of X. It is fastest when all of the dimensions are powers of 2.

See Also
fft
One-dimensional fast Fourier transform
fft2
Two-dimensional fast Fourier transform
fftn
Multidimensional fast Fourier transform
Purpose
Imaginary part of a complex number

Syntax
Y = imag(Z)

Description
Y = imag(Z) returns the imaginary part of the elements of array Z.

Examples
imag(2+3i)
ans =
   3

See Also
conj Complex conjugate
i, j Imaginary unit (\(\sqrt{-1}\))
real Real part of complex number
Purpose
Return information about a graphics file

Synopsis
info = imfinfo(filename, fmt)
info = imfinfo(filename)

Description
info = imfinfo(filename, fmt) returns a structure whose fields contain information about an image in a graphics file. filename is a string that specifies the name of the graphics file, and fmt is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If imfinfo cannot find a file named filename, it looks for a file named filename.fmt.

This table lists the possible values for fmt:

<table>
<thead>
<tr>
<th>Format</th>
<th>File type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bmp'</td>
<td>Windows Bitmap (BMP)</td>
</tr>
<tr>
<td>'hdf'</td>
<td>Hierarchical Data Format (HDF)</td>
</tr>
<tr>
<td>'jpg' or 'jpeg'</td>
<td>Joint Photographic Experts Group (JPEG)</td>
</tr>
<tr>
<td>'pcx'</td>
<td>Windows Paintbrush (PCX)</td>
</tr>
<tr>
<td>'tif' or 'tiff'</td>
<td>Tagged Image File Format (TIFF)</td>
</tr>
<tr>
<td>'xwd'</td>
<td>X Windows Dump (XWD)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename</td>
<td>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</td>
</tr>
<tr>
<td>FileModDate</td>
<td>A string containing the date when the file was last modified</td>
</tr>
<tr>
<td>FileSize</td>
<td>An integer indicating the size of the file in bytes</td>
</tr>
<tr>
<td>Format</td>
<td>A string containing the file format, as specified by <code>fmt</code>; for JPEG and TIFF files, the three-letter variant is returned</td>
</tr>
<tr>
<td>FormatVersion</td>
<td>A string or number describing the version of the format</td>
</tr>
<tr>
<td>Width</td>
<td>An integer indicating the width of the image in pixels</td>
</tr>
<tr>
<td>Height</td>
<td>An integer indicating the height of the image in pixels</td>
</tr>
<tr>
<td>BitDepth</td>
<td>An integer indicating the number of bits per pixel</td>
</tr>
<tr>
<td>ColorType</td>
<td>A string indicating the type of image; either 'truecolor' for a truecolor RGB image, 'grayscale' for a grayscale intensity image, or 'indexed' for an indexed image</td>
</tr>
</tbody>
</table>

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

```matlab
info = imfinfo('flowers.bmp')
```

```matlab
info =
    Filename: 'flowers.bmp'
    FileModDate: '16-Oct-1996 11:41:38'
    FileSize: 182078
    Format: 'bmp'
    FormatVersion: 'Version 3 (Microsoft Windows 3.x)'
    Width: 500
    Height: 362
    BitDepth: 8
    ColorType: 'indexed'
    FormatSignature: 'BM'
    NumColormapEntries: 256
    Colormap: [256x3 double]
    RedMask: []
    GreenMask: []
    BlueMask: []
    ImageDataOffset: 1078
   .BitmapHeaderSize: 40
    NumPlanes: 1
    CompressionType: 'none'
    BitmapSize: 181000
    HorzResolution: 0
    VertResolution: 0
    NumColorsUsed: 256
    NumImportantColors: 0
```

See Also

- `imread` Read image from graphics file
- `imwrite` Write an image to a graphics file
**imread**

**Purpose**
Read image from graphics file

**Synopsis**

- `A = imread(filename, fmt)`
- `[X, map] = imread(filename, fmt)`
- ` [...] = imread(filename)`
- ` [...] = imread(..., idx)` (TIFF only)
- ` [...] = imread(..., ref)` (HDF only)

**Description**

`A = imread(filename, fmt)` reads the image in `filename` into `A`, whose class is `uint8`. If the file contains a grayscale intensity image, `A` is a two-dimensional array. If the file contains a truecolor (RGB) image, `A` is a three-dimensional (m-by-n-by-3) array. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory in the MATLAB path. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for `fmt`:

<table>
<thead>
<tr>
<th>Format</th>
<th>File type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bmp'</td>
<td>Windows Bitmap (BMP)</td>
</tr>
<tr>
<td>'hdf'</td>
<td>Hierarchical Data Format (HDF)</td>
</tr>
<tr>
<td>'jpg'</td>
<td>Joint Photographic Experts Group (JPEG)</td>
</tr>
<tr>
<td>'pcx'</td>
<td>Windows Paintbrush (PCX)</td>
</tr>
<tr>
<td>'tif'</td>
<td>Tagged Image File Format (TIFF)</td>
</tr>
<tr>
<td>'xwd'</td>
<td>X Windows Dump (XWD)</td>
</tr>
</tbody>
</table>

`[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

Reads in one image from a multi-image TIFF file. If dx is an integer value that specifies the order in which the image appears in the file. For example, if dx is 3, imread reads the third image in the file. If you omit this argument, imread reads the first image in the file.

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP</td>
<td>1-bit, 4-bit, 8-bit, and 24-bit uncompressed images; 4-bit and 8-bit run-length encoded (RLE) images</td>
</tr>
<tr>
<td>HDF</td>
<td>8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets</td>
</tr>
<tr>
<td>JPEG</td>
<td>Any baseline JPEG image; JPEG images with some commonly used extensions</td>
</tr>
<tr>
<td>PCX</td>
<td>1-bit, 8-bit, and 24-bit images</td>
</tr>
<tr>
<td>TIFF</td>
<td>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</td>
</tr>
<tr>
<td>XWD</td>
<td>1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps</td>
</tr>
</tbody>
</table>
**imread**

**Examples**

This example reads the sixth image in a TIFF file:

```matlab
[X, map] = imread('flowers.tif', 6);
```

This example reads the fourth image in an HDF file:

```matlab
info = imfinfo('skull.hdf');
[X, map] = imread('skull.hdf', info(4).Reference);
```

**See Also**

- `imfinfo` Return information about a graphics file
- `imwrite` Write an image to a graphics file
**Purpose**
Write an image to a graphics file

**Synopsis**
- `imwrite(A, filename, fmt)`
- `imwrite(X, map, filename, fmt)`
- `imwrite(..., filename)`
- `imwrite(..., Parameter, Value, ...)`

**Description**
`imwrite(A, filename, fmt)` writes the image in `A` to `filename`. `filename` is a string that specifies the name of the output file, and `fmt` is a string that specifies the format of the file. If `A` is a grayscale intensity image or a truecolor (RGB) image of class `uint8`, `imwrite` writes the actual values in the array to the file. If `A` is of class `double`, `imwrite` rescales the values in the array before writing, using `uint8(round(255*A))`. This operation converts the floating-point numbers in the range [0, 1] to 8-bit integers in the range [0, 255].

This table lists the possible values for `fmt`:

<table>
<thead>
<tr>
<th>Format</th>
<th>File type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bmp'</td>
<td>Windows Bitmap (BMP)</td>
</tr>
<tr>
<td>'hdf'</td>
<td>Hierarchical Data Format (HDF)</td>
</tr>
<tr>
<td>'jpg' or 'jpeg'</td>
<td>Joint Photographers Expert Group (JPEG)</td>
</tr>
<tr>
<td>'pcx'</td>
<td>Windows Paintbrush (PCX)</td>
</tr>
<tr>
<td>'tif' or 'tiff'</td>
<td>Tagged Image File Format (TIFF)</td>
</tr>
<tr>
<td>'xwd'</td>
<td>X Windows Dump (XWD)</td>
</tr>
</tbody>
</table>

`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`.

2-367
`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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Compression'</td>
<td>One of these strings: 'none', 'rle', 'jpeg'</td>
<td>'rle'</td>
</tr>
<tr>
<td>'Quality'</td>
<td>A number between 0 and 100; parameter applies only if 'Compression' is 'jpeg'; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger</td>
<td>75</td>
</tr>
<tr>
<td>'WriteMode'</td>
<td>One of these strings: 'overwrite', 'append'</td>
<td>'overwrite'</td>
</tr>
</tbody>
</table>

This table describes the available parameters for JPEG files:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Quality'</td>
<td>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</td>
<td>75</td>
</tr>
</tbody>
</table>
This table describes the available parameters for TIFF files:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Compression'</td>
<td>One of these strings: 'none', 'packbits', 'ccitt'; 'ccitt' is valid for binary images only</td>
<td>'ccitt' for binary images; 'packbits' for all other images</td>
</tr>
<tr>
<td>'Description'</td>
<td>Any string; fills in the ImageDescription field returned by imfinfo</td>
<td>empty</td>
</tr>
</tbody>
</table>

This table summarizes the types of images that imwrite can write:

<table>
<thead>
<tr>
<th>Format</th>
<th>Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP</td>
<td>8-bit uncompressed images with associated colormap; 24-bit uncompressed images</td>
</tr>
<tr>
<td>HDF</td>
<td>8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets</td>
</tr>
<tr>
<td>JPEG</td>
<td>Baseline JPEG images</td>
</tr>
<tr>
<td>PCX</td>
<td>8-bit images</td>
</tr>
<tr>
<td>TIFF</td>
<td>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</td>
</tr>
<tr>
<td>XWD</td>
<td>8-bit ZPixmaps</td>
</tr>
</tbody>
</table>

**Example**

```matlab
imwrite(X,map,'flowers.hdf','Compression','none','WriteMode','append')
```

**See Also**

- **imfinfo** Return information about a graphics file
- **imread** Read image from graphics file
Purpose
Subscripts from linear index

Syntax
\[ [I, J] = \text{ind2sub} (\text{siz}, \text{IND}) \]
\[ [I_1, I_2, \ldots, I_n] = \text{ind2sub} (\text{siz}, \text{IND}) \]

Description
The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

\[ [I, J] = \text{ind2sub} (\text{siz}, \text{IND}) \]
returns the arrays \( I \) and \( J \) containing the equivalent row and column subscripts corresponding to the index matrix \( \text{IND} \) for a matrix of size \( \text{siz} \).

For matrices, \( [I, J] = \text{ind2sub} (\text{size}(A), \text{find}(A>5)) \) returns the same values as
\[ [I, J] = \text{find}(A>5). \]

\[ [I_1, I_2, \ldots, I_n] = \text{ind2sub} (\text{siz}, \text{IND}) \]
returns \( n \) subscript arrays \( I_1, I_2, \ldots, I_n \) containing the equivalent multidimensional array subscripts equivalent to \( \text{IND} \) for an array of size \( \text{siz} \).

Examples
The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is:

```
1   3
2   4
5   7
6   8
1,1,1 1,2,1
2,1,1 2,2,1
1,1,2 1,2,2
2,1,2 2,2,2
```

See Also
`sub2ind`
Single index from subscripts
`find`
Find indices and values of nonzero elements
Purpose
Infinity

Syntax
Inf

Description
Inf returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

Examples
1/0, 1.e1000, 2^1000, and exp(1000) all produce Inf.
log(0) produces -Inf.
Inf - Inf and Inf / Inf both produce NaN, Not-a-Number.

See Also
i s* Detect state
NaN Not-a-Number
inferiorto

Purpose
Inferior class relationship

Syntax
inferiorto('class1','class2',...)

Description
The inferiorto function establishes a hierarchy which determines the order in which MATLAB calls object methods.

inferiorto('class1','class2',...) invoked within a class constructor method (say myclass.m) indicates that myclass's method should not be invoked if a function is called with an object of class myclass and one or more objects of class class1, class2, and so on.

Remarks
Suppose A is of class 'class_a', B is of class 'class_b' and C is of class 'class_c'. Also suppose the constructor class_c.m contains the statement: inferiorto('class_a'). Then e = fun(a,c) or e = fun(c,a) invokes class_a/fun.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So, fun(b,c) calls class_b/fun, while fun(c,b) calls class_c/fun.

See Also
superiorto Superior class relationship
Purpose  Construct an inline object

Syntax  
\[
g = \text{inline(} \text{expr}) \\
g = \text{inline(} \text{expr, arg1, arg2, ...}) \\
g = \text{inline(} \text{expr, n})
\]

Description  \text{inline(} \text{expr}) \text{ constructs an inline function object from the MATLAB expression contained in the string expr. The input argument to the inline function is automatically determined by searching expr for an isolated lower case alphabetic character, other than i or j, that is not part of a word formed from several alphabetic characters. If no such character exists, x is used. If the character is not unique, the one closest to x is used. If there is a tie, the one later in the alphabet is chosen.}

\text{inline(} \text{expr, arg1, arg2, ...}) \text{ constructs an inline function whose input arguments are specified by the strings arg1, arg2, ... Multicharacter symbol names may be used.}

\text{inline(} \text{expr, n}) \text{, where n is a scalar, constructs an inline function whose input arguments are x, P1, P2, ...}

Remarks  Three commands related to \text{inline} allow you to examine an inline function object and determine how it was created.

\text{char(} f \text{un}) \text{ converts the inline function into a character array. This is identical to formula(} f \text{un}).

\text{argnames(} f \text{un}) \text{ returns the names of the input arguments of the inline object } f \text{un as a cell array of strings.}

\text{formula(} f \text{un}) \text{ returns the formula for the inline object } f \text{un.}

A fourth command \text{vectorize(} f \text{un}) inserts a \text{.} before any ^, * or / in the formula for } f \text{un. The result is a vectorized version of the inline function.}
Examples

Create a simple inline function to square a number:

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

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

Create an inline function to compute the formula $f = 3\sin(2x^2)$:

```matlab
     g = inline('3*sin(2*x.^2)')
g =
     inline function:
     g(x) = 3*sin(2*x.^2)

argnames(g)
ans =
     'x'

formula(g)
ans =
     3*sin(2*x.^2)
g(pi)  
ans =  
  2.3306  
g(2*pi)  
ans =  
  -1.2151  
fmin(g, pi, 2*pi)  
ans =  
  3.8630
inmem

Purpose

Functions in memory

Syntax

M = inmem

[M, mex] = inmem

Description

M = inmem returns a cell array of strings containing the names of the M-files that are in the P-code buffer.

[M, mex] = inmem returns a cell array containing the names of the MEX-files that have been loaded.

Examples

clear all
% start with a clean slate
erf(.5)
M = inmem

lists the M-files that were required to run erf.

2-376
Purpose
Detect points inside a polygonal region

Syntax
IN = inpolygon(X, Y, xv, yv)

Description
IN = inpolygon(X, Y, xv, yv) returns a matrix IN the same size as X and Y. Each element of IN is assigned one of the values 1, 0.5 or 0, depending on whether the point (X(p, q), Y(p, q)) is inside the polygonal region whose vertices are specified by the vectors xv and yv. In particular:

IN(p, q) = 1 If (X(p, q), Y(p, q)) is inside the polygonal region
IN(p, q) = 0.5 If (X(p, q), Y(p, q)) is on the polygon boundary
IN(p, q) = 0 If (X(p, q), Y(p, q)) is outside the polygonal region

Examples
L = linspace(0, 2.*pi, 6); xv = cos(L)'; yv = sin(L)';
xv = [xv; xv(1)]; yv = [yv; yv(1)];
x = randn(250, 1); y = randn(250, 1);
in = inpolygon(x, y, xv, yv);
plot(xv, yv, x(in), y(in), 'r+', x(~in), y(~in), 'bo')
input

Purpose
Request user input

Syntax
```
user_entry = input('prompt')
user_entry = input('prompt','s')
```

Description
The response to the `input` prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.

```
user_entry = input('prompt')
```
displays `prompt` as a prompt on the screen, waits for input from the keyboard, and returns the value entered in `user_entry`.

```
user_entry = input('prompt','s')
```
returns the entered string as a text variable rather than as a variable name or numerical value.

Remarks
If you press the Return key without entering anything, `input` returns an empty matrix.

The text string for the prompt may contain one or more `\n` characters. The `\n` means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use `\\`.

Examples
Press Return to select a default value by detecting an empty matrix:
```
i = input('Do you want more? Y/N [Y]: ','s');
if isempty(i)
i = 'Y';
end
```

See Also
The `ginput` and `uicontrol` commands in the MATLAB Graphics Guide, and:
```
keyboard
menu
```
Invoke the keyboard in an M-file
Generate a menu of choices for user input
**Purpose**  
Input argument name

**Syntax**  
`inputname(argnum)`

**Description**  
This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number `argnum` If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

**Examples**  
Suppose the function `myfun.m` is defined as:

```matlab
function c = myfun(a,b)
    disp(sprintf('First calling variable is "%s".', inputname(1)))
end
```

Then

```matlab
x = 5; y = 3; myfun(x,y)
```

produces

`First calling variable is "x".`

But

```matlab
myfun(pi+1,pi-1)
```

produces

`First calling variable is "".`

**See Also**  
`nargin`, `nargout`  
Number of function arguments

`nargchk`  
Check number of input arguments
### int2str

**Purpose**
Integer to string conversion

**Syntax**

```matlab
str = int2str(N)
```

**Description**

The function `int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

**Examples**

- `int2str(2+3)` is the string '5'.
- One way to label a plot is:
  ```matlab
title(['case number ' int2str(n)])
```
- For matrix or vector inputs, `int2str` returns a string matrix:
  ```matlab
  int2str(eye(3))
  ans =
  1 0 0
  0 1 0
  0 0 1
  ```

**See Also**
- `fprintf` Write formatted data to file
- `num2str` Number to string conversion
- `sprintf` Write formatted data to a string
Purpose

One-dimensional data interpolation (table lookup)

Syntax

\[ y_i = \text{interp1}(x, Y, x_i) \]
\[ y_i = \text{interp1}(x, Y, x_i, \text{method}) \]

Description

\( y_i = \text{interp1}(x, Y, x_i) \) returns vector \( y_i \) containing elements corresponding to the elements of \( x_i \) 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 \( y_i \) will be \( \text{length}(x_i) \)-by-size(\( Y \), 2). Out of range values are returned as NaNs.

\( y_i = \text{interp1}(x, Y, x_i, \text{method}) \) 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 \texttt{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 \( x \), \( Y \), \( x_i \), and \( y_i \).

\[ f(x) \]
\[ [ \bullet \quad \bullet \quad \bullet \quad \bullet ] \quad x \]
\[ [ \circ \quad \circ \quad \circ ] \quad Y \]
\[ y_i \]

Interpolation is the same operation as table lookup. Described in table lookup terms, the table is \( \text{tab} = [x, y] \) and \texttt{interp1} looks up the elements of \( x_i \) in \( x \),
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.

```matlab
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

```matlab
ans =
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```matlab
x = 1900:1:2000;
y = interp1(t, p, x, 'spline');
plot(t, p, 'o', x, y)
```

![United States Census](image)
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,

\[
\text{tab} = \\
\begin{array}{ll}
1950 & 150.697 \\
1960 & 179.323 \\
1970 & 203.212 \\
1980 & 226.505 \\
1990 & 249.633
\end{array}
\]

then the population in 1975, obtained by table lookup within the matrix `tab`, is

\[
p = \text{interp1}(\text{tab}(:,1), \text{tab}(:,2), 1975)
\]

\[
p = 214.8585
\]

**Algorithm**
The `interp1` command is a MATLAB M-file. The 'nearest', 'linear' and 'cubic' methods have fairly straightforward implementations. For the 'spline' method, `interp1` calls a function `spline` that uses the M-files `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses them in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see these M-files and the Spline Toolbox.

**See Also**
- `interpft` One-dimensional interpolation using the FFT method.
- `interp2` Two-dimensional data interpolation (table lookup)
- `interp3` Three-dimensional data interpolation (table lookup)
- `interpn` Multidimensional data interpolation (table lookup)
- `spline` Cubic spline interpolation

**References**
Two-dimensional data interpolation (table lookup)

Syntax

\[
\begin{align*}
Z_I &= \text{interp2}(X, Y, Z, XI, YI) \\
Z_I &= \text{interp2}(Z, XI, YI) \\
Z_I &= \text{interp2}(Z, ntimes) \\
Z_I &= \text{interp2}(X, Y, Z, XI, YI, \text{method})
\end{align*}
\]

Description

\[
Z_I = \text{interp2}(X, Y, Z, XI, YI)
\]
returns matrix \(Z_I\) containing elements corresponding to the elements of \(XI\) and \(YI\) and determined by interpolation within the two-dimensional function specified by matrices \(X\), \(Y\), and \(Z\). \(X\) and \(Y\) must be monotonic, and have the same format ("plaid") as if they were produced by \text{meshgrid}. \(X\) and \(Y\) specify the points at which the data \(Z\) is given. Out of range values are returned as NaNs.

\(XI\) and \(YI\) can be matrices, in which case \text{interp2} returns the values of \(Z\) corresponding to the points \((XI(i,j), YI(i,j))\). Alternatively, you can pass in the row and column vectors \(xi\) and \(yi\), respectively. In this case, \text{interp2} interprets these vectors as if you issued the command \text{meshgrid}(xi, yi).

\[
Z_I = \text{interp2}(Z, XI, YI)
\]
assumes that \(X = 1:n\) and \(Y = 1:m\) where \([m,n] = \text{size}(Z)\).

\[
Z_I = \text{interp2}(Z, ntimes)
\]
expands \(Z\) by interleaving interpolates between every element, working recursively for \(ntimes\). \text{interp2}(Z) is the same as \text{interp2}(Z,1).

\[
Z_I = \text{interp2}(X, Y, Z, XI, YI, \text{method})
\]
specifies an alternative interpolation method:

- 'linear' for bilinear interpolation (default)
- 'nearest' for nearest neighbor interpolation
- 'cubic' for bicubic interpolation

All interpolation methods require that \(X\) and \(Y\) be monotonic, and have the same format ("plaid") as if they were produced by \text{meshgrid}. Variable spacing is handled by mapping the given values in \(X, Y, XI,\) and \(YI\) to an equally spaced domain before interpolating. For faster interpolation when \(X\) and \(Y\) are equally spaced and monotonic, use the methods 'linear', 'cubic', or 'nearest'.

2-384
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 $\text{tab} = [\text{NaN}, Y; X, Z]$ and `interp2` looks up the elements of $X_I$ in $X$, $Y_I$ in $Y$, and, based upon their location, returns values $Z_I$ interpolated within the elements of $Z$. 

![Diagram of `interp2` function and data points](image-url)
Examples

Interpolate the `peaks` function over a finer grid:

```matlab
[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,

```matlab
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:

```matlab
w = interp2(service, years, wage, 15, 1975)
w =
190.6287
```
See Also

griddata
interp1
interp3
interp
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**

**Purpose**
Three-dimensional data interpolation (table lookup)

**Syntax**

\[
V_I = \text{interp3}(X, Y, Z, V, XI, YI, ZI)
\]
\[
V_I = \text{interp3}(V, XI, YI, ZI)
\]
\[
V_I = \text{interp3}(V, ntimes)
\]
\[
V_I = \text{interp3}(\ldots, \text{method})
\]

**Description**

\[
V_I = \text{interp3}(X, Y, Z, V, XI, YI, ZI)
\]
interpolates to find \(V_I\), the values of the underlying three-dimensional function \(V\) at the points in matrices \(XI, YI,\) and \(ZI\). Matrices \(X, Y,\) and \(Z\) specify the points at which the data \(V\) is given. Out of range values are returned as NaN.

\(XI, YI,\) and \(ZI\) can be matrices, in which case \text{interp3} returns the values of \(V\) corresponding to the points \((XI(i,j), YI(i,j), ZI(i,j))\). Alternatively, you can pass in the vectors \(xi, yi,\) and \(zi\). Vector arguments that are not the same size are interpreted as if you called \text{meshgrid}.

\[
V_I = \text{interp3}(V, XI, YI, ZI)\]
assumes \(X=1:N, Y=1:M, Z=1:P\) where \([M N P]=\text{size}(V)\).

\[
V_I = \text{interp3}(V, ntimes)\]
expands \(V\) by interleaving interpolates between every element, working recursively for \(ntimes\) iterations. The command \text{interp3}(V, 1)\) is the same as \text{interp3}(V)\).

\[
V_I = \text{interp3}(\ldots, \text{method})\]
specifies alternative methods:

- ‘linear’ for linear interpolation (default)
- ‘cubic’ for cubic interpolation
- ‘nearest’ for nearest neighbor interpolation

**Discussion**

All the interpolation methods require that \(X, Y,\) and \(Z\) be monotonic and have the same format (“plaid”) as if they were produced by \text{meshgrid}. Variable spacing is handled by mapping the given values in \(X, Y, Z, XI, YI,\) and \(ZI\) to an equally spaced domain before interpolating. For faster interpolation when \(X, Y,\) and \(Z\) are equally spaced and monotonic, use the methods ‘linear’, ‘cubic’, or ‘nearest’.
Examples

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

\[
[x, y, z, v] = \text{flow}(10);
\]
\[
[\text{xi}, \text{yi}, \text{zi}] = \text{meshgrid}(0.1:0.25:10, -3:0.25:3, -3:0.25:3);
\]
\[
\text{vi} = \text{interp3}(x, y, z, v, \text{xi}, \text{yi}, \text{zi}); \quad \% \text{VI is 31-by-41-by-27}
\]
\[
\text{slice(xi, yi, zi, vi, [6 9.5], 2, [-2 2]) shading flat}
\]

See Also

\text{interp1} \quad \text{One-dimensional data interpolation (table lookup)}
\text{interp2} \quad \text{Two-dimensional data interpolation (table lookup)}
\text{interpn} \quad \text{Multidimensional data interpolation (table lookup)}
\text{meshgrid} \quad \text{Generate X and Y matrices for three-dimensional plots}
**Purpose**  
One-dimensional interpolation using the FFT method

**Syntax**  
```matlab  
y = interpft(x, n)  
y = interpft(x, n, dim)  
```

**Description**  
`y = interpft(x, n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m` and `x` has sample interval `dx`, then the new sample interval for `y` is  
```
dy = dx * m / n. Note that n cannot be smaller than m  
```

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x, n, dim)` operates along the specified dimension.

**Algorithm**  
The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

**See Also**  
`interp1`  
One-dimensional data interpolation (table lookup)
**Purpose**
Multidimensional data interpolation (table lookup)

**Syntax**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VI = interpn(X1, X2, X3, ... , V, Y1, Y2, Y3, ...)</code></td>
<td>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 <code>2*N+1</code> 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 NaN.</td>
</tr>
<tr>
<td><code>VI = interpn(V, Y1, Y2, Y3, ...)</code></td>
<td>interpolates as above, assuming <code>X1 = 1:size(V,1), X2 = 1:size(V,2), X3 = 1:size(V,3), and so on.</code></td>
</tr>
<tr>
<td><code>VI = interpn(V, ntimes)</code></td>
<td>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>.</td>
</tr>
<tr>
<td><code>VI = interpn(..., method)</code></td>
<td>specifies alternative methods:</td>
</tr>
<tr>
<td></td>
<td><strong>'linear'</strong> for linear interpolation (default)</td>
</tr>
<tr>
<td></td>
<td><strong>'cubic'</strong> for cubic interpolation</td>
</tr>
<tr>
<td></td>
<td><strong>'nearest'</strong> for nearest neighbor interpolation</td>
</tr>
</tbody>
</table>

**Discussion**
All the interpolation methods require that `X,Y` and `Z` be monotonic and have the same format ("plaid") as if they were produced by `ndgrid`. Variable spacing is handled by mapping the given values in `X1,X2,X3,...` and `Y1,Y2,Y3,...` to an equally spaced domain before interpolating. For faster interpolation when `X1,X2,X3,` and so on are equally spaced and monotonic, use the methods **'linear', 'cubic', or 'nearest'**.
See Also

interp1
interp2
ndgrid

One-dimensional data interpolation (table lookup)
Two-dimensional data interpolation (table lookup)
Generate arrays for multidimensional functions and interpolation
### Purpose
Set intersection of two vectors

### Syntax
- `c = intersect(a,b)`
- `c = intersect(A,B,'rows')`
- `[c,ia,ib] = intersect(...)`

### Description
- `c = intersect(a,b)` returns the values common to both `a` and `b`. The resulting vector is sorted in ascending order. In set theoretic terms, this is `a \cap b`.

- `c = intersect(A,B,'rows')` when `A` and `B` are matrices with the same number of columns returns the rows common to both `A` and `B`.

- `[c,ia,ib] = intersect(a,b)` also returns column index vectors `ia` and `ib` such that `c = a(ia)` and `c = b(ib)` (or `c = a(ia,:)` and `c = b(ib,:)`).

### Examples
```matlab
A = [1 2 3 6]; B = [1 2 3 4 6 10 20];
[c,ia,ib] = intersect(A,B);
disp([c;ia;ib])
```
```
1     2     3     6
1     2     3     4
1     2     3     5
```

### See Also
- `ismember`  True for a set member
- `setdiff`  Return the set difference of two vectors
- `setxor`  Set exclusive-or of two vectors
- `union`  Set union of two vectors
- `unique`  Unique elements of a vector
Purpose
Matrix inverse

Syntax
Y = inv(X)

Description
Y = inv(X) returns the inverse of the square matrix X. A warning message is printed if X is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of inv arises when solving the system of linear equations \(Ax = b\). One way to solve this is with \(x = inv(A) \ast b\). 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.

Examples
Here is an example demonstrating the difference between solving a linear system by inverting the matrix with \(inv(A) \ast b\) 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 \(1. \times 10^{10}\), 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 \(b = A \ast x\). Thus the system of linear equations is badly conditioned, but consistent.

On a 20 MHz 386SX notebook computer, the statements

\[
\begin{align*}
\text{tic}, \quad y &= inv(A) \ast b, \quad \text{toc} \\
\text{err} &= \text{norm}(y-x) \\
\text{res} &= \text{norm}(A \ast y-b)
\end{align*}
\]

produce

\[
\begin{align*}
\text{elapsed time} &= 9.6600 \\
\text{err} &= 2.4321e-07 \\
\text{res} &= 1.8500e-09
\end{align*}
\]

while the statements

\[
\begin{align*}
\text{tic}, \quad z &= A \backslash b, \quad \text{toc} \\
\text{err} &= \text{norm}(z-x) \\
\text{res} &= \text{norm}(A \ast z-b)
\end{align*}
\]
It takes almost two and one half times as long to compute the solution with $
y = \text{inv}(A) \ast 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 $\text{inv}(A) \ast b$ 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 `ZGEID` and `ZGEFA` from LINPACK. For more information, see the LINPACK Users' Guide.

**Diagnostics**

From `inv`, if the matrix is singular,

Matrix is singular to working precision.

On machines with IEEE arithmetic, this is only a warning message. `inv` then returns a matrix with each element set to `Inf`. On machines without IEEE arithmetic, like the VAX, this is treated as an error.

If the inverse was found, but is not reliable, this message is displayed.

Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = xxx
See Also

\ Matrix left division (backslash)
/
Matrix right division (slash)
det Matrix determinant
lu
LU matrix factorization
rref Reduced row echelon form

References

Purpose

Inverse of the Hilbert matrix

Syntax

\[ H = \text{invhilb}(n) \]

Description

\[ H = \text{invhilb}(n) \] generates the exact inverse of the exact Hilbert matrix for \( n \) less than about 15. For larger \( n \), \( \text{invhilb}(n) \) generates an approximation to the inverse Hilbert matrix.

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, \( n \), is less than 15.

Comparing \( \text{invhilb}(n) \) with \( \text{inv(hilb(n))} \) involves the effects of two or three sets of roundoff errors:

- The errors caused by representing \( \text{hilb}(n) \)
- The errors in the matrix inversion process
- The errors, if any, in representing \( \text{invhilb}(n) \)

It turns out that the first of these, which involves representing fractions like \( 1/3 \) and \( 1/5 \) in floating-point, is the most significant.

Examples

\( \text{invhilb}(4) \) is

\[
\begin{pmatrix}
16 & -120 & 240 & -140 \\
-120 & 1200 & -2700 & 1680 \\
240 & -2700 & 6480 & -4200 \\
-140 & 1680 & -4200 & 2800
\end{pmatrix}
\]

See Also

hilb

Hilbert matrix

References

Purpose
Inverse permute the dimensions of a multidimensional array

Syntax
A = ipermute(B, order)

Description
A = ipermute(B, order) is the inverse of permute. ipermute rearranges the dimensions of B so that permute(A, order) will produce B. B has the same values as A but the order of the subscripts needed to access any particular element are rearranged as specified by order. All the elements of order must be unique.

Remarks
permute and ipermute are a generalization of transpose (’,’) for multidimensional arrays.

Examples
Consider the 2-by-2-by-3 array a:

\[
a = \text{cat}(3, \text{eye}(2), 2\cdot\text{eye}(2), 3\cdot\text{eye}(2))
\]

\[
a(:,:,1) =
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

\[
a(:,:,2) =
\begin{bmatrix}
2 & 0 \\
0 & 2
\end{bmatrix}
\]

\[
a(:,:,3) =
\begin{bmatrix}
3 & 0 \\
0 & 3
\end{bmatrix}
\]

Permuting and inverse permuting a in the same fashion restores the array to its original form:

\[
B = \text{permute}(a, [3 2 1]);
\]

\[
C = \text{ipermute}(B, [3 2 1]);
\]

\[
i\text{equal}(a, C)
\]

\[
\text{ans} = 1
\]

See Also
permute Rearrange the dimensions of a multidimensional array
**Purpose**
Detect state

**Syntax**

```matlab
k = iscell(C)
k = iscellstr(S)
k = ischar(S)
k = isempty(A)
k = isequal(A,B,...)
k = isfield(S,'field')
k = isfinite(A)
k = isglobal(NAME)
k = ishandle(H)
k = isinf(A)
k = isletter('str')
k = islogical(A)
k = isnan(A)
k = isnumeric(A)
k = object(A)
k = isppc
TF = isprime(A)
TF = isfinite(A)
k = isreal(A)
k = isspace('str')
k = issparse(S)
k = isstruct(S)
k = isstudent
TF = isnan(A)
k = isunichar
TF = isnan(A)
k = isunix
```

**Description**

- `k = iscell(C)` returns logical true (1) if `C` is a cell array and logical false (0) otherwise.

- `k = iscellstr(S)` returns logical true (1) if `S` is a cell array of strings and logical false (0) otherwise. A cell array of strings is a cell array where every element is a character array.

- `k = ischar(S)` returns logical true (1) if `S` is a character array and logical false (0) otherwise.

- `k = isempty(A)` returns logical true (1) if `A` is an empty array and logical false (0) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

- `k = isequal(A,B,...)` returns logical true (1) if the input arrays are the same type and size and hold the same contents, and logical false (0) otherwise.

- `k = isfield(S,'field')` returns logical true (1) if `field` is the name of a field in the structure array `S`.

- `TF = isfinite(A)` returns an array the same size as `A` containing logical true (1) where the elements of the array `A` are finite and logical false (0) where they are infinite or `NaN`.

- `k = isglobal(NAME)` returns logical true (1) if `NAME` is a global variable name and logical false (0) otherwise.

- `TF = ishandle(H)` returns logical true (1) if `H` is a handle value and logical false (0) otherwise.

- `k = isinf(A)` returns logical true (1) if `A` contains at least one infinite or NaN value, and logical false (0) otherwise.

- `k = isletter('str')` returns logical true (1) if `str` contains at least one letter, and logical false (0) otherwise.

- `k = isspace('str')` returns logical true (1) if `str` contains at least one space character, and logical false (0) otherwise.

- `k = issparse(S)` returns logical true (1) if `S` is a sparse matrix and logical false (0) otherwise.

- `k = isreal(A)` returns logical true (1) if `A` is a real matrix and logical false (0) otherwise.

- `k = isstruct(S)` returns logical true (1) if `S` is a structure array and logical false (0) otherwise.

- `TF = isnan(A)` returns logical true (1) if `A` contains at least one `NaN` value, and logical false (0) otherwise.

- `k = isunichar` returns logical true (1) if `A` is a unichar or char array and logical false (0) otherwise.

- `TF = isnan(A)` returns logical true (1) if `A` contains at least one `NaN` value, and logical false (0) otherwise.

- `TF = isprime(A)` returns logical true (1) if `A` is a prime number and logical false (0) otherwise.

- `k = isunichar` returns logical true (1) if `A` is a unichar or char array and logical false (0) otherwise.

- `TF = isunichar` returns logical true (1) if `A` is a unichar or char array and logical false (0) otherwise.

- `k = isunix` returns logical true (1) if `A` is created on a UNIX platform and logical false (0) otherwise.

- `k = isvms` returns logical true (1) if `A` is created on a VMS platform and logical false (0) otherwise.

- `TF = isvms` returns logical true (1) if `A` is created on a VMS platform and logical false (0) otherwise.
For any A, exactly one of the three quantities \( \text{isfinite}(A) \), \( \text{isinf}(A) \), and \( \text{isnan}(A) \) is equal to one.

\[ k = \text{isglobal}(\text{NAME}) \] returns logical true (1) if \( \text{NAME} \) has been declared to be a global variable, and logical false (0) if it has not been so declared.

\[ \text{TF} = \text{ishandle}(\text{H}) \] returns an array the same size as \( \text{H} \) that contains logical true (1) where the elements of \( \text{H} \) are valid graphics handles and logical false (0) where they are not.

\[ k = \text{ishold} \] returns logical true (1) if \( \text{hold} \) is on, and logical false (0) if it is off. When \( \text{hold} \) is on, the current plot and all axis properties are held so that subsequent graphing commands add to the existing graph. \( \text{hold on} \) means the \( \text{Next-Plot} \) property of both figure and axes is set to add.

\[ k = \text{isieee} \] 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).

\[ \text{TF} = \text{isnan}(A) \] 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.

\[ \text{TF} = \text{isletter}('\text{str}') \] returns an array the same size as \( '\text{str}' \) containing logical true (1) where the elements of \( \text{str} \) are letters of the alphabet and logical false (0) where they are not.

\[ k = \text{islogical}(A) \] returns logical true (1) if \( A \) is a logical array and logical false (0) otherwise.

\[ \text{TF} = \text{isnumeric}(A) \] returns an array the same size as \( A \) containing logical true (1) where the elements of \( A \) are numeric arrays and logical false (0) where they are not.

\[ k = \text{isnumeric}(A) \] 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.

\[ k = \text{isobject}(A) \] returns logical true (1) if \( A \) is an object and logical false (0) otherwise.
k = isppc returns logical true (1) if the computer running MATLAB is a Macintosh Power PC and logical false (0) otherwise.

TF = isprime(A) returns an array the same size as A containing logical true (1) for the elements of A which are prime, and logical false (0) otherwise.

k = isreal(A) returns logical true (1) if all elements of A are real numbers, and logical false (0) if either A is not a numeric array, or if any element of A has a nonzero imaginary component. Since strings are a subclass of numeric arrays, isreal always returns 1 for a string input.

Because MATLAB supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use isreal with discretion.

TF = isspace('str') returns an array the same size as 'str' containing logical true (1) where the elements of str are ASCII white spaces and logical false (0) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

k = issparse(S) returns logical true (1) if the storage class of S is sparse and logical false (0) otherwise.

k = isstruct(S) returns logical true (1) if S is a structure and logical false (0) otherwise.

k = isstudent returns logical true (1) for student editions of MATLAB and logical false (0) for commercial editions.

k = isunix returns logical true (1) for UNIX versions of MATLAB and logical false (0) otherwise.

k = isvms returns logical true (1) for VMS versions of MATLAB and logical false (0) otherwise.
Examples

\( s = 'A1, B2, C3'; \)

\[ \text{isletter}(s) \]
\[ \text{ans} = \]
\[ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \]

\( B = \text{rand}(2, 2, 2); \)
\( B(:,:,:) = []; \)

\[ \text{isempty}(B) \]
\[ \text{ans} = \]
\[ 1 \]

Given,

\[
A = \begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0
\end{bmatrix}
\]

\( \text{isequal}(A, B, C) \) returns 0, and \( \text{isequal}(A, B) \) returns 1.

Let

\( a = [-2 \ -1 \ 0 \ 1 \ 2] \)

Then

\[
\text{isfinite}(1./a) = [1 \ 1 \ 0 \ 1 \ 1]
\]
\[
\text{isinf}(1./a) = [0 \ 0 \ 1 \ 0 \ 0]
\]
\[
\text{isnan}(1./a) = [0 \ 0 \ 0 \ 0 \ 0]
\]

and

\[
\text{isfinite}(0./a) = [1 \ 1 \ 0 \ 1 \ 1]
\]
\[
\text{isinf}(0./a) = [0 \ 0 \ 0 \ 0 \ 0]
\]
\[
\text{isnan}(0./a) = [0 \ 0 \ 1 \ 0 \ 0]
\]
Purpose
Detect an object of a given class

Syntax
K = isa(obj,'class_name')

Description
K = isa(obj,'class_name') returns logical true (1) if obj is of class (or a subclass of) class_name, and logical false (0) otherwise.

The argument class_name is the name of a user-defined or pre-defined class of objects. Predefined MATLAB classes include:

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

Examples
isa(rand(3,4),'double') returns 1.

See Also
class: Create object or return class of object
ismember

Purpose
Detect members of a set

Syntax
\[ k = \text{ismember}(a, S) \]
\[ k = \text{ismember}(A, S, 'rows') \]

Description
\( k = \text{ismember}(a, S) \) returns a vector the same length as \( a \) containing logical true (1) where the elements of \( a \) are in the set \( S \), and logical false (0) elsewhere. In set theoretic terms, \( k \) is 1 where \( a \in S \).

\( k = \text{ismember}(A, S, 'rows') \) when \( A \) and \( S \) are matrices with the same number of columns returns a vector containing 1 where the rows of \( A \) are also rows of \( S \) and 0 otherwise.

Examples
\[
\text{set} = [0 2 4 6 8 10 12 14 16 18 20];
\text{a} = \text{reshape}(1:5, [5 1])
\]
\[
a = [1 2 3 4 5]
\]
\[
\text{ismember}(a, \text{set})
\]
\[
\text{ans} = [0 1 0 1 0]
\]

See Also
intersect, setdiff, setxor, union, unique
Set intersection of two vectors
Return the set difference of two vectors
Set exclusive-or of two vectors
Set union of two vectors
Unique elements of a vector
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Detect strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This MATLAB 4 function has been renamed <code>ischar</code> in MATLAB 5.</td>
</tr>
<tr>
<td>See Also</td>
<td><code>is*</code> Detect state</td>
</tr>
</tbody>
</table>
### Purpose
Imaginary unit

### Syntax
- `j`
- `x+yj`
- `x+j*y`

### Description
Use the character `j` in place of the character `i`, if desired, as the imaginary unit.

As the basic imaginary unit `sqrt(-1)`, `j` is used to enter complex numbers. Since `j` is a function, it can be overridden and used as a variable. This permits you to use `j` as an index in `for` loops, etc.

It is possible to use the character `j` without a multiplication sign as a suffix in forming a numerical constant.

### Examples
- `Z = 2+3j`
- `Z = x+j*y`
- `Z = r*exp(j*theta)`

### See Also
- `conj` Complex conjugate
- `i` Imaginary unit
- `imag` Imaginary part of a complex number
- `real` Real part of complex number
keyboard

**Purpose** Invoke the keyboard in an M-file

**Syntax** keyboard

**Description** `keyboard` , when placed in an M-file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `κ` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your M-files.

To terminate the keyboard mode, type the command:

```
return
```

then press the Return key.

**See Also**

- `dbstop` Set breakpoints in an M-file function
- `input` Request user input
- `quit` Terminate MATLAB
- `return` Terminate keyboard mode
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 \( mp \)-by-\( nq \).

Examples  

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

\[
\begin{bmatrix}
X(1, 1) * Y & X(1, 2) * Y & X(1, 3) * Y \\
X(2, 1) * Y & X(2, 2) * Y & X(2, 3) * Y
\end{bmatrix}
\]

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:

\[
\begin{align*}
I &= \text{ speye}(n, n); \\
E &= \text{ sparse}(2:n, 1:n-1, 1, n, n); \\
D &= E + E' - 2*I; \\
A &= \text{ kron}(D, I) + \text{ kron}(I, D);
\end{align*}
\]

Plotting this with the \texttt{spy} function for \( n = 5 \) yields:
Purpose

Last error message

Syntax

\[
\text{str} = \text{lasterr} \\
\text{lasterr}('')
\]

Description

\(\text{str} = \text{lasterr}\) returns the last error message generated by MATLAB.

\(\text{lasterr}('')\) resets \(\text{lasterr}\) so it returns an empty matrix until the next error occurs.

Examples

Here is a function that examines the \(\text{lasterr}\) string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply.

\[
\text{function catch} \\
\text{l} = \text{lasterr}; \\
\text{j} = \text{findstr(l,'Inner matrix dimensions');} \\
\text{if j==[]} \\
\text{disp('Wrong dimensions for matrix multiply')} \\
\text{else} \\
\text{k} = \text{findstr(l,'Undefined function or variable')} \\
\text{if (k==[])} \\
\text{disp('At least one operand does not exist')} \\
\text{end} \\
\text{end}
\]

The lasterr function is useful in conjunction with the two-argument form of the eval function:

\[
\text{eval('str','catchstr')}\]

where catchstr examines the lasterr string to determine the cause of the error and take appropriate action. The eval function evaluates the string \(\text{str}\) and returns if no error occurs. If an error occurs, eval executes catchstr. Using eval with the catch function above:

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

2-408
MATLAB responds with **Wrong dimensions for matrix multiply.**

**See Also**

- `error`  
  Display error messages
- `eval`  
  Interpret strings containing MATLAB expressions
Purpose
Least common multiple

Syntax
L = lcm(A, B)

Description
L = lcm(A, B) returns the least common multiple of corresponding elements of arrays A and B. Inputs A and B must contain positive integer elements and must be the same size (or either can be scalar).

Examples
lcm(8, 40)
ans =
   40

lcm(pascal(3), magic(3))
ans =
   8  1  6
   3 10 21
   4  9  6

See Also
gcd  

Greatest common divisor
Purpose
Associated Legendre functions

Syntax

$P = \text{legendre}(n, X)$
$S = \text{legendre}(n, X, 'sch')$

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:

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

Description
$P = \text{legendre}(n, X)$ computes the associated Legendre functions of degree $n$ and order $m = 0, 1, \ldots, n$, 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 $P(m+1, d1, d2 \ldots)$ contains the associated Legendre function of degree $n$ and order $m$ evaluated at $X(d1, d2 \ldots)$.

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

$\begin{bmatrix}
P_2^0(x(1)) & P_2^0(x(2)) & P_2^0(x(3)) & \ldots \\
P_2^1(x(1)) & P_2^1(x(2)) & P_2^1(x(3)) & \ldots \\
P_2^2(x(1)) & P_2^2(x(2)) & P_2^2(x(3)) & \ldots 
\end{bmatrix}$
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:

<table>
<thead>
<tr>
<th>m = 0</th>
<th>x = 0</th>
<th>x = 0.1</th>
<th>x = 0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5000</td>
<td>0.4850</td>
<td>0.4400</td>
</tr>
<tr>
<td>m = 1</td>
<td>0</td>
<td>0.2985</td>
<td>0.5879</td>
</tr>
<tr>
<td>m = 2</td>
<td>3.0000</td>
<td>2.9700</td>
<td>2.8800</td>
</tr>
</tbody>
</table>

Note that this matrix is of the form shown at the bottom of the previous page.

Given,

```
X = rand(2, 4, 5); N = 2;
P = legendre(N, X)
```

Then `size(P)` is 3-by-2-by-4-by-5, and `P(:,1,2,3)` is the same as `legendre(n, X(:,1,2,3))`.
Purpose
Length of vector

Syntax
\[ n = \text{length}(X) \]

Description
The statement \( \text{length}(X) \) is equivalent to \( \max(\text{size}(X)) \) for nonempty arrays and 0 for empty arrays.

\[ n = \text{length}(X) \] returns the size of the longest dimension of \( X \). If \( X \) is a vector, this is the same as its length.

Examples
\[
\begin{align*}
    x &= \text{ones}(1, 8); \\
    n &= \text{length}(x) \\
    n &= 8 \\
    x &= \text{rand}(2, 10, 3); \\
    n &= \text{length}(x) \\
    n &= 10
\end{align*}
\]

See Also
\begin{align*}
    \text{ndims} & \quad \text{Number of array dimensions} \\
    \text{size} & \quad \text{Array dimensions}
\end{align*}
**lin2mu**

**Purpose**
Linear to mu-law conversion

**Syntax**
\[ \mu = \text{lin2mu}(y) \]

**Description**
\[ \mu = \text{lin2mu}(y) \] converts linear audio signal amplitudes in the range \(-1 \leq Y \leq 1\) to mu-law encoded “flints” in the range \(0 \leq \mu \leq 255\).

**See Also**
- auwrite: Write NeXT/SUN (.au) sound file
- mu2lin: Mu-law to linear conversion
Purpose
Generate linearly spaced vectors

Syntax
\[
\begin{align*}
y &= \text{linspace}(a, b) \\
y &= \text{linspace}(a, b, n)
\end{align*}
\]

Description
The \text{linspace} function generates linearly spaced vectors. It is similar to the colon operator “:”, but gives direct control over the number of points.

\[
y = \text{linspace}(a, b) \quad \text{generates a row vector } y \text{ of 100 points linearly spaced between } a \text{ and } b.
\]

\[
y = \text{linspace}(a, b, n) \quad \text{generates } n \text{ points.}
\]

See Also
: (Colon) Create vectors, matrix subscripting, and \texttt{for} iterations
logspace Generate logarithmically spaced vectors
**load**

**Purpose**
Retrieve variables from disk

**Syntax**
- `load`
- `load filename`
- `load (filename)`
- `load filename.ext`
- `load filename -ascii`
- `load filename -mat`

**Description**
The `load` and `save` commands retrieve and store MATLAB variables on disk.

- `load` by itself, loads all the variables saved in the file 'matlab.mat'.
- `load filename` retrieves the variables from 'filename.mat' given a full pathname or a MATLABPATH relative partial pathname.
- `load (filename)` loads a file whose name is stored in `filename`. The statement:

  ```matlab
  str = 'filename.mat'; load (str)
  ```

  retrieves the variables from the binary file 'filename.mat'.

- `load filename.ext` reads ASCII files that contain rows of space separated values. The resulting data is placed into a variable with the same name as the file (without the extension). ASCII files may contain MATLAB comments (lines that begin with `%`).

- `load filename -ascii` or `load filename -mat` can be used to force `load` to treat the file as either an ASCII file or a MAT file.

**Remarks**
MAT-files are double-precision binary MATLAB format files created by the `save` command and readable by the `load` command. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the disparate formats allow. They can also be manipulated by other programs, external to MATLAB.

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

- fprintf: Write formatted data to file
- fscanf: Read formatted data from file
- save: Save workspace variables on disk
- spconvert: Import matrix from sparse matrix external format

See also partialpath.
log

Purpose  
Natural logarithm

Syntax  
\[ Y = \text{log}(X) \]

Description  
The \text{log} function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

\[ Y = \text{log}(X) \] returns the natural logarithm of the elements of \( X \). For complex or negative \( z \), where \( z = x + y i \), the complex logarithm is returned:

\[ \text{log}(z) = \text{log}(\text{abs}(z)) + i \cdot \text{atan2}(y, x) \]

Examples  
The statement \( \text{abs}(\text{log}(-1)) \) is a clever way to generate \( \pi \):

\[
\begin{align*}
\text{ans} &= \text{abs}(\text{log}(-1)) \\
&= 3.1416
\end{align*}
\]

See Also  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{exp}</td>
<td>Exponential</td>
</tr>
<tr>
<td>\text{log10}</td>
<td>Common (base 10) logarithm</td>
</tr>
<tr>
<td>\text{log2}</td>
<td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td>
</tr>
<tr>
<td>\text{logm}</td>
<td>Matrix logarithm</td>
</tr>
</tbody>
</table>
Purpose
Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax
Y = log2(X)
[F, E] = log2(X)

Description
Y = log2(X) computes the base 2 logarithm of the elements of X.

[F, E] = log2(X) returns arrays F and E. Argument F is an array of real values, usually in the range 0.5 ≤ abs(F) < 1. For real X, F satisfies the equation: X = F .* 2.^E. Argument E is an array of integers that, for real X, satisfy the equation: X = F .* 2.^E.

Remarks
This function corresponds to the ANSI C function frexp() and the IEEE floating-point standard function logb(). Any zeros in X produce F = 0 and E = 0.

Examples
For IEEE arithmetic, the statement [F, E] = log2(X) yields the values:

\[
\begin{array}{ccc}
X & F & E \\
1 & 1/2 & 1 \\
\pi & \pi/4 & 2 \\
-3 & -3/4 & 2 \\
\text{eps} & 1/2 & -51 \\
\text{real max} & 1-\text{eps}/2 & 1024 \\
\text{real min} & 1/2 & -1021 \\
\end{array}
\]

See Also
log
Natural logarithm
pow2
Base 2 power and scale floating-point numbers
log10

**Purpose**
Common (base 10) logarithm

**Syntax**

\[ Y = \log_{10}(X) \]

**Description**
The \( \log_{10} \) function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

\[ Y = \log_{10}(X) \] returns the base 10 logarithm of the elements of \( X \).

**Examples**
On a computer with IEEE arithmetic

\[ \log_{10}(\text{realmax}) \text{ is } 308.2547 \]

and

\[ \log_{10}(\text{eps}) \text{ is } -15.6536 \]

**See Also**
exp \hspace{1em} \text{Exponential}
\log \hspace{1em} \text{Natural logarithm}
\log2 \hspace{1em} \text{Base 2 logarithm and dissect floating-point numbers into exponent and mantissa}
\logm \hspace{1em} \text{Matrix logarithm}
**Purpose** Convert numeric values to logical

**Syntax**

\[ K = \text{logical}(A) \]

**Description**

\[ K = \text{logical}(A) \] returns an array that can be used for logical indexing or logical tests. The array \( K \) is the same size as \( A \) and is displayed using 1 where corresponding elements of \( A \) are nonzero, and 0 where corresponding elements of \( A \) are zero.

**Remarks**

Logical arrays are also created by the relational operators (\( == \), \( < \), \( > \), etc.) and functions like \( \text{any} \), \( \text{all} \), \( \text{isnan} \), \( \text{isinf} \), and \( \text{isfinite} \).

**Examples**

Given \( A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \), the statement \( B = \text{logical}(\text{eye}(3)) \) returns a logical array

\[
B =
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

which can be used in logical indexing that returns \( A \)'s diagonal elements:

\[
A(B)
\]

\[
\text{ans} =
\begin{bmatrix}
1 \\
5 \\
9
\end{bmatrix}
\]

However, attempting to index into \( A \) using the numeric array \( \text{eye}(3) \) results in:

\[
A(\text{eye}(3))
\]

??? Index into matrix is negative or zero.
**Purpose**
Matrix logarithm

**Syntax**
Y = logm(X)
[Y, esterr] = logm(X)

**Description**
Y = logm(X) returns the matrix logarithm: the inverse function of expm(X).
Complex results are produced if X has negative eigenvalues. A warning message is printed if the computed expm(Y) is not close to X.

[Y, esterr] = logm(X) does not print any warning message, but returns an estimate of the relative residual, \(\frac{\text{norm}(\expm(Y) - X)}{\text{norm}(X)}\).

**Remarks**
If X is real symmetric or complex Hermitian, then so is logm(X).
Some matrices, like X = [0 1; 0 0], do not have any logarithms, real or complex, and logm cannot be expected to produce one.

**Limitations**
For most matrices:
\[\logm(\expm(X)) = X = \expm(\logm(X))\]
These identities may fail for some X. For example, if the computed eigenvalues of X include an exact zero, then logm(X) generates infinity. Or, if the elements of X are too large, expm(X) may overflow.

**Examples**
Suppose A is the 3-by-3 matrix

\[
\begin{bmatrix}
1 & 1 & 0 \\
0 & 0 & 2 \\
0 & 0 & -1
\end{bmatrix}
\]

and X = expm(A) is

\[
\begin{bmatrix}
2.7183 & 1.7183 & 1.0862 \\
0 & 1.0000 & 1.2642 \\
0 & 0 & 0.3679
\end{bmatrix}
\]

Then A = logm(X) produces the original matrix A.

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

\begin{verbatim}
ans =
1.0000  0.5413  0.0826
-Inf    0   0.2345
-Inf   -Inf  -1.0000
\end{verbatim}

**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**
- \texttt{expm} Matrix exponential
- \texttt{funm} Evaluate functions of a matrix
- \texttt{sqrtm} Matrix square root

**References**

**logspace**

**Purpose**
Generate logarithmically spaced vectors

**Syntax**

\[
y = \text{logspace}(a, b)\\
y = \text{logspace}(a, b, n)\\
y = \text{logspace}(a, \pi)
\]

**Description**
The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the “:” or colon operator.

- `y = \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.

**Remarks**
All the arguments to `logspace` must be scalars.

**See Also**
- (Colon) Create vectors, matrix subscripting, and for iterations
- `linspace` Generate linearly spaced vectors
Purpose

Keyword search through all help entries

Syntax

lookfor topic
lookfor topic -all

Description

lookfor topic searches for the string topic in the first comment line (the H1 line) of the help text in all M-files found on MATLAB's search path. For all files in which a match occurs, lookfor displays the H1 line.

lookfor topic -all searches the entire first comment block of an M-file looking for topic.

Examples

For example

lookfor inverse

finds at least a dozen matches, including H1 lines containing “inverse hyperbolic cosine,” “two-dimensional inverse FFT,” and “pseudoinverse.” Contrast this with

which inverse

or

what inverse

These commands run more quickly, but probably fail to find anything because MATLAB does not ordinarily have a function inverse.

In summary, what lists the functions in a given directory, which finds the directory containing a given function or file, and lookfor finds all functions in all directories that might have something to do with a given keyword.

See Also

dir Directory listing
help Online help for MATLAB functions and M-files
what Directory listing of M-files, MAT-files, and MEX-files
which Locate functions and files
who List directory of variables in memory
lower

Purpose Convert string to lower case

Syntax \( t = \text{lower('str')} \)

Description \( t = \text{lower('str')} \) returns the string formed by converting any upper-case characters in \( \text{str} \) to the corresponding lower-case characters and leaving all other characters unchanged.

Examples \( \text{lower('MathWorks')} \) is \text{mathworks}.

Remarks Character sets supported:
- Mac: Standard Roman
- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also upper Convert string to upper case
Purpose
Least squares solution in the presence of known covariance

Syntax
\[
x = \text{lscov}(A, b, V)
\]
\[
[x, dx] = \text{lscov}(A, b, V)
\]

Description
\(x = \text{lscov}(A, b, V)\) 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 \(m\times n\) 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^* (i \text{nv}(V) - i \text{nv}(V) * A^* i \text{nv}(V) * A) * A^* i \text{nv}(V) * B \) / (m – n)
\]
\[
dx = \text{sqrt} \left( \text{diag} (i \text{nv}(A^* i \text{nv}(V) * A) * \text{mse}) \right)
\]

Algorithm
The vector \(x\) minimizes the quantity \((A^*x – b)' * i \text{nv}(V) * (A^*x – b)\). The classical linear algebra solution to this problem is

\[
x = i \text{nv}(A^* i \text{nv}(V) * A) * A^* i \text{nv}(V) * b
\]

but the \text{lscov} function instead computes the QR decomposition of \(A\) and then modifies \(Q\) by \(V\).

See Also
\(\backslash\) Matrix left division (backslash)
\(\text{nnls}\) Nonnegative least squares
\(\text{qr}\) Orthogonal-triangular decomposition

Reference
Purpose
LU matrix factorization

Syntax

\[
\begin{align*}
[L, U] &= \text{lu}(X) \\
[L, U, P] &= \text{lu}(X) \\
\text{l u}(X)
\end{align*}
\]

Description
The \text{lu} function expresses any square matrix \(X\) as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the LU, or sometimes the LR, factorization.

\[
[L, U] = \text{lu}(X)
\]
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 \(L*U = P*X\).

\(\text{l u}(X)\)
returns the output from the LINPACK routine \text{ZGEFA}.

Remarks
Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with \text{inv} and the determinant with \text{det}. It is also the basis for the linear equation solution or matrix division obtained with \(\backslash\) and /.

Arguments

\(L\) A factor of \(X\). Depending on the form of the function, \(L\) is either lower triangular, or else the product of a lower triangular matrix with a permutation matrix \(P\).

\(U\) An upper triangular matrix that is a factor of \(X\).

\(P\) The permutation matrix satisfying the equation \(L*U = P*X\).

Examples
Start with

\[
A =
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 0
\end{bmatrix}
\]
To see the LU factorization, call \texttt{lu} with two output arguments:

\[
\begin{bmatrix}
L & U
\end{bmatrix} = \texttt{lu}(A)
\]

\[
L = \\
\begin{bmatrix}
0.1429 & 1.0000 & 0 \\
0.5714 & 0.5000 & 1.0000 \\
1.0000 & 0 & 0
\end{bmatrix}
\]

\[
U = \\
\begin{bmatrix}
7.0000 & 8.0000 & 0.0000 \\
0 & 0.8571 & 3.0000 \\
0 & 0 & 4.5000
\end{bmatrix}
\]

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:

\[
L \cdot U
\]

which returns the original \(A\). Using three arguments on the left-hand side to get the permutation matrix as well:

\[
\begin{bmatrix}
L & U & P
\end{bmatrix} = \texttt{lu}(A)
\]

returns the same value of \(U\), but \(L\) is reordered:

\[
L = \\
\begin{bmatrix}
1.0000 & 0 & 0 \\
0.1429 & 1.0000 & 0 \\
0.5714 & 0.5000 & 1.0000
\end{bmatrix}
\]

\[
U = \\
\begin{bmatrix}
7.0000 & 8.0000 & 0 \\
0 & 0.8571 & 3.0000 \\
0 & 0 & 4.5000
\end{bmatrix}
\]

\[
P = \\
\begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]
To verify that \( L+U \) is a permuted version of \( A \), compute \( L+U \) and subtract it from \( P \cdot A \):

\[
P \cdot A - L + U
\]

The inverse of the example matrix, \( X = \text{inv}(A) \), is actually computed from the inverses of the triangular factors:

\[
X = \text{inv}(U) \ast \text{inv}(L)
\]

The determinant of the example matrix is

\[
d = \det(A)
\]

which gives

\[
d = 27
\]

It is computed from the determinants of the triangular factors:

\[
d = \det(L) \ast \det(U)
\]

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**

The **lu** uses the subroutines **ZGEDI** and **ZGEFA** from LINPACK. For more information, see the LINPACK Users' Guide.

**See Also**

- \( \backslash \) Matrix left division (backslash)
- \( / \) Matrix right division (slash)
- \( \text{cond} \) Condition number with respect to inversion
- \( \text{det} \) Matrix determinant
- \( \text{inv} \) Matrix inverse
- \( \text{qr} \) Orthogonal-triangular decomposition
- \( \text{rref} \) Reduced row echelon form

**References**

Purpose
Incomplete LU matrix factorizations

Syntax
luinc(X,'0')
[L,U] = luinc(X,'0')
[L,U,P] = luinc(X,'0')
luinc(X,droptol)
luinc(X,options)
[L,U] = luinc(X,options)
[L,U] = luinc(X,droptol)
[L,U,P] = luinc(X,options)
[L,U,P] = luinc(X,droptol)

Description
luinc produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

luinc(X,'0') computes the incomplete LU factorization of level 0 of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix X, and their product agrees with the permutated X over its sparsity pattern. luinc(X,'0') returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost, but
\[ \text{nnz(luinc}(X, '0') \text{)} = \text{nnz}(X), \text{ with the possible exception of some zeros due to cancellation.} \]

\[ [L,U] = \text{luinc}(X, '0') \]
returns the product of permutation matrices and a unit lower triangular matrix in L and an upper triangular matrix in U. The exact sparsity patterns of L, U, and X are not comparable but the number of nonzeros is maintained with the possible exception of some zeros in L and U due to cancellation:
\[ \text{nnz}(L) + \text{nnz}(U) = \text{nnz}(X) + n, \text{ where } X \text{ is } n \times n. \]
The product \( L \cdot U \) agrees with \( X \) over its sparsity pattern. \( (L \cdot U) \cdot spones(X) - X \) has entries of the order of \( \text{eps} \).

\[ [L,U,P] = \text{luinc}(X, '0') \]
returns a unit lower triangular matrix in L, an upper triangular matrix in U and a permutation matrix in P. L has the same sparsity pattern as the lower triangle of the permuted X
\[ \text{spones}(L) = \text{spones}(\text{tril}(P \cdot X)) \]
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 $P^*X$

\[ \text{spones}(U) = \text{spones}(\text{triu}(P^*X)) \]

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. $(L^*U)^* \text{spones}(P^*X) - P^*X$ has entries of the order of eps.

**luinc**

computes the incomplete LU factorization of any sparse matrix using a drop tolerance. drop tol must be a non-negative scalar.

$\text{luinc}(X, \text{droptol})$ produces an approximation to the complete LU factors returned by $\text{lu}(X)$. 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 $\text{lu}(X)$.

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 $X$)

\[ \text{droptol} \times \text{norm}(X(:,j)) \]

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.

$\text{luinc}(X, \text{opt ons})$ specifies a structure with up to four fields that may be used in any combination: drop tol, milu, udiag, thresh. Additional fields of opt ons are ignored.

drop tol is the drop tolerance of the incomplete factorization.

If milu is 1, $\text{luinc}$ produces the modified incomplete LU factorization that subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is 0.

If udiag is 1, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is 0.
thresh is the pivot threshold between 0 (forces diagonal pivoting) and 1, the default, which always chooses the maximum magnitude entry in the column to be the pivot. thresh is described in greater detail in lu.

luinc(X,options) is the same as luinc(X,droptol) if options has droptol as its only field.

[L, U] = luinc(X, options) returns a permutation of a unit lower triangular matrix in L and an upper triangular matrix in U. The product L*U is an approximation to X. luinc(X, options) returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

[L, U] = luinc(X, options) is the same as luinc(X, droptol) if options has droptol as its only field.

[L, U, P] = luinc(X, options) returns a unit lower triangular matrix in L, an upper triangular matrix in U, and a permutation matrix in P. The nonzero entries of U satisfy

\[|U(i, j)| \geq \text{droptol} \times \text{norm}(X(:, j)),\]

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

\[|L(i, j)| \geq \text{droptol} \times \text{norm}(X(:, j))/U(j, j).\]

The product L*U is an approximation to the permuted P*X.

[L, U, P] = luinc(X, options) is the same as [L, U, P] = luinc(X, droptol) if options has droptol as its only field.

Remarks

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. The lower triangular factors all have 1’s along the main diagonal but a single 0 on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the udiag option to replace a zero diagonal only gets rid of the symptoms of the problem but does not solve it. The preconditioner may not be singular, but it probably is not useful and a warning message is printed.
**Limitations**  
`luinc(X,'0')` works on square matrices only.

**Examples**  
Start with a sparse matrix and compute its LU factorization.

```matlab
load west0479;
S = west0479;
LU = lu(S);
```

Compute the incomplete LU factorization of level 0.

```matlab
[L, U, P] = luinc(S, '0');
D = (L*U) .* spones(P*S) - P*S;
spones(U) and spones(triu(P*S)) are identical.
```
\texttt{spones(L)} and \texttt{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.

\begin{align*}
[1 \text{ } \text{L0, I0, I P0}] &= \text{luinc}(S, 0); \\
[1 \text{ } \text{L1, I1, I P1}] &= \text{luinc}(S, 1e^{-10}); \\
&\vdots
\end{align*}

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 \( \frac{\text{nnz(} L^*U - P*S, 1) / \text{nnz(} S, 1) }{ \text{nnz(} S, 1) } \) in second figure below.
Algorithm

`luinc(X, '0')` is based on the “KJI” variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `X`.

`luinc(X, droptol)` and `luinc(X, options)` are based on the column-oriented `lu` for sparse matrices.

See Also

`lu`        LU matrix factorization
  `cholinc`  Incomplete Cholesky factorizations
  `bicg`     BiConjugate Gradients method

References

Purpose: Magic square

Syntax: \( M = \text{magic}(n) \)

Description: \( M = \text{magic}(n) \) 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.

Remarks: A magic square, scaled by its magic sum, is doubly stochastic.

Examples: The magic square of order 3 is

\[
M = \text{magic}(3) \\
M = \\
8 \quad 1 \quad 6 \\
3 \quad 5 \quad 7 \\
4 \quad 9 \quad 2
\]

This is called a magic square because the sum of the elements in each column is the same.

\[
\text{sum}(M) = \\
15 \quad 15 \quad 15
\]

And the sum of the elements in each row, obtained by transposing twice, is the same.

\[
\text{sum}(M')' = \\
15 \\
15
\]

This is also a special magic square because the diagonal elements have the same sum.

\[
\text{sum(diag}(M)) = \\
15
\]

The value of the characteristic sum for a magic square of order \( n \) is

\[
\text{sum(1:n^2)}/n
\]

which, when \( n = 3 \), is 15.
Algorithm

There are three different algorithms: one for odd \( n \), one for even \( n \) not divisible by four, and one for even \( n \) divisible by four.

To make this apparent, type:

```matlab
for n = 3:20
    A = magic(n);
    plot(A, '–')
    r(n) = rank(A);
end
r
```

Limitations

If you supply \( n \) less than 3, `magic` returns either a nonmagic square, or else the degenerate magic squares \( 1 \) and \( [ ] \).

See Also

- `ones` Create an array of all ones
- `rand` Uniformly distributed random numbers and arrays
mat2str

Purpose
Convert a matrix into a string

Syntax
str = mat2str(A)
str = mat2str(A,n)

Description
str = mat2str(A) converts matrix A into a string, suitable for input to the
eval function, using full precision.

str = mat2str(A,n) converts matrix A using n digits of precision.

Limitations
The mat2str function is intended to operate on scalar, vector, or rectangular
array inputs only. An error will result if A is a multidimensional array.

Examples
Consider the matrix:

A =
| 1 2 |
| 3 4 |

The statement
b = mat2str(A)
produces:

b =
[ 1 2 ; 3 4 ]

where b is a string of 11 characters, including the square brackets, spaces, and
a semicolon.
eval(mat2str(A)) reproduces A.

See Also
int2str Integer to string conversion
sprintf Write formatted data to a string
str2num 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:

```matlab
if exist('startup') == 2
    startup
end
```

that invoke `startup.m`. Extend this process to create additional startup M-files, if required.

**See Also**

- `exist` Operating system command
- `path` Check if a variable or file exists
- `quit` Control MATLAB's directory search path
- `quit` Terminate MATLAB
**Purpose**
Root directory of MATLAB installation

**Syntax**
\( \text{rd} = \text{matlabroot} \)

**Description**
\( \text{rd} = \text{matlabroot} \) returns the name of the directory in which the MATLAB software is installed.

**Example**
\( \text{fullfile(matlabroot,'toolbox','matlab','general','')} \)
produces a full path to the toolbox/matlab/general directory that is correct for the platform it is executed on.
Purpose
Maximum elements of an array

Syntax
C = max(A)
C = max(A, B)
C = max(A, [], di m)
[C, I] = max(...)

Description
C = max(A) returns the largest elements along different dimensions of an array.
If A is a vector, max(A) returns the largest element in A.
If A is a matrix, max(A) treats the columns of A as vectors, returning a row vector containing the maximum element from each column.
If A is a multidimensional array, max(A) treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.

C = max(A, B) returns an array the same size as A and B with the largest elements taken from A or B.

C = max(A, [], di m) returns the largest elements along the dimension of A specified by scalar di m For example, max(A, [], 1) produces the maximum values along the first dimension (the rows) of A.

[C, I] = max(...) finds the indices of the maximum values of A, and returns them in output vector I. If there are several identical maximum values, the index of the first one found is returned.

Remarks
For complex input A, max returns the complex number with the largest modulus, computed with max( abs(A)) . The max function ignores NaNs.

See Also
isnan Detect Not-A-Number (NaN)
mean Average or mean values of array
median Median values of array
min Minimum elements of an array
sort Sort elements in ascending order
mean

Purpose
Average or mean value of arrays

Syntax
M = mean(A)
M = mean(A, dim)

Description
M = mean(A) returns the mean values of the elements along different dimensions of an array.
If A is a vector, mean(A) returns the mean value of A.
If A is a matrix, mean(A) treats the columns of A as vectors, returning a row vector of mean values.
If A is a multidimensional array, mean(A) treats the values along the first non-singleton dimension as vectors, returning an array of mean values.
M = mean(A, dim) returns the mean values for elements along the dimension of A specified by scalar dim

Examples
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
mean(A)
ans =
3.5000 4.5000 6.5000 6.5000

mean(A, 2)
ans =
2.7500
4.7500
6.7500
6.7500

See Also
corrcoef  Correlation coefficients
cov  Covariance matrix
max  Maximum elements of an array
median  Median value of arrays
min  Minimum elements of an array
std  Standard deviation
Purpose
Median value of arrays

Syntax
M = median(A)
M = median(A, dim)

Description
M = median(A) returns the median values of the elements along different dimensions of an array.
If A is a vector, median(A) returns the median value of A.
If A is a matrix, median(A) treats the columns of A as vectors, returning a row vector of median values.
If A is a multidimensional array, median(A) treats the values along the first nonsingleton dimension as vectors, returning an array of median values.
M = median(A, dim) returns the median values for elements along the dimension of A specified by scalar dim

Examples
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
median(A)
ans =
   4   5   7   7

median(A, 2)
ans =
   3   5
   7
   7

See Also
corrcoef | Correlation coefficients
cov | Covariance matrix
max | Maximum elements of an array
mean | Average or mean value of arrays
min | Minimum elements of an array
std | Standard deviation
### menu

**Purpose**
Generate a menu of choices for user input

**Syntax**

```matlab
k = menu('mtitle', 'opt1', 'opt2', ..., 'optn')
```

**Description**

`k = menu('mtitle', 'opt1', 'opt2', ..., 'optn')` displays the menu whose title is in the string variable `mtitle` and whose choices are string variables `opt1`, `opt2`, and so on. `menu` returns the value you entered.

**Remarks**
To call `menu` from another UI-object, set that object's `Interruptible` property to `'yes'`. For more information, see the MATLAB Graphics Guide.

**Examples**

```matlab
k = menu('Choose a color', 'Red', 'Green', 'Blue')
```

After input is accepted, use `k` to control the color of a graph.

```matlab
color = ['r', 'g', 'b'];
plot(t, s, color(k))
```

**See Also**

The `uicontrol` command in the MATLAB Graphics Guide, and:

- `input` Request user input
**Purpose**
Generate \( X \) and \( Y \) matrices for three-dimensional plots

**Syntax**

\[
\begin{align*}
[X, Y] &= \text{meshgrid}(x, y) \\
[X, Y] &= \text{meshgrid}(x) \\
[X, Y, Z] &= \text{meshgrid}(x, y, z)
\end{align*}
\]

**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 \text{meshgrid} function is similar to \text{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, \text{meshgrid} is better suited to problems in two- or three-dimensional Cartesian space, while \text{ndgrid} is better suited to multidimensional problems that aren’t spatially based.

\text{meshgrid} is limited to two- or three-dimensional Cartesian space.

**Examples**
The function

\[ [X, Y] = \text{meshgrid}(1:3, 10:14) \]
produces two output arrays, \( X \) and \( Y \):

\[
X = \\
\begin{bmatrix}
1 & 2 & 3 \\
1 & 2 & 3 \\
1 & 2 & 3
\end{bmatrix}
\]
meshgrid

\[
\begin{array}{ccc}
1 & 2 & 3 \\
1 & 2 & 3 \\
\end{array}
\]

\[
Y =
\begin{array}{ccc}
10 & 10 & 10 \\
11 & 11 & 11 \\
12 & 12 & 12 \\
13 & 13 & 13 \\
14 & 14 & 14 \\
\end{array}
\]

See Also

mesh, slice, and surf in the MATLAB Graphics Guide, griddata, ndgrid
## methods

### Purpose
Display method names

### Syntax
```
methods class_name
n = methods('class_name')
```

### Description
`methods class_name` displays the names of the methods for the class with the name `class_name`.  

```
n = methods('class_name')` returns the method names in a cell array of strings.
```

### See Also
- `help`
- `what`
- `which`

Online help for MATLAB functions and M-files
List M-, MAT- and MEX-files
Locate functions and files
**mexext**

**Purpose**  
Return the MEX-filename extension

**Syntax**  
\[ \text{ext} = \text{mexext} \]

**Description**  
\[ \text{ext} = \text{mexext} \] returns the filename extension for the current platform.
**Purpose**  
The name of the currently running M-file

**Syntax**  
mfilename

**Description**  
mfilename returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.

When called from the command line, mfilename returns an empty matrix.
**Purpose**
Minimum elements of an array

**Syntax**

\[ C = \min(A) \]

\[ C = \min(A,B) \]

\[ C = \min(A,[],\dim) \]

\[ [C,I] = \min(...) \]

**Description**

\[ C = \min(A) \] returns the smallest elements along different dimensions of an array.

If \( A \) is a vector, \( \min(A) \) returns the smallest element in \( A \).

If \( A \) is a matrix, \( \min(A) \) treats the columns of \( A \) as vectors, returning a row vector containing the minimum element from each column.

If \( A \) is a multidimensional array, \( \min \) operates along the first nonsingleton dimension.

\[ C = \min(A,B) \] returns an array the same size as \( A \) and \( B \) with the smallest elements taken from \( A \) or \( B \).

\[ C = \min(A,[],\dim) \] returns the smallest elements along the dimension of \( A \) specified by scalar \( \dim \). For example, \( \min(A,[],1) \) produces the minimum values along the first dimension (the rows) of \( A \).

\[ [C,I] = \min(...) \] finds the indices of the minimum values of \( A \), and returns them in output vector \( I \). If there are several identical minimum values, the index of the first one found is returned.

**Remarks**

For complex input \( A \), \( \min \) returns the complex number with the smallest modulus, computed with \( \min(\text{abs}(A)) \). The \( \min \) function ignores NaNs.

**See Also**

- \textit{max} Maximum elements of an array
- \textit{mean} Average or mean values of array
- \textit{median} Median values of array
- \textit{sort} Sort elements in ascending order
Purpose
Modulus (signed remainder after division)

Syntax
M = mod(X, Y)

Definition
mod(x, y) is \( x \mod y \).

Description
\( M = \text{mod}(X, Y) \) returns the remainder \( X - Y \cdot \text{floor}(X/Y) \) for nonzero \( Y \), and returns \( X \) otherwise. \( \text{mod}(X, Y) \) always differs from \( X \) by a multiple of \( Y \).

Remarks
So long as operands \( X \) and \( Y \) are of the same sign, the function \( \text{mod}(X, Y) \) returns the same result as \( \text{rem}(X, Y) \). 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 \( \text{mod}(x, m) \equiv \text{mod}(y, m) \).

Examples
\[
\text{mod}(13, 5)
\]
ans =
3

\[
\text{mod}([1:5], 3)
\]
ans =
1 2 0 1 2

\[
\text{mod(magic(3), 3)}
\]
ans =
2 1 0
0 2 1
1 0 2

Limitations
Arguments \( X \) and \( Y \) should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also
rem Remainder after division
**more**

**Purpose**
Control paged output for the command window

**Syntax**
- `more off`
- `more on`
- `more(n)`

**Description**
- `more off` disables paging of the output in the MATLAB command window.
- `more on` enables paging of the output in the MATLAB command window.
- `more(n)` displays `n` lines per page.

When you've enabled `more` and are examining output:

<table>
<thead>
<tr>
<th>Press the...</th>
<th>To...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return key</td>
<td>Advance to the next line of output.</td>
</tr>
<tr>
<td>Space bar</td>
<td>Advance to the next page of output.</td>
</tr>
<tr>
<td>q (for quit) key</td>
<td>Terminate display of the text.</td>
</tr>
</tbody>
</table>

By default, `more` is disabled. When enabled, `more` defaults to displaying 23 lines per page.

**See Also**
- `diary` Save session in a disk file
### Purpose
Mu-law to linear conversion

### Syntax
```matlab
y = mu2lin(mu)
```

### Description
The function `y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as "flints" in the range $0 \leq \mu \leq 255$, to linear signal amplitude in the range $-s < Y < s$ where $s = 32124/32768 \approx 0.9803$. The input `mu` is often obtained using `fread(., 'uchar')` to read byte-encoded audio files. "Flints" are MATLAB's integers - floating-point numbers whose values are integers.

### See Also
- `auread`: Read NeXT/SUN (.au) sound file
- `lin2mu`: Linear to mu-law conversion
**NaN**

**Purpose**
Not-a-Number

**Syntax**
NaN

**Description**
NaN returns the IEEE arithmetic representation for Not-a-Number (NaN). These result from operations which have undefined numerical results.

**Examples**
These operations produce NaN:

- Any arithmetic operation on a NaN, such as sqrt(NaN)
- Addition or subtraction, such as magnitude subtraction of infinities as (-Inf) - (-1Inf)
- Multiplication, such as 0*Inf
- Division, such as 0/0 and Inf/Inf
- Remainder, such as rem(x, y) where y is zero or x is infinity

**Remarks**
Logical operations involving NaNs always return false, except ~=(not equal). Consequently, the statement NaN ~= NaN is true while the statement NaN == NaN is false.

**See Also**
Inf

Infinity
Purpose
Check number of input arguments

Syntax
msg = nargchk(low, high, number)

Description
The nargchk function often is used inside an M-file to check that the correct number of arguments have been passed.

msg = nargchk(low, high, number) returns an error message if number is less than low or greater than high. If number is between low and high (inclusive), nargchk returns an empty matrix.

Arguments
- low, high: The minimum and maximum number of input arguments that should be passed.
- number: The number of arguments actually passed, as determined by the nargin function.

Examples
Given the function foo:

function f = foo(x, y, z)
error(nargchk(2, 3, nargin))

Then typing foo(1) produces:

    Not enough input arguments.

See Also
nargin, nargout: Number of function arguments
**Purpose**

Number of function arguments

**Syntax**

\[
\begin{align*}
    n & = \text{nargin} \\
    n & = \text{nargin('fun')} \\
    n & = \text{nargout} \\
    n & = \text{nargout('fun')} \\
\end{align*}
\]

**Description**

In the body of a function M-file, \text{nargin} and \text{nargout} indicate how many input or output arguments, respectively, a user has supplied. Outside the body of a function M-file, \text{nargin} and \text{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.

\text{nargin} returns the number of input arguments specified for a function.

\text{nargin('fun')} returns the number of declared inputs for the M-file function \text{fun} or \(-1\) if the function has a variable of input arguments.

\text{nargout} returns the number of output arguments specified for a function.

\text{nargout('fun')} returns the number of declared outputs for the M-file function \text{fun}.

**Examples**

This example shows portions of the code for a function called \text{myplot}, which accepts an optional number of input and output arguments:

```plaintext
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;
```

2-458
nargin, nargout

```matlab
    y0 = y;
end

See Also

inputname    Input argument name
nargchk      Check number of input arguments
```
nchoosek

Purpose
All combinations of the n elements in v taken k at a time

Syntax
C = nchoosek(v, k)

Description
C = nchoosek(v, k), where v is a row vector of length n, creates a matrix
whose rows consist of all possible combinations of the n elements of v taken k
at a time. Matrix C contains n! / ((n–k)! k!) rows and k columns.

Examples
The command nchoosek(2:2:10, 4) returns the even numbers from two to ten,
taken four at a time:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

Limitations
This function is only practical for situations where n is less than about 15.

See Also
perms All possible permutations
Purpose
Generate arrays for multidimensional functions and interpolation

Syntax
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
[X1,X2,...] = ndgrid(x)

Description
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...) transforms the domain specified by vectors x1,x2,x3... into arrays X1,X2,X3... that can be used for the evaluation of functions of multiple variables and multidimensional interpolation. The i th dimension of the output array X i are copies of elements of the vector xi.

[X1,X2,...] = ndgrid(x) is the same as [X1,X2,...] = ndgrid(x,x,...).

Examples
To evaluate the function \( x_1 e^{-x_1^2-x_2^2} \) over the range \(-2 < x_1 < 2; -2 < x_2 < 2\):

\[
[X1,X2] = ndgrid([-2:.2:2, -2:.2:2]);
Z = X1 .* exp(-X1.^2 - X2.^2);
mesh(Z)
\]

Remarks
The ndgrid function is like meshgrid except that the order of the first two input arguments are switched. That is, the statement

[X1,X2,X3] = ndgrid(x1,x2,x3)

produces the same result as

[X2,X1,X3] = meshgrid(x2,x1,x3).

Because of this, ndgrid is better suited to multidimensional problems that aren't spatially based, while meshgrid is better suited to problems in two- or three-dimensional Cartesian space.

See Also
meshgrid Generate X and Y matrices for three-dimensional plots
interpn Multidimensional data interpolation (table lookup).
**Purpose**
Number of array dimensions

**Syntax**
\[ n = 	ext{ndims}(A) \]

**Description**
\[ n = \text{ndims}(A) \]
returns the number of dimensions in the array \( A \). The number of dimensions in an array is always greater than or equal to 2. Trailing singleton dimensions are ignored. A singleton dimension is any dimension for which \( \text{size}(A, \text{dim}) = 1 \).

**Algorithm**
\[ \text{ndims}(x) = \text{length}(	ext{size}(x)). \]

**See Also**
\[ \text{size} \]
Array dimensions
Purpose

Next power of two

Syntax

\[
p = \text{nextpow2}(A)
\]

Description

\[
p = \text{nextpow2}(A)
\]
returns the smallest power of two that is greater than or equal to the absolute value of \(A\). (That is, \(p\) that satisfies \(2^p \geq \text{abs}(A)\).)

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

If \(A\) is non-scalar, \text{nextpow2} returns the smallest power of two greater than or equal to \(\text{length}(A)\).

Examples

For any integer \(n\) in the range from 513 to 1024, \text{nextpow2}(n) is 10.

For a 1-by-30 vector \(A\), \text{length}(A) is 30 and \text{nextpow2}(A) is 5.

See Also

- \texttt{fft} One-dimensional fast Fourier transform
- \texttt{log2} Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
- \texttt{pow2} Base 2 power and scale floating-point numbers
nnls

Purpose
Nonnegative least squares

Syntax
\[ x = \text{nnls}(A, b) \]
\[ x = \text{nnls}(A, b, \text{tol}) \]
\[ [x, w] = \text{nnls}(A, b) \]
\[ [x, w] = \text{nnls}(A, b, \text{tol}) \]

Description
\( x = \text{nnls}(A, b) \) solves the system of equations \( Ax = b \) in a least squares sense, subject to the constraint that the solution vector \( x \) has nonnegative elements: \( x_j \geq 0, \ j = 1, 2, \ldots n \). The solution \( x \) minimizes \( \|Ax - b\| \) subject to \( x \geq 0 \).

\( x = \text{nnls}(A, b, \text{tol}) \) solves the system of equations, and specifies a tolerance \( \text{tol} \). By default, \( \text{tol} \) is: \( \max(\text{size}(A)) \times \text{norm}(A, 1) \times \text{eps} \).

\([x, w] = \text{nnls}(A, b)\) also returns the dual vector \( w \), where \( w_i \leq 0 \) when \( x_i = 0 \) and \( w_i \equiv 0 \) when \( x_i > 0 \).

\([x, w] = \text{nnls}(A, b, \text{tol})\) solves the system of equations, returns the dual vector \( w \) and specifies a tolerance \( \text{tol} \).

Examples
Compare the unconstrained least squares solution to the \text{nnls} solution for a 4-by-2 problem:

\[
A =
\begin{bmatrix}
0. 0372 & 0. 2869 \\
0. 6861 & 0. 7071 \\
0. 6233 & 0. 6245 \\
0. 6344 & 0. 6170
\end{bmatrix}
\]

\[
b =
\begin{bmatrix}
0. 8587 \\
0. 1781 \\
0. 0747 \\
0. 8405
\end{bmatrix}
\]

\[
[ A; b, \text{nnls}(A, b)] =
\begin{bmatrix}
-2. 5627 & 0 \\
3. 1108 & 0. 6929
\end{bmatrix}
\]
\[
\begin{bmatrix}
\text{norm}(A(a \backslash b) - b) & \text{norm}(A\text{nnls}(a, b) - b)
\end{bmatrix} = 
\begin{bmatrix}
0.6674 & 0.9118
\end{bmatrix}
\]

The solution from \text{nnls} does not fit as well, but has no negative components.

**Algorithm**

The \text{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**

nnz

Purpose
Number of nonzero matrix elements

Syntax
\[ n = \text{nnz}(X) \]

Description
\[ n = \text{nnz}(X) \] returns the number of nonzero elements in matrix \( X \).
The density of a sparse matrix is \( \text{nnz}(X) / \text{prod(size}(X)) \).

Examples
The matrix
\[
\text{w} = \text{sparse(wilkinson(21))};
\]
is a tridiagonal matrix with 20 nonzeros on each of three diagonals, so \( \text{nnz}(w) = 60 \).

See Also
find Find indices and values of nonzero elements
nonzeros Nonzero matrix elements
nzmax Amount of storage allocated for nonzero matrix elements
size Array dimensions
whos List directory of variables in memory
isa Detect an object of a given class
Purpose
Nonzero matrix elements

Syntax
s = nonzeros(A)

Description
s = nonzeros(A) returns a full column vector of the nonzero elements in A, ordered by columns.

This gives the s, but not the i and j, from [i, j, s] = find(A). Generally,

length(s) = nnz(A) ≤ nzmax(A) ≤ prod(size(A))

See Also
find    Find indices and values of nonzero elements
nnz    Number of nonzero matrix elements
nzmax    Amount of storage allocated for nonzero matrix elements
size    Array dimensions
whos    List directory of variables in memory
isa    Detect an object of a given class
**Purpose**

Vector and matrix norms

**Syntax**

\[
\begin{align*}
n &= \text{norm}(A) \\
n &= \text{norm}(A, p)
\end{align*}
\]

**Description**

The norm of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

\[
n = \text{norm}(A) \quad \text{returns the largest singular value of } A, \quad \max(\text{svd}(A)).
\]

\[
n = \text{norm}(A, p) \quad \text{returns a different kind of norm, depending on the value of } p:
\]

| If \( p \) is... | Then `norm` returns...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The 1-norm, or largest column sum of ( A ), ( \max(\text{sum(abs(A))}) ).</td>
</tr>
<tr>
<td>2</td>
<td>The largest singular value (same as <code>norm(A)</code>).</td>
</tr>
<tr>
<td><code>inf</code></td>
<td>The infinity norm, or largest row sum of ( A ), ( \max(\text{sum(abs(A'))}) ).</td>
</tr>
<tr>
<td><code>'fro'</code></td>
<td>The Frobenius-norm of matrix ( A ), ( \sqrt{\text{sum(diag(A'*A))}} ).</td>
</tr>
</tbody>
</table>

When \( A \) is a vector, slightly different rules apply:

\[
\begin{align*}
\text{norm}(A, p) & \quad \text{Returns } \text{sum(abs(A).}^p)^{1/p}, \text{for any } 1 \leq p \leq \infty. \\
\text{norm}(A) & \quad \text{Returns } \text{norm}(A, 2). \\
\text{norm}(A, \text{inf}) & \quad \text{Returns } \max(\text{abs}(A)). \\
\text{norm}(A, -\text{inf}) & \quad \text{Returns } \min(\text{abs}(A)).
\end{align*}
\]

**Remarks**

To obtain the root-mean-square (RMS) value, use `norm(A)/sqrt(n)`. Note that `norm(A)` where \( A \) is an \( n \)-element vector, is the length of \( A \).

**See Also**

| `cond` | Condition number with respect to inversion |
| `normest` | 2-norm estimate |
| `svd` | Singular value decomposition |

2-468
Purpose

2-norm estimate

Syntax

\[ \text{nrm} = \text{normest}(S) \]
\[ \text{nrm} = \text{normest}(S, \text{tol}) \]
\[ [\text{nrm count}] = \text{normest}(...) \]

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

\[ \text{nrm} = \text{normest}(S) \] returns an estimate of the 2-norm of the matrix \( S \).

\[ \text{nrm} = \text{normest}(S, \text{tol}) \] uses relative error \( \text{tol} \) instead of the default tolerance \( 1.e^{-6} \). The value of \( \text{tol} \) determines when the estimate is considered acceptable.

\[ [\text{nrm count}] = \text{normest}(...) \] returns an estimate of the 2-norm and also gives the number of power iterations used.

Examples

The matrix \( W = \text{gallery}('\text{wilkinson}', 101) \) is a tridiagonal matrix. Its order, 101, is small enough that \( \text{norm}(\text{full}(W)) \), which involves \( \text{svd}(\text{full}(W)) \), is feasible. The computation takes 4.13 seconds (on one computer) and produces the exact norm, 50.7462. On the other hand, \( \text{normest}(\text{sparse}(W)) \) 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

\text{cond} \quad \text{Condition number with respect to inversion}
\text{condest} \quad 1\text{-norm matrix condition number estimate}
\text{norm} \quad \text{Vector and matrix norms}
\text{svd} \quad \text{Singular value decomposition}
**Purpose**
Current date and time

**Syntax**
\[ t = \text{now} \]

**Description**
\( t = \text{now} \) returns the current date and time as a serial date number. To return the time only, use \( \text{rem}(\text{now}, 1) \). To return the date only, use \( \text{floor}(\text{now}) \).

**Examples**
\[ t_1 = \text{now}, \quad t_2 = \text{rem}(\text{now}, 1) \]

\[ t_1 = \]
\[ 7.2908e+05 \]

\[ t_2 = \]
\[ 0.4013 \]

**See Also**
- \( \text{clock} \) Current time as a date vector
- \( \text{date} \) Current date string
- \( \text{datenum} \) Serial date number
Purpose
Null space of a matrix

Syntax
B = null(A)

Description
B = null(A) returns an orthonormal basis for the null space of A.

Remarks
B'*B = I, A*B has negligible elements, and (if B is not equal to the empty matrix) the number of columns of B is the nullity of A.

See Also
orth
Range space of a matrix
qr
Orthogonal-triangular decomposition
svd
Singular value decomposition
num2cell

**Purpose**

Convert a numeric array into a cell array

**Syntax**

\[ c = \text{num2cell}(A) \]
\[ c = \text{num2cell}(A, \text{dims}) \]

**Description**

\[ c = \text{num2cell}(A) \] converts the matrix \( A \) into a cell array by placing each element of \( A \) into a separate cell. Cell array \( c \) will be the same size as matrix \( A \).

\[ c = \text{num2cell}(A, \text{dims}) \] converts the matrix \( A \) into a cell array by placing the dimensions specified by \( \text{dims} \) into separate cells. \( C \) will be the same size as \( A \) except that the dimensions matching \( \text{dims} \) will be 1.

**Examples**

The statement

\[ \text{num2cell}(A, 2) \]

places the rows of \( A \) into separate cells. Similarly

\[ \text{num2cell}(A, [1 3]) \]

places the column-depth pages of \( A \) into separate cells.

**See Also**

cat  
Concatenate arrays
num2str

**Purpose**
Number to string conversion

**Syntax**

```
str = num2str(A)
str = num2str(A, precision)
str = num2str(A, format)
```

**Description**
The `num2str` function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.

- `str = num2str(a)` converts array `A` into a string representation `str` with roughly four digits of precision and an exponent if required.
- `str = num2str(a, precision)` converts the array `A` into a string representation `str` with maximum precision specified by `precision`. Argument `precision` specifies the number of digits the output string is to contain. The default is four.
- `str = num2str(A, format)` converts array `A` using the supplied `format`. By default, this is `'%11.4g'`, which signifies four significant digits in exponential or fixed-point notation, whichever is shorter. (See `fprintf` for format string details).

**Examples**

```matlab
num2str(pi) is 3.142.
num2str(eps) is 2.22e-16.
num2str(magic(2)) produces the string matrix
     1   3
     4   2
```

**See Also**

- `fprintf` Write formatted data to file
- `int2str` Integer to string conversion
- `sprintf` Write formatted data to a string
**Purpose**
Amount of storage allocated for nonzero matrix elements

**Syntax**

\[ n = nzmax(S) \]

**Description**

\( n = nzmax(S) \) returns the amount of storage allocated for nonzero elements.

If \( S \) is a sparse matrix...

\[ \text{nzm}(S) \text{ is the number of storage locations allocated for the nonzero elements in } S. \]

If \( S \) is a full matrix...

\[ \text{nzm}(S) = \text{prod(size}(S)) . \]

Often, \( nnz(S) \) and \( nzmax(S) \) are the same. But if \( S \) is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and \( nzmax(S) \) reflects this. Alternatively, \( \text{sparse}(i, j, s, m, n, nzm) \) or its simpler form, \( \text{spalloc}(m, n, nzm) \), can set \( nzmax \) in anticipation of later fill-in.

**See Also**

- \text{find} Find indices and values of nonzero elements
- \text{nnz} Number of nonzero matrix elements
- \text{nonzeros} Nonzero matrix elements
- \text{size} Array dimensions
- \text{whos} List directory of variables in memory
- \text{isa} Detect an object of a given class
ode45, ode23, ode113, ode15s, ode23s

Purpose
Solve differential equations

Syntax

\[
\begin{align*}
[T, Y] &= \text{solver('F', tspan, y0)} \\
[T, Y] &= \text{solver('F', tspan, y0, options)} \\
[T, Y] &= \text{solver('F', tspan, y0, options, p1, p2...)} \\
[T, Y, TE, YE, IE] &= \text{solver('F', tspan, y0, options)} \\
[T, X, Y] &= \text{solver('model', tspan, y0, options, ut, p1, p2, ...)}
\end{align*}
\]

Arguments

- **F**: Name of the ODE file, a MATLAB function of \( t \) and \( y \) returning a column vector. All solvers can solve systems of equations in the form \( y' = F(t,y) \). ode15s and ode23s can both solve equations of the form \( M y' = F(t,y) \). Only ode15s can solve equations in the form \( M(t)y' = F(t,y) \). For information about ODE file syntax, see the odefile reference page.
- **tspan**: A vector specifying the interval of integration \([t_0 \text{ to } tfinal]\). To obtain solutions at specific times (all increasing or all decreasing), use \( tspan = [t_0,t_1, ... , tfinal] \).
- **y0**: A vector of initial conditions.
- **options**: Optional integration argument created using the odeset function. See odeset for details.
- **p1, p2...**: Optional parameters to be passed to \( F \).
- **T, Y**: Solution matrix \( Y \), where each row corresponds to a time returned in column vector \( T \).

Description

\([T, Y] = \text{solver('F', tspan, y0)} \) with \( tspan = [t_0 \text{ to } tfinal] \) integrates the system of differential equations \( y' = F(t,y) \) from time \( t_0 \) to \( tfinal \) with initial conditions \( y_0 \). \( F' \) is a string containing the name of an ODE file. Function \( F(t,y) \) must return a column vector. Each row in solution array \( Y \) corresponds to a time returned in column vector \( T \). To obtain solutions at the specific times \( t_0, t_1, ..., tfinal \) (all increasing or all decreasing), use \( tspan = [t_0 \ t_1 \ ... \ tfinal] \).

\([T, Y] = \text{solver('F', tspan, y0, options)} \) solves as above with default integration parameters replaced by property values specified in \( options \), an argument created with the odeset function (see odeset for details). Commonly used
ode45, ode23, ode113, ode15s, ode23s

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).

\[ [T, Y] = \text{solver}('F', t\text{span}, y0, \text{options}, p1, p2...) \]
solves as above, passing the additional parameters p1, p2... to the M-file F, whenever it is called. Use options = [ ] as a placeholder if no options are set.

\[ [T, Y, TE, YE, IE] = \text{solver}('F', t\text{span}, y0, \text{options}) \]
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 \( \frac{\partial F}{\partial y} \) is critical to reliability and efficiency so there are special options. Set JConstant to 'on' if \( \frac{\partial F}{\partial y} \) is constant. Set Vectorized to 'on' if the ODE file is coded so that F(t, [y1 y2 ...]) returns \([F(t, y1) F(t, y2) ...]\). Set Jattern to 'on' if \( \frac{\partial F}{\partial y} \) is a sparse matrix and the ODE file is coded so that F([], [], 'jpattern') returns a sparsity pattern matrix of 1's and 0's showing the nonzeros of \( \frac{\partial F}{\partial y} \). Set Jacobian to 'on' if the ODE file is coded so that F(t, y, 'jacobian') returns \( \frac{\partial F}{\partial y} \).

Both ode15s and ode23s can solve problems \( M' = F(t, y) \) with a constant mass matrix M that is nonsingular and (usually) sparse. Set Mass to 'on' if the ODE file is coded so that F([], [], 'mass') returns M (see fem2ode). Only ode15s can solve problems M(t)y' = F(t, y) with a time-dependent mass matrix M(t) that is nonsingular and (usually) sparse. Set Mass to 'on' if the ODE file is coded so that F(t, [], 'mass') returns M(t) (see fem1ode). For ode15s set MassConstant to 'on' if M is constant.
ode45, ode23, ode113, ode15s, ode23s

<table>
<thead>
<tr>
<th>Solver</th>
<th>Problem Type</th>
<th>Order of Accuracy</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode45</td>
<td>Nonstiff</td>
<td>Medium</td>
<td>Most of the time. This should be the first solver you try.</td>
</tr>
<tr>
<td>ode23</td>
<td>Nonstiff</td>
<td>Low</td>
<td>If using crude error tolerances or solving moderately stiff problems.</td>
</tr>
<tr>
<td>ode113</td>
<td>Nonstiff</td>
<td>Low to high</td>
<td>If using stringent error tolerances or solving a computationally intensive ODE file.</td>
</tr>
<tr>
<td>ode15s</td>
<td>Stiff</td>
<td>Low to medium</td>
<td>If ode45 is slow (stiff systems) or there is a mass matrix.</td>
</tr>
<tr>
<td>ode23s</td>
<td>Stiff</td>
<td>Low</td>
<td>If using crude error tolerances to solve stiff systems or there is a constant mass matrix.</td>
</tr>
</tbody>
</table>

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 2-480 for more details.

It is possible to specify tspan, y0 and options in the ODE file (see odefile). If tspan or y0 is empty, then the solver calls the ODE file:

\[
[tspan, y0, options] = F([], [], 'init')
\]

to obtain any values not supplied in the solver’s argument list. Empty arguments at the end of the call list may be omitted. This permits you to call the solvers with other syntaxes such as:

\[
[T, Y] = solver('F')
\]

\[
[T, Y] = solver('F', [], y0)
\]

\[
[T, Y] = solver('F', tspan, [], options)
\]

\[
[T, Y] = solver('F', [], [], options)
\]

Integration parameters (options) can be specified both in the ODE file and on the command line. If an option is specified in both places, the command line specification takes precedence. For information about constructing an ODE file, see the odefile reference page.
ode45, ode23, ode113, ode15s, ode23s

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and Using MATLAB.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ode45</th>
<th>ode23</th>
<th>ode113</th>
<th>ode15s</th>
<th>ode23s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel Tol, AbsTol</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>OutputFcn, OutputSel, Refine, Stats</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Events</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>MaxStep, InitialStep</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>J Constant, Jacobian, Pattern, Vectorized</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Mass</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>MassConstant</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>—</td>
</tr>
<tr>
<td>MaxOrder, BDF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>—</td>
</tr>
</tbody>
</table>

Examples

Example 1. An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces:

\[
\begin{align*}
    y_1' &= y_2 y_3 \quad y_1(0) = 0 \\
    y_2' &= -y_1 y_3 \quad y_2(0) = 1 \\
    y_3' &= -0.51 y_1 y_2 \quad y_3(0) = 1
\end{align*}
\]

To simulate this system, create a function M-file `rigid` containing the equations:

```matlab
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);
```

2-478
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.

```matlab
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:

```matlab
plot(T, Y(:, 1), '-', T, Y(:, 2), '-.', T, Y(:, 3), '.')
```

**Example 2.** An example of a stiff system is provided by the van der Pol equations governing relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

\[
\begin{align*}
y'_{1} &= y_{2} \\
y'_{2} &= 1000(1 - y_{1}^2)y_{2} - y_{1} \\
y_{1}(0) &= 0 \\
y_{2}(0) &= 1
\end{align*}
\]
ode45, ode23, ode113, ode15s, ode23s

To simulate this system, create a function M-file `vdp1000` containing the equations:

```matlab
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.

```matlab
[T,Y] = ode15s('vdp1000',[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution:

```matlab
plot(T,Y(:,1),'-o');
```

Algorithms

ode45 is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver – in computing \( y(t_n) \), it needs only the solution at the immediately preceding time point, \( y(t_{n-1}) \). In general, ode45 is the best function to apply as a “first try” for most problems. [1]
ode23 is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of moderate stiffness. Like ode45, ode23 is a one-step solver. [2]

ode113 is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances and when the ODE file function is particularly expensive to evaluate. ode113 is a multistep solver - it normally needs the solutions at several preceding time points to compute the current solution. [3]

The above algorithms are intended to solve non-stiff systems. If they appear to be unduly slow, try using one of the stiff solvers (ode15s and ode23s) instead.

ode15s is a variable order solver based on the numerical differentiation formulas, NDFs. Optionally, it uses the backward differentiation formulas, BDFs (also known as Gear’s method) that are usually less efficient. Like ode113, ode15s is a multistep solver. If you suspect that a problem is stiff or if ode45 has failed or was very inefficient, try ode15s. [7]

ode23s is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective. [7]

See Also
odeset, odeget, odefile

References
ode45, ode23, ode113, ode15s, ode23s


**Purpose**

Define a differential equation problem for ODE solvers

**Description**

*odefile* is not a command or function. It is a help entry that describes how to create an M-file defining the system of equations to be solved. This definition is the first step in using any of MATLAB's ODE solvers. In MATLAB documentation, this M-file is referred to as *odefile*, although you can give your M-file any name you like.

You can use the *odefile* M-file to define a system of differential equations in one of these forms:

\[ y' = F(t, y) \]
\[ My' = F(t, y) \]
\[ M(t)y' = F(t, y) \]

where:

- \( t \) is a scalar independent variable, typically representing time.
- \( y \) is a vector of dependent variables.
- \( F \) is a function of \( t \) and \( y \) returning a column vector the same length as \( y \).
- \( M \) and \( M(t) \) represent nonsingular constant or time dependent mass matrices.

The ODE file must accept the arguments \( t \) and \( y \), although it does not have to use them. By default, the ODE file must return a column vector the same length as \( y \).

Only the stiff solver *ode15s* can solve \( M(t)y' = F(t, y) \). Both *ode15s* and *ode23s* can solve equations of the form \( My' = F(t, y) \).

Beyond defining a system of differential equations, you can specify an entire initial value problem (IVP) within the ODE M-file, eliminating the need to supply time and initial value vectors at the command line (see Examples on page 2-486).
To use the ODE file template:

- Enter the command `help odefile` to display the help entry.
- Cut and paste the ODE file text into a separate file.
- Edit the file to eliminate any cases not applicable to your IVP.
- Insert the appropriate information where indicated. The definition of the ODE system is required information. (See item 2 as well as Examples on page 2-486). Here is an annotated version of the result:

```matlab
function [out1, out2, out3] = odefile(t, y, flag, p1, p2) % ODEFILE The template for ODE files.
if nargin < 3 | isempty(flag) % Return dy/dt = F(t,y)
    out1 = < Insert a function of t and/or y, p1, and p2 here >;
else
    switch(flag) % Return default [tspan, y0, and options]
        case 'init' % Return default [tspan, y0, and options]
            out1 = < Insert tspan here >;
            out2 = < Insert y0 here >;
            out3 = < Insert options = odeset(...) or [] here >;
        case 'jacobian' % Return matrix J(t,y) = dF/dy
            out1 = < Insert Jacobian matrix here >;
        case 'jpattern' % Return sparsity pattern matrix S
            out1 = < Insert Jacobian matrix sparsity pattern here >;
        case 'mass' % Return mass matrix M(t) or M
            out1 = < Insert mass matrix here >;
        case 'events' % Return event vector and info
            out1 = < Insert event function vector here >;
            out2 = < Insert logical isterminal vector here >;
            out3 = < Insert direction vector here >;
        otherwise
            error(['Unknown flag'' flag''.'']);
    end
end
```
Notes

1 The ODE file must accept $t$ and $y$ vectors from the ODE solvers and must return a column vector the same length as $y$. The optional input argument \texttt{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 \texttt{switch} statement determines the type of output required, so that the ODE file can pass the appropriate information to the solver. (See steps 4 - 9.)

4 In the default initial conditions ('init') case, the ODE file returns basic information (time span, initial conditions, options) to the solver. If you omit this case, you must supply all the basic information on the command line.

5 In the 'jacobian' case, the ODE file returns a Jacobian matrix to the solver. You need only provide this case when you wish to improve the performance of the stiff solvers \texttt{ode15s} and \texttt{ode23s}.

6 In the 'jpattern' case, the ODE file returns the Jacobian sparsity pattern matrix to the solver. You need provide this case only when you want to generate sparse Jacobian matrices numerically for a stiff solver.

7 In the 'mass' case, the ODE file returns a mass matrix to the solver. You need provide this case only when you want to solve a system in either of the forms $M\frac{dy}{dt} = F(t,y)$ or $M(t)\frac{dy}{dt} = F(t,y)$.

8 In the 'events' case, the ODE file returns to the solver the values that it needs to perform event location. When the \texttt{Events} property is set to 1, the ODE solvers examine any elements of the \texttt{event} vector for transitions to, from, or through zero. If the corresponding element of the logical \texttt{isterminal} vector is set to 1, integration will halt when a zero-crossing is detected. The elements of the \texttt{direction} vector are -1, 1, or 0, specifying that the corresponding event must be decreasing, increasing, or that any crossing is to be detected. See the Applying MATLAB and also the examples \texttt{ballode} and \texttt{orbitode}.

9 An unrecognized \texttt{flag} generates an error.
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:

\[
\begin{align*}
    y'_1 &= y_2 \\
    y'_2 &= \mu (1 - y_1^2)y_2 - y_1
\end{align*}
\]

The M-file

```matlab
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:

```matlab
[t, y] = ode45('vdp1', [0 20], [2; 0]);
plot(t, y(:,1), '-', t, y(:,2), '-.')
```

![Graph of van der Pol equation solution](image)
To specify the entire initial value problem (IVP) within the M-file, rewrite `vdp1` as follows:

```matlab
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:

```matlab
[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:

```matlab
[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`, `brusode`, `vdpode`, or `biode`, and `rigode`. Use `type filename` from the MATLAB command line to see the coding for a specific ODE file.

**See Also**

The Applying MATLAB and the reference entries for the ODE solvers and their associated functions:

`ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `odeget`, `odeset`
odeget

Purpose
Extract properties from options structure created with odeset

Syntax

o = odeget(options,'name')
o = odeget(options,'name',default)

Description

o = odeget(options,'name') extracts the value of the property specified by string 'name' from integrator options structure options, returning an empty matrix if the property value is not specified in options. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix [ ] is a valid options argument.

o = odeget(options,'name',default) returns o = default if the named property is not specified in options.

Example
Having constructed an ODE options structure,

options = odeset('RelTol',1e–4,'AbsTol',[1e–3 2e–3 3e–3]);

you can view these property settings with odeget:

odeget(options,'RelTol')
ans =

1.0000e-04

odeget(options,'AbsTol')
ans =

0.0010 0.0020 0.0030

See Also
odeset
Purpose
Create or alter options structure for input to ODE solvers

Syntax
options = odeset('name1',value1,'name2',value2,...)
options = odeset(oldopts,'name1',value1,...)
options = odeset(oldopts,newopts)

Description
The odeset function lets you adjust the integration parameters of the ODE solvers. See below for information about the integration parameters.

options = odeset('name1',value1,'name2',value2,...) creates an integrator options structure in which the named properties have the specified values. The odeset function sets any unspecified properties to the empty matrix [].

It is sufficient to type only the leading characters that uniquely identify the property name. Case is ignored for property names.

options = odeset(oldopts,'name1',value1,...) alters an existing options structure with the values supplied.

options = odeset(oldopts,newopts) alters an existing options structure oldopts by combining it with a new options structure newopts. Any new options not equal to the empty matrix overwrite corresponding options in oldopts. For example:

oldopts

| F | 1 | [] | 4 | 's' | 's' | [] | [] | [] | ...
|

newopts

| T | 3 | F | [] | '' | [] | [] | [] | [] | ...
|

odeset(oldopts,newopts)

| T | 3 | F | 4 | '' | 's' | [] | [] | [] | ...
|
odeset by itself, displays all property names and their possible values:

```matlab
odeset
    AbsTol: [ positive scalar or vector {1e–6} ]
    BDF: [ on | {off} ]
    Events: [ on | {off} ]
    InitialStep: [ positive scalar ]
    Jacobian: [ on | {off} ]
    JConstant: [ on | {off} ]
    JPattern: [ on | {off} ]
    Mass: [ on | {off} ]
    MassConstant: [ on | off ]
    MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
    MaxStep: [ positive scalar ]
    OutputFcn: [ string ]
    OutputSel: [ vector of integers ]
    Refine: [ positive integer ]
    RelTol: [ positive scalar {1e–3} ]
    Stats: [ on | {off} ]
    Vectorized: [ on | {off} ]
```

**Properties**

The available properties depend upon the ODE solver used. There are seven principal categories of properties:

- Error tolerance
- Solver output
- Jacobian
- Event location
- Mass matrix
- Step size
- ode15s
Table 1-1: Error Tolerance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel Tol</td>
<td>Positive scalar {1e-3}</td>
<td>A relative error tolerance that applies to all components of the solution vector.</td>
</tr>
<tr>
<td>Abs Tol</td>
<td>Positive scalar or vector {1e-6}</td>
<td>The absolute error tolerance. If scalar, the tolerance applies to all components of the solution vector. Otherwise the tolerances apply to corresponding components.</td>
</tr>
</tbody>
</table>

Table 1-2: Solver Output Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputFcn</td>
<td>String</td>
<td>The name of an installable output function (for example, odeplot, odephas2, odephas3, and odeprint). 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 <code>(T, Y)</code>. The <code>status</code> return value should be 1 if integration should be halted (e.g., a STOP button has been pressed) and 0 otherwise. When the integration is complete, the solver calls <code>outputfcn([],[],'done')</code>.</td>
</tr>
<tr>
<td>OutputSel</td>
<td>Vector of indices</td>
<td>Specifies which components of the solution vector are to be passed to the output function.</td>
</tr>
</tbody>
</table>
Refine: 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: Specifies whether statistics about the computational cost of the integration should be displayed.

### Table 1-2: Solver Output Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refine</td>
<td>Positive Integer</td>
<td>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.</td>
</tr>
<tr>
<td>Stats</td>
<td>on</td>
<td>{off}</td>
</tr>
</tbody>
</table>

### Table 1-3: Jacobian Matrix Properties (for ode15s and ode23s)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JConstant</td>
<td>on</td>
<td>{off}</td>
</tr>
<tr>
<td>Jacobian</td>
<td>on</td>
<td>{off}</td>
</tr>
<tr>
<td>JPattern</td>
<td>on</td>
<td>{off}</td>
</tr>
</tbody>
</table>
Vectorized on \{off\}

Informs the solver that the ODE file $F(t, y)$ has been vectorized so that $F(t, [y_1 \ y_2 \ ...])$ returns $[F(t, y_1) \ F(t, y_2) \ ...]$. 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 \texttt{odeset} has been used to set \texttt{Vectorized} to 'on'.

### Table 1-4: Event Location Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>on {off}</td>
<td>Instructs the solver to locate events. The ODE file must respond to the arguments $(t, y, \text{'events'})$ by returning the appropriate values. See \texttt{odefile}.</td>
</tr>
</tbody>
</table>

### Table 1-5: Mass Matrix Properties (for ode15s and ode23s)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mass</td>
<td>on {off}</td>
<td>Informs the solver that the ODE file is coded so that $F(t, [], \text{'mass'})$ returns $M$ or $M(t)$ (see \texttt{odefile}).</td>
</tr>
<tr>
<td>MassConstant</td>
<td>on {off}</td>
<td>Informs the solver that the mass matrix $M(t)$ is constant.</td>
</tr>
</tbody>
</table>
In addition there are two options that apply only to the ode15s solver.

**Table 1-6: Step Size Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxStep</td>
<td>Positive scalar</td>
<td>An upper bound on the magnitude of the step size that the solver uses.</td>
</tr>
<tr>
<td>InitialStep</td>
<td>Positive scalar</td>
<td>Suggested initial step size. The solver tries this first, but if too large an error results, the solver uses a smaller step size.</td>
</tr>
</tbody>
</table>

**Table 1-7: ode15s Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxOrder</td>
<td>1</td>
<td>The maximum order formula used.</td>
</tr>
<tr>
<td>BDF</td>
<td>on</td>
<td>Specifies whether the Backward Differentiation Formulas (BDF’s) are to be used instead of the default Numerical Differentiation Formulas (NDF’s).</td>
</tr>
</tbody>
</table>

**See Also**

code file, odeget, ode45, ode23, ode113, ode15s, ode23s
Purpose
Create an array of all ones

Syntax
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones([d1 d2 d3...])
Y = ones([d1 d2 d3...])
Y = ones(size(A))

Description
Y = ones(n) returns an n-by-n matrix of 1s. An error message appears if n is not a scalar.
Y = ones(m,n) or Y = ones([m n]) returns an m-by-n matrix of ones.
Y = ones(d1,d2,d3...) or Y = ones([d1 d2 d3...]) returns an array of 1s with dimensions d1-by-d2-by-d3-by...
Y = ones(size(A)) returns an array of 1s that is the same size as A.

See Also
eye Identity matrix
rand Uniformly distributed random numbers and arrays
randn Normally distributed random numbers and arrays
zeros Create an array of all zeros
**Purpose**  
Range space of a matrix

**Syntax**  
\[ B = \text{orth}(A) \]

**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**  
- `null`  
  Null space of a matrix  
- `svd`  
  Singular value decomposition  
- `rank`  
  Rank of a matrix
Purpose

Default part of switch statement

Description

otherwise is part of the switch statement syntax, which allows for conditional execution. The statements following otherwise are executed only if none of the preceding case expressions (case_expr) match the switch expression (sw_expr).

Examples

The general form of the switch statement is:

```
switch sw_expr
  case case_expr
    statement
    statement
  case {case_expr1, case_expr2, case_expr3}
    statement
    statement
  otherwise
    statement
    statement
end
```

See switch for more details.

See Also

switch Switch among several cases based on expression
otherwise
**Purpose**
Consolidate workspace memory

**Syntax**
pack
pack filename

**Description**
pack, by itself, frees up needed space by compressing information into the minimum memory required.

pack filename accepts an optional filename for the temporary file used to hold the variables. Otherwise it uses the file named pack.tmp.

**Remarks**
The pack command doesn't affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the Out of memory message from MATLAB, the pack command may find you some free memory without forcing you to delete variables.

The pack command frees space by:

- Saving all variables on disk in a temporary file called pack.tmp.
- Clearing all variables and functions from memory.
- Reloading the variables back from pack.tmp.
- Deleting the temporary file pack.tmp.
If you use pack and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, here are some system-specific tips:

- **MS-Windows**: Increase the swap space by opening the Control Panel, double-clicking on the 386 Enhanced icon, and pressing the **Virtual Memory** button.

- **Macintosh**: Change the application memory size by using **Get Info** on the program icon. You may also want to turn on virtual memory via the Memory Control Panel.

- **VAX/VMS**: Ask your system manager to increase your working set and/or pagefile quota.

- **UNIX**: Ask your system manager to increase your swap space.

**See Also**

- clear
  - Remove items from memory
Purpose
Partial pathname

Description
A partial pathname is a MATLABPATH relative pathname used to locate private and method files, which are usually hidden, or to restrict the search for files when more than one file with the given name exists.

A partial pathname contains the last component, or last several components, of the full pathname separated by / . For example, matfun/trace, private/children, inline/formula, and demos/clown.mat are valid partial pathnames. Specifying the @ in method directory names is optional, so funfun/inline/formula is also a valid partial pathname.

Partial pathnames make it easy to find toolbox or MATLAB relative files on your path in a portable way independent of the location where MATLAB is installed.
Purpose
Pascal matrix

Syntax
A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)

Description
A = pascal(n) returns the Pascal matrix of order n: a symmetric positive definite matrix with integer entries taken from Pascal’s triangle. The inverse of A has integer entries.

A = pascal(n,1) returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is involutary, that is, it is its own inverse.

A = pascal(n,2) returns a transposed and permuted version of pascal(n,1). A is a cube root of the identity matrix.

Examples
pascal(4) returns

\[
\begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20 \\
\end{pmatrix}
\]

A = pascal(3,2) produces

\[
A =
\begin{pmatrix}
0 & 0 & -1 \\
0 & -1 & 2 \\
-1 & -1 & 1 \\
\end{pmatrix}
\]

See Also
chol
Cholesky factorization
### Purpose
Control MATLAB’s directory search path

### Syntax
```matlab
pat h
p = pat h
pat h(‘newpat h’)
pat h(pat h,’newpat h’)
pat h(‘newpat h’,pat h)
```

### Description
`pat h` prints out the current setting of MATLAB’s search path. On all platforms except the Macintosh, the path resides in `pat h def .m` (in `toolbox/local`). The Macintosh stores its path in the `Matlab Settings File` (usually in the `Preferences` folder).

- `p = pat h` returns the current search path in string variable `p`.
- `pat h(‘newpat h’)` changes the path to the string ‘newpat h’.
- `pat h(pat h,’newpat h’)` appends a new directory to the current path.
- `pat h(‘newpat h’,pat h)` prepends a new directory to the current path.

### Remarks
MATLAB has a search path. If you enter a name, such as `fox`, the MATLAB interpreter:

1. Looks for `fox` as a variable.
2. Checks for `fox` as a built-in function.
3. Looks in the current directory for `fox.mex` and `fox.m`.
4. Searches the directories specified by `pat h` for `fox.mex` and `fox.m`.

### Examples
Add a new directory to the search path on various operating systems:

- **UNIX:**
  ```matlab
  pat h(pat h,’/home/myfriend/goodstuff’) 
  ```
- **VMS:**
  ```matlab
  pat h(pat h,’DISK1:[MYFRIEND.GOODSTUFF]’) 
  ```
- **MS-DOS:**
  ```matlab
  pat h(pat h,’TOOLS\GOODSTUFF’) 
  ```
- **Macintosh:**
  ```matlab
  pat h(pat h,’Tools:GoodStuff’) 
  ```
See Also

addpath    Add directories to MATLAB’s search path
            Change working directory
            Directory listing
            Remove directories from MATLAB’s search path
            Directory listing of M-files, MAT-files, and MEX-files
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Halt execution temporarily</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>pause</td>
</tr>
<tr>
<td></td>
<td>pause(n)</td>
</tr>
<tr>
<td></td>
<td>pause on</td>
</tr>
<tr>
<td></td>
<td>pause off</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>pause, by itself, causes M-files to stop and wait for you to press any key before continuing.</td>
</tr>
<tr>
<td></td>
<td>pause(n) pauses execution for n seconds before continuing.</td>
</tr>
<tr>
<td></td>
<td>pause on allows subsequent pause commands to pause execution.</td>
</tr>
<tr>
<td></td>
<td>pause off ensures that any subsequent pause or pause(n) statements do not pause execution. This allows normally interactive scripts to run unattended.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>The <code>drawnow</code> command in the MATLAB Graphics Guide</td>
</tr>
</tbody>
</table>
Preconditioned Conjugate Gradients method

\[ x = \text{pcg}(A, b) \]

\[ \text{pcg}(A, b, \text{tol}) \]

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}) \]

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}, M) \]

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2) \]

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ x = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ [x, \text{flag}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ [x, \text{flag}, \text{relres}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ [x, \text{flag}, \text{relres}, \text{iter}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ [x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \]

\[ x = \text{pcg}(A, b) \] 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 \( \|b-Ax\|/\|b\| \) 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.

\[ \text{pcg}(A, b, \text{tol}) \] specifies the tolerance of the method, \( \text{tol} \).

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}) \] additionally specifies the maximum number of iterations, \( \text{maxit} \).

\[ \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1) \] and \[ \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2) \] use left preconditioner \( M = M_1M_2 \) and effectively solve the system \( M^{-1}A^+x = M^{-1}b \) for \( x \). If \( M_1 \) or \( M_2 \) 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 \text{backslash} within \text{pcg}, it is wise to factor...
preconditioners into their Cholesky factors first. For example, replace \( \text{pcg}(A, b, \text{tol}, \text{maxit}, M) \) with:

\[
R = \text{chol}(M); \\
\text{pcg}(A, b, \text{tol}, \text{maxit}, R', R).
\]

The preconditioner \( M \) must be symmetric and positive definite.

\( \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \) specifies the initial estimate \( x_0 \). If \( x_0 \) is given as the empty matrix (\([\ ]\)), the default all zero vector is used.

\( x = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \) returns a solution \( x \). If \( \text{pcg} \) converged, a message to that effect is displayed. If \( \text{pcg} \) failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual \( \| b - A^* x \| / \| b \| \) and the iteration number at which the method stopped or failed.

\( [x, \text{flag}] = \text{pcg}(A, b, \text{tol}, \text{maxit}, M_1, M_2, x_0) \) returns a solution \( x \) and a flag which describes the convergence of \( \text{pcg} \):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{pcg} ) converged to the desired tolerance ( \text{tol} ) within ( \text{maxit} ) iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>( \text{pcg} ) iterated ( \text{maxit} ) times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>One of the systems of equations of the form ( M \cdot y = r ) involving the preconditioner was ill-conditioned and did not return a useable result when solved by ( \backslash ) (backslash).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during ( \text{pcg} ) became too small or too large to continue computing</td>
</tr>
</tbody>
</table>

Whenever \( \text{flag} \) is not 0, the solution \( x \) returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the \( \text{flag} \) output is specified.
[x, flag, relres] = pcg(A, b, tol, maxit, M1, M2, x0) also returns the relative residual norm(b–A*x)/norm(b). If flag is 0, then relres ≤ tol.

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

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

Examples

A = delsq(numgrid('C', 25))
b = ones(length(A), 1)
[x, flag] = pcg(A, b)

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

R = cholinc(A, 1e-3)
[x2, flag2, relres2, iter2, resvec2] = pcg(A, b, 1e-8, 10, R', R)

flag2 is 0 since pcg will converge to the tolerance of 1.2e-9 (the value of relres2) at the sixth iteration (the value of iter2) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of 1e-3. resvec2(1) = norm(b) and resvec2(7) = norm(b–A*x2). You may follow the progress of pcg.
by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with
\[
\text{semilogy}(0: \text{iter}2, \text{resvec2/ norm(b)}, '–o').
\]

See Also

- **bicg** BiConjugate Gradients method
- **bicgstab** BiConjugate Gradients Stabilized method
- **cgs** Conjugate Gradients Squared method
- **cholinc** Incomplete Cholesky factorizations
- **gmres** Generalized Minimum Residual method (with restarts)
- **qmr** Quasi-Minimal Residual method
- **\** Matrix left division

References

pcode

**Purpose**
Create preparsed pseudocode file (P-file)

**Syntax**
- `pcode fun`
- `pcode *.m`
- `pcode fun1 fun2 ...`
- `pcode... -inplace`

**Description**
pcode fun parses the M-file fun.m into the P-file fun.p and puts it into the current directory. The original M-file can be anywhere on the search path.

pcode *.m creates P-files for all the M-files in the current directory.

pcode fun1 fun2 ... creates P-files for the listed functions.

pcode... -inplace creates P-files in the same directory as the M-files. An error occurs if the files can't be created.
Purpose

All possible permutations

Syntax

\[ P = \text{perms}(v) \]

Description

\( P = \text{perms}(v) \), where \( v \) is a row vector of length \( n \), creates a matrix whose rows consist of all possible permutations of the \( n \) elements of \( v \). Matrix \( P \) contains \( n! \) rows and \( n \) columns.

Examples

The command \( \text{perms}(2:2:6) \) returns all the permutations of the numbers 2, 4, and 6:

\[
\begin{array}{ccc}
6 & 4 & 2 \\
4 & 6 & 2 \\
6 & 2 & 4 \\
2 & 6 & 4 \\
4 & 2 & 6 \\
2 & 4 & 6 \\
\end{array}
\]

Limitations

This function is only practical for situations where \( n \) is less than about 15.

See Also

- \text{nchoosek}  
  All combinations of the \( n \) elements in \( v \) taken \( k \) at a time
- \text{permute}  
  Rearrange the dimensions of a multidimensional array
- \text{randperm}  
  Random permutation
Purpose  
Rearrange the dimensions of a multidimensional array

Syntax  
B = permute(A, order)

Description  
B = permute(A, order) rearranges the dimensions of A so that they are in the order specified by the vector order. B has the same values of A but the order of the subscripts needed to access any particular element is rearranged as specified by order. All the elements of order must be unique.

Remarks  
permute and ipermute are a generalization of transpose (.') for multidimensional arrays.

Examples  
Given any matrix A, the statement
permute(A, [2 1])
is the same as A'.

For example:
A = [1 2; 3 4]; permute(A, [2 1])
ans =
    1     3
    2     4

The following code permutes a three-dimensional array:
X = rand(12, 13, 14);
Y = permute(X, [2 3 1]);
size(Y)
ans =
    13    14    12

See Also  
ipermute  
Inverse permute the dimensions of a multidimensional array
Purpose

Ratio of a circle's circumference to its diameter, \( \pi \)

Syntax

`pi`

Description

`pi` returns the floating-point number nearest the value of \( \pi \). The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples

The expression `sin(pi)` is not exactly zero because `pi` is not exactly \( \pi \):

```matlab
    sin(pi)
    ans = 1.2246e-16
```

See Also

- `ans` The most recent answer
- `eps` Floating-point relative accuracy
- `i` Imaginary unit
- `Inf` Infinity
- `j` Imaginary unit
- `NaN` Not-a-Number
**pinv**

**Purpose**  
Moore-Penrose pseudoinverse of a matrix

**Syntax**  
B = pinv(A)
B = pinv(A,tol)

**Definition**  
The Moore-Penrose pseudoinverse is a matrix B of the same dimensions as A' satisfying four conditions:

- \( A^*B^*A = A \)
- \( B^*A^*B = B \)
- \( A^*B \) is Hermitian
- \( B^*A \) is Hermitian

The computation is based on \( \text{svd}(A) \) and any singular values less than \( \text{tol} \) are treated as zero.

**Description**  
B = pinv(A) returns the Moore-Penrose pseudoinverse of A.

B = pinv(A,tol) returns the Moore-Penrose pseudoinverse and overrides the default tolerance, \( \max(\text{size}(A)) \times \text{norm}(A) \times \text{eps} \).

**Examples**  
If A is square and not singular, then \( \text{pinv}(A) \) is an expensive way to compute \( \text{inv}(A) \). If A is not square, or is square and singular, then \( \text{inv}(A) \) does not exist. In these cases, \( \text{pinv}(A) \) has some of, but not all, the properties of \( \text{inv}(A) \).

If A has more rows than columns and is not of full rank, then the overdetermined least squares problem

\[
\text{minimize } \| A^*x - b \|
\]

does not have a unique solution. Two of the infinitely many solutions are

\( x = \text{pinv}(A) \cdot b \)

and

\( y = A \setminus b \)

These two are distinguished by the facts that \( \text{norm}(x) \) is smaller than the norm of any other solution and that \( y \) has the fewest possible nonzero components.
For example, the matrix generated by

\[ A = \text{magc}(8); \quad A = A(:, 1:6) \]

is an 8-by-6 matrix that happens to have \( \text{rank}(A) = 3 \).

\[
A = \\
\begin{bmatrix}
64 & 2 & 3 & 61 & 60 & 6 \\
9 & 55 & 54 & 12 & 13 & 51 \\
17 & 47 & 46 & 20 & 21 & 43 \\
40 & 26 & 27 & 37 & 36 & 30 \\
32 & 34 & 35 & 29 & 28 & 38 \\
41 & 23 & 22 & 44 & 45 & 19 \\
49 & 15 & 14 & 52 & 53 & 11 \\
8 & 58 & 59 & 5 & 4 & 62 \\
\end{bmatrix}
\]

The right-hand side is \( b = 260 \times \text{ones}(8, 1) \),

\[
b = \\
\begin{bmatrix}
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
\end{bmatrix}
\]

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to \( A \times x = b \) 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

\[
x = \text{pinv}(A) \times b
\]
which is

\[
\begin{align*}
x &= \\
 1.1538 & \\
 1.4615 & \\
 1.3846 & \\
 1.3846 & \\
 1.4615 & \\
 1.1538 & \\
\end{align*}
\]

and

\[
\begin{align*}
y &= A \backslash b \\
\end{align*}
\]

which is

\[
\begin{align*}
y &= \\
 3.0000 & \\
 4.0000 & \\
 0 & \\
 0 & \\
 1.0000 & \\
 0 & \\
\end{align*}
\]

Both of these are exact solutions in the sense that \(\text{norm}(A \cdot x - b)\) and \(\text{norm}(A \cdot y - b)\) 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.
Purpose
Transform polar or cylindrical coordinates to Cartesian

Syntax
[X, Y] = pol2cart(THETA, RHO)
[X, Y, Z] = pol2cart(THETA, RHO, Z)

Description
[X, Y] = pol2cart(THETA, RHO) transforms the polar coordinate data stored in corresponding elements of THETA and RHO to two-dimensional Cartesian, or xy, coordinates. The arrays THETA and RHO must be the same size (or either can be scalar). The values in THETA must be in radians.

[X, Y, Z] = pol2cart(THETA, RHO, Z) transforms the cylindrical coordinate data stored in corresponding elements of THETA, RHO, and Z to three-dimensional Cartesian, or XYZ, coordinates. The arrays THETA, RHO, and Z must be the same size (or any can be scalar). The values in THETA must be in radians.

Algorithm
The mapping from polar and cylindrical coordinates to Cartesian coordinates is:

\[
\begin{align*}
\text{Polar to Cartesian Mapping} \\
\theta &= \text{atan2}(y, x) \\
r &\rho = \sqrt{x^2 + y^2}
\end{align*}
\]

\[
\begin{align*}
\text{Cylindrical to Cartesian Mapping} \\
\theta &= \text{atan2}(y, x) \\
r &\rho = \sqrt{x^2 + y^2} \\
z &= z
\end{align*}
\]

See Also
cart2pol
Transform Cartesian coordinates to polar or cylindrical
cart2sph
Transform Cartesian coordinates to spherical
 sph2cart
Transform spherical coordinates to Cartesian
poly

Purpose
Polynomial with specified roots

Syntax
p = poly(A)
p = poly(r)

Description
p = poly(A) where A is an n-by-n matrix returns an n+1 element row vector
whose elements are the coefficients of the characteristic polynomial, det(sI – A).
The coefficients are ordered in descending powers: if a vector c has n+1 com-
nents, the polynomial it represents is c_1 s^n + ... + c_n s + c_{n+1}.
p = poly(r) where r is a vector returns a row vector whose elements are the
coefficients of the polynomial whose roots are the elements of r.

Remarks
Note the relationship of this command to
r = roots(p)
which returns a column vector whose elements are the roots of the polynomial
specified by the coefficients row vector p. For vectors, roots and poly are
inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples
MATLAB displays polynomials as row vectors containing the coefficients
ordered by descending powers. The characteristic equation of the matrix

A =
1 2 3
4 5 6
7 8 0

is returned in a row vector by poly:
p = poly(A)
p =
1 -6 -72 -27
The roots of this polynomial (eigenvalues of matrix A) are returned in a column vector by `roots`:

\[ r = \text{roots}(p) \]
\[ r = \begin{align*}
12 &. 1229 \\
-5 &. 7345 \\
-0 &. 3884 
\end{align*} \]

**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

\[ \text{det}(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1} \]

The algorithm is expressed in an M-file:

```matlab
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1) - z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

\[ (\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n) \]

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of A. This is true even if the eigenvalues of A are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.
**See Also**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>Convolution and polynomial multiplication</td>
</tr>
<tr>
<td>polyval</td>
<td>Polynomial evaluation</td>
</tr>
<tr>
<td>residue</td>
<td>Convert between partial fraction expansion and polynomial coefficients</td>
</tr>
<tr>
<td>roots</td>
<td>Polynomial roots</td>
</tr>
</tbody>
</table>
Purpose

Area of polygon

Syntax

A = polyarea(X,Y)
A = polyarea(X,Y,dim)

Description

A = polyarea(X,Y) returns the area of the polygon specified by the vertices in the vectors X and Y.

If X and Y are matrices of the same size, then polyarea returns the area of polygons defined by the columns X and Y.

If X and Y are multidimensional arrays, polyarea returns the area of the polygons in the first nonsingleton dimension of X and Y.

A = polyarea(X,Y,dim) operates along the dimension specified by scalar dim.

Examples

L = linspace(0,2.*pi,6); xv = cos(L)'; yv = sin(L)';
xv = [xv ; xv(1) ]; yv = [yv ; yv(1) ];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image

See Also

convhull
inpolygon
Convex hull
Detect points inside a polygonal region
Purpose

Polynomial derivative

Syntax

\[
\begin{align*}
k &= \text{polyder}(p) \\
k &= \text{polyder}(a,b) \\
[q,d] &= \text{polyder}(b,a)
\end{align*}
\]

Description

The \text{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 \(b/a\).

Examples

The derivative of the product

\[
(3x^2 + 6x + 9)(x^2 + 2x)
\]
is obtained with

\[
\begin{align*}
a &= [3 \ 6 \ 9] \\
b &= [1 \ 2 \ 0] \\
k &= \text{polyder}(a,b) \\
k &= \begin{bmatrix} 12 & 36 & 42 & 18 \end{bmatrix}
\end{align*}
\]

This result represents the polynomial

\[
12x^3 + 36x^2 + 42x + 18
\]

See Also

\text{conv} \quad \text{Convolution and polynomial multiplication}
\text{deconv} \quad \text{Deconvolution and polynomial division}
**Purpose**
Polynomial eigenvalue problem

**Syntax**
\[ [X, e] = \text{polyeig}(A0, A1, \ldots, Ap) \]

**Description**
\[ [X, e] = \text{polyeig}(A0, A1, \ldots, Ap) \]
solves the polynomial eigenvalue problem of degree \( p \):
\[
(A_0 + \lambda A_1 + \ldots + \lambda^p A_p)X = 0
\]

where polynomial degree \( p \) is a non-negative integer, and \( A0, A1, \ldots, Ap \) are input matrices of order \( n \). Output matrix \( X \), of size \( n \times np \), contains eigenvectors in its columns. Output vector \( e \), of length \( np \), contains eigenvalues.

**Remarks**
Based on the values of \( p \) and \( n \), \text{polyeig} handles several special cases:

- \( p = 0 \), or \text{polyeig}(A) is the standard eigenvalue problem: \text{eig}(A).
- \( p = 1 \), or \text{polyeig}(A, B) is the generalized eigenvalue problem: \text{eig}(A, -B).
- \( n = 1 \), or \text{polyeig}(a0, a1, \ldots, ap) for scalars \( a0, a1, \ldots, ap \) is the standard polynomial problem: \text{roots([ap \ldots a1 a0])}.

**Algorithm**
If both \( A0 \) and \( Ap \) are singular, the problem is potentially ill posed; solutions might not exist or they might not be unique. In this case, the computed solutions may be inaccurate. \text{polyeig} attempts to detect this situation and display an appropriate warning message. If either one, but not both, of \( A0 \) and \( Ap \) is singular, the problem is well posed but some of the eigenvalues may be zero or infinite (\( \text{Inf} \)).

The \text{polyeig} function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of \text{eig} and \text{qz} for more on this, as well as the EISPACK Guide.

**See Also**
- \text{eig} — Eigenvalues and eigenvectors
- \text{qz} — QZ factorization for generalized eigenvalues
polyfit

Purpose
Polynomial curve fitting

Syntax
\[ p = \text{polyfit}(x, y, n) \]
\[ [p, s] = \text{polyfit}(x, y, n) \]

Description
\( p = \text{polyfit}(x, y, n) \) finds the coefficients of a polynomial \( p(x) \) of degree \( n \) that fits the data, \( p(x(i)) \) to \( y(i) \), in a least squares sense. The result \( p \) is a row vector of length \( n+1 \) containing the polynomial coefficients in descending powers:
\[
p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}
\]

\[ [p, s] = \text{polyfit}(x, y, n) \] returns the polynomial coefficients \( p \) and a structure \( S \) for use with \text{polyval} to obtain error estimates or predictions. If the errors in the data \( Y \) are independent normal with constant variance; \text{polyval} will produce error bounds that contain at least 50% of the predictions.

Examples
This example involves fitting the error function, \( \text{erf}(x) \), by a polynomial in \( x \). This is a risky project because \( \text{erf}(x) \) is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of \( x \)-points, equally spaced in the interval \([0, 2.5]\); then evaluate \( \text{erf}(x) \) at those points.
\[
x = (0: 0.1: 2.5)';
y = \text{erf}(x);
\]
The coefficients in the approximating polynomial of degree 6 are
\[
p = \text{polyfit}(x, y, 6)
\]
\[
p =
0.0084  -0.0983  0.4217  -0.7435  0.1471  1.1064  0.0004
\]
There are seven coefficients and the polynomial is
\[
0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004
\]
To see how good the fit is, evaluate the polynomial at the data points with
\[
f = \text{polyval}(p, x);
\]
A table showing the data, fit, and error is

```matlab
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 0.9993 0.9995 -0.0002
2.4000 0.9996 0.9994 0.0002
2.5000 0.9989 0.9991 -0.0003
```

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.

```matlab
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p, x);
plot(x, y, 'o', x, f, '–')
axis([0 5 0 2])
```

![Graph showing the fit and approximation](image.png)
\textbf{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

$$V_p \approx y$$

The M-file can be modified to use other functions of $x$ as the basis functions.

\textbf{See Also} 
polyval \hspace{1cm} \text{Polynomial evaluation}
roots \hspace{1cm} \text{Polynomial roots}
**Purpose**
Polynomial evaluation

**Syntax**
y = polyval (p, x)
[y, delta] = polyval (p, x, S)

**Description**
y = polyval (p, x) returns the value of the polynomial p evaluated at x. Polynomial p is a vector whose elements are the coefficients of a polynomial in descending powers.

x can be a matrix or a vector. In either case, polyval evaluates p at each element of x.

[y, delta] = polyval (p, x, S) uses the optional output structure S generated by polyfit to generate error estimates, y±delta. If the errors in the data input to polyfit are independent normal with constant variance, y±delta contains at least 50% of the predictions.

**Remarks**
The polyvalm(p, x) function, with x a matrix, evaluates the polynomial in a matrix sense. See polyvalm for more information.

**Examples**
The polynomial p(x) = 3x^2 + 2x + 1 is evaluated at x = 5, 7, and 9 with
\[
p = [3 2 1];
polyval (p, [5 7 9])
\]
which results in
\[
an = 
86   162   262
\]
For another example, see polyfit.

**See Also**
polyfit  Polynomial curve fitting
polyvalm  Matrix polynomial evaluation
Purpose

Matrix polynomial evaluation

Syntax

\[ Y = \text{polyvalm}(p, X) \]

Description

\[ Y = \text{polyvalm}(p, X) \] 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.

Examples

The Pascal matrices are formed from Pascal’s triangle of binomial coefficients. Here is the Pascal matrix of order 4.

\[
X = \text{pascal}(4)
\]

\[
X =
\begin{array}{cccc}
  1 & 1 & 1 & 1 \\
  1 & 2 & 3 & 4 \\
  1 & 3 & 6 & 10 \\
  1 & 4 & 10 & 20 \\
\end{array}
\]

Its characteristic polynomial can be generated with the \text{poly} function.

\[
p = \text{poly}(X)
\]

\[
p =
\begin{array}{cccc}
  1 & -29 & 72 & -29 & 1 \\
\end{array}
\]

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.

\[
\text{polyval}(p, X)
\]

\[
\begin{array}{cccc}
  16 & 16 & 16 & 16 \\
  16 & 15 & -140 & -563 \\
  16 & -140 & -2549 & -12089 \\
  16 & -563 & -12089 & -43779 \\
\end{array}
\]
But evaluating it in a matrix sense is interesting.

```plaintext
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**

**Purpose**
Base 2 power and scale floating-point numbers

**Syntax**

\[
X = \text{pow2}(Y) \\
X = \text{pow2}(F, E)
\]

**Description**

\[
X = \text{pow2}(Y)
\]
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.

**Remarks**

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

**Examples**

For IEEE arithmetic, the statement \(X = \text{pow2}(F, E)\) 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>\pi / 4</td>
<td>2</td>
<td>\pi</td>
</tr>
<tr>
<td>-3/4</td>
<td>2</td>
<td>-3</td>
</tr>
<tr>
<td>1/2</td>
<td>-51</td>
<td>\text{eps}</td>
</tr>
<tr>
<td>1-\text{eps} / 2</td>
<td>1024</td>
<td>\text{real max}</td>
</tr>
<tr>
<td>1/2</td>
<td>-1021</td>
<td>\text{real min}</td>
</tr>
</tbody>
</table>

**See Also**

`\log2` Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

`\wedge` Matrix power

`.^` Array power

`\exp` Exponential

`\text{hex2num}` Hexadecimal to double number conversion

`\text{real max}` Largest positive floating-point number

`\text{real min}` Smallest positive floating-point number
Purpose  Generate list of prime numbers

Syntax  \( p = \text{primes}(n) \)

Description  \( p = \text{primes}(n) \) returns a row vector of the prime numbers less than or equal to \( n \). A prime number is one that has no factors other than 1 and itself.

Examples  \( p = \text{primes}(37) \)

\[
p =
\begin{bmatrix}
 2 & 3 & 5 & 7 & 11 & 13 & 17 & 19 & 23 & 29 & 31 & 37
\end{bmatrix}
\]

See Also  \text{factor}  Prime factors
Purpose

Product of array elements

Syntax

B = prod(A)
B = prod(A, dim)

Description

B = prod(A) returns the products along different dimensions of an array. If A is a vector, prod(A) returns the product of the elements. If A is a matrix, prod(A) treats the columns of A as vectors, returning a row vector of the products of each column. If A is a multidimensional array, prod(A) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

B = prod(A, dim) takes the products along the dimension of A specified by scalar dim.

Examples

The magic square of order 3 is

\[
M = \text{magic}(3)
\]

\[
M =
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]

The product of the elements in each column is

\[
\text{prod}(M) =
\begin{bmatrix}
96 \\
45 \\
84
\end{bmatrix}
\]

The product of the elements in each row can be obtained by:

\[
\text{prod}(M, 2) =
\begin{bmatrix}
48 \\
105 \\
72
\end{bmatrix}
\]

See Also

cumprod Cumulative product
diff Difference
sum Sum of array elements
**Purpose**

Measure and display M-file execution profiles

**Syntax**

```
profile function
profile report
profile report n
profile report frac
profile on
profile off
profile done
profile reset
info = profile
```

**Description**

The profiler utility helps you debug and optimize M-files by tracking the cumulative execution time of each line of code. The utility creates a vector of “bins,” one bin for every line of code in the M-file being profiled. As MATLAB executes the M-file code, the profiler updates each bin with running counts of the time spent executing the corresponding line.

- **profile function** starts the profiler for function. function must be the name of an M-file function or a `MATLABPATH` relative partial pathname.
- **profile report** displays a profile summary report for the M-file currently being profiled.
- **profile report n**, where `n` is an integer, displays a report showing the `n` lines that take the most time.
- **profile report frac**, where `frac` is a number between 0.0 and 1.0, displays a report of each line that accounts for more than `frac` of the total time.
- **profile on** and **profile off** enable and disable profiling, respectively.
- **profile done** turns off the profiler and clears its data.
- **profile reset** erases the bin contents without disabling profiling or changing the M-file under inspection.
info = profile returns a structure with the fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Full path to the function being profiled.</td>
</tr>
<tr>
<td>function</td>
<td>Name of function being profiled.</td>
</tr>
<tr>
<td>interval</td>
<td>Sampling interval in seconds.</td>
</tr>
<tr>
<td>count</td>
<td>Vector of sample counts</td>
</tr>
<tr>
<td>state</td>
<td>on if the profiler is running and off otherwise.</td>
</tr>
</tbody>
</table>

**Remarks**

You can also profile built-in functions. The profiler tracks the number of intervals in which the built-in function was called (an estimate of how much time was spent executing the built-in function).

The profiler's behavior is defined by root object properties and can be manipulated using the set and get commands. See the Applying MATLAB for more details.

**Limitations**

The profiler utility can accommodate only one M-file at a time.

**See Also**

See also partialpath.
Purpose

Quasi-Minimal Residual method

Syntax

\[
\begin{align*}
x &= \text{qmr} \left( A, b \right) \\
\text{qmr} \left( A, b, t ol \right) \\
\text{qmr} \left( A, b, t ol, m axi t \right) \\
\text{qmr} \left( A, b, t ol, m axi t, M L \right) \\
\text{qmr} \left( A, b, t ol, m axi t, M L, M 2 \right) \\
\text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right) \\
x &= \text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right) \\
\left[ x, f l a g \right] &= \text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right) \\
\left[ x, f l a g, r e l r e s \right] &= \text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right) \\
\left[ x, f l a g, r e l r e s, i t e r \right] &= \text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right) \\
\left[ x, f l a g, r e l r e s, i t e r, r e s v e c \right] &= \text{qmr} \left( A, b, t ol, m axi t, M L, M 2, x 0 \right)
\end{align*}
\]

Description

\( x = \text{qmr} \left( A, b \right) \) attempts to solve the system of linear equations \( A \times x = b \) 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-\( n \). \text{qmr} will start iterating from an initial estimate that by default is an all zero vector of length \( n \). Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate \( x \) has relative residual \( \frac{\text{norm}(b - A \times x)}{\text{norm}(b)} \) less than or equal to the tolerance of the method. The default tolerance is \( 1 \times 10^{-6} \). The default maximum number of iterations is the minimum of \( n \) and 20. No preconditioning is used.

\( \text{qmr} \left( A, b, t ol \right) \) specifies the tolerance of the method, \( t ol \).

\( \text{qmr} \left( A, b, t ol, m axi t \right) \) additionally specifies the maximum number of iterations, \( m axi t \).

\( \text{qmr} \left( A, b, t ol, m axi t, M L \right) \) and \( \text{qmr} \left( A, b, t ol, m axi t, M L, M 2 \right) \) use left and right preconditioners \( M L \) and \( M 2 \) and effectively solve the system \( \text{inv}(M L) \times A \times \text{inv}(M 2) \times y = \text{inv}(M L) \times b \) for \( y \), where \( x = \text{inv}(M 2) \times y \). If \( M L \) or \( M 2 \) 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 \( M L \times y = r \) are solved using backslash within \text{qmr}, it is wise to factor precondi-
tationers into their LU factorizations first. For example, replace
qmr(A, b, tol, maxi t, M []) or qmr(A, b, tol, maxi t, [], M) with:

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

qmr(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 = qmr(A, b, tol, maxi t, 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, maxi t, M1, M2, x0) returns a solution x and a flag
which describes the convergence of qmr:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>qmr converged to the desired tolerance tol within maxi t iterations without failing for any reason.</td>
</tr>
<tr>
<td>1</td>
<td>qmr iterated maxi t times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>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).</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during qmr became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.
[x, flag, relres] = qmr(A, b, tol, maxit, M1, M2, x0) also returns the relative residual norm(b - A*x) / norm(b). If flag is 0, then relres ≤ tol.

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

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

Examples

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

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

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

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

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

flag2 is 0 since qmr will converge to the tolerance of 1.9e-16 (the value of relres2) at the eighth iteration (the value of iter2) when preconditioned by the incomplete LU factorization with a drop tolerance of 1e-6. resvec2(1) = norm(b) and resvec2(9) = norm(b - A*x2). You may follow the progress of qmr.
by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with `semilogy(0:iter2, resvec2/norm(b), '–o')`.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{relative_residual_plot.png}
\caption{Relative residuals vs iteration number.}
\end{figure}

See Also

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

References


Purpose
Orthogonal-triangular decomposition

Syntax
\[
\begin{align*}
[Q, R] &= \text{qr}(X) \\
[Q, R, E] &= \text{qr}(X) \\
[Q, R] &= \text{qr}(X, 0) \\
[Q, R, E] &= \text{qr}(X, 0) \\
A &= \text{qr}(X)
\end{align*}
\]

Description
The \text{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.

\[ [Q, R] = \text{qr}(X) \]
produces an upper triangular matrix \( R \) of the same dimension as \( X \) and a unitary matrix \( Q \) so that \( X = Q \ast R \).

\[ [Q, R, E] = \text{qr}(X) \]
produces a permutation matrix \( E \), an upper triangular matrix \( R \) with decreasing diagonal elements, and a unitary matrix \( Q \) so that \( X \ast E = Q \ast R \). The column permutation \( E \) is chosen so that \( \text{abs}(\text{diag}(R)) \) is decreasing.

\[ [Q, R] = \text{qr}(X, 0) \quad \text{and} \quad [Q, R, E] = \text{qr}(X, 0) \]
produce “economy-size” decompositions in which \( E \) is a permutation vector, so that \( Q \ast R = X(:, E) \). The column permutation \( E \) is chosen so that \( \text{abs}(\text{diag}(R)) \) is decreasing.

\[ A = \text{qr}(X) \]
returns the output of the LINPACK subroutine ZQRDC. \( \text{triu(} \text{qr}(X)) \) is \( R \).

Examples
Start with
\[
A =
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}
\]
This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

$$[Q, R] = qr(A)$$

$$Q =
\begin{bmatrix}
-0.0776 & -0.8331 & 0.5444 & 0.0605 \\
-0.3105 & -0.4512 & -0.7709 & 0.3251 \\
-0.5433 & -0.0694 & -0.0913 & -0.8317 \\
-0.7762 & 0.3124 & 0.3178 & 0.4461
\end{bmatrix}$$

$$R =
\begin{bmatrix}
-12.8841 & -14.5916 & -16.2992 \\
0 & -1.0413 & -2.0826 \\
0 & 0 & 0.0000 \\
0 & 0 & 0
\end{bmatrix}$$

The triangular structure of $R$ gives it zeros below the diagonal; the zero on the diagonal in $R(3,3)$ implies that $R$, and consequently $A$, does not have full rank.

The QR factorization is used to solve linear systems with more equations than unknowns. For example

$$b =
\begin{bmatrix}
1 \\
3 \\
5 \\
7
\end{bmatrix}$$

The linear system $Ax = b$ represents four equations in only three unknowns. The best solution in a least squares sense is computed by

$$x = A \backslash b$$

which produces

Warning: Rank deficient, rank = 2, tol = 1.4594E-014

$$x =
\begin{bmatrix}
0.5000 \\
0 \\
0.1667
\end{bmatrix}$$
The quantity \( \text{tol} \) is a tolerance used to decide if a diagonal element of \( R \) is negligible. If \([Q, R, E] = \text{qr}(A)\), then
\[
\text{tol} = \max(\text{size}(A)) \times \text{eps} \times \text{abs}(R(1,1))
\]
The solution \( x \) was computed using the factorization and the two steps
\[
y = Q' \ast b;
x = R \backslash y
\]
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 \texttt{qr} function uses the LINPACK routines \texttt{ZQRDC} and \texttt{ZQRSL}. \texttt{ZQRDC} computes the QR decomposition, while \texttt{ZQRSL} applies the decomposition.

**See Also**
- \texttt{\textbackslash} \hspace{1cm} Matrix left division (backslash)
- \texttt{/} \hspace{1cm} Matrix right division (slash)
- \texttt{lu} \hspace{1cm} LU matrix factorization
- \texttt{nul1} \hspace{1cm} Null space of a matrix
- \texttt{orth} \hspace{1cm} Range space of a matrix
- \texttt{qrdelete} \hspace{1cm} Delete column from QR factorization
- \texttt{qrinsert} \hspace{1cm} Insert column in QR factorization

**References**
### qrdelete

**Purpose**
Delete column from QR factorization

**Syntax**

```
[Q,R] = qrdelete(Q,R,j)
```

**Description**

The `qrdelete` function changes `Q` and `R` to be the factorization of the matrix `A` with its `j`th column, `A(:,j)`, removed.

Inputs `Q` and `R` represent the original QR factorization of matrix `A`, as returned by the statement `[Q,R] = qr(A)`. Argument `j` specifies the column to be removed from matrix `A`.

**Algorithm**
The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

**See Also**
- `qr` Orthogonal-triangular decomposition
- `qrinsert` Insert column in QR factorization
**Purpose**

Insert column in QR factorization

**Syntax**

\[ [Q, R] = qrinsert(Q, R, j, x) \]

**Description**

\[ [Q, R] = qrinsert(Q, R, j, x) \] changes \( Q \) and \( R \) to be the factorization of the matrix obtained by inserting an extra column, \( x \), before \( A(:,j) \). If \( A \) has \( n \) columns and \( j = n+1 \), then \( qrinsert \) inserts \( x \) after the last column of \( A \).

Inputs \( Q \) and \( R \) represent the original QR factorization of matrix \( A \), as returned by the statement \( [Q, R] = qr(A) \). Argument \( x \) is the column vector to be inserted into matrix \( A \). Argument \( j \) specifies the column before which \( x \) is inserted.

**Algorithm**

The \( qrinsert \) function inserts the values of \( x \) into the \( j \)th 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 \( j \)th column.

**See Also**

- \( qr \) Orthogonal-triangular decomposition
- \( qrdelete \) Delete column from QR factorization
qtwrite

**Purpose**  
Write QuickTime movie file to disk

**Syntax**  
qtwrite(d,siz,map,'filename')
qtwrite(mov,map,'filename')
qtwrite(...,options)

**Description**  
qtwrite(d,siz,map,'filename')  
writes the indexed image deck d with size siz and colormap map to the QuickTime movie file 'filename'. If 'filename' already exists, it will be replaced.

qtwrite(mov,map,'filename')  
writes the MATLAB movie matrix mov with colormap map to the QuickTime movie file 'filename'.

qtwrite(...,options)  
can be used to set the frame rate, spacial quality, and compressor type:

options(1): frame rate (frames per second) (10 fps default)

options(2): compressor type:
1 - video (default), 2 - jpeg, 3 - animation

options(3): spacial quality:
1 - minimum, 2 - low, 3 - normal (default), 4 - high,
5 - maximum, 6 - lossless

qtwrite requires QuickTime and works only on the Macintosh.
Purpose
Numerical evaluation of integrals

Syntax
\[
q = \text{quad}('fun', a, b) \\
q = \text{quad}('fun', a, b, tol) \\
q = \text{quad}('fun', a, b, tol, trace) \\
q = \text{quad}('fun', a, b, tol, trace, P1, P2, \ldots) \\
q = \text{quad8}(\ldots)
\]

Description
Quadrature is a numerical method of finding the area under the graph of a function, that is, computing a definite integral.

\[
q = \int_a^b f(x) \, dx
\]

\(q = \text{quad}('fun', a, b)\) 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.

\(q = \text{quad}('fun', a, b, tol)\) iterates until the relative error is less than \(tol\). The default value for \(tol\) is 1.e-3. Use a two element tolerance vector, \(tol = [\text{rel\_tol} \quad \text{abs\_tol}]\), to specify a combination of relative and absolute error.

\(q = \text{quad}('fun', a, b, tol, trace)\) integrates to a relative error of \(tol\), and for non-zero \(trace\), plots a graph showing the progress of the integration.

\(q = \text{quad}('fun', a, b, tol, trace, P1, P2, \ldots)\) allows coefficients \(P1, P2, \ldots\) to be passed directly to the specified function: \(G = \text{fun}(X, P1, P2, \ldots)\). To use default values for \(tol\) or \(trace\), pass in the empty matrix, for example: \(\text{quad}('fun', a, b, [], [], P1)\).

Remarks
quad8, a higher-order method, has the same calling sequence as quad.

Examples
Integrate the sine function from 0 to \(\pi\):
\[
a = \text{quad('sin', 0, pi)} \\
a =
\]
\[
2.0000
\]
quad, quad8

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:

\[ \int_{0}^{1} \frac{1}{\sqrt{x}} \, dx \]

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.

Purpose: Terminate MATLAB

Syntax: quit

Description: quit terminates MATLAB without saving the workspace. To save your workspace variables, use the `save` command before quitting.

See Also: save, startup
            Save workspace variables on disk
            MATLAB startup M-file
Purpose
QZ factorization for generalized eigenvalues

Syntax
[ AA, BB, Q, Z, V ] = qz( A, B )

Description
The qz function gives access to what are normally only intermediate results in the computation of generalized eigenvalues.

[ AA, BB, Q, Z, V ] = qz( A, B ) produces upper triangular matrices AA and BB, and matrices Q and Z containing the products of the left and right transformations, such that

Q\cdot A \cdot Z = AA
Q\cdot B \cdot Z = BB

The qz function also returns the generalized eigenvector matrix V.

The generalized eigenvalues are the diagonal elements of AA and BB so that

A \cdot V \cdot \text{diag}(BB) = B \cdot V \cdot \text{diag}(AA)

Arguments
A, B        Square matrices.
AA, BB      Upper triangular matrices.
Q, Z        Transformation matrices.
V           Matrix whose columns are eigenvectors.

Algorithm
Complex generalizations of the EISPACK routines QZHES, QZI T, QZVAL, and QZVEC implement the QZ algorithm.

See Also
eig              Eigenvalues and eigenvectors

References
**Purpose**
Uniformly distributed random numbers and arrays

**Syntax**
- \( Y = \text{rand}(n) \)
- \( Y = \text{rand}(m,n) \)
- \( Y = \text{rand}([m\ n]) \)
- \( Y = \text{rand}(m\ n\ p\ldots) \)
- \( Y = \text{rand}([m\ n\ p\ldots]) \)
- \( Y = \text{rand}(\text{size}(A)) \)
- \( \text{rand} \)
- \( s = \text{rand}('\text{state}') \)

**Description**
The \textit{rand} function generates arrays of random numbers whose elements are uniformly distributed in the interval \((0,1)\).

- \( Y = \text{rand}(n) \) returns an \( n \)-by-\( n \) matrix of random entries. An error message appears if \( n \) is not a scalar.
- \( Y = \text{rand}(m,n) \) or \( Y = \text{rand}([m\ n]) \) returns an \( m \)-by-\( n \) matrix of random entries.
- \( Y = \text{rand}(m\ n\ p\ldots) \) or \( Y = \text{rand}([m\ n\ p\ldots]) \) generates random arrays.
- \( Y = \text{rand}(\text{size}(A)) \) returns an array of random entries that is the same size as \( A \).
- \( \text{rand} \), by itself, returns a scalar whose value changes each time it's referenced.
- \( s = \text{rand}('\text{state}') \) returns a 35-element vector containing the current state of the uniform generator. To change the state of the generator:
  - \( \text{rand}('\text{state}',s) \) resets the state to \( s \).
  - \( \text{rand}('\text{state}',0) \) resets the generator to its initial state.
  - \( \text{rand}('\text{state}',j) \) for integer \( j \), resets the generator to its \( j \)-th state.
  - \( \text{rand}('\text{state}',\text{sum(100*clock)}) \) resets it to a different state each time.
Remarks

MATLAB 5 uses a new multiseed random number generator that can generate all the floating-point numbers in the closed interval \([2^{-53}, 1 - 2^{-53}]\). Theoretically, it can generate over \(2^{1492}\) values before repeating itself. MATLAB 4 used random number generators with a single seed. `rand('seed', 0)` and `rand('seed', j)` use the MATLAB 4 generator. `rand('seed')` returns the current seed of the MATLAB 4 uniform generator. `rand('state', j)` and `rand('state', s)` use the MATLAB 5 generator.

Examples

```matlab
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.

```matlab
if rand < .5
    'heads'
else
    'tails'
end
```

See Also

- `randn` Normally distributed random numbers and arrays
- `randperm` Random permutation
- `sprand` Sparse uniformly distributed random matrix
- `sprandn` Sparse normally distributed random matrix
**Purpose**

Normally distributed random numbers and arrays

**Syntax**

\[
Y = \text{randn}(n)
\]

\[
Y = \text{randn}(m,n)
\]

\[
Y = \text{randn}([m \ n])
\]

\[
Y = \text{randn}(m,n,p,\ldots)
\]

\[
Y = \text{randn}([m \ n \ p\ldots])
\]

\[
Y = \text{randn}(\text{size}(A))
\]

\[
\text{randn}
\]

\[
s = \text{randn}('\text{state}')
\]

**Description**

The \texttt{randn} function generates arrays of random numbers whose elements are normally distributed with mean 0 and variance 1.

\[
Y = \text{randn}(n)
\]

returns an \(n\)-by-\(n\) matrix of random entries. An error message appears if \(n\) is not a scalar.

\[
Y = \text{randn}(m,n) \text{ or } Y = \text{randn}([m \ n])
\]

returns an \(m\)-by-\(n\) matrix of random entries.

\[
Y = \text{randn}(m,n,p,\ldots) \text{ or } Y = \text{randn}([m \ n \ p\ldots])
\]

generates random arrays.

\[
Y = \text{randn}(\text{size}(A))
\]

returns an array of random entries that is the same size as \(A\).

\texttt{randn}, by itself, returns a scalar whose value changes each time it’s referenced.

\[
s = \text{randn}('\text{state}')
\]

returns a 2-element vector containing the current state of the normal generator. To change the state of the generator:

\[
\text{randn}('\text{state}',s)
\]

Resets the state to \(s\).

\[
\text{randn}('\text{state}',0)
\]

Resets the generator to its initial state.

\[
\text{randn}('\text{state}',j)
\]

For integer \(j\), resets the generator to its \(j\)th state.

\[
\text{randn}('\text{state}',\text{sum(100*clock)})
\]

Resets it to a different state each time.
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**

\[
R = \text{randn}(3, 4) \text{ may produce }
\]
\[
R =
\begin{bmatrix}
1.1650 & 0.3516 & 0.0591 & 0.8717 \\
0.6268 & -0.6965 & 1.7971 & -1.4462 \\
0.0751 & 1.6961 & 0.2641 & -0.7012
\end{bmatrix}
\]

For a histogram of the \text{randn} distribution, see \text{hist}.

**See Also**

- `rand`: Uniformly distributed random numbers and arrays
- `randper m`: Random permutation
- `sprand`: Sparse uniformly distributed random matrix
- `sprandn`: Sparse normally distributed random matrix
Purpose
Random permutation

Syntax
p = randperm(n)

Description
p = randperm(n) returns a random permutation of the integers 1:n.

Remarks
The randperm function calls rand and therefore changes rand’s seed value.

Examples
randperm(6) might be the vector
[3 2 6 4 1 5]
or it might be some other permutation of 1:6.

See Also
permute Rearrange the dimensions of a multidimensional array
Purpose
Rank of a matrix

Syntax
\[
k = rank(A)
k = rank(A,tol)
\]

Description
The `rank` function provides an estimate of the number of linearly independent rows or columns of a matrix.

\[
k = rank(A) \text{ returns the number of singular values of } A \text{ that are larger than the default tolerance, } \max(\text{size}(A)) \times \text{norm}(A) \times \text{eps}.
\]

\[
k = rank(A,tol) \text{ returns the number of singular values of } A \text{ that are larger than tol}.
\]

Algorithm
There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition, or SVD, described in Chapter 11 of the LINPACK Users' Guide. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

\[
s = svd(A);
\text{tol} = \max(\text{size}(A)) \times s(1) \times \text{eps};
\text{r} = \text{sum}(s > \text{tol});
\]

References
Purpose  
Rational fraction approximation

Syntax  
\[
\begin{align*}
[N, D] &= \text{rat}(X) \\
[N, D] &= \text{rat}(X, \text{tol}) \\
\text{rat}(\ldots) \\
S &= \text{rats}(X, \text{strlen}) \\
S &= \text{rats}(X)
\end{align*}
\]

Description  
Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The \text{rat} function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The \text{rats} function calls \text{rat}, and returns strings.

\[
\begin{align*}
[N, D] &= \text{rat}(X) \text{ returns arrays } N \text{ and } D \text{ so that } N/D \text{ approximates } X \text{ to within the default tolerance, } 1 \times 10^{-6} \text{norm}(X(:), 1). \\
[N, D] &= \text{rat}(X, \text{tol}) \text{ returns } N/D \text{ approximating } X \text{ to within tol.} \\
\text{rat}(X), \text{ with no output arguments, simply displays the continued fraction.}
\end{align*}
\]

\[
S = \text{rats}(X, \text{strlen}) \text{ returns a string containing simple rational approximations to the elements of } X. \text{ Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in } X. \text{ strlen is the length of each string element returned by the rats function. The default is strlen = 13, which allows 6 elements in 78 spaces.}
\]

\[
S = \text{rats}(X) \text{ returns the same results as those printed by MATLAB with format rat.}
\]

Examples  
Ordinarily, the statement  
\[
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
\]
produces  
\[
s = 0.7595
\]
However, with

```
format rat
```
or with

```
rats(s)
```
the printed result is

```
s = 319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity `s` is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)` produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n, d] = rat(s)
```
produces

```
n = 319, d = 420
```

The mathematical quantity $\pi$ 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
```
and so it agrees with \( \pi \) to seven significant figures. The statement

\[
\text{rat}(\pi)
\]

produces

\[
3 + \frac{1}{7 + \frac{1}{16}}
\]

This shows how the \( \frac{355}{113} \) was obtained. The less accurate, but more familiar approximation \( \frac{22}{7} \) is obtained from the first two terms of this continued fraction.

**Algorithm**

The \( \text{rat}(X) \) function approximates each element of \( X \) by a continued fraction of the form:

\[
\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{d_3 + \cdots + \frac{1}{d_k}}}
\]

The \( d \)'s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when \( X = \sqrt{2} \). For \( x = \sqrt{2} \), the error with \( k \) terms is about \( 2.68 \times (0.173)^k \), so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.
Purpose
Matrix reciprocal condition number estimate

Syntax
\[ c = \text{rcond}(A) \]

Description
\[ c = \text{rcond}(A) \] returns an estimate for the reciprocal of the condition of \( A \) in 1-norm using the LINPACK condition estimator. If \( A \) is well conditioned, \( \text{rcond}(A) \) is near 1.0. If \( A \) is badly conditioned, \( \text{rcond}(A) \) is near 0.0. Compared to \text{cond}, \text{rcond} is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm
The \text{rcond} function uses the condition estimator from the LINPACK routine \text{ZGECO}.

See Also
- \text{cond} — Condition number with respect to inversion
- \text{condest} — 1-norm matrix condition number estimate
- \text{norm} — Vector and matrix norms
- \text{normest} — 2-norm estimate
- \text{rank} — Rank of a matrix
- \text{svd} — Singular value decomposition

References
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')
Purpose
Real part of complex number

Syntax
X = real(Z)

Description
X = real(Z) returns the real part of the elements of the complex array Z.

Examples
real(2+3*i) is 2.

See Also
abs          Absolute value and complex magnitude
angle        Phase angle
conj         Complex conjugate
i, j         Imaginary unit (\(\sqrt{-1}\))
i mag        Imaginary part of a complex number
Purpose
Largest positive floating-point number

Syntax
\( n = \text{realmax} \)

Description
\( n = \text{realmax} \) returns the largest floating-point number representable on a particular computer. Anything larger overflows.

Examples
On machines with IEEE floating-point format, \( \text{realmax} \) is one bit less than \( 2^{1024} \) or about \( 1.7977 \times 10^{308} \).

Algorithm
The \( \text{realmax} \) function is equivalent to \( \text{pow2}(2 - \text{eps}, \text{maxexp}) \), where \( \text{maxexp} \) is the largest possible floating-point exponent.

Execute \( \text{type realmax} \) to see \( \text{maxexp} \) for various computers.

See Also
\begin{align*}
\text{eps} & \quad \text{Floating-point relative accuracy} \\
\text{realmin} & \quad \text{Smallest positive floating-point number}
\end{align*}
### Purpose
Smallest positive floating-point number

### Syntax
\( n = \text{realmin} \)

### Description
\( n = \text{realmin} \) returns the smallest positive normalized floating-point number on a particular computer. Anything smaller underflows or is an IEEE “denormal.”

### Examples
On machines with IEEE floating-point format, \( \text{realmin} \) is \( 2^{(-1022)} \) or about 2.2251e–308.

### Algorithm
The \( \text{realmin} \) function is equivalent to \( \text{pow2}(1, \text{minexp}) \) where \( \text{minexp} \) is the smallest possible floating-point exponent.

Execute \( \text{type realmin} \) to see \( \text{minexp} \) for various computers.

### See Also
- \( \text{eps} \) Floating-point relative accuracy
- \( \text{realmax} \) Largest positive floating-point number
### recordsound

**Purpose**  
Record sound

**Syntax**

\[
y = \text{recordsound}( \text{seconds} )
\]

\[
y = \text{recordsound}( \text{seconds}, \text{samplerate} )
\]

\[
y = \text{recordsound}( \text{seconds}, \text{numchannels} )
\]

\[
y = \text{recordsound}( \text{seconds}, \text{samplerate}, \text{numchannels} )
\]

**Description**

\[
y = \text{recordsound}( \text{seconds} ) \quad \text{records a monophonic sound for seconds number of seconds at the lowest sampling rate (usually 11 or 22 kHz) and highest resolution (usually 8 or 16 bits) that the Macintosh supports.}
\]

\[
y = \text{recordsound}( \text{seconds}, \text{samplerate} ) \quad \text{records a sound at a sampling rate greater than or equal to samplerate, or at the maximum sampling rate that the Macintosh supports.}
\]

\[
y = \text{recordsound}( \text{seconds}, \text{numchannels} ) \quad \text{records a sound with numchannels (usually 1 or 2) channels. If numchannels is 2 and the Macintosh does not support stereo recording, a monophonic sound is recorded instead.}
\]

\[
y = \text{recordsound}( \text{seconds}, \text{samplerate}, \text{numchannels} ) \quad \text{records a sound at the specified sampling rate and with the specified number of channels.}
\]

**Examples**

\[
y = \text{recordsound}( 10 )
\]

\[
y = \text{recordsound}( 5, 22050 )
\]

\[
y = \text{recordsound}( 5, 2 )
\]

\[
y = \text{recordsound}( 5, 44100, 2 )
\]
Purpose  
Remainder after division

Syntax  
\[ R = \text{rem}(X, Y) \]

Description  
\[ R = \text{rem}(X, Y) \] returns \( X - \text{fix}(X ./ Y) \times Y \), where \( \text{fix}(X ./ Y) \) is the integer part of the quotient, \( X ./ Y \).

Remarks  
So long as operands \( X \) and \( Y \) are of the same sign, the statement \( \text{rem}(X, Y) \) 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 \( \text{rem} \) function returns a result that is between 0 and \( \text{sign}(X) \times \text{abs}(Y) \). If \( Y \) is zero, \( \text{rem} \) returns NaN.

Limitations  
Arguments \( X \) and \( Y \) should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also  
\( \text{mod} \)  
Modulus (signed remainder after division)
Purpose
Replicate and tile an array

Syntax
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p ...])
repmat(A,m,n)

Description
B = repmat(A,m,n) creates a large matrix B consisting of an m-by-n tiling of copies of A. The statement repmat(A,n) creates an n-by-n tiling.

B = repmat(A,[m n]) accomplishes the same result as repmat(A,m,n).

B = repmat(A,[m n p ...]) produces a multidimensional (m-by-n-by-p-by-...) array composed of copies of A. A may be multidimensional.

repmat(A,m,n) when A is a scalar, produces an m-by-n matrix filled with A’s value. This can be much faster than a*ones(m,n) when m or n is large.

Examples
In this example, repmat replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

B = repmat(eye(2),3,4)

B =
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1

The statement N = repmat(NaN,[2 3]) creates a 2-by-3 matrix of NaNs.
Reshape array

B = reshape(A, m, n)
B = reshape(A, m, n, p, ...)
B = reshape(A, [m n p, ...])
B = reshape(A, siz)

Purpose

Syntax

Description

Examples

See Also

(: (colon)) Colon:
shiftdim Shift dimensions
squeeze Remove singleton dimensions
**Purpose**
Convert between partial fraction expansion and polynomial coefficients

**Syntax**

\[
[r, p, k] = \text{residue}(b, a)\\
[b, a] = \text{residue}(r, p, k)
\]

**Description**
The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

\[
[r, p, k] = \text{residue}(b, a)
\]
finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, \(b(s)\) and \(a(s)\), of the form:

\[
\frac{b(s)}{a(s)} = \frac{b_1 + b_2 s^{-1} + b_3 s^{-2} + \ldots + b_{m+1} s^{-m}}{a_1 + a_2 s^{-1} + a_3 s^{-2} + \ldots + a_{n+1} s^{-n}}
\]

\[
[b, a] = \text{residue}(r, p, k)
\]
converts the partial fraction expansion back to the polynomials with coefficients in \(b\) and \(a\).

**Definition**
If there are no multiple roots, then:

\[
\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k(s)
\]

The number of poles \(n\) is

\[
n = \text{length}(h(a)) - 1 = \text{length}(h(r)) = \text{length}(h(p))
\]

The direct term coefficient vector is empty if \(\text{length}(h(b)) < \text{length}(h(a))\); otherwise

\[
\text{length}(h(k)) = \text{length}(h(b)) - \text{length}(h(a)) + 1
\]

If \(p(j) = \ldots = p(j+m-1)\) is a pole of multiplicity \(m\) then the expansion includes terms of the form

\[
\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s-p_j)^m}
\]
### residue

**Arguments**
- b, a Vectors that specify the coefficients of the polynomials in descending powers of s
- r Column vector of residues
- p Column vector of poles
- k Row vector of direct terms

**Algorithm**
The `residue` function is an M-file. It first obtains the poles with roots. Next, if the fraction is nonproper, the direct term k is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, the M-file `resi2` computes the residues at the repeated root locations.

**Limitations**
Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, \( a(s) \), is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

**See Also**
- `deconv` Deconvolution and polynomial division
- `poly` Polynomial with specified roots
- `roots` Polynomial roots

**References**
Purpose  
Return to the invoking function

Syntax  
\texttt{return}

Description
\texttt{return} causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples
If the determinant function were an M-file, it might use a \texttt{return} statement in handling the special case of an empty matrix as follows:

\begin{verbatim}
function \texttt{d = det(A)}
    \%\texttt{DET} \texttt{det(A)} \texttt{is the determinant of \texttt{A}}.
    if isempty(A)
        \texttt{d} = 1;
        \texttt{return}
    else
        \texttt{...}
    end
\end{verbatim}

See Also
break  
Break out of flow control structures

disp  
Display text or array

day  
Terminate \texttt{for}, \texttt{while}, \texttt{switch}, and \texttt{if} statements or indicate last index

date  
Display error messages

end  
Repeat statements a specific number of times

if  
Conditionally execute statements

keyboard  
Invoke the keyboard in an M-file

switch  
Switch among several cases based on expression

while  
Repeat statements an indefinite number of times
**rmfield**

**Purpose**
Remove structure fields

**Syntax**

```matlab
s = rmfield(s, 'field')
s = rmfield(s, FIELDS)
```

**Description**

`s = rmfield(s, 'field')` removes the specified field from the structure array `s`.

`s = rmfield(s, FIELDS)` removes more than one field at a time when `FIELDS` is a character array of field names or cell array of strings.

**See Also**

- `fields` Field names of a structure
- `getfield` Get field of structure array
- `setfield` Set field of structure array
- `strvcat` Vertical concatenation of strings
### rmpath

**Purpose**  
Remove directories from MATLAB’s search path

**Syntax**  
`rmpath directory`

**Description**  
`rmpath directory` removes the specified directory from MATLAB’s current search path.

**Remarks**  
The function syntax form is also acceptable:

```
rmpath('directory')
```

**Examples**  
```
rmpath /usr/local/matlab/mytools
```

**See Also**  
- `addpath`  
  Add directories to MATLAB’s search path  
- `path`  
  Control MATLAB’s directory search path
Purpose
Polynomial roots

Syntax
\[ r = \text{roots}(c) \]

Description
\[ r = \text{roots}(c) \] returns a column vector whose elements are the roots of the polynomial \( c \).

Row vector \( c \) contains the coefficients of a polynomial, ordered in descending powers. If \( c \) has \( n+1 \) components, the polynomial it represents is \[ c_1 s^n + ... + c_n s + c_{n+1}. \]

Remarks
Note the relationship of this function to \( p = \text{poly}(r) \), which returns a row vector whose elements are the coefficients of the polynomial. For vectors, \( \text{roots} \) and \( \text{poly} \) are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples
The polynomial \( s^3 - 6s^2 - 72s - 27 \) is represented in MATLAB as
\[ p = [1 -6 -72 -27] \]

The roots of this polynomial are returned in a column vector by
\[ r = \text{roots}(p) \]
\[ r = \\
12.1229 \\
-5.7345 \\
-0.3884 \]

Algorithm
The algorithm simply involves computing the eigenvalues of the companion matrix:
\[ A = \text{diag}(\text{ones}(n-2, 1), -1); \]
\[ A(1,:) = -c(2:n-1)/c(1); \]
\[ \text{eig}(A) \]

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix \( A \), but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in \( c \).
<table>
<thead>
<tr>
<th>See Also</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fzero</td>
<td>Zero of a function of one variable</td>
</tr>
<tr>
<td>poly</td>
<td>Polynomial with specified roots</td>
</tr>
<tr>
<td>residue</td>
<td>Convert between partial fraction expansion and polynomial coefficients</td>
</tr>
</tbody>
</table>
rot90

**Purpose**
Rotate matrix 90°

**Syntax**

\[ B = \text{rot90}(A) \]
\[ B = \text{rot90}(A, k) \]

**Description**

- \( B = \text{rot90}(A) \) rotates matrix \( A \) counterclockwise by 90 degrees.
- \( B = \text{rot90}(A, k) \) rotates matrix \( A \) counterclockwise by \( k \times 90 \) degrees, where \( k \) is an integer.

**Examples**

The matrix

\[ X = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix} \]

rotated by 90 degrees is

\[ Y = \text{rot90}(X) \]
\[ Y = \begin{bmatrix}
3 & 6 & 9 \\
2 & 5 & 8 \\
1 & 4 & 7
\end{bmatrix} \]

**See Also**

- flipdim
  - Flip array along a specified dimension
- fliplr
  - Flip matrices left-right
- flipud
  - Flip matrices up-down
Purpose       Round to nearest integer

Syntax       \( Y = \text{round}(X) \)

Description  \( Y = \text{round}(X) \) rounds the elements of \( X \) to the nearest integers. For complex \( X \), the imaginary and real parts are rounded independently.

Examples

\[
a =
\begin{bmatrix}
-1.9000 & -0.2000 & 3.4000 & 5.6000 \\
7.0000 & 2.4000 + 3.6000i
\end{bmatrix}
\]
\[
\text{round}(a)
\]
\[
\begin{bmatrix}
-2.0000 & 0 & 3.0000 & 6.0000 \\
7.0000 & 2.0000 + 4.0000i
\end{bmatrix}
\]

See Also
- \( \text{ceil} \)    Round toward infinity
- \( \text{fix} \)        Round towards zero
- \( \text{floor} \)      Round towards minus infinity
rref, rrefmovie

Purpose
Reduced row echelon form

Syntax
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
rrefmovie(A)

Description
R = rref(A) produces the reduced row echelon form of A using Gauss Jordan elimination with partial pivoting. A default tolerance of 
(max(size(A))*eps *norm(A,inf)) tests for negligible column elements.

[R,jb] = rref(A) also returns a vector jb so that:
• r = length(jb) is this algorithm's idea of the rank of A,
• x(jb) are the bound variables in a linear system Ax = b,
• A(:,jb) is a basis for the range of A,
• R(1:r,jb) is the r-by-r identity matrix.

[R,jb] = rref(A,tol) uses the given tolerance in the rank tests.
Roundoff errors may cause this algorithm to compute a different value for the rank than rank, orth and null.

rrefmovie(A) shows a movie of the algorithm working.

Examples
Use rref on a rank-deficient magic square:
A = magic(4), R = rref(A)
A =
16  2  3  13
5 11 10  8
9  7  6 12
4 14 15  1
R =
1  0  0  1
0  1  0  3
0  0  1 -3
0  0  0  0
### See Also

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv</td>
<td>Matrix inverse</td>
</tr>
<tr>
<td>lu</td>
<td>LU matrix factorization</td>
</tr>
<tr>
<td>rank</td>
<td>Rank of a matrix</td>
</tr>
</tbody>
</table>
rsf2csf

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:

\[ A = U \ast T \ast U' \text{ and } U' \ast U = \text{eye(size}(A)) \]. See schur for details.

Examples

Given matrix \( A \),

\[
\begin{bmatrix}
1 & 1 & 1 & 3 \\
1 & 2 & 1 & 1 \\
1 & 1 & 3 & 1 \\
-2 & 1 & 1 & 4
\end{bmatrix}
\]

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

\[
[u, t] = \text{schur}(A);
[U, T] = \text{rsf2csf}(u, t)
\]

yields a triangular matrix \( T \) whose diagonal consists of the eigenvalues of \( A \).

\[
U =
\begin{bmatrix}
-0.4576 + 0.3044i & 0.5802 - 0.4934i & -0.0197 & -0.3428 \\
0.1616 + 0.3556i & 0.4235 + 0.0051i & 0.1666 & 0.8001 \\
0.3963 + 0.2333i & 0.1718 + 0.2458i & 0.7191 & -0.4260 \\
-0.4759 - 0.3278i & -0.2709 - 0.2778i & 0.6743 & 0.2466
\end{bmatrix}
\]
\[ T = \begin{bmatrix}
1.9202 + 1.4742i & 0.7691 - 1.0772i & -1.5895 - 0.9940i & -1.3798 + 0.1864i \\
0 & 1.9202 - 1.4742i & 1.9296 + 1.6909i & 0.2511 + 1.0844i \\
0 & 0 & 4.8121 & 1.1314 \\
0 & 0 & 0 & 1.3474
\end{bmatrix} \]

**See Also**

schur  
Schur decomposition
Purpose

Save workspace variables on disk

Syntax

save
save filename
save filename variables
save filename options
save filename variables options

Description

save, by itself, stores all workspace variables in a binary format in the file named matlab.mat. The data can be retrieved with load.

save filename stores all workspace variables in filename.mat instead of the default matlab.mat. If filename is the special string stdio, the save command sends the data as standard output.

save filename variables saves only the workspace variables you list after the filename.

Options

The forms of the save command that use options are:

save filename options
save filename variables options,

Each specifies a particular ASCII data format, as opposed to the binary MAT-file format, in which to save data. Valid option combinations are:

<table>
<thead>
<tr>
<th>With these options...</th>
<th>Data is stored in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ascii</td>
<td>8-digit ASCII format</td>
</tr>
<tr>
<td>-ascii -double</td>
<td>16-digit ASCII format</td>
</tr>
<tr>
<td>-ascii -tabs</td>
<td>8-digit ASCII format, tab-separated</td>
</tr>
<tr>
<td>-ascii -double -tabs</td>
<td>16-digit ASCII format, tab-separated</td>
</tr>
</tbody>
</table>

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...
save

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.

<table>
<thead>
<tr>
<th>Element Range</th>
<th>Bytes per Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 255</td>
<td>1</td>
</tr>
<tr>
<td>0 to 65535</td>
<td>2</td>
</tr>
<tr>
<td>–32767 to 32767</td>
<td>2</td>
</tr>
<tr>
<td>(-2^{31} + 1) to (2^{31} - 1)</td>
<td>4</td>
</tr>
<tr>
<td>other</td>
<td>8</td>
</tr>
</tbody>
</table>

The Application Program Interface Libraries contain C and Fortran routines to read and write MAT-files from external programs. It is important to use recommended access methods, rather than rely upon the specific file format, which is likely to change in the future.
See Also

- fprintf: Write formatted data to file
- fwrite: Write binary data to a file
- load: Retrieve variables from disk
Purpose
Schur decomposition

Syntax

\[
[U, T] = \text{schur}(A)
\]

\[
T = \text{schur}(A)
\]

Description
The \text{schur} 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
\[
A = U T U^* \quad \text{and} \quad U^* U = \text{eye(size}(A))
\]

\[
T = \text{schur}(A)
\]
returns just the Schur matrix \(T\).

Remarks
The complex Schur form of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The real Schur form has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

If the matrix \(A\) is real, \text{schur} returns the real Schur form. If \(A\) is complex, \text{schur} returns the complex Schur form. The function \text{rsf2csf} converts the real form to the complex form.

Examples
\(H\) is a 3-by-3 eigenvalue test matrix:

\[
H =
\begin{pmatrix}
-149 & -50 & -154 \\
537 & 180 & 546 \\
-27 & -9 & -25
\end{pmatrix}
\]

Its Schur form is

\[
\text{schur}(H) =
\begin{pmatrix}
1.0000 & 7.1119 & 815.8706 \\
0 & 2.0000 & -55.0236 \\
0 & 0 & 3.0000
\end{pmatrix}
\]

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Algorithm
For real matrices, \text{schur} uses the EISPACK routines \text{ORTRAN}, \text{ORTHES}, and \text{HQR2}. \text{ORTHES} converts a real general matrix to Hessenberg form using orthogonal
similarity transformations. 

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, QZT, 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**


Purpose

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, subsequent MATLAB input is obtained from the file. Script files have a filename extension of .m and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (%) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

echo     Echo M-files during execution
function Function M-files
type     List file
Purpose
Secant and hyperbolic secant

Syntax
Y = sec(X)
Y = sech(X)

Description
The sec and sech commands operate element-wise on arrays. The functions’ domains and ranges include complex values. All angles are in radians.

Y = sec(X) returns an array the same size as X containing the secant of the elements of X.

Y = sech(X) returns an array the same size as X containing the hyperbolic secant of the elements of X.

Examples
Graph the secant over the domains \(-\pi/2 < x < \pi/2\) and \(\pi/2 < x < 3\pi/2\), and the hyperbolic secant over the domain \(-2\pi \leq x \leq 2\pi\).

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))
**sec, sech**

The expression \( \text{sec}(\pi/2) \) does not evaluate as infinite but as the reciprocal of the floating-point accuracy \( \varepsilon_p \), 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
**Purpose**

Return the set difference of two vectors

**Syntax**

\[
c = \text{setdiff}(a, b)
\]

\[
c = \text{setdiff}(A, B, 'rows')
\]

\[
[c, i] = \text{setdiff}(...)
\]

**Description**

\( c = \text{setdiff}(a, b) \) returns the values in \( a \) that are not in \( b \). The resulting vector is sorted in ascending order. In set theoretic terms, \( c = a - b \).

\( c = (A, B, 'rows') \) when \( A \) and \( B \) are matrices with the same number of columns returns the rows from \( A \) that are not in \( B \).

\([c, i] = \text{setdiff}(...)\) also returns an index vector \( i \) such that \( c = a(i) \) or \( c = a(i,:) \).

**Examples**

\[
A = \text{magic}(5);
B = \text{magic}(4);
[c, i] = \text{setdiff}(A, B);
\]

\[
c' = 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23 \ 24 \ 25
\]

\[
i' = 1 \ 10 \ 14 \ 18 \ 19 \ 23 \ 2 \ 6 \ 15
\]

**See Also**

- intersect
- ismember
- setxor
- union
- unique

Set intersection of two vectors
True for a set member
Set exclusive-or of two vectors
Set union of two vectors
Unique elements of a vector
setfield

Purpose
Set field of structure array

Syntax
s = setfield(s,'field',v)
s = setfield(s,{i,j},'field',{k},v)

Description
s = setfield(s,'field',v), where s is a 1-by-1 structure, sets the contents of the specified field to the value v. This is equivalent to the syntax s.field = v.

s = setfield(s,{i,j},'field',{k},v) sets the contents of the specified field to the value v. This is equivalent to the syntax s(i,j).field(k) = v. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to {i,j} and {k} above). Pass field references as strings.

Examples
Given the structure:
```matlab
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1
```
Then the command `mystr = setfield(mystr,{2,1},'name','ted')` yields
```matlab
mystr =

2x1 struct array with fields:
   name
   ID
```

See Also
fields Field names of a structure
getfield Get field of structure array
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Set string flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>This MATLAB 4 function has been renamed <code>char</code> in MATLAB 5.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td><code>char</code> Create character array (string)</td>
</tr>
</tbody>
</table>
Purpose
Set exclusive-or of two vectors

Syntax
\[
c = \text{setxor}(a, b)
\]
\[
c = \text{setxor}(A, B, 'rows')
\]
\[
[c, i, a, i b] = \text{setxor}(\ldots)
\]

Description
\[
c = \text{setxor}(a, b)
\]
returns the values that are not in the intersection of \(a\) and \(b\). The resulting vector is sorted.

\[
c = \text{setxor}(A, B, 'rows')\] when \(A\) and \(B\) are matrices with the same number of columns returns the rows that are not in the intersection of \(A\) and \(B\).

\[
[c, i, a, i b] = \text{setxor}(\ldots)
\] also returns index vectors \(i\), \(a\), and \(i\) such that \(c\) is a sorted combination of the elements \(c = a(i, a)\) and \(c = b(i, b)\) or, for row combinations, \(c = a(i, :)\) and \(c = b(i, :)\).

Examples
\[
a = [-1 0 1 \infty -\infty \text{NaN}];
b = [-2 \pi 0 \infty];
c = \text{setxor}(a, b)
\]
\[
c =
\]
\[
-\infty -2.0000 -1.0000 1.0000 3.1416 \text{NaN}
\]

See Also
\text{intersect} \quad \text{Set intersection of two vectors}
\text{ismember} \quad \text{True for a set member}
\text{setdiff} \quad \text{Set difference of two vectors}
\text{union} \quad \text{Set union of two vectors}
\text{unique} \quad \text{Unique elements of a vector}
Purpose
Shift dimensions

Syntax
B = shiftdim(X, n)
[B, nshifts] = shiftdim(X)

Description
B = shiftdim(X, n) shifts the dimensions of X by n. When n is positive, shiftdim shifts the dimensions to the left and wraps the n leading dimensions to the end. When n is negative, shiftdim shifts the dimensions to the right and pads with singletons.

[B, nshifts] = shiftdim(X) returns the array B with the same number of elements as X but with any leading singleton dimensions removed. A singleton dimension is any dimension for which size(A, dim) = 1. nshifts is the number of dimensions that are removed.

If X is a scalar, shiftdim has no effect.

Examples
The shiftdim command is handy for creating functions that, like sum or diff, work along the first nonsingleton dimension.

a = rand(1, 1, 3, 1, 2);
[b, n] = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b, -n); % c == a.
d = shiftdim(a, 3); % d is 1-by-2-by-1-by-1-by-3.

See Also
reshape
squeeze
Reshape array
Remove singleton dimensions
Purpose

Signum function

Syntax

\[ Y = \text{sign}(X) \]

Description

\[ Y = \text{sign}(X) \] returns an array \( Y \) the same size as \( X \), where each element of \( Y \) is:

- 1 if the corresponding element of \( X \) is greater than zero
- 0 if the corresponding element of \( X \) equals zero
- -1 if the corresponding element of \( X \) is less than zero

For nonzero complex \( X \), \( \text{sign}(X) = X ./ \text{abs}(X) \).

See Also

- \text{abs} Absolute value and complex magnitude
- \text{conj} Complex conjugate
- \text{imag} Imaginary part of a complex number
- \text{real} Real part of complex number
**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 \(-5 \leq x \leq 5\).

\[
x = -\pi : 0.01 : \pi; \quad \text{plot}(x, \sin(x)) \\
x = -5 : 0.01 : 5; \quad \text{plot}(x, \sinh(x))
\]

The expression \(\sin(\pi)\) is not exactly zero, but rather a value the size of the floating-point accuracy \(\text{eps}\), because \(\pi\) is only a floating-point approximation to the exact value of \(\pi\).
**sin, sinh**

**Algorithm**

\[
\sin(x + iy) = \sin(x) \cos(y) + i \cos(x) \sin(y)
\]

\[
\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}
\]

\[
\sinh(z) = \frac{e^z - e^{-z}}{2}
\]

**See Also**

asin, asinh

Inverse sine and inverse hyperbolic sine
Purpose

Array dimensions

Syntax

\[
d = \text{size}(X) \\
[m \ n] = \text{size}(X) \\
m = \text{size}(X, \text{dim}) \\
[d_1, d_2, \ldots, d_n] = \text{size}(X)
\]

Description

\(d = \text{size}(X)\) 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 \(\text{dim}\).

\([d_1, d_2, \ldots, d_n] = \text{size}(X)\) 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 \(\text{ndims}(X)\).

Examples

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.

\[
d = \text{size}(\text{rand}(2, 3, 4))
\]

\[
d = \\
2 \quad 3 \quad 4
\]
Here the size of each dimension is assigned to a separate variable.

\[
\begin{bmatrix}
m, n, p \end{bmatrix} = \text{size}(\text{rand}(2, 3, 4))
\]

\[
m = 2
\]

\[
n = 3
\]

\[
p = 4
\]

If \(X = \text{ones}(3, 4, 5)\), then

\[
\begin{bmatrix}
d_1, d_2, d_3 \end{bmatrix} = \text{size}(X)
\]

\[
d_1 = 3 \\
d_2 = 4 \\
d_3 = 5
\]

but when the number of output variables is less than \(\text{ndims}(X)\):

\[
\begin{bmatrix}
d_1, d_2 \end{bmatrix} = \text{size}(X)
\]

\[
d_1 = 3 \\
d_2 = 20
\]

The “extra” dimensions are collapsed into a single product.

If \(n > \text{ndims}(X)\), the “extra” variables all represent singleton dimensions:

\[
\begin{bmatrix}
d_1, d_2, d_3, d_4, d_5, d_6 \end{bmatrix} = \text{size}(X)
\]

\[
d_1 = 3 \\
d_2 = 4 \\
d_3 = 5
\]

\[
d_4 = 1 \\
d_5 = 1 \\
d_6 = 1
\]

**See Also**
- `exist`: Check if a variable or file exists
- `length`: Length of vector
- `whos`: List directory of variables in memory
**Purpose**
Sort elements in ascending order

**Syntax**

B = sort(A)

[B, INDEX] = sort(A)

B = sort(A, dim)

**Description**

B = sort(A) sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

Real, complex, and string elements are permitted. For identical values in A, the location in the input array determines location in the sorted list. When A is complex, the elements are sorted by magnitude, and where magnitudes are equal, further sorted by phase angle on the interval \([-\pi, \pi]\). If A includes any NaN elements, sort places these at the end.

If A is a vector, sort(A) arranges those elements in ascending order.

If A is a matrix, sort(A) treats the columns of A as vectors, returning sorted columns.

If A is a multidimensional array, sort(A), treats the values along the first non-singleton dimension as vectors, returning an array of sorted vectors.

[B, INDEX] = sort(A) also returns an array of indices. INDEX is an array of size(A), each column of which is a permutation vector of the corresponding column of A. If A has repeated elements of equal value, indices are returned that preserve the original relative ordering.

B = sort(A, dim) sorts the elements along the dimension of A specified by scalar dim.

If dim is a vector, sort works iteratively on the specified dimensions. Thus, sort(A, [1 2]) is equivalent to sort(sort(A, 2), 1).

**See Also**

max
Minimum elements of an array

mean
Average or mean value of arrays

median
Median value of arrays

min
Minimum elements of an array

sortrows
Sort rows in ascending order
sortrows

Purpose
Sort rows in ascending order

Syntax
B = sortrows(A)
B = sortrows(A, column)
[B, index] = sortrows(A)

Description
B = sortrows(A) sorts the rows of A as a group in ascending order. Argument A must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When A is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval \([-\pi, \pi]\).

B = sortrows(A, column) sorts the matrix based on the columns specified in the vector column. For example, sortrows(A, [2 3]) sorts the rows of A by the second column, and where these are equal, further sorts by the third column.

[B, index] = sortrows(A) also returns an index vector index.

If A is a column vector, then B = A(index).

If A is an m-by-n matrix, then B = A(index,:).

Examples
Given the 5-by-5 string matrix,

\[
A = ['one ';'two ';'three ';'four ';'five '];
\]

The commands B = sortrows(A) and C = sortrows(A, 1) yield

\[
B = \\
five 
four 
one 
three 
two
\]

\[
C = \\
four 
five 
one 
two 
three
\]

See Also
sort Sort elements in ascending order
Purpose
Convert vector into sound

Syntax
sound(y, Fs)
sound(y)
sound(y, Fs, bits)

Description
sound(y, Fs), sends the signal in vector y (with sample frequency Fs) to the speaker on PC, Macintosh, and most UNIX platforms. Values in y are assumed to be in the range \(-1.0 \leq y \leq 1.0\). Values outside that range are clipped. Stereo sound is played on platforms that support it when y is an n-by-2 matrix.

sound(y) plays the sound at the default sample rate or 8192 Hz.

sound(y, Fs, bits) plays the sound using bits bits/sample if possible. Most platforms support bits = 8 or bits = 16.

Remarks
MATLAB supports all Windows-compatible sound devices.

See Also
auread Read NeXT/SUN (.au) sound file
auwrite Write NeXT/SUN (.au) sound file
soundscale Scale data and play as sound
wavread Read Microsoft WAVE (.wav) sound file
wavwrite Write Microsoft WAVE (.wav) sound file
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Sound capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>soundcap</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><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.</td>
</tr>
</tbody>
</table>
Purpose
Scale data and play as sound

Syntax
soundsc(y, Fs)
soundsc(y)
soundsc(y, Fs, bits)
soundsc(y, ..., slim)

Description
soundsc(y, Fs) sends the signal in vector y (with sample frequency Fs) to the speaker on PC, Macintosh, and most UNIX platforms. The signal y is scaled to the range \(-1.0 \leq y \leq 1.0\) before it is played, resulting in a sound that is played as loud as possible without clipping.

soundsc(y) plays the sound at the default sample rate or 8192 Hz.

soundsc(y, Fs, bits) plays the sound using bits bits/sample if possible. Most platforms support bits = 8 or bits = 16.

soundsc(y, ..., slim) where slim = [slow shigh] maps the values in y between slow and shigh to the full sound range. The default value is slim = [min(y) max(y)].

Remarks
MATLAB supports all Windows-compatible sound devices.

See Also
auread Read NeXT/SUN (.au) sound file
auwrite Write NeXT/SUN (.au) sound file
sound Convert vector into sound
wavread Read Microsoft WAVE (.wav) sound file
wavwrite Write Microsoft WAVE (.wav) sound file
Purpose
Allocate space for sparse matrix

Syntax
$S = \text{spalloc}(m, n, \text{nzmax})$

Description
$S = \text{spalloc}(m, n, \text{nzmax})$ creates an all zero sparse matrix $S$ of size $m$-by-$n$ with room to hold $\text{nzmax}$ nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

$\text{spalloc}(m, n, \text{nzmax})$ is shorthand for

$$\text{sparse}([], [], [], m, n, \text{nzmax})$$

Examples
To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

$$S = \text{spalloc}(n, n, 3*n);$$
$$\text{for} \ j = 1: n$$
$$S(:, j) = [\text{zeros}(n-3, 1)' \ \text{round(rand(3, 1))}' ];$$
$$\text{end}$$
Purpose
Create sparse matrix

Syntax
- \( S = \text{sparse}(A) \)
- \( S = \text{sparse}(i, j, s, m, n, nzmax) \)
- \( S = \text{sparse}(i, j, s, m, n) \)
- \( S = \text{sparse}(i, j, s) \)
- \( S = \text{sparse}(m, n) \)

Description
The `sparse` function generates matrices in MATLAB's sparse storage organization.

- \( S = \text{sparse}(A) \) converts a full matrix to sparse form by squeezing out any zero elements. If \( S \) is already sparse, `sparse(S)` returns \( S \).

- \( S = \text{sparse}(i, j, s, m, n, nzmax) \) uses vectors \( i, j, \) and \( s \) to generate an \( m \times n \) sparse matrix with space allocated for \( nzmax \) nonzeros. Any elements of \( s \) that are zero are ignored, along with the corresponding values of \( i \) and \( j \). Vectors \( i, j, \) and \( s \) are all the same length.

To simplify this six-argument call, you can pass scalars for the argument \( s \) and one of the arguments \( i \) or \( j \) — in which case they are expanded so that \( i, j, \) and \( s \) all have the same length.

- \( S = \text{sparse}(i, j, s, m, n) \) uses \( nzmax = \text{length}(s) \).

- \( S = \text{sparse}(i, j, s) \) uses \( m = \max(i) \) and \( n = \max(j) \). The maxima are computed before any zeros in \( S \) are removed, so one of the rows of \([i \ j \ s]\) might be \([m \ n \ 0]\).

- \( S = \text{sparse}(m, n) \) abbreviates \( \text{sparse([],[],[],m,n,0)} \). This generates the ultimate sparse matrix, an \( m \times n \) all zero matrix.

Remarks
All of MATLAB's built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, \( A \cdot S \) is at least as sparse as \( S \).
sparse

Examples

S = sparse(1: n, 1: n, 1) generates a sparse representation of the n-by-n identity matrix. The same S results from $S = \text{sparse}(\text{eye}(n, n))$, 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:

$$[i, j, s] = \text{find}(S);$$
$$[m, n] = \text{size}(S);$$
$$S = \text{sparse}(i, j, s, m, n);$$

So does this, if the last row and column have nonzero entries:

$$[i, j, s] = \text{find}(S);$$
$$S = \text{sparse}(i, j, s);$$

See Also

The sparfun directory, and:

- diag
- find
- full
- nnz
- nonzeros
- nzmax
- spones
- sprandn
- sprandsym
- spy
**Purpose**
Import matrix from sparse matrix external format

**Syntax**

\[ S = \text{spconvert}(D) \]

**Description**

\text{spconvert} is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. \text{spconvert} is the second step in the process:

1. Load an ASCII data file containing \([i, j, v]\) or \([i, j, r, e, i m]\) as rows into a MATLAB variable.
2. Convert that variable into a MATLAB sparse matrix.

\[ S = \text{spconvert}(D) \] converts a matrix \(D\) with rows containing \([i, j, s]\) or \([i, j, r, s]\) to the corresponding sparse matrix. \(D\) must have an \(\text{nnz}\) or \(\text{nnz}+1\) row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form \([m \ n \ 0]\) or \([m \ n \ 0 \ 0]\) anywhere in \(D\) can be used to specify \(\text{size}(S)\). If \(D\) is already sparse, no conversion is done, so \text{spconvert} can be used after \(D\) is loaded from either a MAT-file or an ASCII file.

**Examples**

Suppose the ASCII file \texttt{uphill.dat} contains

\[
\begin{array}{llll}
1 & 1 & 1.000000000000000 & 0.000000000000000 \\
1 & 2 & 0.500000000000000 & 0.000000000000000 \\
2 & 2 & 0.333333333333333 & 0.000000000000000 \\
1 & 3 & 0.333333333333333 & 0.000000000000000 \\
2 & 3 & 0.250000000000000 & 0.000000000000000 \\
3 & 3 & 0.200000000000000 & 0.000000000000000 \\
1 & 4 & 0.250000000000000 & 0.000000000000000 \\
2 & 4 & 0.200000000000000 & 0.000000000000000 \\
3 & 4 & 0.166666666666667 & 0.000000000000000 \\
4 & 4 & 0.142857142857143 & 0.000000000000000 \\
4 & 4 & 0.000000000000000 & 0.000000000000000 \\
\end{array}
\]

Then the statements

\[
\begin{align*}
\text{load} & \quad \text{uphill.dat} \\
H & = \text{spconvert}(\text{uphill})
\end{align*}
\]
spconvert

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

**Purpose**
Extract and create sparse band and diagonal matrices

**Syntax**

```matlab
[B, d] = spdiags(A)
B = spdiags(A, d)
A = spdiags(B, d, A)
A = spdiags(B, d, m, n)
```

**Description**
The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible:

- `[B, d] = spdiags(A)` extracts all nonzero diagonals from the `m` by `n` matrix `A`. `B` is an `m` by `n` matrix whose columns are the `p` nonzero diagonals of `A`. `d` is a vector of length `p` whose integer components specify the diagonals in `A`

- `B = spdiags(A, d)` extracts the diagonals specified by `d`.

- `A = spdiags(B, d, A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

- `A = spdiags(B, d, m, n)` creates an `m` by `n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

**Remarks**
If a column of `B` is longer than the diagonal it's replacing, `spdiags` takes elements from `B`'s tail.

**Arguments**
The `spdiags` function deals with three matrices, in various combinations, as both input and output:

- `A` An `m` by `n` matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on `p` diagonals.

- `B` An `m` by `n` (`m` by `n`) -by- `p` matrix, usually (but not necessarily) full, whose columns are the diagonals of `A`.

- `d` A vector of length `p` whose integer components specify the diagonals in `A`.

2-609
Roughly, A, B, and d are related by

```matlab
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.

```matlab
e = ones(n, 1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson’s test matrix (see gallery):

```matlab
A = spdiags(abs(-(n-1)/2:(n-1)/2)', 0, A)
```

Finally, recover the three diagonals:

```matlab
B = spdiags(A)
```

The second example is not square.

```matlab
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

```matlab
B = [41 11 0
     52 22 0
     63 33 13
     74 44 24]
```
Conversely, with the above B and d, the expression `spdiags(B, d, 7, 4)` reproduces the original A.

See Also

`diag` Diagonal matrices and diagonals of a matrix
speak

**Purpose**
Speak text string

**Syntax**
speak(y)
speak(y, voice)

**Description**
speak(y) speaks the text string y using the default voice.

speak(y, voice) speaks the text string y using the voice specified by voice.

speak requires the Speech Manager and works only on the Macintosh.

**Examples**

speak('I like math.')
speak('I really like matlab', 'good news')
Purpose

Sparse identity matrix

Syntax

S = speye(m,n)
S = speye(n)

Description

S = speye(m,n) forms an m-by-n sparse matrix with 1s on the main diagonal.
S = speye(n) abbreviates speye(n,n).

Examples

I = speye(1000) forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as I = sparse(eye(1000,1000)), but the latter requires eight megabytes for temporary storage for the full representation.

See Also

spalloc  Allocate space for sparse matrix
spones   Replace nonzero sparse matrix elements with ones
spdiags  Extract and create sparse band and diagonal matrices
sprand  Sparse uniformly distributed random matrix
sprandn Sparse normally distributed random matrix
Purpose
Apply function to nonzero sparse matrix elements

Syntax
f = spfun('function', S)

Description
The spfun function selectively applies a function to only the nonzero elements of a sparse matrix, preserving the sparsity pattern of the original matrix (except for underflow).

f = spfun('function', S) evaluates function(S) on the nonzero elements of S. function must be the name of a function, usually defined in an M-file, which can accept a matrix argument, S, and evaluate the function at each element of S.

Remarks
Functions that operate element-by-element, like those in the elfun directory, are the most appropriate functions to use with spfun.

Examples
Given the 4-by-4 sparse diagonal matrix

\[
S = \\
(1, 1) 1 \\
(2, 2) 2 \\
(3, 3) 3 \\
(4, 4) 4
\]

f = spfun('exp', S) has the same sparsity pattern as S:

\[
f = \\
(1, 1) 2.7183 \\
(2, 2) 7.3891 \\
(3, 3) 20.0855 \\
(4, 4) 54.5982
\]

whereas exp(S) has 1s where S has 0s.

\[
\text{full(exp(S))}
\]

\[
ans = \\
2.7183 1.0000 1.0000 1.0000 \\
1.0000 7.3891 1.0000 1.0000 \\
1.0000 1.0000 20.0855 1.0000 \\
1.0000 1.0000 1.0000 54.5982
\]
**Purpose**
Transform spherical coordinates to Cartesian

**Syntax**
\[ [x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, R) \]

**Description**
\[ [x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, R) \]
transforms the corresponding elements of spherical coordinate arrays to Cartesian, or \( x, y, z \), coordinates. \( \text{THETA}, \text{PHI} \), and \( R \) must all be the same size. \( \text{THETA} \) and \( \text{PHI} \) are angular displacements in radians from the positive \( x \)-axis and from the \( x-y \) plane, respectively.

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

\[
\begin{align*}
x &= r \cdot \cos(\phi) \cdot \cos(\theta) \\
y &= r \cdot \cos(\phi) \cdot \sin(\theta) \\
z &= r \cdot \sin(\phi)
\end{align*}
\]

**See Also**
cart2pol
Transform Cartesian coordinates to polar or cylindrical
cart2sph
Transform Cartesian coordinates to spherical
pol2cart
Transform polar or cylindrical coordinates to Cartesian
spline

Purpose
Cubic spline interpolation

Syntax
yi = spline(x, y, xi)
p = spline(x, y)

Description
The spline function interpolates between data points using cubic spline fits.

yi = spline(x, y, xi) accepts vectors x and y that contain coarsely spaced data, and vector xi that specifies a new, more finely spaced abscissa. The function uses cubic spline interpolation to find a vector yi corresponding to xi.

pp = spline(x, y) returns the pp-form of the cubic spline interpolant, for later use with ppval and other spline functions.

Examples
The two vectors

\[
t = 1900:10:1990;
p = [75.995 91.972 105.711 123.203 131.669 \ldots \\
150.697 179.323 203.212 226.505 249.633 ]';
\]

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

\[
spline(t, p, 2000)
\]

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

\[
ans = \\
270.6060
\]

The statements

\[
x = 1900:1:2000;
y = spline(t, p, x);
plot(t, p, 'o', x, y)
\]
interpolate the data with a cubic spline, evaluate that spline for each year from 1900 to 2000, and plot the result.

Algorithm

spline is a MATLAB M-file. It uses the M-files ppval, mkpp, and unmkpp. These routines form a small suite of functions for working with piecewise polynomials. spline uses these functions in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see the M-files and the Spline Toolbox.

See Also

interp1 One-dimensional data interpolation (table lookup)
ppval Evaluate piecewise polynomial

References

spones

Purpose
Replace nonzero sparse matrix elements with ones

Syntax
R = spones(S)

Description
R = spones(S) generates a matrix R with the same sparsity structure as S, but with 1’s in the nonzero positions.

Examples
c = sum(spones(S)) is the number of nonzeros in each column.
r = sum(spones(S'))’ is the number of nonzeros in each row.
sum(c) and sum(r) are equal, and are equal to nnz(S).

See Also
nnz Number of nonzero matrix elements
spalloc Allocate space for sparse matrix
spfun Apply function to nonzero sparse matrix elements
Purpose
Set parameters for sparse matrix routines

Syntax
spparms('key', value)
spparms(values)
[keys, values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')

Description
spparms('key', value) sets one or more of the tunable parameters used in the sparse linear equation operators, \ and /, and the minimum degree orderings, colmd and symmd. In ordinary use, you should never need to deal with this function.

The meanings of the key parameters are

'spumoni' Sparse Monitor flag.
0 produces no diagnostic output, the default.
1 produces information about choice of algorithm based on matrix structure, and about storage allocation.
2 also produces very detailed information about the minimum degree algorithms.

'thr_rel', 'thr_abs' Minimum degree threshold is thr_rel * mindegree + thr_abs.

'exact_d' Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.

'supernd' If positive, minimum degree amalgamates the supernodes every supernd stages.

'rreduce' If positive, minimum degree does row reduction every rreduce stages.

'wh_frac' Rows with density > wh_frac are ignored in colmd.
spparms

's pparms', 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

'autommd' Nonzero to use minimum degree orderings with \ and /.
'aug_rel', 'aug_abs' Residual scaling parameter for augmented equations is
    aug_rel * max(max(abs(A))) + aug_abs.
    For example, aug_rel = 0, aug_abs = 1 puts an unscaled identity matrix in the (1,1) block of the augmented matrix.
### spparms

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default</th>
<th>Tight</th>
</tr>
</thead>
<tbody>
<tr>
<td>val ues(1) 'spumoni'</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>val ues(2) 'thr_rel'</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>val ues(3) 'thr_abs'</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>val ues(4) 'exact_d'</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>val ues(5) 'supernd'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>val ues(6) 'rreduce'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>val ues(7) 'wh_frac'</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>val ues(8) 'autommd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>val ues(9) 'aug_rel'</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>val ues(10) 'aug_abs'</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

### See Also
- \Matrix left division (backslash)
- colmmd Sparse column minimum degree permutation
- symmmd Sparse symmetric minimum degree ordering

### References
**sprand**

**Purpose**
Sparse uniformly distributed random matrix

**Syntax**

- \( R = \text{sprand}(S) \)
- \( R = \text{sprand}(m,n,density) \)
- \( R = \text{sprand}(m,n,density,rc) \)

**Description**

- \( R = \text{sprand}(S) \) has the same sparsity structure as \( S \), but uniformly distributed random entries.
- \( R = \text{sprand}(m,n,density) \) is a random, \( m \times n \), sparse matrix with approximately \( density \times m \times n \) uniformly distributed nonzero entries \( (0 \leq \text{density} \leq 1) \).
- \( R = \text{sprand}(m,n,density,rc) \) also has reciprocal condition number approximately equal to \( rc \). \( R \) is constructed from a sum of matrices of rank one.

If \( rc \) is a vector of length \( lr \), where \( lr \leq \min(m,n) \), then \( R \) has \( rc \) as its first \( lr \) singular values, all others are zero. In this case, \( R \) is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

**See Also**

- **sprandn**
  Sparse normally distributed random matrix
- **sprandsym**
  Sparse symmetric random matrix
### Purpose
Sparse normally distributed random matrix

### Syntax
- `R = sprandn(S)`
- `R = sprandn(m, n, density)`
- `R = sprandn(m, n, density, rc)`

### Description
- **`R = sprandn(S)`** has the same sparsity structure as `S`, but normally distributed random entries with mean 0 and variance 1.
- **`R = sprandn(m, n, density)`** is a random, m-by-n, sparse matrix with approximately `density * m * n` normally distributed nonzero entries (`0 ≤ density ≤ 1`).
- **`R = sprandn(m, n, density, rc)`** also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.
  - If `rc` is a vector of length `lr`, where `lr ≤ min(m, n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

### See Also
- **sprand** Sparse uniformly distributed random matrix
- **sprandn** Sparse normally distributed random matrix
Purpose

Sparse symmetric random matrix

Syntax

\[
\begin{align*}
R &= \text{sprandsym}(S) \\
R &= \text{sprandsym}(n, \text{density}) \\
R &= \text{sprandsym}(n, \text{density}, rc) \\
R &= \text{sprandsym}(n, \text{density}, rc, \text{kind})
\end{align*}
\]

Description

\[
\begin{align*}
\text{R = sprandsym}(S) & \text{ returns a symmetric random matrix whose lower triangle and diagonal have the same structure as S. Its elements are normally distributed, with mean 0 and variance 1.} \\
\text{R = sprandsym}(n, \text{density}) & \text{ returns a symmetric random, n-by-n, sparse matrix with approximately density \(n \times n\) nonzeros; each entry is the sum of one or more normally distributed random samples, and (0 \leq \text{density} \leq 1).} \\
\text{R = sprandsym}(n, \text{density}, rc) & \text{ returns a matrix with a reciprocal condition number equal to \(rc\). The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in [-1, 1].} \\
\text{If } rc & \text{ is a vector of length } n, \text{ then } R \text{ has eigenvalues } rc. \text{ Thus, if } rc \text{ is a positive (nonnegative) vector then } R \text{ is a positive definite matrix. In either case, } R \text{ 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.} \\
\text{R = sprandsym}(n, \text{density}, rc, \text{kind}) & \text{ returns a positive definite matrix. Argument } \text{kind} \text{ can be:} \\
\text{1 to generate } R & \text{ by random Jacobi rotation of a positive definite diagonal matrix. } R \text{ has the desired condition number exactly.} \\
\text{2 to generate an } R & \text{ that is a shifted sum of outer products. } R \text{ has the desired condition number only approximately, but has less structure.} \\
\text{3 to generate an } R & \text{ that has the same structure as the matrix } S \text{ and approximate condition number } 1/rc. \text{ density is ignored.}
\end{align*}
\]

See Also

\[
\begin{align*}
\text{sprand} & \quad \text{Sparse uniformly distributed random matrix} \\
\text{sprandn} & \quad \text{Sparse normally distributed random matrix}
\end{align*}
\]
Purpose
Write formatted data to a string

Syntax
`s = sprintf(format, A,...)`  
`[s, errmsg] = sprintf(format, A,...)`

Description
`s = sprintf(format, A,...)` formats the data in matrix `A` (and in any additional matrix arguments) under control of the specified `format` string, and returns it in the MATLAB string variable `s`. `sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

The `format` string specifies notation, alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters; along with escape characters, conversion specifiers, and other characters, organized as shown below:

```
%–12.5e
```

- Initial `%` character
- Flag
- Field width and precision
- Conversion character

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.
**Remarks**

The `sprintf` function behaves like its ANSI C language `sprintf()` namesake with certain exceptions and extensions. These include:

1. The following nonstandard subtype specifiers are supported for conversion specifiers `%o`, `%u`, `%x`, and `%X`.
   - The underlying C data type is a float rather than an unsigned integer.
   - 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**

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\ \</td>
<td>Backslash</td>
</tr>
<tr>
<td>\ &quot; or &quot;</td>
<td>Single quotation mark</td>
</tr>
<tr>
<td>%%</td>
<td>Percent character</td>
</tr>
</tbody>
</table>
Conversion characters specify the notation of the output.

### Conversion Specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td>%d</td>
<td>Decimal notation (signed)</td>
</tr>
<tr>
<td>%e</td>
<td>Exponential notation (using a lowercase e as in 3. 1415e+00)</td>
</tr>
<tr>
<td>%E</td>
<td>Exponential notation (using an uppercase E as in 3. 1415E+00)</td>
</tr>
<tr>
<td>%f</td>
<td>Fixed-point notation</td>
</tr>
<tr>
<td>%g</td>
<td>The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.</td>
</tr>
<tr>
<td>%G</td>
<td>Same as %g, but using an uppercase E</td>
</tr>
<tr>
<td>%o</td>
<td>Octal notation (unsigned)</td>
</tr>
<tr>
<td>%s</td>
<td>String of characters</td>
</tr>
<tr>
<td>%u</td>
<td>Decimal notation (unsigned)</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal notation (using lowercase letters a–f)</td>
</tr>
<tr>
<td>%X</td>
<td>Hexadecimal notation (using uppercase letters A–F)</td>
</tr>
</tbody>
</table>

Other characters can be inserted into the conversion specifier between the `%` and the conversion character.
Other Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A minus sign (-)</td>
<td>Left-justifies the converted argument in its field.</td>
<td>% 5. 2d</td>
</tr>
<tr>
<td>A plus sign (+)</td>
<td>Always prints a sign character (+ or -).</td>
<td>%+5. 2d</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Pad with zeros rather than spaces.</td>
<td>%05. 2d</td>
</tr>
<tr>
<td>Digits (field width)</td>
<td>A digit string specifying the minimum number of digits to be printed.</td>
<td>%6f</td>
</tr>
<tr>
<td>Digits (precision)</td>
<td>A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.</td>
<td>%6. 2f</td>
</tr>
</tbody>
</table>

Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>sprintf(’%0.5g’, (1+sqrt(5))/2)</td>
<td>1.618</td>
</tr>
<tr>
<td>sprintf(’%0.5g’, 1/eps)</td>
<td>4.5036e+15</td>
</tr>
<tr>
<td>sprintf(’%4.5f’, 1/eps)</td>
<td>4503599627370496.00000</td>
</tr>
<tr>
<td>sprintf(’%d’, round(pi))</td>
<td>3</td>
</tr>
<tr>
<td>sprintf(’%s’, ’hello’)</td>
<td>hello</td>
</tr>
<tr>
<td>sprintf(’The array is %dx%d.’, 2, 3)</td>
<td>The array is 2x3</td>
</tr>
<tr>
<td>sprintf(’\n’)</td>
<td>Line termination character on all platforms</td>
</tr>
</tbody>
</table>

See Also

int2str, num2str, sscanf

References


2-628
Purpose

Visualize sparsity pattern

Syntax

spy(S)
spy(S, markerSize)
spy(S, 'LineSpec')
spy(S, 'LineSpec', markerSize)

Description

spy(S) plots the sparsity pattern of any matrix S.

spy(S, markerSize), where markerSize is an integer, plots the sparsity pattern using markers of the specified point size.

spy(S, 'LineSpec'), where LineSpec is a string, uses the specified plot marker type and color.

spy(S, 'LineSpec', markerSize) uses the specified type, color, and size for the plot markers.

S is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

spy replaces format +, which takes much more space to display essentially the same information.

See Also

The gplot and LineSpec reference entries in the MATLAB Graphics Guide, and:

find Find indices and values of nonzero elements
symmd Sparse symmetric minimum degree ordering
symrcm Sparse reverse Cuthill-McKee ordering
Purpose

Square root

Syntax

\[ B = \text{sqrt}(A) \]

Description

\[ B = \text{sqrt}(A) \] returns the square root of each element of the array \( X \). For the elements of \( X \) that are negative or complex, \( \text{sqrt}(X) \) produces complex results.

Remarks

See \text{sqrtm} for the matrix square root.

Examples

\[
\text{sqrt}((-2:2)')
\]

\[
\begin{array}{c}
\text{ans} = \\
0 + 1.4142i \\
0 + 1.0000i \\
0 \\
1.0000 \\
1.4142 \\
\end{array}
\]

See Also

\text{sqrtm} Matrix square root
Purpose

Matrix square root

Syntax

```matlab
Y = sqrtm(X)
[Y, esterr] = sqrtm(X)
```

Description

Y = sqrtm(X) is the matrix square root of X. Complex results are produced if X has negative eigenvalues. A warning message is printed if the computed Y*Y is not close to X.

[Y, esterr] = sqrtm(X) does not print any warning message, but returns an estimate of the relative residual, norm(Y*Y−X) / norm(X).

Remarks

If X is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like X = [0 1; 0 0], do not have any square roots, real or complex, and sqrtm cannot be expected to produce one.

Examples

A matrix representation of the fourth difference operator is

```matlab
X =
    5   -4    1    0    0
   -4     6   -4    1    0
    1   -4     6   -4    1
    0    1   -4     6   -4
    0    0    1   -4     5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, Y = sqrtm(X), is a representation of the second difference operator.

```matlab
Y =
    2   -1   -0    0   -0
   -1    2   -1   -0   -0
   -0   -1    2   -1    0
    0   -0   -1    2   -1
   -0   -0    0   -1    2
```
The matrix
\[ X = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix} \]
has four square roots. Two of them are
\[ Y_1 = \begin{bmatrix} 1.5667 & 1.7408 \\ 2.6112 & 4.1779 \end{bmatrix} \]
and
\[ Y_2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \]
The other two are \(-Y_1\) and \(-Y_2\). All four can be obtained from the eigenvalues and vectors of \(X\).

\[
[V, D] = \text{eig}(X);
\]
\[ D = \begin{bmatrix} 0.1386 & 0 \\ 0 & 28.8614 \end{bmatrix} \]
The four square roots of the diagonal matrix \(D\) result from the four choices of sign in
\[ S = \begin{bmatrix} \pm 0.3723 & 0 \\ 0 & \pm 5.3723 \end{bmatrix} \]
All four \(Y\)s are of the form
\[ Y = V^*S/V \]
The \texttt{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
\[ X = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \]
does not have any square roots. There is no matrix \( Y \), real or complex, for which \( Y^* Y = X \). The statement

\[
    Y = \sqrt{\text{m}(X)}
\]

produces several warning messages concerning accuracy and the answer

\[
    Y =
\]

\[
    1.0e+03 *
\]

\[
    0.0000 + 0.0000i \quad 4.9354 - 7.6863i \\
    0.0000 + 0.0000i \quad 0.0000 + 0.0000i
\]

**Algorithm**

The function \( \sqrt{\text{m}}(X) \) is an abbreviation for \( \text{funm}(X, 'sqrt') \). The algorithm used by \( \text{funm} \) is based on a Schur decomposition. It can fail in certain situations where \( X \) has repeated eigenvalues. See \( \text{funm} \) for details.

**See Also**

- \( \text{expm} \)  
  Matrix exponential
- \( \text{funm} \)  
  Evaluate functions of a matrix
- \( \text{logm} \)  
  Matrix logarithm
**Purpose**
Remove singleton dimensions

**Syntax**
\[ B = \text{squeeze}(A) \]

**Description**
\[ B = \text{squeeze}(A) \] returns an array \( B \) with the same elements as \( A \), but with all singleton dimensions removed. A singleton dimension is any dimension for which \( \text{size}(A, \text{dim}) = 1 \).

**Examples**
Consider the 2-by-1-by-3 array \( Y = \text{rand}(2, 1, 3) \). This array has a singleton column dimension — that is, there's only one column per page.

\[
\begin{array}{ll}
Y(:,:,1) &= Y(:,:,2) = \\
0.5194 & 0.0346 \\
0.8310 & 0.0535 \\
Y(:,:,3) &= \\
& 0.5297 \\
& 0.6711
\end{array}
\]

The command \( Z = \text{squeeze}(Y) \) yields a 2-by-3 matrix:

\[
\begin{array}{lll}
0.5194 & 0.0346 & 0.5297 \\
0.8310 & 0.0535 & 0.6711
\end{array}
\]

**See Also**
- reshape
- shiftdim

Reshape array
Shift dimensions
Purpose
Read string under format control

Syntax
A = sscanf(s,format)
A = sscanf(s,format,size)
[A,count,errmsg,nextindex] = sscanf(...)

Description
A = sscanf(s,format) reads data from the MATLAB string variable s, converts it according to the specified format string, and returns it in matrix A. format is a string specifying the format of the data to be read. See “Remarks” for details. sscanf is the same as fscanf except that it reads the data from a MATLAB string variable rather than reading it from a file.

A = sscanf(s,format,size) reads the amount of data specified by size and converts it according to the specified format string. size is an argument that determines how much data is read. Valid options are:
n Read n elements into a column vector.
inf Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
[m,n] Read enough elements to fill an m-by-n matrix, filling the matrix in column order. n can be Inf, but not m.

If the matrix A results from using character conversions only and size is not of the form [M,N], a row vector is returned.

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

[A,count,errmsg,nextindex] = sscanf(...) reads data from MATLAB string variable s, converts it according to the specified format string, and returns it in matrix A. count is an optional output argument that returns the number of elements successfully read. errmsg is an optional output argument that returns an error message string if an error occurred or an empty matrix if an error did not occur. nextindex is an optional output argument specifying one more than the number of characters scanned in s.
Remarks

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

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

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

- An asterisk (*) Skip over the matched value, if the value is matched but not stored in the output matrix.
- A digit string Maximum field width.
- A letter The size of the receiving object; for example, h for short as in %d for a short integer, or l for long as in %ld for a long integer or %g 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
- % Signed integer
- %o Signed octal integer
- %s A series of non-whitespace characters
- %u Signed decimal integer
If `%` is used, an element read may use several MATLAB matrix elements, each holding one character. Use `%c` to read space characters; the format `%` 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

```matlab
s = '2.7183 3.1416';
A = sscanf(s, '%f')
```

create a two-element vector containing poor approximations to e and pi.

**See Also**

- `eval` Interpret strings containing MATLAB expressions
- `sprintf` Write formatted data to a string
**Purpose**

MATLAB startup M-file

**Syntax**

startup

**Description**

At startup time, MATLAB automatically executes the master M-file `matlabrc.m`, if it exists, and `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on MATLAB's search path.

You can create a startup file in your own MATLAB directory. The file can include physical constants, handle graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

**Algorithm**

Only `matlabrc.m` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```matlab
if exist('startup') == 2
    startup
end
```

that invoke `startup.m`. You can extend this process to create additional startup M-files, if required.

**See Also**

- Operating system command: `exist`
- Check if a variable or file exists
- `matlabrc` MATLAB startup M-file
- `path` Control MATLAB's directory search path
- `quit` Terminate MATLAB
**Purpose**
Standard deviation

**Syntax**

\[
s = \text{std}(X) \\
s = \text{std}(X, \text{flag}) \\
s = \text{std}(X, \text{flag}, \text{dim})
\]

**Definition**
There are two common textbook definitions for the standard deviation \( s \) of a data vector \( X \):

\[
(1) \quad s = \left( \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}
\quad \text{and} \quad
(2) \quad s = \left( \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}
\]

where

\[
\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i
\]

and \( n \) is the number of elements in the sample. The two forms of the equation differ only in \( n - 1 \) versus \( n \) in the divisor.

**Description**

\( s = \text{std}(X) \), 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 unbiased 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 \( X \)

\[
X =
\begin{pmatrix}
1 & 5 & 9 \\
7 & 15 & 22
\end{pmatrix}
\]

\[
s = \text{std}(X, 0, 1)
\]

\[
s =
\begin{pmatrix}
4.2426 & 7.0711 & 9.1924
\end{pmatrix}
\]

\[
s = \text{std}(X, 0, 2)
\]

\[
s =
\begin{pmatrix}
4.000 \\
7.5056
\end{pmatrix}
\]

**See Also**

`corrcoef`, `cov`, `mean`, `median`
Purpose
String to number conversion

Syntax
\[ x = \text{str2num}('\text{str}') \]

Description
\[ x = \text{str2num}('\text{str}') \] converts the string \( \text{str} \), which is an ASCII character representation of a numeric value, to MATLAB’s numeric representation. The string can contain:

- Digits
- A decimal point
- A leading + or – sign
- A letter \( e \) preceding a power of 10 scale factor
- A letter \( i \) indicating a complex or imaginary number.

The \text{str2num} function can also convert string matrices.

Examples
\[ \text{str2num('3.14159e0')} \] is approximately \( \pi \).

To convert a string matrix:
\[ \text{str2num(['1 2';'3 4'])} \]

\[
\begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array}
\]

See Also
- \[] (special characters) – Build arrays
- ; (special characters) – End array rows; suppress printing; separate statements.
- \text{hex2num} – Hexadecimal to double number conversion
- \text{num2str} – Number to string conversion
- \text{sparse} – Create sparse matrix
- \text{sscanf} – Read string under format control
**Purpose**
String concatenation

**Syntax**
\[ t = \text{strcat}(s_1, s_2, s_3, \ldots) \]

**Description**
\[ t = \text{strcat}(s_1, s_2, s_3, \ldots) \] horizontally concatenates corresponding rows of the character arrays \( s_1, s_2, s_3, \ldots \). The trailing padding is ignored. All the inputs must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, \text{strcat} returns a cell array of strings formed by concatenating corresponding elements of \( s_1, s_2, \ldots \). The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be a character array.

**Examples**
Given two 1-by-2 cell arrays \( a \) and \( b \),
\[
a = \begin{bmatrix} 'abcde' \\ 'fghi' \\ 'jkl' \\ 'mn' \end{bmatrix},
\]
\[
b = \begin{bmatrix} 'abcde' \\ 'fghi' \\ 'jkl' \\ 'mn' \end{bmatrix}.
\]
the command \( t = \text{strcat}(a, b) \) yields:
\[
t = \begin{bmatrix} 'abcdejkl' \\ 'fghimn' \end{bmatrix}.
\]
Given the 1-by-1 cell array \( c = \{ 'Q' \} \), the command \( t = \text{strcat}(a, b, c) \) yields:
\[
t = \begin{bmatrix} 'abcdejklQ' \\ 'fghimnQ' \end{bmatrix}.
\]

**Remarks**
\text{strcat} and matrix operation are different for strings that contain trailing spaces:
\[
a = 'hello',
b = 'goodby',
\]
\[
\text{strcat}(a, b)
\]
\[
\text{ans} =
\]
\[
\text{hel l o goodby}
\]
\[
[ a \ b]
\]
\[
\text{ans} =
\]
\[
\text{hel l o goodby}
\]
### See Also

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat</code></td>
<td>Concatenate arrays</td>
</tr>
<tr>
<td><code>cellstr</code></td>
<td>Create cell array of strings from character array</td>
</tr>
<tr>
<td><code>strvcat</code></td>
<td>Vertical concatenation of strings</td>
</tr>
</tbody>
</table>
**strcmp**

**Purpose**
Compare strings

**Syntax**

```
k = strcmp('str1','str2')
TF = strcmp(S,T)
```

**Description**

`k = strcmp('str1','str2')` compares the strings `str1` and `str2` and returns logical true (1) if the two are identical, and logical false (0) otherwise.

`TF = strcmp(S,T)` where either `S` or `T` is a cell array of strings, returns an array `TF` the same size as `S` and `T` containing 1 for those elements of `S` and `T` that match, and 0 otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

**Remarks**

Note that the value returned by `strcmp` is not the same as the C language convention. In addition, the `strcmp` function is case sensitive; any leading and trailing blanks in either of the strings are explicitly included in the comparison.
Examples

```matlab
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

- `findstr`: Find one string within another
- `strncmp`: Compare the first n characters of two strings
- `strmatch`: Find possible matches for a string
strings

**Purpose**  
MATLAB string handling

**Syntax**  
\[
S = 'Any Characters' \\
S = string(X) \\
X = numeric(S)
\]

**Description**  
\( S = 'Any Characters' \) is a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The length of \( S \) is the number of characters. A quote within the string is indicated by two quotes.

\( S = string(X) \) can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

\( X = double(S) \) converts the string to its equivalent numeric codes.

\( isstr(S) \) tells if \( S \) is a string variable.

Use the \( strcat \) 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 \( strcat \).

**Example**  
\[
s = ['It is 1 o'clock', 7]
\]

**See Also**  
char Create character array (string)  
strcat String concatenation
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Justify a character array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>strjust(S)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>strjust(S)</code> returns a right-justified version of the character array <code>S</code>.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td><code>deblank</code> Strip trailing blanks from the end of a string</td>
</tr>
</tbody>
</table>
**Purpose**

Find possible matches for a string

**Syntax**

```matlab
i = strmatch('str', STRS)
i = strmatch('str', STRS, 'exact')
```

**Description**

`i = strmatch('str', STRS)` looks through the rows of the character array or cell array of strings `STRS` to find strings that begin with string `str`, returning the matching row indices. `strmatch` is fastest when `STRS` is a character array.

`i = strmatch('str', STRS, 'exact')` returns only the indices of the strings in `STRS` matching `str` exactly.

**Examples**

The statement

```matlab
i = strmatch('max', strvcat('max', 'minimax', 'maximum'))
```
returns `i = [1; 3]` since rows 1 and 3 begin with 'max'. The statement

```matlab
i = strmatch('max', strvcat('max', 'minimax', 'maximum'), 'exact')
```
returns `i = 1`, since only row 1 matches 'max' exactly.

**See Also**

- `findstr` Find one string within another
- `strcmp` Compare strings
- `strncmp` Compare the first `n` characters of two strings
- `strvcat` Vertical concatenation of strings
Purpose

Compare the first \( n \) characters of two strings

Syntax

\[
k = \text{strncmp}(\text{'str1'},\text{'str2'},n)\\
TF = \text{strncmp}(S,T,n)
\]

Description

\( k = \text{strncmp}(\text{'str1'},\text{'str2'},n) \) returns logical true (1) if the first \( n \) characters of the strings \( \text{str1} \) and \( \text{str2} \) are the same, and returns logical false (0) otherwise. Arguments \( \text{str1} \) and \( \text{str2} \) may also be cell arrays of strings.

\( TF = \text{strncmp}(S,T,n) \) where either \( S \) or \( T \) is a cell array of strings, returns an array \( TF \) the same size as \( S \) and \( T \) containing 1 for those elements of \( S \) and \( T \) that match (up to \( n \) characters), and 0 otherwise. \( S \) and \( T \) must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

Remarks

The command \text{strncmp} is case sensitive. Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

See Also

\begin{align*}
\text{findstr} & \quad \text{Find one string within another} \\
\text{strcmp} & \quad \text{Compare strings} \\
\text{strmatch} & \quad \text{Find possible matches for a string}
\end{align*}
Purpose
String search and replace

Syntax
\[
str = \text{strrep}(str1, str2, str3)
\]

Description
\[
str = \text{strrep}(str1, str2, str3)
\]
replaces all occurrences of the string \( str2 \) within string \( str1 \) with the string \( str3 \).

\text{strrep}(str1, str2, str3), when any of \( str1, str2, \) or \( str3 \) is a cell array of strings, returns a cell array the same size as \( str1, str2 \) and \( str3 \) obtained by performing a \text{strrep} using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples
\[
s1 = 'This is a good example.';
str = \text{strrep}(s1, 'good', 'great')
\]
\[
str =
This is a great example.
\]

\[
A =
\begin{array}{c}
'MATLAB' \\
'Tool boxes'
\end{array}
\begin{array}{c}
'SIMULINK' \\
'The MathWorks'
\end{array}
\]

\[
B =
\begin{array}{c}
'Handle Graphics' \\
'Tool boxes'
\end{array}
\begin{array}{c}
'Real Time Workshop' \\
'The MathWorks'
\end{array}
\]

\[
C =
\begin{array}{c}
'Signal Processing' \\
'MATLAB'
\end{array}
\begin{array}{c}
'Image Processing' \\
'SIMULINK'
\end{array}
\]

\[
\text{strrep}(A, B, C)
\]
\[
\text{ans} =
\begin{array}{c}
'MATLAB' \\
'MATLAB'
\end{array}
\begin{array}{c}
'SIMULINK' \\
'SIMULINK'
\end{array}
\]

See Also
findstr
Find one string within another
**Purpose**

First token in string

**Syntax**

Token = `strtok('str', delimiter)`

Token = `strtok('str')`

[token, rem] = `strtok(...)`

**Description**

`token = strtok('str', delimiter)` returns the first token in the text string `str`, that is, the first set of characters before a delimiter is encountered. The vector `delimiter` contains valid delimiter characters.

`token = strtok('str')` uses the default delimiters, the white space characters. These include tabs (ASCII 9), carriage returns (ASCII 13), and spaces (ASCII 32).

[token, rem] = `strtok(...)` returns the remainder `rem` of the original string. The remainder consists of all characters from the first delimiter on.

**Examples**

```matlab
s = 'This is a good example.';
[token, rem] = strtok(s)
```

**See Also**

`findstr` Find one string within another

`strmatch` Find possible matches for a string
Purpose
Create structure array

Syntax
s = struct('field1',values1,'field2',values2,...)

Description
s = struct('field1',values1,'field2',values2,...) creates a structure array with the specified fields and values. The value arrays values1, values2, etc. must be cell arrays of the same size or scalar cells. Corresponding elements of the value arrays are placed into corresponding structure array elements. The size of the resulting structure is the same size as the value cell arrays or 1-by-1 if none of the values is a cell.

Examples
The command

s = struct('type',{'big','little'},'color',{'red'},'x',{3 4})

produces a structure array s:

s =
1x2 struct array with fields:
   type
   color
   x

The value arrays have been distributed among the fields of s:

s(1)
ans =
   type: 'big'
   color: 'red'
   x: 3

s(2)
ans =
   type: 'little'
   color: 'red'
   x: 4

See Also
fieldnames Field names of a structure
getfield Get field of structure array
rmfield Remove structure fields
setfield Set field of structure array
**Purpose**
Structure to cell array conversion

**Syntax**
\[ c = \text{struct2cell}(s) \]

**Description**
\( c = \text{struct2cell}(s) \) converts the \( m \times n \) structure \( s \) (with \( p \) fields) into a \( p \times m \times n \) cell array \( c \).

If structure \( s \) is multidimensional, cell array \( c \) has size \([p \text{ size}(s)]\).

**Examples**
The commands

\[
\text{clear } s, s.\text{category} = \text{'tree'}; \\
s.\text{height} = 37.4; s.\text{name} = \text{'birch'};
\]

create the structure

\[
s = \\
\text{category: 'tree'} \\
\text{height: 37.4000} \\
\text{name: 'birch'}
\]

Converting the structure to a cell array,

\[
c = \text{struct2cell}(s)
\]

\[
c = \\
\text{'tree'} \\
[37.4000] \\
\text{'birch'}
\]

**See Also**
\text{cell2struct, fields}
strvcat

Purpose
Vertical concatenation of strings

Syntax
S = strvcat(t1,t2,t3,...)

Description
S = strvcat(t1,t2,t3,...) forms the character array S containing the text strings (or string matrices) t1, t2, t3,... as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

Remarks
If each text parameter, ti, is itself a character array, strvcat appends them vertically to create arbitrarily large string matrices.

Examples
The command strvcat('Hello','Yes') is the same as ['Hello';'Yes '], except that strvcat performs the padding automatically.

t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';

S1 = strvcat(t1,t2,t3) S2 = strvcat(t4,t2,t3)

S1 =
first
string
matrix

S2 =
second
string
matrix

S3 = strvcat(S1,S2)

S3 =
first
string
matrix
second
string
matrix

See Also
cat Concatenate arrays
int2str Integer to string conversion
mat2str Convert a matrix into a string
num2str Number to string conversion
string Convert numeric values to string
**Purpose**
Single index from subscripts

**Syntax**

\[
\text{IND} = \text{sub2ind} (\text{siz}, I, J)
\]

\[
\text{IND} = \text{sub2ind} (\text{siz}, I_1, I_2, \ldots, I_n)
\]

**Description**
The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

\[
\text{IND} = \text{sub2ind} (\text{siz}, I, J)
\]
returns the linear index equivalent to the row and column subscripts in the arrays `I` and `J` for an matrix of size `siz`.

\[
\text{IND} = \text{sub2ind} (\text{siz}, I_1, I_2, \ldots, I_n)
\]
returns the linear index equivalent to the `n` subscripts in the arrays `I_1`, `I_2`, ..., `I_n` for an array of size `siz`.

**Examples**
The mapping from subscripts to linear index equivalents for a 2-by-2-by-2 array is:

\[
\begin{array}{ccc}
1, 1, 1 & 1, 2, 1 & 1, 1, 2 \\
2, 1, 1 & 2, 2, 1 & 2, 1, 2 \\
1, 1, 2 & 1, 2, 2 & 2, 2, 2 \\
2, 1, 2 & 2, 2, 2 & 2, 2, 1
\end{array}
\]

\[
\begin{array}{cc}
1 & 3 \\
2 & 4 \\
5 & 7 \\
6 & 8
\end{array}
\]

**See Also**
`ind2sub` Subscripts from linear index
`find` Find indices and values of nonzero elements
subsasgn

Purpose
Overloaded method for \( A(i) = B, A\{i\} = B, \) and \( A.\text{field} = B \)

Syntax
\[
A = \text{subsasgn}(A, S, B)
\]

Description
\( A = \text{subsasgn}(A, S, B) \) 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:

- **type**: A string containing '\(', '\}', or '.', where '\(' specifies integer subscripts; '\}' specifies cell array subscripts, and '.' specifies subscripted structure fields.
- **subs**: A cell array or string containing the actual subscripts.

Examples
The syntax \( A(1:2,:) = B \) calls \( A = \text{subsasgn}(A, S, B) \) 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 \( A = \text{subsasgn}(A, S, B) \) where \( S.\text{type} = '.' \) and \( S.\text{subs} = 'field' \).

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 a 3-by-1 structure array with the following values:

\[
S(1).\text{type} = '(' \quad S(2).\text{type} = '.' \quad S(3).\text{type} = '(' \)
S(1).\text{subs} = \{1, 2\} \quad S(2).\text{sub} = 'name' \quad S(3).\text{subs} = \{3:5\}
\]

See Also
subsref
Overloaded method for \( A(i), A\{i\} \) and \( A.\text{field} \)

See Using MATLAB for more information about overloaded methods and \( \text{subsasgn} \).
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Overloaded method for ( X(A) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>( i = \text{subsindex}(A) )</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>( i = \text{subsindex}(A) ) is called for the syntax ' ( X(A) )' when ( A ) is an object. ( \text{subsindex} ) must return the value of the object as a zero-based integer index (( i ) must contain integer values in the range 0 to ( \text{prod} (\text{size}(X)) - 1 )). ( \text{subsindex} ) is called by the default ( \text{subsref} ) and ( \text{subsasgn} ) functions, and you can call it if you overload these functions.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>( \text{subsasgn} ) Overloaded method for ( A(i) = B, A{i} = B ), and ( A.field = B ) ( \text{subsref} ) Overloaded method for ( A(i), A{i} ) and ( A.field )</td>
</tr>
</tbody>
</table>
Purpose

Overloaded method for \texttt{A(i)}, \texttt{A{I}} and \texttt{A.field}

Syntax

\begin{verbatim}
B = subsref(A,S)
\end{verbatim}

Description

\texttt{B = subsref(A,S)} is called for the syntax \texttt{A(i)}, \texttt{A{i}}, or \texttt{A.i} when \texttt{A} is an object. \texttt{S} is a structure array with the fields:

- \texttt{type}: A string containing \texttt{'( )'}, \texttt{'{}'} or \texttt{'.'}, where \texttt{'( )'} specifies integer subscripts; \texttt{'}{}' specifies cell array subscripts, and \texttt{'.'} specifies subscripted structure fields.
- \texttt{subs}: A cell array or string containing the actual subscripts.

Examples

The syntax \texttt{A(1:2,:)} calls \texttt{subsref(A,S)} where \texttt{S} is a 1-by-1 structure with \texttt{S.type='()' and S.subs = {1:2::}}. A colon used as a subscript is passed as the string \texttt{:}.

The syntax \texttt{A{1:2}} calls \texttt{subsref(A,S)} where \texttt{S.type='{}'}.

The syntax \texttt{A.field} calls \texttt{subsref(A,S)} where \texttt{S.type='.' and S.subs='field'}.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases \texttt{length(S)} is the number of subscripting levels. For instance, \texttt{A(1,2).name(3:5)} calls \texttt{subsref(A,S)} where \texttt{S} is 3-by-1 structure array with the following values:

\begin{verbatim}
S(1).type='()' S(2).type='.' S(3).type='()'
S(1).subs={1,2} S(2).subs='name' S(3).subs={3:5}
\end{verbatim}

See Also

\texttt{subsasgn} Overloaded method for \texttt{A(i)=B}, \texttt{A{I}=B}, and \texttt{A.field=B}

See Using MATLAB for more information about overloaded methods and \texttt{subsref}.

2-658
subspace

Purpose
Angle between two subspaces

Syntax
\[ \theta = \text{subspace}(A, B) \]

Description
\[ \theta = \text{subspace}(A, B) \] finds the angle between two subspaces specified by the columns of \( A \) and \( B \). If \( A \) and \( B \) are column vectors of unit length, this is the same as \( \text{acos}(A' \cdot B) \).

Remarks
If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations \( A \), and a second realization of the experiment described by \( B \), \( \text{subspace}(A, B) \) gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples
Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

\[
H = \text{hadamard}(8); \\
A = H(:, 2:4); \\
B = H(:, 5:8);
\]

Note that matrices \( A \) and \( B \) are different sizes—\( A \) has three columns and \( B \) four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

\[
\theta = \text{subspace}(A, B) \\
\theta = 1.5708
\]

That \( A \) and \( B \) are orthogonal is shown by the fact that \( \theta \) is equal to \( \pi/2 \).

\[
\theta - \pi/2 \\
\text{ans} = 0
\]
**Purpose**

Sum of array elements

**Syntax**

B = sum(A)

B = sum(A, dim)

**Description**

B = sum(A) returns sums along different dimensions of an array.

If A is a vector, sum(A) returns the sum of the elements.

If A is a matrix, sum(A) treats the columns of A as vectors, returning a row vector of the sums of each column.

If A is a multidimensional array, sum(A) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

B = sum(A, dim) sums along the dimension of A specified by scalar dim.

**Remarks**

sum(diag(X)) is the trace of X.

**Examples**

The magic square of order 3 is

```matlab
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2
```

This is called a magic square because the sums of the elements in each column are the same.

```matlab
sum(M) =
    15    15    15
```

as are the sums of the elements in each row, obtained by transposing:

```matlab
sum(M') =
    15    15    15
```

**See Also**

cumsum Cumulative sum
diff Differences and approximate derivatives
prod Product of array elements
trace Sum of diagonal elements

2-660
Purpose
Superior class relationship

Syntax
superiorto('class1', 'class2', ...)

Description
The superiorto function establishes a hierarchy that determines the order in which MATLAB calls object methods.

superiorto('class1', 'class2', ...) invoked within a class constructor method (say myclass.m) indicates that myclass's method should be invoked if a function is called with an object of class myclass and one or more objects of class class1, class2, and so on.

Remarks
Suppose A is of class 'class_a', B is of class 'class_b' and C is of class 'class_c'. Also suppose the constructor class_c.m contains the statement: superiorto('class_a'). Then e = fun(a, c) or e = fun(c, a) invokes class_c/fun.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So, fun(b, c) calls class_b/fun, while fun(c, b) calls class_c/fun.

See Also
inferiorto Inferior class relationship
svd

**Purpose**
Singular value decomposition

**Syntax**
\[
s = \text{svd}(X)
\]
\[
[U, S, V] = \text{svd}(X)
\]
\[
[U, S, V] = \text{svd}(X, 0)
\]

**Description**
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 \( X = US V' \).

\( [U, S, V] = \text{svd}(X, 0) \) produces the “economy size” decomposition. If \( X \) is \( m \times n \) with \( m > n \), then `svd` computes only the first \( n \) columns of \( U \) and \( S \) is \( n \times n \).

**Examples**
For the matrix
\[
X =
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6 \\
7 & 8
\end{bmatrix}
\]
the statement
\[
[U, S, V] = \text{svd}(X)
\]
produces
\[
U =
\begin{bmatrix}
0.1525 & 0.8226 & -0.3945 & -0.3800 \\
0.3499 & 0.4214 & 0.2428 & 0.8007 \\
0.5474 & 0.0201 & 0.6979 & -0.4614 \\
0.7448 & -0.3812 & -0.5462 & 0.0407
\end{bmatrix}
\]
svd

\[ S = \begin{bmatrix} 14.2691 & 0 \\ 0 & 0.6268 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 0.6414 & -0.7672 \\ 0.7672 & 0.6414 \end{bmatrix} \]

The economy size decomposition generated by
\[
[U, S, V] = \text{svd}(X, 0)
\]
produces
\[ U = \begin{bmatrix} 0.1525 & 0.8226 \\ 0.3499 & 0.4214 \\ 0.5474 & 0.0201 \\ 0.7448 & -0.3812 \end{bmatrix}, \quad V = \begin{bmatrix} 0.6414 & -0.7672 \\ 0.7672 & 0.6414 \end{bmatrix} \]

Algorithm
The \text{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:
\[ \text{Solution will not converge.} \]

References
svds

Purpose
A few singular values

Syntax
s = svds(A)
s = svds(A, k)
s = svds(A, k, 0)
[U, S, V] = svds(A, ...)

Description
svds(A) computes the five largest singular values and associated singular vectors of the matrix A.

svds(A, k) computes the k largest singular values and associated singular vectors of the matrix A.

svds(A, k, 0) computes the k smallest singular values and associated singular vectors.

With one output argument, s is a vector of singular values. With three output arguments and if A is m-by-n:

• U is m-by-k with orthonormal columns
• S is k-by-k diagonal
• V is n-by-k with orthonormal columns
• U*S*V is the closest rank k approximation to A

Algorithm
svds(A, k) uses eigs to find the k largest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0].

svds(A, k, 0) uses eigs to find the 2k smallest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0], and then selects the k positive eigenvalues and their eigenvectors.

Example
west0479 is a real 479-by-479 sparse matrix. svd calculates all 479 singular values. svds picks out the largest and smallest singular values.

load west0479
s = svd(full(west0479))
sl = svds(west0479, 4)
ss = svds(west0479, 6, 0)

2-664
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
- normest(west0479) = 3.189385666549991e+05

and estimated:

- normest(west0479) = 3.189385666549991e+05

See Also

- svd           Singular value decomposition
- eig           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:

```matlab
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`
**Purpose**  
Sparse symmetric minimum degree ordering

**Syntax**  
\( p = \text{symmmd}(S) \)

**Description**  
\( p = \text{symmmd}(S) \) returns a symmetric minimum degree ordering of \( S \). For a symmetric positive definite matrix \( S \), this is a permutation \( p \) such that \( S(p, p) \) tends to have a sparser Cholesky factor than \( S \). Sometimes \text{symmmd} works well for symmetric indefinite matrices too.

**Remarks**  
The minimum degree ordering is automatically used by \( \backslash \) 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 \text{spparms}.

**Algorithm**  
The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, \text{symmmd}(A) just creates a nonzero structure \( K \) such that \( K^T \ast K \) has the same nonzero structure as \( A \) and then calls the column minimum degree code for \( K \).

**Examples**  
Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the \text{symrcm} reference page.

```plaintext
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))')
```

2-668
Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

See Also

- **colmmd**        Sparse column minimum degree permutation
- **colperm**        Sparse column permutation based on nonzero count
- **symrcm**         Sparse reverse Cuthill-McKee ordering

References

**Purpose**  
Sparse reverse Cuthill-McKee ordering

**Syntax**

\[ r = \text{symrcm}(S) \]

**Description**

\[ r = \text{symrcm}(S) \] returns the symmetric reverse Cuthill-McKee ordering of \( S \). This is a permutation \( r \) such that \( S(r, r) \) tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric \( S \).

For a real, symmetric sparse matrix, \( S \), the eigenvalues of \( S(r, r) \) are the same as those of \( S \), but \( \text{eig}(S(r, r)) \) probably takes less time to compute than \( \text{eig}(S) \).

**Algorithm**

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

**Examples**

The statement

\[ B = \text{bucky} \]

uses an M-file in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

\[ \text{subplot}(1,2,1), \text{spy}(B), \text{title('B')} \]

The reverse Cuthill-McKee ordering is obtained with

\[ p = \text{symrcm}(B); \]
\[ R = B(p, p); \]
The spy plot shows a much narrower bandwidth:

```
subplot(1,2,2), spy(R), title('B(p,p)')
```

This example is continued in the reference pages for `symmmd`.

The bandwidth can also be computed with

```
[i, j] = find(B);
bw = max(i-j) + 1
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

**See Also**

- `colmmd` — Sparse column minimum degree permutation
- `colperm` — Sparse column permutation based on nonzero count
- `symmmd` — Sparse symmetric minimum degree ordering

**References**


**Purpose**
Tangent and hyperbolic tangent

**Syntax**

Y = tan(X)
Y = tanh(X)

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

Y = tan(X) returns the circular tangent of each element of X.

Y = tanh(X) returns the hyperbolic tangent of each element of X.

**Examples**
Graph the tangent function over the domain \(-\pi/2 < x < \pi/2\), and the hyperbolic tangent function over the domain \(-5 \leq x \leq 5\).

x = (-pi/2) +0.01:0.01:(pi/2)-0.01; plot(x,tan(x))

x = -5:0.01:5; plot(x,tanh(x))

The expression tan(pi/2) does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of \(\pi\).
**Algorithm**

\[
\tan(z) = \frac{\sin(z)}{\cos(z)} \\
\tanh(z) = \frac{\sinh(z)}{\cosh(z)}
\]

**See Also**
atan, atan2
tempdir

**Purpose**
Return the name of the system's temporary directory

**Syntax**
```plaintext
tmp_dir = tempdir
```

**Description**
`tmp_dir = tempdir` returns the name of the system's temporary directory, if one exists. This function does not create a new directory.

**See Also**
- `tempname` Unique name for temporary file
**tempname**

**Purpose**
Unique name for temporary file

**Syntax**
tempname

**Description**
tempname returns a unique string beginning with the characters tp. This string is useful as a name for a temporary file.

**See Also**
tempdir Return the name of the system's temporary directory
**tic, toc**

**Purpose**
Stopwatch timer

**Syntax**

```
tic
    any statements
toc
```

```
t = toc
```

**Description**
tic starts a stopwatch timer.
toc prints the elapsed time since tic was used.
t = toc returns the elapsed time in t.

**Examples**
This example measures how the time required to solve a linear system varies with the order of a matrix.

```
for n = 1:100
    A = rand(n, n);
    b = rand(n, 1);
    tic
    x = A \ b;
    t(n) = toc;
end
plot(t)
```

**See Also**

- **clock** Current time as a date vector
- **cputime** Elapsed CPU time
- **etime** Elapsed time
**Purpose**

Toeplitz matrix

**Syntax**

\[
T = \text{toeplitz}(c, r)
\]

\[
T = \text{toeplitz}(r)
\]

**Description**

A Toeplitz matrix is defined by one row and one column. A symmetric Toeplitz matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

\[
T = \text{toeplitz}(c, r)
\] 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.

**Examples**

A Toeplitz matrix with diagonal disagreement is

\[
c = [1 \ 2 \ 3 \ 4 \ 5];
r = [1.5 \ 2.5 \ 3.5 \ 4.5 \ 5.5];
\text{toeplitz}(c, r)
\]

Column wins diagonal conflict:

\[
\text{ans} =
\begin{bmatrix}
1.000 & 2.500 & 3.500 & 4.500 & 5.500 \\
2.000 & 1.000 & 2.500 & 3.500 & 4.500 \\
3.000 & 2.000 & 1.000 & 2.500 & 3.500 \\
4.000 & 3.000 & 2.000 & 1.000 & 2.500 \\
5.000 & 4.000 & 3.000 & 2.000 & 1.000
\end{bmatrix}
\]

**See Also**

hankel  
Hankel matrix
### Purpose
Sum of diagonal elements

### Syntax
\[ b = \text{trace}(A) \]

### Description
\[ b = \text{trace}(A) \] is the sum of the diagonal elements of the matrix \( A \).

### Algorithm
\( \text{trace} \) is a single-statement M-file.

\[ t = \text{sum(diag(A))}; \]

### See Also
- **det**  
  Matrix determinant

- **eig**  
  Eigenvalues and eigenvectors
Purpose

Trapezoidal numerical integration

Syntax

Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)

Description

Z = trapz(Y) computes an approximation of the integral of Y via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply Z by the spacing increment.

If Y is a vector, trapz(Y) is the integral of Y.
If Y is a matrix, trapz(Y) is a row vector with the integral over each column.
If Y is a multidimensional array, trapz(Y) works across the first nonsingleton dimension.

Z = trapz(X,Y) computes the integral of Y with respect to X using trapezoidal integration.

If X is a column vector and Y an array whose first nonsingleton dimension is length(X), trapz(X,Y) operates across this dimension.

Z = trapz(...,dim) integrates across the dimension of Y specified by scalar dim. The length of X, if given, must be the same as size(Y,dim).

Examples

The exact value of \( \int_{0}^{\pi} \sin(x) \, dx \) is 2.
To approximate this numerically on a uniformly spaced grid, use

\[
X = 0: \pi / 100: \pi ;
Y = \sin(x) ;
\]

Then both

\[
Z = \text{trapz}(X,Y)
\]
and

\[
Z = \pi / 100 \times \text{trapz}(Y)
\]
trapz

produce

\[ Z = 1.9998 \]

A nonuniformly spaced example is generated by

\[ X = \text{sort(rand(1, 101)*pi)}; \]
\[ Y = \sin(X); \]
\[ Z = \text{trapz}(X, Y); \]

The result is not as accurate as the uniformly spaced grid. One random sample produced

\[ Z = 1.9984 \]

See Also

cumsum Cumulative sum
cumtrapz Cumulative trapezoidal numerical integration
Purpose

Lower triangular part of a matrix

Syntax

L = tril(X)
L = tril(X,k)

Description

L = tril(X) returns the lower triangular part of X.

L = tril(X,k) returns the elements on and below the kth diagonal of X. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Examples

tril(ones(4,4),-1) is

0 0 0 0
1 0 0 0
1 1 0 0
1 1 1 0

See Also

diag Diagonal matrices and diagonals of a matrix
triu Upper triangular part of a matrix
Purpose
Upper triangular part of a matrix

Syntax
U = triu(X)
U = triu(X, k)

Description
U = triu(X) returns the upper triangular part of X.

U = triu(X, k) returns the element on and above the kth diagonal of X. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Examples
triu(ones(4, 4), -1) is

1 1 1 1
1 1 1 1
0 1 1 1
0 0 1 1

See Also
diag
Diagonal matrices and diagonals of a matrix
triu
Lower triangular part of a matrix
Purpose
Search for enclosing Delaunay triangle

Syntax
T = tsearch(x, y, TRI, xi, yi)

Description
T = tsearch(x, y, TRI, xi, yi) returns an index into the rows of TRI for each point in xi, yi. The tsearch command returns NaN for all points outside the convex hull. Requires a triangulation TRI of the points x,y obtained from delaunay.

See Also
delaunay Delaunay triangulation
dsearch Search for nearest point
**type**

**Purpose**
List file

**Syntax**
type filename

**Description**
type filename displays the contents of the specified file in the MATLAB command window given a full pathname or a MATLABPATH relative partial pathname. Use pathnames and drive designators in the usual way for your computer's operating system.

If you do not specify a filename extension, the type command adds the .m extension by default. The type command checks the directories specified in MATLAB's search path, which makes it convenient for listing the contents of M-files on the screen.

**Examples**
type foo.bar lists the file foo.bar.
type foo lists the file foo.m

**See Also**
Operating system command
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>Change working directory</td>
</tr>
<tr>
<td>dbtype</td>
<td>List M-file with line numbers</td>
</tr>
<tr>
<td>delete</td>
<td>Delete files and graphics objects</td>
</tr>
<tr>
<td>dir</td>
<td>Directory listing</td>
</tr>
<tr>
<td>path</td>
<td>Control MATLAB's directory search path</td>
</tr>
<tr>
<td>what</td>
<td>Directory listing of M-files, MAT-files, and MEX-files</td>
</tr>
<tr>
<td>who</td>
<td>List directory of variables in memory</td>
</tr>
</tbody>
</table>

See also partialpath.
**Purpose**
Convert to unsigned 8-bit integer

**Syntax**

```matlab
i = uint8(x)
```

**Description**

`i = uint8(x)` converts the vector `x` into an unsigned 8-bit integer. `x` can be any numeric object (such as a `double`). The elements of an `uint8` range from 0 to 255. The result for any elements of `x` outside this range is not defined (and may vary from platform to platform). If `x` is already an unsigned 8-bit integer, `uint8` has no effect.

The `uint8` class is primarily meant to store integer values. Most operations that manipulate arrays without changing their elements are defined (examples are `reshape`, `size`, subscripted assignment and subscripted reference). No math operations are defined for `uint8` since such operations are ambiguous on the boundary of the set (for example they could wrap or truncate there). You can define your own methods for `uint8` (as you can for any object) by placing the appropriately named method in an `@uint8` directory within a directory on your path. The Image Processing Toolbox does just that to define additional methods for the `uint8` (such as the logical operators `<`, `>`, `&`, etc.).

Type `help oopfun` for the names of the methods you can overload.

**See Also**

- `double` Convert to double precision
**union**

**Purpose**
Set union of two vectors

**Syntax**
- `c = union(a, b)`
- `c = union(A, B, 'rows')`
- `[c, ia, ib] = union(...)`

**Description**
c = union(a, b) returns the combined values from a and b but with no repetitions. The resulting vector is sorted in ascending order. In set theoretic terms, c = a ∪ b.

c = union(A, B, 'rows') when A and B are matrices with the same number of columns returns the combined rows from A and B with no repetitions.

[c, ia, ib] = union(...) also returns index vectors ia and ib such that c = a(ia) and c = b(ib) or, for row combinations, c = a(ia,:) and c = b(ib,:).  

**Examples**
```matlab
a = [-1 0 2 4 6];
b = [-1 0 1 3];
[c, ia, ib] = union(a, b);
c =
-1     0     1     2     3     4     6
ia =
3     4     5
ib =
1     2     3     4
```

**See Also**
- `intersect` Set intersection of two vectors
- `setdiff` Return the set difference of two vectors
- `setxor` Set exclusive-or of two vectors
- `unique` Unique elements of a vector

2-686
Purpose
Unique elements of a vector

Syntax
\[
b = \text{unique}(a) \\
\]
\[
b = \text{unique}(A, 'rows') \\
\]
\[
[b, i, j] = \text{unique}(...) \\
\]

Description
\[
b = \text{unique}(a) \text{ returns the same values as in } a \text{ but with no repetitions. The resulting vector is sorted in ascending order.} \\
\]
\[
b = \text{unique}(A, 'rows') \text{ returns the unique rows of } A. \\
\]
\[
[b, i, j] = \text{unique}(...) \text{ also returns index vectors } i \text{ and } j \text{ such that } b = a(i) \text{ and } a = b(j) \text{ (or } b = a(:, i) \text{ and } a = b(:, j)). \\
\]

Examples
\[
a = [1 1 5 6 2 3 3 9 8 6 2 4] \\
a = 1 1 5 6 2 3 3 9 8 6 2 4 \\
[b, i, j] = \text{unique}(a) \\
b = 1 2 3 4 5 6 8 9 \\
i = 2 11 7 12 3 10 9 8 \\
j = 1 1 5 6 2 3 3 8 7 6 2 4 \\
a(i) \text{ ans} = \\
1 2 3 4 5 6 8 9 \\
b(j) \text{ ans} = \\
1 1 5 6 2 3 3 9 8 6 2 4 \\
\]

See Also
intersect
Set intersection of two vectors
ismember
True for a set member
setdiff
Return the set difference of two vectors
setxor
Set exclusive-or of two vectors
union
Set union of two vectors
unwrap

Purpose
Correct phase angles

Syntax
Q = unwrap(P)
Q = unwrap(P, tol)
Q = unwrap(P, [], dim)
Q = unwrap(P, tol, dim)

Description
Q = unwrap(P) corrects the radian phase angles in array P by adding multiples of ±2π when absolute jumps between consecutive array elements are greater than π radians. If P is a matrix, unwrap operates columnwise. If P is a multidimensional array, unwrap operates on the first nonsingleton dimension.

Q = unwrap(P, tol) uses a jump tolerance tol instead of the default value, π.

Q = unwrap(P, [], dim) unwraps along dim using the default tolerance.

Q = unwrap(P, tol, dim) uses a jump tolerance of tol.

Examples
Array P features smoothly increasing phase angles except for discontinuities at elements (3,1) and (1,2).

P =
0  7.0686  1.5708  2.3562
0.1963  0.9817  1.7671  2.5525
6.6759  1.1781  1.9635  2.7489
0.5890  1.3744  2.1598  2.9452

The function Q = unwrap(P) eliminates these discontinuities.

Q =
0  0.7854  1.5708  2.3562
0.1963  0.9817  1.7671  2.5525
0.3927  1.1781  1.9635  2.7489
0.5890  1.3744  2.1598  2.9452

Limitations
The unwrap function detects branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

See Also
abs
angle

2-688
**Purpose**
Convert string to upper case

**Syntax**

t = upper('str')

**Description**
t = upper('str') converts any lower-case characters in the string str to the corresponding upper-case characters and leaves all other characters unchanged.

**Examples**
upper('attention!') is ATTENTION!.

**Remarks**
Character sets supported:

- Mac: Standard Roman
- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

**See Also**
lower Convert string to lower case
**varargin, varargout**

**Purpose**
Pass or return variable numbers of arguments

**Syntax**
```matlab
function varargout = foo(n)
y = function bar(varargin)
```

**Description**
- `function varargout = foo(n)` returns a variable number of arguments from function `foo.m`
- `y = function bar(varargin)` accepts a variable number of arguments into function `bar.m`

The `varargin` and `varargout` statements are used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, `varargin` and `varargout` must be lowercase.

**Examples**
The function
```matlab
function myplot(x, varargin)
plot(x, varargin{:})
```
collects all the inputs starting with the second input into the variable `varargin`. `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to `plot`. The call
```matlab
myplot(sin(0:.1:1), 'color', [.5 .7 .3], 'linestyle', ':')
```
results in `varargin` being a 1-by-4 cell array containing the values `'color'`, `[.5 .7 .3]`, `'linestyle'`, and `':'`.

The function
```matlab
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for i = 1:nout, varargout(i) = {s(i)}; end
```
returns the size vector and, optionally, individual sizes. So
```matlab
[s, rows, cols] = mysize(rand(4,5));
```
returns `s = [4 5], rows = 4, cols = 5.`
See Also

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nargin</td>
<td>Number of function arguments</td>
</tr>
<tr>
<td>nargout</td>
<td>Number of function arguments</td>
</tr>
<tr>
<td>nargchk</td>
<td>Check number of input arguments</td>
</tr>
</tbody>
</table>
vectorize

Purpose
Vectorize expression

Syntax
vectorize(string)
vectorize(function)

Description
vectorize(string) inserts a . before any ^, * or / in string. The result is a character string.

vectorize(function) when function is an inline function object, vectorizes the formula for function. The result is the vectorized version of the inline function.

See Also
inline Construct an inline object
Purpose
MATLAB version number

Syntax
v = version
[v, d] = version

Description
v = version returns a string v containing the MATLAB version number.

[v, d] = version also returns a string d containing the date of the version.

See Also
help         Online help for MATLAB functions and M-files
whatsnew     Display README files for MATLAB and toolboxes
version      MATLAB version number
**voronoi**

**Purpose**  
Voronoi diagram

**Syntax**

```
voronoide(x, y)
voronoide(x, y, TRI)
h = voronoide(...,'LineSpec')
[vx, vy] = voronoide(...)
```

**Definition**  
Consider a set of coplanar points P. For each point P_x in the set P, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P. Such a boundary is called a Voronoi polygon, and the set of all Voronoi polygons for a given point set is called a Voronoi diagram.

**Description**

`voronoide(x, y)` plots the Voronoi diagram for the points x, y.  

`voronoide(x, y, TRI)` uses the triangulation TRI instead of computing it via delaunay.  

`h = voronoide(...,'LineSpec')` plots the diagram with color and line style specified and returns handles to the line objects created in h.  

`[vx, vy] = voronoide(...) ` returns the vertices of the Voronoi edges in vx and vy so that `plot(vx, vy, '-', x, y, '.')` creates the Voronoi diagram.
This code plots the Voronoi diagram for 10 randomly generated points.

```matlab
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

- `convhull` - Convex hull
- `delaunay` - Delaunay triangulation
- `dsearch` - Search for nearest point
**warning**

**Purpose**
Display warning message

**Syntax**
- `warning('message')`
- `warning on`
- `warning off`
- `warning backtrace`
- `warning debug`
- `warning once`
- `warning always`
- `[s,f] = warning`

**Description**
- `warning('message')` displays the text `message` as does the `disp` function, except that with `warning`, message display can be suppressed.
- `warning off` suppresses all subsequent warning messages.
- `warning on` re-enables them.
- `warning backtrace` is the same as `warning on` except that the file and line number that produced the warning are displayed.
- `warning debug` is the same as `dbstop if warning` and triggers the debugger when a warning is encountered.
- `warning once` displays Handle Graphics backwards compatibility warnings only once per session.
- `warning always` displays Handle Graphics backwards compatibility warnings as they are encountered (subject to current warning state).

- `[s,f] = warning` returns the current warning state `s` and the current warning frequency `f` as strings.

**Remarks**
Use `dbstop on warning` to trigger the debugger when a warning is encountered.

**See Also**
- `dbstop` Set breakpoints in an M-file function
- `disp` Display text or array
- `error` Display error messages
Purpose
Read Microsoft WAVE (.wav) sound file

Syntax
\[ y = \text{wavread('filename')} \]
\[ [y, Fs, bits] = \text{wavread('filename')} \]
\[ [...] = \text{wavread('filename', N)} \]
\[ [...] = \text{wavread('filename', [N1 N2])} \]
\[ [...] = \text{wavread('filename','size')} \]

Description
\text{wavread} supports multichannel data, with up to 16 bits per sample.

\[ y = \text{wavread('filename')} \] loads a WAVE file specified by the string \text{filename}, returning the sampled data in \( y \). The .wav extension is appended if no extension is given. Amplitude values are in the range \([-1, +1]\).

\[ [y, Fs, bits] = \text{wavread('filename')} \] returns the sample rate (\( Fs \)) in Hertz and the number of bits per sample (\( bits \)) used to encode the data in the file.

\[ [...] = \text{wavread('filename', N)} \] returns only the first \( N \) samples from each channel in the file.

\[ [...] = \text{wavread('filename', [N1 N2])} \] returns only samples \( N1 \) through \( N2 \) from each channel in the file.

\[ size = \text{wavread('filename','size')} \] returns the size of the audio data contained in the file in place of the actual audio data, returning the vector \( size = [\text{samples channel s}] \).

See Also
\text{auread} Read NeXT/SUN (.au) sound file
\text{wavwrite} Write Microsoft WAVE (.wav) sound file
**wavwrite**

**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 $N \leq 16$.

**See Also**

- `auwrite`  
  Write NeXT/SUN (.au) sound file
- `wavread`  
  Read Microsoft WAVE (.wav) sound file
Purpose
Point Web browser at file or Web site

Syntax
web url

Description
web url opens a Web browser and loads the file or Web site specified in the URL (Uniform Resource Locator). The URL can be in any form your browser supports. Generally, the URL specifies a local file or a Web site on the Internet.

Examples
web file:/disk/dir1/dir2/foo.html points the browser to the file foo.html. If the file is on the MATLAB path, web(['file:' which('foo.html')]) also works.
web http://www.mathworks.com loads The MathWorks Web page into your browser. Use web mailto:email_address to send e-mail to another site.

The Web browser used is specified in the docopt M-file.

See Also
doc Load hypertext documentation
docopt Configure local doc access defaults (in online help)
weekday

Purpose
Day of the week

Syntax

\[ [N, S] = \text{weekday}(D) \]

Description

\[ [N, S] = \text{weekday}(D) \]
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:

<table>
<thead>
<tr>
<th>N</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sun</td>
</tr>
<tr>
<td>2</td>
<td>Mon</td>
</tr>
<tr>
<td>3</td>
<td>Tue</td>
</tr>
<tr>
<td>4</td>
<td>Wed</td>
</tr>
<tr>
<td>5</td>
<td>Thu</td>
</tr>
<tr>
<td>6</td>
<td>Fri</td>
</tr>
<tr>
<td>7</td>
<td>Sat</td>
</tr>
</tbody>
</table>

Examples

Either

\[ [n, s] = \text{weekday}(728647) \]
or

\[ [n, s] = \text{weekday}('19- Dec-1994') \]
returns \( n = 2 \) and \( s = \text{Mon} \).

See Also

datenum — Serial date number
datetector — Date components
eomday — End of month
Purpose
Directory listing of M-files, MAT-files, and MEX-files

Syntax
what
what dirname

Description
what by itself, lists the M-files, MAT-files, and MEX-files in the current directory.
what dirname lists the files in directory dirname on MATLAB's search path. It is not necessary to enter the full pathname of the directory. The last component, or last couple of components, is sufficient. Use what class or what dirname/private to list the files in a method directory or a private directory (for the class named class).

Examples
The statements
   what general
and
   what matlab/general
both list the M-files in the general directory. The syntax of the path depends on your operating system:

UNIX:       matlab/general
VMS:        MATLAB.GENERAL
MS-DOS:      MATLAB\GENERAL
Macintosh:   MATLAB:General

See Also
dir Directory listing
lookfor Keyword search through all help entries
pat h Control MATLAB's directory search path
whi ch Locate functions and files
who List directory of variables in memory
whatsnew

Purpose
Display README files for MATLAB and toolboxes

Syntax
whatsnew
whatsnew matlab
whatsnew toolboxpath

Description
whatsnew by itself, displays the README file for the MATLAB product or a specified toolbox. If present, the README file summarizes new functionality that is not described in the documentation.

whatsnew matlab displays the README file for MATLAB.

whatsnew toolboxpath displays the README file for the toolbox specified by the string toolboxpath.

Examples
whatsnew matlab % MATLAB README file
whatsnew signal % Signal Processing Toolbox README file

See Also
help Online help for MATLAB functions and M-files
lookfor Keyword search through all help entries
path Control MATLAB's directory search path
version MATLAB version number
which Locate functions and files
which

**Purpose**
Locate functions and files

**Syntax**

```matlab
which fun
which fun -all
which file.ext
which fun1 in fun2
which fun(a,b,c,...)
s = which(...)
```

**Description**
which fun displays the full pathname of the specified function. The function can be an M-file, MEX-file, workspace variable, built-in function, or SIMULINK model. The latter three display a message indicating that they are variable, built in to MATLAB, or are part of SIMULINK. Use which private/fun or which class/fun or which class/private/fun to further qualify the function name for private functions, methods, and private methods (for the class named class).

which fun -all displays the paths to all functions with the name fun. The first one in the list is the one normally returned by which. The others are either shadowed or can be executed in special circumstances. The -all flag can be used with all forms of which.

which file.ext displays the full pathname of the specified file.

which fun1 in fun2 displays the pathname to function fun1 in the context of the M-file fun2. While debugging fun2, which fun1 does the same thing. You can use this to determine if a local or private version of a function is being called instead of a function on the path.

which fun(a,b,c,...) displays the path to the specified function with the given input arguments. For example, which feval(g), when g=inline('sin(x)'), indicates that inline/feval.m is invoked.

s = which(...) returns the results of which in the string s instead of printing it to the screen. s will be the string built-in/variable for built-in functions or variables in the workspace. You must use the functional form of which when there is an output argument.
which

**Examples**

For example,

```matlab
which inv
```

reveals that `inv` is a built-in function, and

```matlab
which pinv
```

indicates that `pinv` is in the `matfun` directory of the MATLAB Toolbox.

The statement

```matlab
which jacobian
```

probably says

```matlab
jacobian not found
```

because there is no file `jacobian.m` on MATLAB's search path. Contrast this with `lookfor jacobian`, which takes longer to run, but finds several matches to the keyword `jacobian` in its search through all the help entries. (If `jacobian.m` does exist in the current directory, or in some private directory that has been added to MATLAB's search path, `which jacobian` finds it.)

**See Also**

`dir`, `exist`, `help`, `lookfor`, `path`, `what`, `who`
while

**Purpose**
Repeat statements an indefinite number of times

**Syntax**
```
while expression
    statements
end
```

**Description**
while repeats statements an indefinite number of times. The statements are executed while the real part of expression has all nonzero elements. expression is usually of the form
```
expression rop expression
```
where rop is ==, <, <=, >=, or ~=.
The scope of a while statement is always terminated with a matching end.

**Examples**
The variable eps is a tolerance used to determine such things as near singularity and rank. Its initial value is the machine epsilon, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates while loops:
```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

**See Also**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Test to determine if all elements are nonzero</td>
</tr>
<tr>
<td>any</td>
<td>Test for any nonzero</td>
</tr>
<tr>
<td>break</td>
<td>Break out of flow control structures</td>
</tr>
<tr>
<td>end</td>
<td>Terminate for, while, switch, 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>
</tbody>
</table>
**Purpose**
List directory of variables in memory

**Syntax**

```matlab
who
whos
who global
whos global
who -file filename
whos -file filename
who ... var1 var2
whos ... var1 var2
s = who(...)  
s = whos(...)```

**Description**

- `who` by itself, lists the variables currently in memory.
- `whos` by itself, lists the current variables, their sizes, and whether they have nonzero imaginary parts.
- `who global` and `whos global` list the variables in the global workspace.
- `who -file filename` and `whos -file filename` list the variables in the specified MAT-file.
- `who ... var1 var2` and `whos ... var1 var2` restrict the display to the variables specified. The wildcard character `*` can be used to display variables that match a pattern. For instance, `who A*` finds all variables in the current workspace that start with `A`. Use the functional form, such as `whos(' -file ', filename, v1, v2)`, when the filename or variable names are stored in strings.
- `s = who(...)` returns a cell array containing the names of the variables in the workspace or file. Use the functional form of `who` when there is an output argument.
who, whos

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

**Purpose**  
Wilkinson's eigenvalue test matrix

**Syntax**  
`W = wilkinson(n)`

**Description**  
`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

**Examples**  
`wilkinson(7)` is

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

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

**See Also**  
eig  
Eigenvalues and eigenvectors  
gallery  
Test matrices  
pascal  
Pascal matrix
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
  \[
  \begin{bmatrix}
  \text{upper\_left\_row} & \text{upper\_left\_col} \\
  \text{lower\_right\_row} & \text{lower\_right\_col}
  \end{bmatrix}
  \]

- A cell range specified as a string; for example, 'A1...C5'.
- A named range specified as a string; for example, 'Sales'.

See Also
wk1write Write a matrix to a Lotus123 WK1 spreadsheet file
**wk1write**

**Purpose**
Write a matrix to a Lotus123 WK1 spreadsheet file

**Syntax**

```matlab
wk1write(filename, M)
wk1write(filename, M, r, c)
```

**Description**

`wk1write(filename, M)` writes the matrix `M` into a Lotus123 WK1 spreadsheet file named `filename`.

`wk1write(filename, M, r, c)` writes the matrix starting at the spreadsheet location `(r, c)`. `r` and `c` are zero based so that `r=0, c=0` specifies the first cell in the spreadsheet.

![Spreadsheet Diagram]

**See Also**

`wk1read` Read a Lotus123 WK1 spreadsheet file into a matrix
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Write <code>snd</code> resources and files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>writesnd(data, samplerate, bitspersample, filename)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>writesnd(data, samplerate, bitspersample, filename)</code> writes the sound information specified by <code>data</code> and <code>samplerate</code> into an <code>snd</code> resource in <code>filename</code>.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code>writesnd(y, Fs, 16, 'gong.snd')</code></td>
</tr>
</tbody>
</table>
**xlgetrange**

**Purpose**
Get range of cells from Microsoft Excel worksheet

**Syntax**
`xlgetrange([rmin, cmin, rmax, cmax], workbookname, worksheetnum)`

**Description**
`xlgetrange([rmin, cmin, rmax, cmax], workbookname, worksheetnum)` returns the data in the range `r<rmin>c<cmin>:r<rmax>c<cmax>` of sheet `worksheetnum` of the Microsoft Excel workbook `workbookname`. `worksheetnum` defaults to 1 if not specified. Only numerical data is supported.

**See Also**
- `applescript` Load a compiled AppleScript from a file and execute it
- `xlsetrange` Set range of cells in Microsoft Excel worksheet
**Purpose**
Set range of cells in Microsoft Excel worksheet

**Syntax**
xlsetrange(data, [rmin, cmin, rmax, cmax], workbookname, worksheetnum)

**Description**
xlsetrange(data, [rmin, cmin, rmax, cmax], workbookname, worksheetnum)
sets the cells in the range r<rmin>c<cmin>:r<rmax>c<cmax> of sheet
worksheetnum of the Microsoft Excel workbook workbookname to data.
worksheetnum defaults to 1 if not specified. Only numerical data is supported.

**See Also**
applescript  Load a compiled AppleScript from a file and execute it
xlgetrange Get range of cells from Microsoft Excel worksheet
**xor**

**Purpose**
Exclusive or

**Syntax**
\[ C = \text{xor}(A, B) \]

**Description**
\[ C = \text{xor}(A, B) \] performs an exclusive OR operation on the corresponding elements of arrays \( A \) and \( B \). The resulting element \( C(i,j,\ldots) \) is logical true (1) if \( A(i,j,\ldots) \) or \( B(i,j,\ldots) \), but not both, is nonzero.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>

**Examples**
Given \( A = [0 \ 0 \ \pi \ \text{eps}] \) and \( B = [0 \ -2.4 \ 0 \ 1] \), then
\[ C = \text{xor}(A, B) \]
\[ C = \\
0 \ 1 \ 1 \ 0 \]

To see where either \( A \) or \( B \) has a nonzero element and the other matrix does not,
\[ \text{spy}(\text{xor}(A, B)) \]

**See Also**
& \hspace{2cm} \text{Logical AND operator}
| \hspace{2cm} \text{Logical OR operator}
all \hspace{2cm} \text{Test to determine if all elements are nonzero}
any \hspace{2cm} \text{Test for any nonzeros}
find \hspace{2cm} \text{Find indices and values of nonzero elements}
Purpose
Create an array of all zeros

Syntax
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(d1, d2, d3)
B = zeros([d1 d2 d3])
B = zeros(size(A))

Description
B = zeros(n) returns an n-by-n matrix of zeros. An error message appears if n is not a scalar.

B = zeros(m,n) or B = zeros([m n]) returns an m-by-n matrix of zeros.

B = zeros(d1, d2, d3) or B = zeros([d1 d2 d3]) returns an array of zeros with dimensions d1-by-d2-by-d3-by-...

B = zeros(size(A)) returns an array the same size as A consisting of all zeros.

Remarks
The MATLAB language does not have a dimension statement—MATLAB automatically allocates storage for matrices. Nevertheless, most MATLAB programs execute faster if the zeros function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time.

Examples
With n = 1000, the for loop

```matlab
for i = 1:n, x(i) = i; end
```
takes about 1.2 seconds to execute on a Sun SPARC-1. If the loop is preceded by the statement

```matlab
x = zeros(1, n);
```
the computations require less than 0.2 seconds.

See Also
eye Identity matrix
ones Create an array of all ones
rand Uniformly distributed random numbers and arrays
randn Normally distributed random numbers and arrays
zeros
List of Commands

This appendix lists MATLAB commands and functions alphabetically. For a list of commands grouped by functional category, see the Command Summary.
### Function Names

**Arithmetic Operators** + * / \ ^

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2-2</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
</tbody>
</table>

**Relational Operators** < > <= >= == ~=

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>2-9</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td></td>
</tr>
<tr>
<td>~=</td>
<td></td>
</tr>
</tbody>
</table>

**Logical Operators** & | ~

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>2-11</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

**Special Characters** [ ] ( ) {} = ' . ...

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>2-13</td>
</tr>
<tr>
<td>]</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td>'</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td></td>
</tr>
</tbody>
</table>

**Colon** :

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>2-16</td>
</tr>
</tbody>
</table>

**Miscellaneous**

- abs          2-18
- acopy        2-19
- acos, acosh  2-20
- acot, acoth  2-21
- acsc, acsch  2-22
- addpath      2-24
- airy         2-25
- all          2-27
- amove        2-29
- angle        2-30
- ans          2-31
- any          2-32
- applescript  2-34
- arename      2-35
- areveal      2-36
- asec, asech  2-37
- asin, asinh  2-38
- assignin     2-39
- atan, atanh  2-40
- atan2       2-42
- auread       2-43
- auwrite      2-44
- balance      2-45
- base2dec     2-48
- besselh      2-49
- besseli, besselk 2-51
- besselj, bessely 2-53
- beta, betaic, betain 2-56
- bicc         2-58
- big斯塔b     2-65
- bin2dec      2-69
- bitand       2-70
- bitcmp       2-71
- bitget       2-72
- bitmax       2-73
- bitor        2-74
- bitset       2-75
- bitshift     2-76
- bitxor       2-77
- blanks       2-78
- break        2-79
- builtin      2-80
- calendar     2-82
- cart2pol     2-83
- cart2spht    2-85
- case         2-86
- cat          2-87
- cd           2-88
- cdf2rdf      2-89
- ceil         2-91
- cell         2-92
- cell2struct  2-93
- celldisp     2-94
- cellplot     2-95
- cellstr      2-96
- cgs          2-97
- char         2-101
- chol         2-103
- cholnc       2-105
- class        2-110
- clear        2-111
- clock        2-113
- colmd        2-114
- colperm      2-117
- comple       2-118
- compan       2-119
- computer     2-121
- cond         2-122
- condeig      2-123
- condest      2-124
- conj         2-125
- conv         2-126
- conv2        2-128
- convhull     2-129
- convn        2-130
- corrcoeff    2-131
- cos, cosh    2-132
- cot, coth    2-133
- cov          2-135
- cplxpair     2-136
- cputime      2-137
- csc, csch    2-138
- cumprod      2-139
- csumsum      2-140
- cumtrapz     2-141
- date         2-143
- datenum      2-144
- datestr      2-145
- datevec      2-147
- dbclear      2-148
- dbcont       2-150
- dbdown       2-151
- dbflagd      2-152
- dbmode       2-154
- dbquit       2-155
- dbstack      2-156
- dbstatus     2-157
- debstep      2-158
- dbstop       2-159
- dbtype       2-162
- dbup         2-163
- deadv        2-164
- deexec       2-166
- deinit       2-167
- depeoe       2-168
- ddereq       2-170
- determ       2-172
- deinadv      2-173
- deal         2-174
- deblank      2-177
- dec2base     2-178
- dec2bin      2-179
- dec2hex      2-180
- dec2time     2-181
- del2         2-182
- delaunay     2-185
- delete       2-188
- det          2-189
- diag         2-190
- diary        2-191
- diff         2-192
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>2-194</td>
</tr>
<tr>
<td>disp</td>
<td>2-195</td>
</tr>
<tr>
<td>dlmread</td>
<td>2-196</td>
</tr>
<tr>
<td>dlmwrite</td>
<td>2-197</td>
</tr>
<tr>
<td>dmperm</td>
<td>2-198</td>
</tr>
<tr>
<td>doc</td>
<td>2-199</td>
</tr>
<tr>
<td>double</td>
<td>2-200</td>
</tr>
<tr>
<td>dsearch</td>
<td>2-201</td>
</tr>
<tr>
<td>echo</td>
<td>2-202</td>
</tr>
<tr>
<td>edit</td>
<td>2-203</td>
</tr>
<tr>
<td>eig</td>
<td>2-204</td>
</tr>
<tr>
<td>eigs</td>
<td>2-207</td>
</tr>
<tr>
<td>ellipj</td>
<td>2-213</td>
</tr>
<tr>
<td>ellipke</td>
<td>2-215</td>
</tr>
<tr>
<td>else</td>
<td>2-217</td>
</tr>
<tr>
<td>elseif</td>
<td>2-218</td>
</tr>
<tr>
<td>end</td>
<td>2-220</td>
</tr>
<tr>
<td>eomday</td>
<td>2-221</td>
</tr>
<tr>
<td>eps</td>
<td>2-222</td>
</tr>
<tr>
<td>erf, erfc, erfinv</td>
<td>2-223</td>
</tr>
<tr>
<td>error</td>
<td>2-225</td>
</tr>
<tr>
<td>errortrap</td>
<td>2-226</td>
</tr>
<tr>
<td>etime</td>
<td>2-227</td>
</tr>
<tr>
<td>eval</td>
<td>2-228</td>
</tr>
<tr>
<td>evalin</td>
<td>2-230</td>
</tr>
<tr>
<td>exist</td>
<td>2-231</td>
</tr>
<tr>
<td>exp</td>
<td>2-233</td>
</tr>
<tr>
<td>expint</td>
<td>2-234</td>
</tr>
<tr>
<td>expm</td>
<td>2-236</td>
</tr>
<tr>
<td>eye</td>
<td>2-238</td>
</tr>
<tr>
<td>factor</td>
<td>2-240</td>
</tr>
<tr>
<td>fclose</td>
<td>2-241</td>
</tr>
<tr>
<td>feof</td>
<td>2-242</td>
</tr>
<tr>
<td>ferror</td>
<td>2-243</td>
</tr>
<tr>
<td>feval</td>
<td>2-244</td>
</tr>
<tr>
<td>fft</td>
<td>2-245</td>
</tr>
<tr>
<td>fft2</td>
<td>2-248</td>
</tr>
<tr>
<td>fttn</td>
<td>2-249</td>
</tr>
<tr>
<td>fftshift</td>
<td>2-250</td>
</tr>
<tr>
<td>fgeti</td>
<td>2-251</td>
</tr>
<tr>
<td>fgets</td>
<td>2-252</td>
</tr>
<tr>
<td>fieldnames</td>
<td>2-253</td>
</tr>
<tr>
<td>fileparts</td>
<td>2-254</td>
</tr>
<tr>
<td>filter</td>
<td>2-255</td>
</tr>
<tr>
<td>filter2</td>
<td>2-257</td>
</tr>
<tr>
<td>find</td>
<td>2-258</td>
</tr>
<tr>
<td>findstr</td>
<td>2-260</td>
</tr>
<tr>
<td>fix</td>
<td>2-261</td>
</tr>
<tr>
<td>flipdim</td>
<td>2-262</td>
</tr>
<tr>
<td>flipr</td>
<td>2-263</td>
</tr>
<tr>
<td>floor</td>
<td>2-265</td>
</tr>
<tr>
<td>flops</td>
<td>2-266</td>
</tr>
<tr>
<td>fopen</td>
<td>2-267</td>
</tr>
<tr>
<td>fopen</td>
<td>2-268</td>
</tr>
<tr>
<td>fopen</td>
<td>2-269</td>
</tr>
<tr>
<td>fmins</td>
<td>2-270</td>
</tr>
<tr>
<td>for</td>
<td>2-271</td>
</tr>
<tr>
<td>fprintf</td>
<td>2-272</td>
</tr>
<tr>
<td>fread</td>
<td>2-273</td>
</tr>
<tr>
<td>format</td>
<td>2-274</td>
</tr>
<tr>
<td>fprintf</td>
<td>2-275</td>
</tr>
<tr>
<td>fscanf</td>
<td>2-276</td>
</tr>
<tr>
<td>fseek</td>
<td>2-277</td>
</tr>
<tr>
<td>ftell</td>
<td>2-278</td>
</tr>
<tr>
<td>full</td>
<td>2-279</td>
</tr>
<tr>
<td>fullfile</td>
<td>2-280</td>
</tr>
<tr>
<td>funm</td>
<td>2-281</td>
</tr>
<tr>
<td>fwrite</td>
<td>2-282</td>
</tr>
<tr>
<td>fzero</td>
<td>2-283</td>
</tr>
<tr>
<td>gamma</td>
<td>2-284</td>
</tr>
<tr>
<td>gamma, gammaic, gammai</td>
<td>2-285</td>
</tr>
<tr>
<td>gigamain</td>
<td>2-286</td>
</tr>
<tr>
<td>ginc</td>
<td>2-287</td>
</tr>
<tr>
<td>gcd</td>
<td>2-288</td>
</tr>
<tr>
<td>gseek</td>
<td>2-289</td>
</tr>
<tr>
<td>getfield</td>
<td>2-290</td>
</tr>
<tr>
<td>global</td>
<td>2-291</td>
</tr>
<tr>
<td>gmesh</td>
<td>2-292</td>
</tr>
<tr>
<td>gradient</td>
<td>2-293</td>
</tr>
<tr>
<td>griddata</td>
<td>2-294</td>
</tr>
<tr>
<td>griddata</td>
<td>2-295</td>
</tr>
<tr>
<td>hadamard</td>
<td>2-296</td>
</tr>
<tr>
<td>help</td>
<td>2-297</td>
</tr>
<tr>
<td>hankel</td>
<td>2-298</td>
</tr>
<tr>
<td>hess</td>
<td>2-299</td>
</tr>
<tr>
<td>hex2dec</td>
<td>2-300</td>
</tr>
<tr>
<td>hex2num</td>
<td>2-301</td>
</tr>
<tr>
<td>hilb</td>
<td>2-302</td>
</tr>
<tr>
<td>imfinfo</td>
<td>2-303</td>
</tr>
<tr>
<td>imread</td>
<td>2-304</td>
</tr>
<tr>
<td>imwrite</td>
<td>2-305</td>
</tr>
<tr>
<td>ind2sub</td>
<td>2-306</td>
</tr>
<tr>
<td>inline</td>
<td>2-307</td>
</tr>
<tr>
<td>inmem</td>
<td>2-308</td>
</tr>
<tr>
<td>inpolygon</td>
<td>2-309</td>
</tr>
<tr>
<td>input</td>
<td>2-310</td>
</tr>
<tr>
<td>int2str</td>
<td>2-311</td>
</tr>
<tr>
<td>intersect</td>
<td>2-312</td>
</tr>
<tr>
<td>interp</td>
<td>2-313</td>
</tr>
<tr>
<td>interp1</td>
<td>2-314</td>
</tr>
<tr>
<td>interp2</td>
<td>2-315</td>
</tr>
<tr>
<td>interp3</td>
<td>2-316</td>
</tr>
<tr>
<td>interpft</td>
<td>2-317</td>
</tr>
<tr>
<td>interpn</td>
<td>2-318</td>
</tr>
<tr>
<td>inv</td>
<td>2-319</td>
</tr>
<tr>
<td>is*</td>
<td>2-320</td>
</tr>
<tr>
<td>isequal</td>
<td>2-321</td>
</tr>
<tr>
<td>isequal</td>
<td>2-322</td>
</tr>
<tr>
<td>isequal</td>
<td>2-323</td>
</tr>
<tr>
<td>isequal</td>
<td>2-324</td>
</tr>
<tr>
<td>isequal</td>
<td>2-325</td>
</tr>
<tr>
<td>isequal</td>
<td>2-326</td>
</tr>
<tr>
<td>isequal</td>
<td>2-327</td>
</tr>
<tr>
<td>isequal</td>
<td>2-328</td>
</tr>
<tr>
<td>isequal</td>
<td>2-329</td>
</tr>
<tr>
<td>isequal</td>
<td>2-330</td>
</tr>
<tr>
<td>isequal</td>
<td>2-331</td>
</tr>
<tr>
<td>isequal</td>
<td>2-332</td>
</tr>
<tr>
<td>isequal</td>
<td>2-333</td>
</tr>
<tr>
<td>isequal</td>
<td>2-334</td>
</tr>
<tr>
<td>isequal</td>
<td>2-335</td>
</tr>
<tr>
<td>isequal</td>
<td>2-336</td>
</tr>
<tr>
<td>isequal</td>
<td>2-337</td>
</tr>
<tr>
<td>isequal</td>
<td>2-338</td>
</tr>
<tr>
<td>isequal</td>
<td>2-339</td>
</tr>
<tr>
<td>isequal</td>
<td>2-340</td>
</tr>
<tr>
<td>isequal</td>
<td>2-341</td>
</tr>
<tr>
<td>isequal</td>
<td>2-342</td>
</tr>
<tr>
<td>isequal</td>
<td>2-343</td>
</tr>
<tr>
<td>isequal</td>
<td>2-344</td>
</tr>
<tr>
<td>isequal</td>
<td>2-345</td>
</tr>
<tr>
<td>isequal</td>
<td>2-346</td>
</tr>
<tr>
<td>isequal</td>
<td>2-347</td>
</tr>
<tr>
<td>isequal</td>
<td>2-348</td>
</tr>
<tr>
<td>isequal</td>
<td>2-349</td>
</tr>
<tr>
<td>isequal</td>
<td>2-350</td>
</tr>
<tr>
<td>isequal</td>
<td>2-351</td>
</tr>
<tr>
<td>isequal</td>
<td>2-352</td>
</tr>
<tr>
<td>isequal</td>
<td>2-353</td>
</tr>
<tr>
<td>isequal</td>
<td>2-354</td>
</tr>
<tr>
<td>isequal</td>
<td>2-355</td>
</tr>
<tr>
<td>isequal</td>
<td>2-356</td>
</tr>
<tr>
<td>isequal</td>
<td>2-357</td>
</tr>
<tr>
<td>isequal</td>
<td>2-358</td>
</tr>
<tr>
<td>isequal</td>
<td>2-359</td>
</tr>
<tr>
<td>isequal</td>
<td>2-360</td>
</tr>
<tr>
<td>isequal</td>
<td>2-361</td>
</tr>
<tr>
<td>isequal</td>
<td>2-362</td>
</tr>
<tr>
<td>isequal</td>
<td>2-363</td>
</tr>
<tr>
<td>isequal</td>
<td>2-364</td>
</tr>
<tr>
<td>isequal</td>
<td>2-365</td>
</tr>
<tr>
<td>isequal</td>
<td>2-366</td>
</tr>
<tr>
<td>isequal</td>
<td>2-367</td>
</tr>
<tr>
<td>isequal</td>
<td>2-368</td>
</tr>
<tr>
<td>isequal</td>
<td>2-369</td>
</tr>
<tr>
<td>isequal</td>
<td>2-370</td>
</tr>
<tr>
<td>isequal</td>
<td>2-371</td>
</tr>
<tr>
<td>isequal</td>
<td>2-372</td>
</tr>
<tr>
<td>isequal</td>
<td>2-373</td>
</tr>
<tr>
<td>isequal</td>
<td>2-374</td>
</tr>
<tr>
<td>isequal</td>
<td>2-375</td>
</tr>
<tr>
<td>isequal</td>
<td>2-376</td>
</tr>
<tr>
<td>isequal</td>
<td>2-377</td>
</tr>
<tr>
<td>isequal</td>
<td>2-378</td>
</tr>
<tr>
<td>isequal</td>
<td>2-379</td>
</tr>
<tr>
<td>isequal</td>
<td>2-380</td>
</tr>
<tr>
<td>isequal</td>
<td>2-381</td>
</tr>
<tr>
<td>isequal</td>
<td>2-382</td>
</tr>
<tr>
<td>isequal</td>
<td>2-383</td>
</tr>
<tr>
<td>isequal</td>
<td>2-384</td>
</tr>
<tr>
<td>isequal</td>
<td>2-385</td>
</tr>
<tr>
<td>isequal</td>
<td>2-386</td>
</tr>
<tr>
<td>isequal</td>
<td>2-387</td>
</tr>
<tr>
<td>isequal</td>
<td>2-388</td>
</tr>
<tr>
<td>isequal</td>
<td>2-389</td>
</tr>
<tr>
<td>isequal</td>
<td>2-390</td>
</tr>
<tr>
<td>isequal</td>
<td>2-391</td>
</tr>
<tr>
<td>isequal</td>
<td>2-392</td>
</tr>
<tr>
<td>isequal</td>
<td>2-393</td>
</tr>
<tr>
<td>isequal</td>
<td>2-394</td>
</tr>
<tr>
<td>isequal</td>
<td>2-395</td>
</tr>
<tr>
<td>isequal</td>
<td>2-396</td>
</tr>
<tr>
<td>isequal</td>
<td>2-397</td>
</tr>
<tr>
<td>isequal</td>
<td>2-398</td>
</tr>
<tr>
<td>isequal</td>
<td>2-399</td>
</tr>
<tr>
<td>isequal</td>
<td>2-400</td>
</tr>
<tr>
<td>isequal</td>
<td>2-401</td>
</tr>
<tr>
<td>isequal</td>
<td>2-402</td>
</tr>
<tr>
<td>isequal</td>
<td>2-403</td>
</tr>
<tr>
<td>isequal</td>
<td>2-404</td>
</tr>
<tr>
<td>isequal</td>
<td>2-405</td>
</tr>
<tr>
<td>isequal</td>
<td>2-406</td>
</tr>
<tr>
<td>isequal</td>
<td>2-407</td>
</tr>
<tr>
<td>isequal</td>
<td>2-408</td>
</tr>
<tr>
<td>isequal</td>
<td>2-409</td>
</tr>
<tr>
<td>isequal</td>
<td>2-410</td>
</tr>
<tr>
<td>isequal</td>
<td>2-411</td>
</tr>
<tr>
<td>isequal</td>
<td>2-412</td>
</tr>
<tr>
<td>isequal</td>
<td>2-413</td>
</tr>
<tr>
<td>isequal</td>
<td>2-414</td>
</tr>
<tr>
<td>isequal</td>
<td>2-415</td>
</tr>
<tr>
<td>isequal</td>
<td>2-416</td>
</tr>
<tr>
<td>isequal</td>
<td>2-417</td>
</tr>
<tr>
<td>isequal</td>
<td>2-418</td>
</tr>
<tr>
<td>isequal</td>
<td>2-419</td>
</tr>
<tr>
<td>isequal</td>
<td>2-420</td>
</tr>
</tbody>
</table>
A-4
Function Names

A logical …………………… 2-421
logm ……………………… 2-422
logspace ………………… 2-424
lookfor ………………… 2-425
lower …………………… 2-426
lscov ……………………… 2-427
lu ………………………… 2-428
luinc …………………… 2-431
magic …………………… 2-438
mat2str ………………… 2-440
matlabrc ………………… 2-441
matlabroot ……………… 2-442
max ……………………… 2-443
mean …………………… 2-444
median ………………… 2-445
menu …………………… 2-446
meshgrid ……………… 2-447
methods ………………… 2-449
mexext ………………… 2-450
mfilename ……………… 2-451
min ……………………… 2-452
mod ……………………… 2-453
more …………………… 2-454
mu2lin ………………… 2-455
NaN ……………………… 2-456
nargchk ………………… 2-457
nargin, nargsout ……… 2-458
nchoosek ……………… 2-459
ndgrid …………………… 2-461
ndims …………………… 2-462
nextpow2 ……………… 2-463
nnls …………………… 2-464
nnz …………………… 2-465
nonzeros ………………… 2-466
norm ……………………… 2-467
normest ………………… 2-468
normest ………………… 2-469
now ……………………… 2-470
null ……………………… 2-471
num2cell ……………… 2-472
num2str ……………… 2-473
nzmax …………………… 2-474
ode45, ode23, ode113, ode15s, ode23s …………… 2-475
odefile ………………… 2-483
 odeget ………………… 2-488
odeset ………………… 2-489
odeget ………………… 2-490
ones …………………… 2-495
orth …………………… 2-496
pack ……………………… 2-497
partialpath …………… 2-498
pascal ………………… 2-499
path …………………… 2-500
pause …………………… 2-501
perms …………………… 2-506
pi ……………………… 2-506
pinv ……………………… 2-507
poly …………………… 2-508
polyarea ……………… 2-509
polyder ………………… 2-511
polyeig ………………… 2-512
polyfit ………………… 2-513
polyval ………………… 2-514
polyvalm ……………… 2-515
pow2 …………………… 2-516
primes ………………… 2-517
prod …………………… 2-518
profile ………………… 2-519
profile ………………… 2-520
qmr ……………………… 2-521
qr ……………………… 2-522
qrdelete ……………… 2-523
qreinsert ……………… 2-524
qrs ……………………… 2-525
qrupdate ……………… 2-526
quadratic ……………… 2-527
quad …………………… 2-528
quad2d ………………… 2-529
quad3d ………………… 2-530
quad …………………… 2-531
quad …………………… 2-532
quad …………………… 2-533
quad …………………… 2-534
quad …………………… 2-535
quad …………………… 2-536
quad …………………… 2-537
quad …………………… 2-538
quad …………………… 2-539
quad …………………… 2-540
quad …………………… 2-541
quad …………………… 2-542
quad …………………… 2-543
quad …………………… 2-544
quad …………………… 2-545
quad …………………… 2-546
quad …………………… 2-547
quad …………………… 2-548
quad …………………… 2-549
quad …………………… 2-550
quad …………………… 2-551
quad …………………… 2-552
quad …………………… 2-553
quad …………………… 2-554
quad …………………… 2-555
quad …………………… 2-556
quad …………………… 2-557
quad …………………… 2-558
quad …………………… 2-559
quad …………………… 2-560
quad …………………… 2-561
quad …………………… 2-562
quad …………………… 2-563
quad …………………… 2-564
quad …………………… 2-565
quad …………………… 2-566
quad …………………… 2-567
quad …………………… 2-568
quad …………………… 2-569
quad …………………… 2-570
quad …………………… 2-571
quad …………………… 2-572
quad …………………… 2-573
quad …………………… 2-574
quad …………………… 2-575
quad …………………… 2-576
quad …………………… 2-577
quad …………………… 2-578
quad …………………… 2-579
quad …………………… 2-580
quad …………………… 2-581
quad …………………… 2-582
quad …………………… 2-583
quad …………………… 2-584
quad …………………… 2-585
quad …………………… 2-586
quad …………………… 2-587
quad …………………… 2-588
quad …………………… 2-589
quad …………………… 2-590
quad …………………… 2-591
quad …………………… 2-592
quad …………………… 2-593
quad …………………… 2-594
quad …………………… 2-595
quad …………………… 2-596
quad …………………… 2-597
quad …………………… 2-598
quad …………………… 2-599
quad …………………… 2-600
quad …………………… 2-601
quad …………………… 2-602
quad …………………… 2-603
quad …………………… 2-604
quad …………………… 2-605
quad …………………… 2-606
quad …………………… 2-607
quad …………………… 2-608
quad …………………… 2-609
quad …………………… 2-610
quad …………………… 2-611
quad …………………… 2-612
quad …………………… 2-613
quad …………………… 2-614
quad …………………… 2-615
quad …………………… 2-616
quad …………………… 2-617
quad …………………… 2-618
quad …………………… 2-619
quad …………………… 2-620
quad …………………… 2-621
quad …………………… 2-622
A-5
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sprandn</td>
<td>2-623</td>
</tr>
<tr>
<td>sprandsym</td>
<td>2-624</td>
</tr>
<tr>
<td>sprintf</td>
<td>2-625</td>
</tr>
<tr>
<td>spy</td>
<td>2-629</td>
</tr>
<tr>
<td>sqrt</td>
<td>2-630</td>
</tr>
<tr>
<td>squrtm</td>
<td>2-631</td>
</tr>
<tr>
<td>squeeze</td>
<td>2-634</td>
</tr>
<tr>
<td>sscanf</td>
<td>2-635</td>
</tr>
<tr>
<td>startup</td>
<td>2-638</td>
</tr>
<tr>
<td>std</td>
<td>2-639</td>
</tr>
<tr>
<td>str2num</td>
<td>2-641</td>
</tr>
<tr>
<td>strcat</td>
<td>2-642</td>
</tr>
<tr>
<td>strcmp</td>
<td>2-642</td>
</tr>
<tr>
<td>strings</td>
<td>2-646</td>
</tr>
<tr>
<td>strjust</td>
<td>2-647</td>
</tr>
<tr>
<td>strcmp</td>
<td>2-648</td>
</tr>
<tr>
<td>strncmp</td>
<td>2-649</td>
</tr>
<tr>
<td>strtok</td>
<td>2-650</td>
</tr>
<tr>
<td>struct</td>
<td>2-651</td>
</tr>
<tr>
<td>struct2cell</td>
<td>2-652</td>
</tr>
<tr>
<td>strvcat</td>
<td>2-653</td>
</tr>
<tr>
<td>sub2ind</td>
<td>2-654</td>
</tr>
<tr>
<td>subsasgn</td>
<td>2-655</td>
</tr>
<tr>
<td>subsindex</td>
<td>2-656</td>
</tr>
<tr>
<td>subsref</td>
<td>2-657</td>
</tr>
<tr>
<td>subspace</td>
<td>2-658</td>
</tr>
<tr>
<td>sum</td>
<td>2-659</td>
</tr>
<tr>
<td>superiorto</td>
<td>2-660</td>
</tr>
<tr>
<td>svd</td>
<td>2-661</td>
</tr>
<tr>
<td>svds</td>
<td>2-662</td>
</tr>
<tr>
<td>switch</td>
<td>2-663</td>
</tr>
<tr>
<td>symmmd</td>
<td>2-664</td>
</tr>
<tr>
<td>symrcm</td>
<td>2-665</td>
</tr>
<tr>
<td>tan, tanh</td>
<td>2-666</td>
</tr>
<tr>
<td>tempdir</td>
<td>2-667</td>
</tr>
<tr>
<td>tempname</td>
<td>2-668</td>
</tr>
<tr>
<td>tic, toc</td>
<td>2-669</td>
</tr>
<tr>
<td>toeplitz</td>
<td>2-670</td>
</tr>
<tr>
<td>trace</td>
<td>2-671</td>
</tr>
<tr>
<td>trapz</td>
<td>2-672</td>
</tr>
<tr>
<td>triu</td>
<td>2-673</td>
</tr>
<tr>
<td>triu</td>
<td>2-674</td>
</tr>
<tr>
<td>tsearch</td>
<td>2-675</td>
</tr>
</tbody>
</table>
Function Names
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
{} 2-13
| 2-11
~ 2-11
^= 2-9
≤ 2-9
≥ 2-9

Numerics

π (pi) 2-513, 2-556, 2-595
1-norm 2-468, 2-558
2-norm (estimate of) 2-469

A
abs 2-18
absolute value 2-18

accuracy
of linear equation solution 2-121
of matrix inversion 2-121
relative floating-point 2-222
acopy 2-19
acos 2-20
acosh 2-20
acot 2-21
acoth 2-21
acsc 2-22
acsch 2-22
Adams-Bashforth-Moulton ODE solver 2-481
addition (arithmetic operator) 2-2
addressing selected array elements 2-16
adjacency graph 2-198
airy 2-25
aligning scattered data
multi-dimensional 2-461
two-dimensional 2-341
all 2-27
allocation of storage (automatic) 2-715
and (M-file function equivalent for &) 2-11
AND, logical
bit-wise 2-70
angle 2-30
ans 2-31
anti-diagonal 2-345
any 2-32
applescript 2-34
arccosecant 2-34
arccosine 2-20
arccotangent 2-21
arccscant 2-37
arcsine 2-38
Index

arctangent 2-40
  (four-quadrant) 2-42
ar ename 2-35
arguments, M-file
  checking number of input 2-457
  number of input 2-458
  number of output 2-458
  passing variable numbers of 2-690
arithmetic operations, matrix and array distinguished 2-2
arithmetic operators 2-2
array
  addressing selected elements of 2-16
  displaying 2-195
  finding indices of 2-258
  left division (arithmetic operator) 2-3
  maximum elements of 2-443
  maximum size of 2-119
  mean elements of 2-444
  median elements of 2-445
  minimum elements of 2-452
  multiplication (arithmetic operator) 2-2
  of all ones 2-495
  power (arithmetic operator) 2-3
  product of elements 2-532
  of random numbers 2-549, 2-551
  removing first n singleton dimensions of 2-593
  removing singleton dimensions of 2-634
  reshaping 2-566
  right division (arithmetic operator) 2-3
  shifting dimensions of 2-593
  size of 2-597
  sorting elements of 2-599
  structure 2-253, 2-331, 2-570, 2-590
  sum of elements 2-660
  swapping dimensions of 2-397, 2-512
  transpose (arithmetic operator) 2-3
  of all zeros 2-715
arrowhead matrix 2-117
ASCII data
  converting sparse matrix after loading from
    2-607
  printable characters (list of) 2-101
  reading from disk 2-416
  saving to disk 2-581
ASCII delimited file
  reading 2-196
  writing matrix to 2-197
asech 2-37
asin 2-38
asinh 2-38
atan 2-40
atan2 2-42
atanh 2-40
auread 2-43
auwrite 2-44
average of array elements 2-444
axis crossing See zero of a function
azimuth (spherical coordinates) 2-615

B
badly conditioned 2-558
balance 2-45
bank format 2-278
base to decimal conversion 2-48
base two operations
  conversion from decimal to binary 2-179
  logarithm 2-419
  next power of two 2-463
base2dec 2-48
Bessel functions 2-49, 2-53
  first kind 2-51
  modified 2-51
second kind 2-52  
third kind 2-54  
Bessel’s equation  
  (defined) 2-49, 2-53  
  modified (defined) 2-51  
bessel h 2-49  
bessel i 2-51  
bessel j 2-53  
bessel k 2-51  
bessel y 2-53  
beta 2-56  
beta function  
  (defined) 2-56  
  incomplete (defined) 2-56  
  natural logarithm of 2-56  
betainc 2-56  
betaln 2-56  
bicgstab 2-65  
Big Endian formats 2-274  
bin2dec 2-69  
binary data  
  reading from disk 2-416  
  saving to disk 2-581  
  writing to file 2-300  
binary to decimal conversion 2-69  
bit and 2-70  
bit cmp 2-71  
bit get 2-72  
bit max 2-73  
bit or 2-74  
bit set 2-75  
bit shift 2-76  
XOR 2-77  
bitxor 2-77  
blanks  
  removing trailing 2-177  
  braces, curly (special characters) 2-13  
  brackets (special characters) 2-13  
break 2-79  
brackets  
  listing 2-157  
  setting 2-159  
Buckminster Fuller 2-670  
built-in 2-80  
built-in functions 2-703  
C  
calendar 2-82  
cart2pol 2-83  
cart2sph 2-85  
Cartesian coordinates 2-83, 2-85, 2-517, 2-615  
case  
  in switch statement (defined) 2-666  
  lower to upper 2-689  
  upper to lower 2-426  
case 2-86  
cat 2-87  
catching errors 2-228  
Cayley-Hamilton theorem 2-529  
cd 2-88  
cdf 2r df 2-89  
cell 2-91  
cell array  
  conversion to from numeric array 2-472  
  creating 2-92  
  structure of, displaying 2-95  
cell2struct 2-93
Index

cellplot 2-95

cgs 2-97
changing working directory 2-88
char 2-101
characters (in format specification string)
  conversion 2-281, 2-627
  escape 2-280, 2-626
checkerboard pattern (example) 2-565
chol 2-103
Cholesky factorization 2-103
  (as algorithm for solving linear equations) 2-6
  lower triangular factor 2-502
  minimum degree ordering and (sparse) 2-668
  preordering for 2-117
cholinc 2-105
class 2-110
class, object See object classes
clear 2-111
clearing variables from workspace 2-111
clock 2-113
collapse dimensions, functions that 2-28
colmmd 2-114
colperm 2-117
combinations of n elements 2-460
combs 2-460
comma (special characters) 2-15
command window
  controlling number of lines per page in 2-454
common elements See set operations, intersection
companion matrix 2-118
complementary error function
  (defined) 2-223
  scaled (defined) 2-223
complete elliptic integral (defined) 2-215
  modulus of 2-213, 2-215
complex
  exponential (defined) 2-233
  logarithm 2-418, 2-420
  modulus (magnitude) 2-18
  numbers 2-353
  numbers, sorting 2-599, 2-600
  phase angle 2-30
  unitary matrix 2-539
  See also imaginary
  complex conjugate 2-124
  sorting pairs of 2-135
  complex Schur form 2-584
computer 2-119
computers supported by MATLAB
  numeric file formats of 2-274
  numeric precision of 2-285, 2-300
concatenating arrays 2-87
cond 2-121
condeig 2-122
condest 2-123
condition number of matrix 2-45, 2-121, 2-558
  estimated 2-123
conditional execution See flow control
conj 2-124
conjugate, complex 2-124
  sorting pairs of 2-135
contents file (cont ext s. n) 2-346
continuation (. . . , special characters) 2-14
continued fraction expansion 2-555
conv 2-125
conv2 2-126
conversion
  base to decimal 2-48
  binary to decimal 2-69
  Cartesian to cylindrical 2-83
| Cartesian to polar | 2-83 |
| complex diagonal to real block diagonal | 2-89 |
| cylindrical to Cartesian | 2-517 |
| decimal number to base | 2-174, 2-178 |
| decimal to binary | 2-179 |
| decimal to hexadecimal | 2-180 |
| full to sparse | 2-605 |
| hexadecimal to decimal | 2-350 |
| hexadecimal to double precision | 2-351 |
| integer to string | 2-379 |
| lowercase to uppercase | 2-689 |
| matrix to string | 2-440 |
| numeric array to cell array | 2-472 |
| numeric array to logical array | 2-421 |
| numeric array to string | 2-473 |
| partial fraction expansion to pole-residue | 2-567 |
| polar to Cartesian | 2-517 |
| pole-residue to partial fraction expansion | 2-567 |
| real to complex Schur form | 2-578 |
| spherical to Cartesian | 2-615 |
| string matrix to cell array | 2-96 |
| string to matrix (formatted) | 2-635 |
| string to numeric array | 2-641 |
| uppercase to lowercase | 2-426 |
| vector to character string | 2-101 |
| conversion characters (in format specification) | string 2-281, 2-627 |
| convhull | 2-128 |
| conv | 2-129 |
| convolution | 2-125 |
| inverse See deconvolution |
| two-dimensional | 2-126 |
| coordinates |
| Cartesian | 2-83, 2-85, 2-517, 2-615 |
| cylindrical | 2-83, 2-85, 2-517 |
| polar | 2-83, 2-85, 2-517 |
| spherical | 2-615 |
| See also conversion |
| corrcoef | 2-130 |
| cos | 2-131 |
| cosecant | 2-138 |
| hyperbolic | 2-138 |
| inverse | 2-22 |
| inverse hyperbolic | 2-22 |
| cosh | 2-131 |
| cosine | 2-131 |
| hyperbolic | 2-131 |
| inverse | 2-20 |
| inverse hyperbolic | 2-20 |
| cot | 2-132 |
| cotangent | 2-132 |
| hyperbolic | 2-132 |
| inverse | 2-21 |
| inverse hyperbolic | 2-21 |
| coth | 2-132 |
| cov | 2-133 |
| covariance |
| least squares solution and | 2-427 |
| cplxpair | 2-135 |
| cputime | 2-136 |
| creating online help for your own M-files | 2-346 |
| creating your own MATLAB functions | 2-296 |
| cross | 2-137 |
| cross product | 2-137 |
| csc | 2-138 |
| csch | 2-138 |
| ctranspose (M-file function equivalent for ' ) | 2-4 |
| cubic interpolation | 2-380, 2-383 |
| cubic spline interpolation | 2-380 |
| cumprod | 2-139 |
| cumsum | 2-140 |
| cumtrapz | 2-141 |
cumulative
    product 2-139
    sum 2-140
curly braces (special characters) 2-13
curve fitting (polynomial) 2-524
customizing your workspace 2-441, 2-638
Cuthill-McKee ordering, reverse 2-668, 2-670
cylindrical coordinates 2-83, 2-85, 2-517

data, aligning scattered
    multi-dimensional 2-461
    two-dimensional 2-341
data, ASCII
    converting sparse matrix after loading from 2-607
    reading from disk 2-416
    saving to disk 2-581
data, binary
    dependence upon array size and type 2-582
    reading from disk 2-416
    saving to disk 2-581
    writing to file 2-300
data, formatted
    reading from file 2-289
    writing to file 2-279
date 2-143
date and time functions 2-221
date string
    format of 2-145
date vector 2-147
datenum 2-144
datetin 2-145
datelat 2-147
datetime 2-148
dbcont 2-150
dbdown 2-151
dbexec 2-154
dbquit 2-155
dbstat 2-156
dbstatus 2-157
dbstep 2-158
dbstop 2-159
dbtype 2-162
dbup 2-163
ddeadv 2-164
ddeexec 2-166
ddeinit 2-167
ddepoke 2-168
ddereq 2-170
ddestep 2-172
ddeunadv 2-173
deal 2-174
deblank 2-177
debugging
    M-files 2-148-2-163, 2-406
    quitting debug mode 2-155
dec2base 2-174, 2-178
dec2bin 2-179
dec2hex 2-180
decimal number to base conversion 2-174, 2-178
decimal point (.)
    (special characters) 2-14
    to distinguish matrix and array operations 2-2
de decomposition
    Dulmage-Mendelsohn 2-198
    “economy-size” 2-539, 2-662
    orthogonal-triangular (QR) 2-427, 2-539
    Schur 2-584
    singular value 2-554, 2-662
deconv 2-181
deconvolution 2-181
default tolerance 2-222
definite integral 2-545
del operator 2-182
del 2 2-182
del aunay 2-185
del ete 2-188
deleting
  files 2-188
  workspace variables 2-111
delimiter
  in ASCII files 2-196, 2-197
density
  of sparse matrix 2-466
dependence, linear 2-659
derivative
  approximate 2-192
  polynomial 2-522
det 2-189
Detect 2-398
detecting
  alphabetic characters 2-399
  empty arrays 2-398
  equal arrays 2-398
  finite numbers 2-398
  global variables 2-399
  infinite elements 2-399
  logical arrays 2-399
  members of a set 2-403
  NaNs 2-399
  objects of a given class 2-402
  positive, negative, and zero array elements 2-594
  prime numbers 2-400
  real numbers 2-400
determinant of a matrix 2-189
di ag 2-190
diagonal 2-190
  anti- 2-345
  k-th (illustration) 2-681
  main 2-190
  sparse 2-609
di ary 2-191
diff 2-192
  differences
    between adjacent array elements 2-192
    between sets 2-589
differential equation solvers 2-475
  adjusting parameters of 2-489
  extracting properties of 2-488
digits
  controlling number of displayed 2-278
dimension statement (lack of in MATLAB) 2-715
dimensions
  functions that collapse 2-28
  size of 2-597
Diophantine equations 2-328
dir 2-194
direct term of a partial fraction expansion 2-567
directory
  changing working 2-88
  listing contents of 2-194, 2-701
  root 2-442
  temporary system 2-674
  See also search path
discontinuities, eliminating (in arrays of phase angles) 2-688
disp 2-195
distribution
  Gaussian 2-223
division
  array, left (arithmetic operator) 2-3
  array, right (arithmetic operator) 2-3
  by zero 2-370
  matrix, left (arithmetic operator) 2-3
  matrix, right (arithmetic operator) 2-2
modulo 2-453
of polynomials 2-181
remainer after 2-564
divisor
greatest common 2-328
dlmread 2-196
dlmwrite 2-197
dnper m 2-198
doc 2-199
documentation, hypertext-based
loading 2-199
dot product 2-137
double 2-200
dsearch 2-201
dual vector 2-464
Dulmage-Mendelsohn decomposition 2-198

E
echo 2-202
edge finding, Sobel technique 2-126
eig 2-204
eigensystem
transforming 2-89
eigenvalue
accuracy of 2-45, 2-204
complex 2-89
matrix logarithm and 2-422
modern approach to computation of 2-519
of companion matrix 2-118
poorly conditioned 2-45
problem 2-204, 2-523
problem, generalized 2-205, 2-523
problem, polynomial 2-523
repeated 2-205, 2-298
Wilkinson test matrix and 2-708
eigenvector
left 2-204
matrix, generalized 2-548
right 2-204
eig 2-207
elevation (spherical coordinates) 2-615
elipj 2-213
elipke 2-215
elliptic functions, Jacobian
(defined) 2-213
elliptic integral
complete (defined) 2-215
modulus of 2-213, 2-215
else 2-217
elseif 2-218
end 2-220
end of line, indicating 2-15
end-of-file indicator 2-242
eomday 2-221
eps 2-222
equal sign (special characters) 2-14
equations, linear
accuracy of solution 2-121
erf 2-223
erfc 2-223
erfcx 2-223
error
catching 2-228, 2-408
in file I/O 2-243
roundoff See roundoff error
error 2-225
error function
(defined) 2-223
complementary 2-223
scaled complementary 2-223
error message
displaying 2-225
Index

Index into matrix is negative or zero 2-421
Out of memory 2-499
retreiving last generated 2-408
escape characters (in format specification string) 2-280, 2-626
etime 2-227
eval 2-228
evalin 2-230
evaluating strings 2-228
exclamation point (special characters) 2-15
executing statements repeatedly 2-276, 2-705
execution
   conditional See flow control
   improving speed of by setting aside storage 2-715
   pausing M-file 2-505
   resuming from breakpoint 2-150
exist 2-231
exp 2-233
expint 2-234
expm 2-236
exponential 2-233
   complex (defined) 2-233
   integral 2-234
   matrix 2-236
exponentiation
   array (arithmetic operator) 2-3
   matrix (arithmetic operator) 2-3
expression, MATLAB 2-354
extension, filename
   .m 2-296, 2-684
   .mat 2-416, 2-581, 2-582
eye 2-238

F
factor 2-240
factorization
   LU 2-428
   QZ 2-523, 2-548
   See also decomposition
   factorization, Cholesky 2-103
   (as algorithm for solving linear equations) 2-6
   minimum degree ordering and (sparse) 2-668
   preordering for 2-117
factors, prime 2-240
fclose 2-241
feof 2-242
ferror 2-243
feval 2-244
fft 2-245
FFT See Fourier transform
fft2 2-248
fftn 2-249
fftshift 2-250
fgetl 2-251
fgets 2-252
fid See file identifier
field names of a structure, obtaining 2-253
fields, noncontiguous
   inserting data into 2-300
file
   closing 2-241
   deleting 2-188
   finding position within 2-293
   listing contents of 2-684
   listing names of 2-194
   MATLAB format 2-416
   opening 2-273
   reading ASCII delimited 2-196
   reading formatted data from 2-289
returning next line of (with carriage returns) 2-251
returning next line of (without carriage returns) 2-252
rewinding to beginning of 2-288
setting position within 2-292
status of 2-231
testing for end of 2-242
text versus binary, opening 2-274
writing binary data to 2-300
writing formatted data to 2-279
writing matrix as ASCII delimited 2-197
file identifier (fid) 2-242
file position indicator
finding 2-293
setting 2-292
setting to start of file 2-288
filename extension .mat 2-416, 2-581, 2-582
fileparts 2-254
filter 2-255
two-dimensional 2-126
filter 2-255
filter2 2-257
find 2-258
finding
file position indicator 2-293
indices of arrays 2-258
sign of array elements 2-594
zero of a function 2-303
See also detecting
findstr 2-260
finite numbers
detecting 2-398
FIR filter See filter
fix 2-261
fixed-point
output format 2-278
flint See floating-point, integer
flipd 2-262
fliplr 2-263
flipud 2-264
floating-point
integer 2-71, 2-75
integer, maximum 2-73
numbers, interval between 2-222
operations, count of 2-266
output format 2-278
floating-point arithmetic, IEEE
largest positive number 2-561
relative accuracy of 2-222
smallest positive number 2-562
floor 2-265
flows 2-266
flow control
break 2-79
case 2-86
else 2-217
elseif 2-218
end 2-220
error 2-225
for 2-276
if 2-354
keyboard 2-406
otherwise 2-497
return 2-569
switch 2-666
while 2-705
fmin 2-267
fmins 2-269
F-norm 2-468
fopen 2-273
for 2-276
Index

format
  for saving ASCII data 2-581
  output display 2-278
format 2-278
format specification string 2-279, 2-625
  matching file data to 2-289, 2-636
formatted data
  reading from file 2-289
  writing to file 2-279
Fourier transform
  algorithm, optimal performance of 2-246, 2-356, 2-357, 2-463
  convolution theorem and 2-125
  discrete, one-dimensional 2-245
  discrete, two-dimensional 2-248
  fast 2-245
  as method of interpolation 2-389
  inverse, one-dimensional 2-356
  inverse, two-dimensional 2-357
  shifting the DC component of 2-250
fprintf 2-279
fraction, continued 2-555
fragmented memory 2-499
fread 2-284
freqspace 2-287
frequency response
  desired response matrix
  frequency spacing 2-287
frequency vector 2-424
fwrite 2-300
fzero 2-303
G
gallery 2-306
gamma 2-326
gamma function
  (defined) 2-326
  incomplete 2-326
  logarithm of 2-326
gammainc 2-326
gammaln 2-326
Gaussian distribution function 2-223
Gaussian elimination
  (as algorithm for solving linear equations) 2-7, 2-393
  Gauss Jordan elimination with partial pivoting 2-576
  LU factorization and 2-428
gcd 2-328
generalized eigenvalue problem 2-205, 2-523
generating a sequence of matrix names (M1 through M12) 2-228
geodesic dome 2-670
gestalt 2-330
getfield 2-331
Givens rotations 2-542, 2-543
global 2-332
global variable
clearing 2-111
    defining 2-332
gmres 2-334
gradient 2-338
    gradient, numerical 2-338
graph
    adjacency 2-198
graphics objects
    deleting 2-188
greatest common divisor 2-328
grid
    aligning data to a 2-341
    grid arrays
        for volumetric plots 2-447
        multi-dimensional 2-461
griddata 2-341
hilb 2-352
    Hilbert matrix 2-352
        inverse 2-396
horzcat (M-file function equivalent for [ , ] ) 2-15
Householder reflections (as algorithm for solving
    linear equations) 2-7
hyperbolic
    cosecant 2-138
    cosecant, inverse 2-22
    cosine 2-131
    cosine, inverse 2-20
    cotangent 2-132
    cotangent, inverse 2-21
    secant 2-37, 2-587
    secant, inverse 2-37
    sine 2-38, 2-595
    sine, inverse 2-38
    tangent 2-40, 2-672
    tangent, inverse 2-40
hyperplanes, angle between 2-659
hypertext-based documentation
    loading 2-199
H
    H1 line 2-347, 2-425
hadamard 2-344
    Hadamard matrix 2-344
        subspaces of 2-659
    Hager's method 2-123
hankel 2-345
    Hankel functions, relationship to Bessel of 2-54
    Hankel matrix 2-345
help
    keyword search 2-425
    online 2-346
help 2-346
    Hermite transformations, elementary 2-328
hess 2-348
    Hessenberg form of a matrix 2-348
hex2dec 2-350
    hex2num 2-351
    hexadecimal format 2-278
hi l b 2-352
    Hilbert matrix 2-352
        inverse 2-396
IEEE floating-point arithmetic
    largest positive number 2-561
        relative accuracy of 2-222
        smallest positive number 2-562
    if 2-354
        i f f t 2-356
        i f f t 2 2-357
        i f f t n 2-358
    I I R filter See filter
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>imag</td>
<td>2-359</td>
</tr>
<tr>
<td>imaginary</td>
<td>part of complex number 2-359</td>
</tr>
<tr>
<td></td>
<td>parts of inverse FFT 2-356, 2-357</td>
</tr>
<tr>
<td></td>
<td>unit (sqrt(-1)) 2-353, 2-405</td>
</tr>
<tr>
<td>See also complex</td>
<td></td>
</tr>
<tr>
<td>iminfo</td>
<td>2-360</td>
</tr>
<tr>
<td>imread</td>
<td>2-363</td>
</tr>
<tr>
<td>imwrite</td>
<td>2-366</td>
</tr>
<tr>
<td>incomplete</td>
<td>beta function (defined) 2-56</td>
</tr>
<tr>
<td></td>
<td>gamma function (defined) 2-326</td>
</tr>
<tr>
<td>ind2sub</td>
<td>2-369</td>
</tr>
<tr>
<td>Index into matrix is negative or zero (error message)</td>
<td>2-421</td>
</tr>
<tr>
<td>Indexing</td>
<td>logical 2-421</td>
</tr>
<tr>
<td></td>
<td>indicator</td>
</tr>
<tr>
<td></td>
<td>end-of-file 2-242</td>
</tr>
<tr>
<td></td>
<td>file position 2-288, 2-292, 2-293</td>
</tr>
<tr>
<td>Indices, array</td>
<td>finding 2-258</td>
</tr>
<tr>
<td></td>
<td>of sorted elements 2-599</td>
</tr>
<tr>
<td>Inf</td>
<td>2-370</td>
</tr>
<tr>
<td>InferiorTo</td>
<td>2-371</td>
</tr>
<tr>
<td>infinity</td>
<td>2-370, 2-399</td>
</tr>
<tr>
<td>norm</td>
<td>2-468</td>
</tr>
<tr>
<td>Inheritance, of objects</td>
<td>2-110</td>
</tr>
<tr>
<td>inline</td>
<td>2-372</td>
</tr>
<tr>
<td>inpolygon</td>
<td>2-376</td>
</tr>
<tr>
<td>Input</td>
<td>checking number of M-file arguments 2-457</td>
</tr>
<tr>
<td></td>
<td>name of array passed as 2-378</td>
</tr>
<tr>
<td></td>
<td>number of M-file arguments 2-458</td>
</tr>
<tr>
<td></td>
<td>prompting users for 2-377, 2-446</td>
</tr>
<tr>
<td>input</td>
<td>2-377</td>
</tr>
<tr>
<td>Installation, root directory of</td>
<td>2-442</td>
</tr>
<tr>
<td>int2str</td>
<td>2-379</td>
</tr>
<tr>
<td>Integer</td>
<td>floating-point 2-71, 2-75</td>
</tr>
<tr>
<td></td>
<td>floating-point, maximum 2-73</td>
</tr>
<tr>
<td></td>
<td>integrable singularities 2-546</td>
</tr>
<tr>
<td></td>
<td>integration</td>
</tr>
<tr>
<td></td>
<td>quadrature 2-545</td>
</tr>
<tr>
<td>Interp1</td>
<td>2-380</td>
</tr>
<tr>
<td>Interp2</td>
<td>2-383</td>
</tr>
<tr>
<td>Interp3</td>
<td>2-387</td>
</tr>
<tr>
<td>Interplft</td>
<td>2-389</td>
</tr>
<tr>
<td>Interplpn</td>
<td>2-390</td>
</tr>
<tr>
<td>Interpolation</td>
<td>one-dimensional 2-380</td>
</tr>
<tr>
<td></td>
<td>two-dimensional 2-383</td>
</tr>
<tr>
<td></td>
<td>three-dimensional 2-387</td>
</tr>
<tr>
<td></td>
<td>multidimensional 2-390</td>
</tr>
<tr>
<td></td>
<td>cubic method 2-341, 2-380, 2-383, 2-387, 2-390</td>
</tr>
<tr>
<td></td>
<td>cubic spline method 2-380, 2-616</td>
</tr>
<tr>
<td></td>
<td>FFT method 2-389</td>
</tr>
<tr>
<td></td>
<td>linear method 2-380, 2-383</td>
</tr>
<tr>
<td></td>
<td>nearest neighbor method 2-341, 2-380, 2-383, 2-387, 2-390</td>
</tr>
<tr>
<td></td>
<td>trilinear method 2-341, 2-387, 2-390</td>
</tr>
<tr>
<td>Interpreter, MATLAB</td>
<td>search algorithm of 2-297</td>
</tr>
<tr>
<td>intersect</td>
<td>2-392</td>
</tr>
<tr>
<td>inv</td>
<td>2-393</td>
</tr>
<tr>
<td>Inverse</td>
<td>cosecant 2-22</td>
</tr>
<tr>
<td></td>
<td>cosine 2-20</td>
</tr>
<tr>
<td></td>
<td>cotangent 2-21</td>
</tr>
<tr>
<td></td>
<td>Fourier transform 2-356, 2-357</td>
</tr>
<tr>
<td></td>
<td>four-quadrant tangent 2-42</td>
</tr>
<tr>
<td></td>
<td>Hilbert matrix 2-396</td>
</tr>
<tr>
<td></td>
<td>hyperbolic cosecant 2-22</td>
</tr>
<tr>
<td></td>
<td>hyperbolic cosine 2-20</td>
</tr>
</tbody>
</table>
Index

hyperbolic cotangent 2-21
hyperbolic secant 2-37
hyperbolic sine 2-38
hyperbolic tangent 2-40
of a matrix 2-393
secant 2-37
sine 2-38
tangent 2-40
inversion, matrix
  accuracy of 2-121
involuntary matrix 2-502
inversion, matrix
  accuracy of 2-121
invhilb 2-396
involutary matrix 2-502
ipermute 2-397
is* 2-398
isa 2-402
iscell 2-398
iscellstr 2-398
ischar 2-398
isempty 2-398
isequal 2-398
isfield 2-398
isfinite 2-398
isglobal 2-399
ishandle 2-399
ishold 2-399
isieee 2-399
isinf 2-399
isletter 2-399
islogical 2-399
ismember 2-403
isnan 2-399
isnumeric 2-399
isobject 2-399
isppc 2-400
isprime 2-400
isreal 2-400
isspace 2-400
issparse 2-400
istr 2-404
isstruct 2-400
isstudent 2-400
isunix 2-400
isvms 2-400

J
J 2-405
J acobi rotations 2-624
J acobian elliptic functions
  (defined) 2-213
joining arrays See concatenating arrays

K
K>> prompt 2-406
keyboard 2-406
keyboard mode 2-406
  terminating 2-569
kr on 2-407
Kronecker tensor product 2-407

L
labeling
  matrix columns 2-195
  plots (with numeric values) 2-473
Laplacian 2-182
largest array elements 2-443
lasterr 2-408
lcm 2-410
ldivide (M-file function equivalent for .\) 2-4
least common multiple 2-410
least squares
  polynomial curve fitting 2-524
Index

problem 2-427
problem, nonnegative 2-464
problem, overdetermined 2-514
Legendre functions
(defined) 2-411
Schmidt semi-normalized 2-411
length 2-413
line numbers
M-file, listing 2-162
linear dependence (of data) 2-659
linear equation systems
accuracy of solution 2-121
solving overdetermined 2-540-2-541
linear equation systems, methods for solving
Cholesky factorization 2-6
Gaussian elimination 2-7
Householder reflections 2-7
least squares 2-464
matrix inversion (inaccuracy of) 2-393
linear interpolation 2-380, 2-383
linearly spaced vectors, creating 2-415
lines per page, controlling in command window 2-454
linspace 2-415
listing
breakpoints 2-157
directory contents 2-194
file contents 2-684
line numbers 2-162
M-files, MAT-files, and MEX-files 2-701
workspace variables 2-706
Little Endian formats 2-274
load 2-416
loading
WK1 spreadsheet files 2-709
local variables 2-296, 2-332
locating MATLAB functions 2-425, 2-703
log 2-418
log of MATLAB session, creating 2-191
log10 [log10] 2-420
log2 2-419
logarithm
base ten 2-420
base two 2-419
complex 2-418, 2-420
matrix (natural) 2-422
natural 2-418
of beta function (natural) 2-56
of gamma function (natural) 2-326
logarithmically spaced vectors, creating 2-424
logical 2-421
logical array
converting numeric array to 2-421
detecting 2-399
logical indexing 2-421
logical operations
AND, bit-wise 2-70
OR, bit-wise 2-74
XOR 2-714
XOR, bit-wise 2-77
logical operators 2-11
logical tests
all 2-27
any 2-32
See also detecting
logm 2-422
logspace 2-424
lookfor 2-425
Lotus123 WK1 spreadsheet file
reading data from a 2-709
writing a matrix to 2-710
lower 2-426
Index

lower triangular matrix 2-681
lowercase to uppercase 2-689
\texttt{lscov} 2-427
\texttt{l u} 2-428
LU factorization 2-428
storage requirements of (sparse) 2-474
\texttt{l u n c} 2-431

\textbf{M}
machine epsilon 2-705
\texttt{magic} 2-438
magic squares 2-438
\texttt{mat 2 str} 2-440
MAT-file 2-416, 2-582
converting sparse matrix after loading from 2-607
listing 2-701
MATLAB format files 2-416
MATLAB interpreter
search algorithm of 2-297
MATLAB search path
adding directories to 2-24
removing directories from 2-571
MATLAB startup file 2-441, 2-638
MATLAB version number 2-693
\texttt{mat la b. nat} 2-416, 2-581
\texttt{mat la brc} 2-441
\texttt{mat la broot} 2-442
matrix
addressing selected rows and columns of 2-16
arrowhead 2-117
companion 2-118
complex unitary 2-539
condition number of 2-45, 2-121, 2-558
converting to formatted data file 2-279
converting to vector 2-16
decomposition 2-539
defective (defined) 2-205
determinant of 2-189
diagonal of 2-190
Dulmage-Mendelsohn decomposition of 2-198
estimated condition number of 2-123
evaluating functions of 2-298
exponential 2-236
flipping left-right 2-263
flipping up-down 2-264
Hadamard 2-344, 2-659
Hankel 2-345
Hermitian Toeplitz 2-677
Hessenberg form of 2-348
Hilbert 2-352
identity 2-238
inverse 2-393
inverse Hilbert 2-396
inversion, accuracy of 2-121
involuntary 2-502
left division (arithmetic operator) 2-3
lower triangular 2-681
magic squares 2-438, 2-660
maximum size of 2-119
modal 2-204
multiplication (defined) 2-2
orthonormal 2-539
Pascal 2-502, 2-528
permutation 2-428, 2-539
poorly conditioned 2-352
power (arithmetic operator) 2-3
pseudo-inverse 2-514
reduced row echelon form of 2-576
replicating 2-565
right division (arithmetic operator) 2-2
Rosser 2-321
rotating $90^\circ$ 2-574
Schur form of 2-578, 2-584
singularity, test for 2-189
sorting rows of 2-600
sparse See sparse matrix
specialized 2-306
square root of 2-631
storing as binary data 2-300
subspaces of 2-659
test 2-306
Toeplitz 2-677
trace of 2-190, 2-678
transpose (arithmetic operator) 2-3
transposing 2-14
unimodular 2-328
unitary 2-662
upper triangular 2-682
Vandermonde 2-526
Wilkinson 2-610, 2-708
writing formatted data to 2-289
See also array
matrix functions
evaluating 2-298
matrix names, (M through M12) generating a sequence of 2-228
matrix power See matrix, exponential
max 2-443
maximum array size 2-119
mean 2-444
median 2-445
median value of array elements 2-445
memory, consolidating information to minimize use of 2-499
menu 2-446
menu (of user input choices) 2-446
meshgrid 2-447
message
   error See error message
   warning See warning message
methods
   inheritance of 2-110
MEX-file
clearing from memory 2-111
listing 2-701
M-file
displaying during execution 2-202
function 2-296
function file, echoing 2-202
listing 2-701
listing conventions 2-296
pausing execution of 2-505
programming 2-296
script 2-296
script file, echoing 2-202
min 2-452
minimizing, function
   of one variable 2-267
   of several variables 2-269
minimum degree ordering 2-668
minus (M-file function equivalent for -) 2-4
mod 2-453
modulo arithmetic 2-453
modulus, complex 2-18
Moore-Penrose pseudoinverse 2-514
mpower (M-file function equivalent for ^) 2-4
mrdivide (M-file function equivalent for \) 2-4
mldivide (M-file function equivalent for /) 2-4
mu2lin 2-455
multidimensional arrays
   concatenating 2-87
   interpolation of 2-390

max 2-443
mean 2-444
median 2-445
min 2-452
minus 2-4
mod 2-453
mpower 2-4
mrdivide 2-4
mldivide 2-4
mu2lin 2-455
max 2-443
mean 2-444
median 2-445
min 2-452
minus 2-4
mod 2-453
mpower 2-4
mrdivide 2-4
mldivide 2-4
mu2lin 2-455
longest dimension of 2-413
number of dimensions of 2-462
rearranging dimensions of 2-397, 2-512
removing singleton dimensions of 2-634
reshaping 2-566
size of 2-597
sorting elements of 2-599
See also array
multiple
  least common 2-410
multiplication
  array (arithmetic operator) 2-2
  matrix (defined) 2-2
  of polynomials 2-125
multistep ODE solver 2-481

nonzero entries (in sparse matrix)
  allocated storage for 2-474
  number of 2-466
  replacing with ones 2-618
  vector of 2-467
nonzeros 2-467

norm
  1-norm 2-468, 2-558
  2-norm (estimate of) 2-469
  F-norm 2-468
  infinity 2-468
  matrix 2-468
  pseudoinverse and 2-514-2-516
  vector 2-468
norm 2-468
normest 2-469
not (M-file function equivalent for ~) 2-11
now 2-470
null l 2-471
null space 2-471
num2cell 2-472
num2str 2-473
number
  of array dimensions 2-462
  of digits displayed 2-278
numbers
  complex 2-30, 2-353
  finite 2-398
  imaginary 2-359
  largest positive 2-561
  minus infinity 2-399
  NaN 2-399, 2-456
  plus infinity 2-370, 2-399
  prime 2-400, 2-531
  random 2-549, 2-551
  real 2-400, 2-560
  smallest positive 2-562
numeric file formats See computers supported by MATLAB.

numeric precision (of hardware) 2-285, 2-300

numerical differentiation formula ODE solvers 2-481

nzmax 2-474

0

object
determining class of 2-402
inheretance 2-110

object classes, list of predefined 2-110, 2-402

ODE See differential equation solvers ode45 and other solvers 2-475

odefile 2-483

odeget 2-488

odeset 2-489

ones 2-495

one-step ODE solver 2-480

online
documentation 2-199
help 2-346
keyword search 2-425

operating system command, issuing 2-15

operators
arithmetic 2-2
logical 2-11
precedence of 2-12
relational 2-9, 2-421
special characters 2-13

Optimization Toolbox 2-267, 2-270

logical OR
bit-wise 2-74
or (M-file function equivalent for | ) 2-11
ordering
minimum degree 2-668

reverse Cuthill-McKee 2-668, 2-670

orth 2-496

orthogonal-triangular decomposition 2-427, 2-539

orthonormal matrix 2-539

otherwise 2-497

Out of memory (error message) 2-499

output
controlling format of 2-278
controlling paging of 2-347, 2-454
number of M-file arguments 2-458

overdetermined equation systems, solving 2-540-2-541

overflow 2-370

P

pack 2-499

Padé approximation (of matrix exponential) 2-236

paging
controlling output in command window 2-347, 2-454

parentheses (special characters) 2-14

Parlett’s method (of evaluating matrix functions) 2-298

partial fraction expansion 2-567

partialpath 2-501

pascal 2-502

Pascal matrix 2-502, 2-528

path 2-503

pathname
of functions or files 2-703
partial 2-501
See also search path

pause 2-505

pausing M-file execution 2-505

pcode 2-506

pcopy 2-510
percent sign (special characters) 2-15
period (.), to distinguish matrix and array operations 2-2
period (special characters) 2-14
perms 2-511
permutation
  of array dimensions 2-512
  matrix 2-428, 2-539
  random 2-553
permutations of n elements 2-511
permute 2-512
phase, complex 2-30
  correcting angles 2-688
pi 2-513
pi (π) 2-513, 2-556, 2-595
pi mv 2-514
platform See computers
plot, volumetric
  generating grid arrays for 2-447
plotting See visualizing
plus (M-file function equivalent for +) 2-4
poly 2-518
pol2cart 2-517
polar coordinates 2-83, 2-85, 2-517
poles of transfer function 2-567
polyval 2-527
polyvalm 2-528
poorly conditioned
  eigenvalues 2-45
  matrix 2-352
pow2 2-530
power
  matrix See matrix exponential
  of two, next 2-463
power (M-file function equivalent for .^) 2-4
precedence of operators 2-12
prime factors 2-240
  dependence of Fourier transform on 2-248
prime numbers 2-400, 2-531
primes 2-531
printing, suppressing 2-15
prod 2-532
product
  cumulative 2-139
  Kronecker tensor 2-407
  of array elements 2-532
  of vectors (cross) 2-137
  scalar (dot) 2-137
profile 2-533
K>> prompt 2-406
prompting users for input 2-377, 2-446
pseudo inverse 2-514

division 2-181
eigenvalue problem 2-523
evaluation 2-527
evaluation (matrix sense) 2-528
multiplication 2-125
pol yval 2-527
pol yval m 2-528
prime numbers 2-400, 2-531
prim es 2-531
printing, suppressing 2-15
prod 2-532
product
  cumulative 2-139
  Kronecker tensor 2-407
  of array elements 2-532
  of vectors (cross) 2-137
  scalar (dot) 2-137
profile 2-533
K>> prompt 2-406
prompting users for input 2-377, 2-446
pseudo inverse 2-514

Q
qmr 2-535
qr 2-539
QR decomposition 2-427, 2-539
  deleting a column from 2-542
  inserting a column into 2-543
qd defeated 2-542
qr insert 2-543
qr write 2-544
quad 2-545
quad8 2-545
quadrature 2-545
quit 2-547
quitting MATLAB 2-547
quotation mark, inserting in a string 2-283
qz 2-548
QZ factorization 2-523, 2-548

R
rand 2-549, 2-661
randn 2-371, 2-551
random
  numbers 2-549, 2-551
  permutation 2-553
  sparse matrix 2-622, 2-623
  symmetric sparse matrix 2-624
randperm 2-553
range space 2-496
rank 2-554
rank of a matrix 2-554
rat 2-555
rational fraction approximation 2-555
rats 2-555
rcond 2-558
divide (M-file function equivalent for ./) 2-4
reading
  ASCII delimited file 2-196
  formatted data from file 2-289
  WK1 spreadsheet files 2-709
readme file 2-702
readsnd 2-559
real 2-560
real numbers 2-400, 2-560
real Schur form 2-584
realmax 2-561
realm 2-562
rearranging arrays
  converting to vector 2-16
  removing first n singleton dimensions 2-593
  removing singleton dimensions 2-634
  reshaping 2-566
  shifting dimensions 2-593
  swapping dimensions 2-397, 2-512
rearranging matrices
  converting to vector 2-16
  flipping left-right 2-263
  flipping up-down 2-264
  rotating 90° 2-574
  transposing 2-14
recordsound 2-563
reduced row echelon form 2-576
regularly spaced vectors, creating 2-16, 2-415
relational operators 2-9, 2-421
relative accuracy
  floating-point 2-222
rem 2-564
remainder after division 2-564
repeatedly executing statements 2-276, 2-705
replicating a matrix 2-565
repmat 2-565
reshape 2-566
residue 2-567
residues of transfer function 2-567
resume execution (from breakpoint) 2-150
return 2-569
reverse Cuthill-McKee ordering 2-668, 2-670
Index

rmfield 2-570
rmpath 2-571
RMS See root-mean-square
root directory 2-442
root-mean-square
  of vector 2-468
roots 2-572
roots of a polynomial 2-518-2-519, 2-572
Rosenbrock banana function 2-270
Rosenbrock ODE solver 2-481
Rosser matrix 2-321
rot90 2-574
rotations
  Givens 2-542, 2-543
  Jacobi 2-624
round
  to nearest integer 2-575
  towards infinity 2-91
  towards minus infinity 2-265
  towards zero 2-261
round 2-575
roundoff error
  characteristic polynomial and 2-519
  convolution theorem and 2-125
  effect on eigenvalues 2-45
  evaluating matrix functions 2-298
  in inverse Hilbert matrix 2-396
  partial fraction expansion and 2-568
  polynomial roots and 2-572
  sparse matrix conversion and 2-608
rref 2-576
rrefmovie 2-576
rsf2csf 2-578
Runge-Kutta ODE solvers 2-480
S
save 2-581
saving
  ASCII data 2-581
  WK1 spreadsheet files 2-710
  workspace variables 2-581
scalar product (of vectors) 2-137
scaled complementary error function (defined) 2-223
scattered data, aligning
  multi-dimensional 2-461
  two-dimensional 2-341
Schmidt semi-normalized Legendre functions 2-411
schur 2-584
Schur decomposition 2-584
  matrix functions and 2-298
  Schur form of matrix 2-578, 2-584
script 2-586
scrolling, screen See paging
search path
  adding directories to 2-24
  MATLAB’s 2-503, 2-684
  removing directories from 2-571
search, string 2-260
sec 2-587
secant 2-587
secant, inverse 2-37
secant, inverse hyperbolic 2-37
sech 2-587
semicolon (special characters) 2-15
sequence of matrix names (M1 through M12)
  generating 2-228
session
  saving log of 2-191
set operations
  difference 2-589
exclusive or 2-592
intersection 2-392
membership 2-403
union 2-686
unique 2-687
setdiff 2-589
setfield 2-590
setstr 2-591
setxor 2-592
shiftdim 2-593
sign 2-594
signum function 2-594
Simpson's rule, adaptive recursive 2-546
sin 2-595
sine 2-595
sine, inverse 2-38
sine, inverse hyperbolic 2-38
single quote (special characters) 2-14
singular value
  decomposition 2-554, 2-662
  largest 2-468
  rank and 2-554
singularities
  integrable 2-546
  soft 2-546
sinh 2-595
size 2-597
size of array dimensions 2-597
size vector 2-566, 2-597
skipping bytes (during file I/O) 2-300
smallest array elements 2-452
soccer ball (example) 2-670
soft singularities 2-546
sort 2-599
sorting
  array elements 2-599
  complex conjugate pairs 2-135
matrix rows 2-600
sortrows 2-600
sound
  converting vector into 2-601, 2-603
sound 2-601, 2-603
soundcap 2-602
spaloc 2-604
sparse 2-605
sparse matrix
  allocating space for 2-604
  applying function only to nonzero elements of
    2-614
  density of 2-466
  diagonal 2-609
  finding indices of nonzero elements of 2-258
  identity 2-613
  minimum degree ordering of 2-114
  number of nonzero elements in 2-466, 2-605
  permuting columns of 2-117
  random 2-622, 2-623
  random symmetric 2-624
  replacing nonzero elements of with ones 2-618
  results of mixed operations on 2-605
  vector of nonzero elements 2-467
  visualizing sparsity pattern of 2-629
sparse storage
  criterion for using 2-294
spconvert 2-607
spdiags 2-609
speak 2-612
speye 2-613
spfun 2-614
sph2cart 2-615
spherical coordinates 2-615
splinest 2-616
spline interpolation (cubic) 2-380
Spline Toolbox 2-382
spones 2-618
sparses 2-619
sprand 2-622
sprandn 2-623
sprandsym 2-624
spreadsheet
  loading WK1 files 2-709
  reading ASCII delimited file into a matrix
  2-196
  writing matrix as ASCII delimited file 2-197
  writing WK1 files 2-710
sprintf 2-625
spy 2-629
sqrt 2-630
sqrtm 2-631
square root
  of a matrix 2-631
  of array elements 2-630
squeeze 2-634
sscanf 2-635
standard deviation 2-639
startup 2-638
startup file 2-441, 2-638
status
  of file or variable 2-231
std 2-639
stopwatch timer 2-676
storage
  allocated for nonzero entries (sparse) 2-474
  sparse 2-605
str2cell 2-96
str2num 2-641
strcat 2-642
strcmp 2-644
string
  comparing one to another 2-644
  comparing the first n characters of two 2-649
  converting from vector to 2-101
  converting matrix into 2-440, 2-473
  converting to lowercase 2-426
  converting to matrix (formatted) 2-635
  converting to numeric array 2-641
  converting to uppercase 2-689
  dictionary sort of 2-600
  evaluating as expression 2-228
  finding first token in 2-651
  inserting a quotation mark in 2-283
  searching and replacing 2-650
  searching for 2-260
  string matrix to cell array conversion 2-96
  strings 2-646
  strjust 2-647
  strcmp 2-648
  strncmp 2-649
  strrep 2-650
  strtok 2-651
  struct2cell 2-653
  structure array
    field names of 2-253
    getting contents of field of 2-331
    remove field from 2-570
    setting contents of a field of 2-590
  strvcat 2-654
  sub2ind 2-655
  subfunction 2-296
  subsasgn 2-656
  subspace 2-659
  subsref 2-658
  subsref (M-file function equivalent for
              A(i,j,k...)) 2-15
  subtraction (arithmetic operator) 2-2
    cumulative 2-140
    of array elements 2-660
Index

sum 2-660
superior to 2-661
svd 2-662
svds 2-664
switch 2-666
symmd 2-668
symrcm 2-670
syntaxes
  of M-file functions, defining 2-296

T
table lookup See interpolation
  tab-separated ASCII format 2-581
tag 2-672
tangent 2-672
  hyperbolic 2-672
tangent (four-quadrant), inverse 2-42
tangent, inverse 2-40
tangent, inverse hyperbolic 2-40
tanh 2-672
  Taylor series (matrix exponential approximation) 2-236
tempdir 2-674
tempname 2-675
temporary
  file 2-675
  system directory 2-674
tensor, Kronecker product 2-407
test matrices 2-306
test, logical See logical tests and detecting
tic 2-676
tiling (copies of a matrix) 2-565
time
  CPU 2-136
  elapsed (stopwatch timer) 2-676
  required to execute commands 2-227
times (M-file function equivalent for \texttt{.*}) 2-4
toc 2-676
toeplitz 2-677
  Toeplitz matrix 2-677
token See also string 2-651
tolerance, default 2-222
  Toolbox
    Optimization 2-267, 2-270
    Spline 2-382
toolbox
  undocumented functionality in 2-702
trace 2-678
  trace of a matrix 2-190, 2-678
trailing blanks
  removing 2-177
transform, Fourier
  discrete, one-dimensional 2-245
  discrete, two-dimensional 2-248
  inverse, one-dimensional 2-356
  inverse, two-dimensional 2-357
  shifting the DC component of 2-250
transformation
  elementary Hermite 2-328
  left and right (QZ) 2-548
  See also conversion
transpose
  array (arithmetic operator) 2-3
  matrix (arithmetic operator) 2-3
transpose (M-file function equivalent for \texttt{.'}) 2-4
trapz 2-679
tricubic interpolation 2-341
tril 2-681
  trilinear interpolation 2-341, 2-387, 2-390
triu 2-682
  truth tables (for logical operations) 2-11
tsearch 2-683
Index

I-26

type 2-684

U
uint8 2-685
uminus (M-file function equivalent for unary –) 2-4
undefined numerical results 2-456
undocumented functionality 2-702
unimodular matrix 2-328
union 2-686
uniq 2-687
unitary matrix (complex) 2-539
unw ap 2-688
uplus (M-file function equivalent for unary +) 2-4
upper 2-689
upper triangular matrix 2-682
uppercase to lowercase 2-426

V
vander 2-690
Vandermonde matrix 2-526
varargin 2-690
varargout 2-690
variable numbers of M-file arguments 2-690
variables
clearing 2-111
global 2-332
/workspace listing 2-706
/local 2-296, 2-332
name of passed 2-378
retrieving from disk 2-416
saving to disk 2-581
sizes of 2-706
status of 2-231

See also workspace

vector
dual 2-464
frequency 2-424
length of 2-413
product (cross) 2-137
vectorize 2-692
vectors, creating
logarithmically spaced 2-424
regularly spaced 2-16, 2-415
ver 2-693
version 2-693
vertcat (M-file function equivalent for [ ; ] ) 2-15
visualizing
cell array structure 2-95
sparse matrices 2-629
voronoi 2-694

W
warning 2-696
warning message (enabling, suppressing, and displaying) 2-696
wavread 2-697
wavwrite 2-698
weekday 2-700
well conditioned 2-558
what 2-701
whatsnew 2-702
which 2-703
while 2-705
white space characters, ASCII 2-400, 2-651
who 2-706
whos 2-706
wildcard (*)
using with clear command 2-111
using with dir command 2-194
Wilkinson 2-708
Wilkinson matrix 2-610, 2-708
wk1read 2-709
wk1write 2-710
workspace
  changing context while debugging 2-151, 2-163
  clearing variables from 2-111
  consolidating memory 2-499
  predefining variables 2-441, 2-638
  saving 2-581
  See also variables
writesnd 2-711
writing
  binary data to file 2-300
  formatted data to file 2-279
  matrix as ASCII delimited file 2-197
  string to matrix (formatted) 2-635
  WK1 spreadsheet files 2-710

X
xlgetrange 2-712
logical XOR 2-714
  bit-wise 2-77
xor 2-714
xyz coordinates See Cartesian coordinates

Z
zero of a function, finding 2-303
zero-padding
  while converting hexadecimal numbers 2-351
  while reading binary files 2-284
zeros 2-715