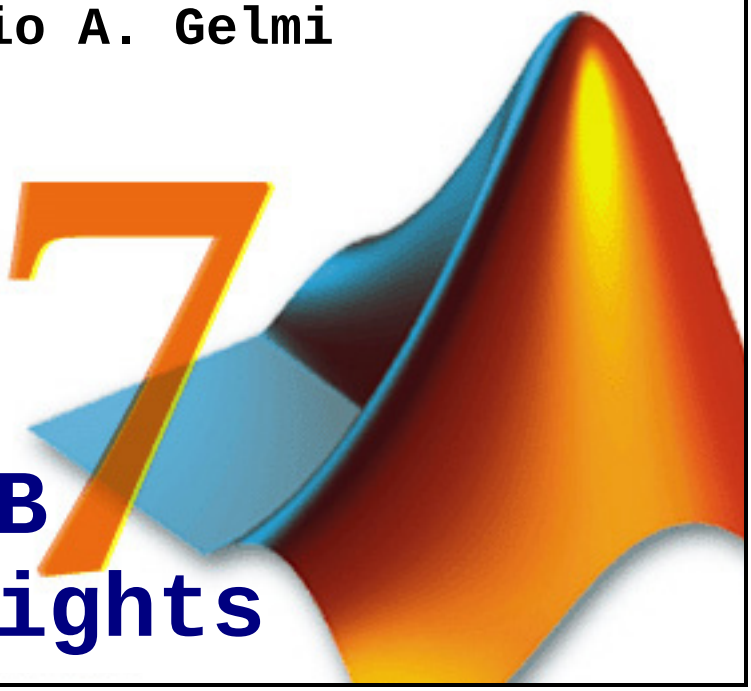
MATLAB®

The Language of Technical Computing

>> Claudio A. Gelmi



MATLAB 7 Highlights

INDEX

Useful commands sorted by category

First: where to look for help

HELP

This command can be used in multiple forms:

HELP	by itself, lists all primary help topics. Each primary topic corresponds to a directory name on the MATLABPATH.
HELP /	lists a description of all operators and special characters.
HELP FUN	displays a description of and syntax for the function FUN.
HELP NAME OF THE TOOLBOX	lists all the commands associated with that toolbox.

Another way to get help is using the ? bottom in MATLAB. Once it is pressed it will automatically open a help navigator for contents, index, search and demos. Extremely helpful and fast.

Also The Mathworks (www.mathworks.com) website is full of resources and helpful articles, including Matlab Central (File Exchange, Newsgroup, Link Exchange, and Blogs) which is an open exchange for the MATLAB and Simulink user community.

Algebraic Operations

DIFF

Difference and approximate derivative.

INLINE

Construct INLINE object (this function is being replaced by anonymous functions (@) in versions of MATLAB® 7.0 or above).

Examples:

```
g = inline('t^2')
g = inline('sin(2*pi*f + theta)', 'f', 'theta')

F = inline('x^2-1'); x0 = fzero(F,4.5)
```

LAPLACE DOMAIN

The following example generates a transfer function and plots the response given a unitary step:

```
g1 = tf([a1 a0], [b2 b1 b0])
step(g2, [0:1:50])
```

LINSOLVE

Solve linear system A*X=B. X = LINSOLVE(A,B) solves the linear system A*X=B using LU factorization with partial pivoting when A is square, and QR factorization with column pivoting otherwise. Warning is given if A is ill conditioned for square matrices and rank deficient for rectangular matrices.

NANMEAN

Mean value, ignoring NaNs.

QUAD

Numerically evaluate integral, adaptive Simpson quadrature.

Example:

```
Q = quad(@myfun,0,2);
```

where myfun.m is the M-file function:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

QUADL

Numerically evaluate integral, adaptive Lobatto quadrature.

ZSCORE

Standardized z score. Z = ZSCORE(X) returns a centered, scaled version of X, known as the Z scores of X. For a vector input, Z = (X - MEAN(X)) ./ STD(X). For a matrix input, Z is a row vector containing the Z scores of each column of X. For N-D arrays, ZSCORE operates along the first non-singleton dimension.

This function is commonly used to preprocess data before computing distances for cluster analysis.

Command Window and DOS commands

CD

Change current working directory.

CLC

Clears the command window and homes the cursor.

CLF

CLF deletes all children of the current figure with visible handles.
CLF RESET deletes all children (including ones with hidden handles) and also resets all figure properties, except Position and Units, to their default values.
CLF(FIG) or CLF(FIG,'RESET') clears the single figure with handle FIG.
FIG_H = CLF(...) returns the handle of the figure.

DIARY

Save text of MATLAB session. DIARY FILENAME causes a copy of all subsequent command window input and most of the resulting command window output to be appended to the named file. If no file is specified, the file 'diary' is used.

HOME

Moves the cursor to the upper left corner of the Command Window and clears the visible portion of the window. You can use the scroll bar to see what was on the screen previously.

PWD

Displays the current working directory.

WHOS

WHOS is a long form of WHO. It lists all the variables in the current workspace, together with information about their size, bytes, class, etc.

Files

DLMWRITE

Write ASCII delimited file.

LOAD & SAVE

```
load file.txt -ascii
load('c:\work\filename.txt')
save variable.txt var  -ascii  -tab
save('filename' 'var1' 'var2')
save('c:\work\filename.txt', 'var1', 'var2', '-ascii')
```

XLSREAD

Get data and text from a spreadsheet in an Excel workbook.

Graphic and Plot commands

AREA

Filled area plot. AREA(X,Y) produces a stacked area plot suitable for showing the contributions of various components to a whole.

AXIS

AXIS([XMIN XMAX YMIN YMAX])
sets scaling for the x- and y-axes on the current plot.

AXIS AUTO	returns the axis scaling to its default, automatic mode where, for each dimension, 'nice' limits are chosen based on the extents of all line, surface, patch, and image children.
AXIS EQUAL	sets the aspect ratio so that equal tick mark increments on the x-,y- and z-axis are equal in size. This makes SPHERE(25) look like a sphere, instead of an ellipsoid.
AXIS TIGHT	sets the axis limits to the range of the data.
AXIS SQUARE	makes the current axis box square in size.
AXIS OFF	turns off all axis labeling, tick marks and background.
AXIS ON	turns axis labeling, tick marks and background back on.

DRAWNOW

"Flushes the event queue" and forces MATLAB to update the screen.

EZPLOT

EZPLOT(FUN) plots the function FUN(X) over the default domain -2*PI < X < 2*PI.
EZPLOT(FUN,[A,B]) plots FUN(X) over A < X < B.

Examples:
The easiest way to express a function is via a string:

Example:
```
ezplot('x^2 - 2*x + 1')
```

One programming technique is to vectorize the string expression using the array operators .* (TIMES), ./ (RDIVIDE), .\ (LDIVIDE), .^ (POWER). This makes the algorithm more efficient since it can perform multiple function evaluations at once.
Example:
```
ezplot('x.*y + x.^2 - y.^2 - 1')
```

You may also use a function handle to an existing function. Function handles are more powerful and efficient than string expressions.

```
ezplot(@humps)
ezplot(@cos,@sin)
```

If your function has additional parameters, for example k in myfun:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you may use an anonymous function to specify that parameter:

Example:
```
ezplot(@(x,y)myfun(x,y,2))
```

EZSURFC

EZSURFC(FUN) plots a graph of the function FUN(X,Y) using SURFC. FUN is plotted over the default domain -2*PI < X < 2*PI, -2*PI < Y < 2*PI.

EZSURFC(FUN,DOMAIN) plots FUN over the specified DOMAIN instead of the default domain. DOMAIN can be the vector [XMIN,XMAX,YMIN,YMAX] or the [A,B] (to plot over A < X < B, A < Y < B).

Example:
```
ezsurfc('x.*exp(-x.^2 - y.^2)')
```

FPLOT

FPLOT(FUN,LIMS) plots the function FUN between the x-axis limits specified by LIMS = [XMIN XMAX]. Using LIMS = [XMIN XMAX YMIN YMAX] also controls the y-axis limits. FUN(x) must return a row vector for each element of vector x.

Examples:

```
fplot(@humps,[0 1])
fplot(@(x)[tan(x),sin(x),cos(x)], 2*pi*[-1 1 -1 1])
fplot(@(x) sin(1./x), [0.01 0.1], 1e-3)
f = @(x,n)abs(exp(-1j*x*(0:n-1))*ones(n,1));
fplot(@(x)f(x,10),[0 2*pi])
```

GRIDDATA

Data gridding and surface fitting. Suitable for plotting experimental data. ZI = GRIDDATA(X,Y,Z,XI,YI) fits a surface of the form Z = F(X,Y) to the data in the (usually) nonuniformly-spaced vectors (X,Y,Z). GRIDDATA interpolates this surface at the points specified by (XI,YI) to produce ZI. The surface always goes through the data points. XI and YI are usually a uniform grid (as produced by MESHGRID) and is where GRIDDATA gets its name.

XI can be a row vector, in which case it specifies a matrix with constant columns. Similarly, YI can be a column vector and it specifies a matrix with constant rows.

[XI,YI,ZI] = GRIDDATA(X,Y,Z,XI,YI) also returns the XI and YI formed this way (the results of [XI,YI] = MESHGRID(XI,YI)).

[...] = GRIDDATA(X,Y,Z,XI,YI,METHOD) where METHOD is one of
- 'linear' - Triangle-based linear interpolation (default)
- 'cubic' - Triangle-based cubic interpolation
- 'nearest' - Nearest neighbor interpolation
- 'v4' - MATLAB 4 griddata method

Examples:
Sample a function at 100 random points between ±2.0:

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

x, y, and z are now vectors containing nonuniformly sampled data. Define a regular grid, and grid the data to it:

```
ti = -2:.25:2;
[XI,YI] = meshgrid(ti,ti);
ZI = griddata(x,y,z,XI,YI);
```

Plot the gridded data along with the nonuniform data points used to generate it:

```
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```

IMAGESC

Scale data and display as image (nice figure!).

MARKERSIZE

A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the '.' symbol) at one-third the specified size.

MESHGRID

X and Y arrays for 3-D plots. [X,Y] = MESHGRID(x,y) transforms the domain specified by vectors x and y into arrays X and Y that can be used for the evaluation of functions of two variables and 3-D surface plots. The rows of the output array X are copies of the vector x and the columns of the output array Y are copies of the vector y.

[X,Y] = MESHGRID(x) is an abbreviation for [X,Y] = MESHGRID(x,x).
[X,Y,Z] = MESHGRID(x,y,z) produces 3-D arrays that can be used to evaluate functions of three variables and 3-D volumetric plots.

For example, to evaluate the function x*exp(-x^2-y^2) over the range -2 < x < 2, -2 < y < 2,

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```

MOVIE

Play recorded movie frames. MOVIE(M) plays the movie in array M once. M must be an array of movie frames (usually from GETFRAME).

PLOTYY

Graphs with y tick labels on the left and right PLOTYY(X1,Y1,X2,Y2) plots Y1 versus X1 with y-axis labeling on the left and plots Y2 versus X2 with y-axis labeling on the right.

PLOTYY(X1,Y1,X2,Y2,FUN) uses the plotting function FUN instead of PLOT to produce each graph. FUN can be a function handle or a string that is the name of a plotting function, e.g. plot, semilogx, semilogy, loglog, stem, etc. or any function that accepts the syntax H = FUN(X,Y). For example

```
PLOTYY(X1,Y1,X2,Y2,@loglog)  % Function handle
PLOTYY(X1,Y1,X2,Y2,'loglog') % String
```

XLIM, YLIM & ZLIM

For plots, set X and Y limits.

Matrices

DIAG

Diagonal matrices and diagonals of a matrix.

Examples:

```
m = 5;
diag(-m:m) + diag(ones(2*m,1),1) + diag(ones(2*m,1),-1)
produces a tridiagonal matrix of order 2*m+1.

a = 5*ones(1,6)
diag(a)
```

EYE

Is the N-by-N identity matrix.

FLIPLR

Flip matrix in left/right direction.

FLIPUD

Flip matrix in up/down direction.

PERMUTE

Permute array dimensions. B = PERMUTE(A,ORDER) rearranges the dimensions of A so that they are in the order specified by the vector ORDER.

RANDN

Normally distributed random numbers.

RANDPERM

Random permutation. RANDPERM(n) is a random permutation of the integers from 1 to n. For example, RANDPERM(6) might be [2 4 5 6 1 3].

REPMAT

Replicate and tile an array. B = repmat(A,M,N) creates a large matrix B consisting of an M-by-N tiling of copies of A. The size of B is [size(A,1)*M, size(A,2)*N]. Faster that ones()

RESHAPE

Change size. RESHAPE(X,M,N) returns the M-by-N matrix whose elements are taken columnwise from X. An error results if X does not have M*N elements.

Shortcuts

```
A(2,:) = [] % deletes the second column of matrix A

A(:)        % transform any matrix in a vector
```

Miscellaneous

CHAR

Create character array (string).

Example:

```
T = char('Hi' 'There')
```

DEAL

Distribute inputs to outputs.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})

a =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

b =

    1
    1
    1

c =

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

```
d =

     0
     0
     0
```

(http://www.mathworks.com/company/newsletters/digest/june98/deal.html).

DEE

In Simulink, DEE is a differential equation editor.

MEMBRANE

Generates the MATLAB logo.

NUM2STR

Convert numbers to a string.

Example:

```
disp(['My favorite number is ' num2str(7)])
```

ODE15i

Solve fully implicit differential DAEs equations.

PCODE

Create pre-parsed pseudo-code file (P-file).

SENDMAIL

Send e-mail.

SUBS

Symbolic substitution.

STRING OPERATIONS

Function	Description
'str'	Create the string specified between quotes.
blanks	Create a string of blanks.
sprintf	Write formatted data to a string.
strcat	Concatenate strings.
strvcat	Concatenate strings vertically.

Functions to Modify Character Arrays	
Function	**Description**
deblank	Remove trailing blanks.
lower	Make all letters lowercase.
sort	Sort elements in ascending or descending order.
strjust	Justify a string.
strrep	Replace one string with another.
strtrim	Remove leading and trailing white space.
upper	Make all letters uppercase.

Functions to Read and Operate on Character Arrays	
Function	**Description**
eval	Execute a string with MATLAB expression.
sscanf	Read a string under format control.

Functions to Search or Compare Character Arrays	
Function	**Description**
findstr	Find one string within another.
strcmp	Compare strings.
strcmpi	Compare strings, ignoring case.
strmatch	Find matches for a string.
strncmp	Compare the first N characters of strings.
strncmpi	Compare the first N characters, ignoring case.

strtok	Find a token in a string.

Functions to Determine Data Type or Content	
Function	**Description**
iscellstr	Return `true` for a cell array of strings.
ischar	Return `true` for a character array.
isletter	Return `true` for letters of the alphabet.
isstrprop	Determine if a string is of the specified category.
isspace	Return `true` for white-space characters.

Functions to Convert Between Numeric and String Data Types	
Function	**Description**
char	Convert to a character or string.
cellstr	Convert a character array to a cell array of strings.
double	Convert a string to numeric codes.
int2str	Convert an integer to a string.
mat2str	Convert a matrix to an `eval`'able string.
num2str	Convert a number to a string.
str2num	Convert a string to a number.
str2double	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets	
Function	**Description**
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector

STRCAT

Concatenate strings. T = STRCAT(S1,S2,S3,...) horizontally concatenates corresponding rows of the character arrays S1, S2, S3 etc. All input arrays 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.

Example:
```
filename = strcat(['save E',  num2str(alpha), '.txt varname  -ascii'])
eval(filename)
```

STRCMP

Compare strings.

TYPE

List M-file.

Programming

ASSIGNIN

Assign variable in workspace. ASSIGNIN(WS,'name',V) assigns the variable 'name' in the workspace WS the value V. WS can be one of 'caller' or 'base'.

Example:
```
% Change the value of a variable from an M-file
assignin('base', 'a',5)
```

BREAK

Terminates the execution of the loop (for or while).

CONTINUE

Passes control to the next iteration of FOR or WHILE loop in which it appears, skipping any remaining statements in the body of the FOR or WHILE loop.

FEVAL

Execute the specified function. FEVAL(F,x1,...,xn) evaluates the function specified by a function handle or function name, F, at the given arguments, x1,...,xn.

For example, if F = @foo, FEVAL(F,9.64) is the same as foo(9.64).

INPUT

Prompt for user input.

The following example gives the prompt in the text string and waits for character string input:

```
R = input('What is your name','s');     % s tell MATLAB to wait for a
string.
```

INPUTNAME

Input argument name. Inside the body of a user-defined function, INPUTNAME(ARGNO) returns the caller's workspace variable name corresponding to the argument number ARGNO. If the input has no name, for example, when it is the result of a calculation or an expression such as, a(1), varargin{:}, eval(expr), etc, then INPUTNAME returns an empty string.

Example:
Suppose the function myfun is defined as:

```
function y = myfun(a,b)
disp(sprintf('My first input is "%s".' ,inputname(1)))
disp(sprintf('My second input is "%s".',inputname(2)))
y = a+b;
```

then

```
x = 5; myfun(x,5)
```

produces

```
My first input is "x".
My second input is "".
```

LASTERR

Last error message.

NARGIN

Number of function input arguments. Inside the body of a user-defined function, NARGIN returns the number of input arguments that were used to call the function.

NARGOUT

Number of function output arguments. Inside the body of a user-defined function, NARGOUT returns the number of output arguments that were used to call the function.

INTERSECT

Set intersection. INTERSECT(A,B) when A and B are vectors returns the values common to both A and B. The result will be sorted. A and B can be cell arrays of strings.

RETURN

Simply returns the control to the invoking function.

SHORT-CIRCUIT OPERATORS && ||

The following operators perform AND and OR operations on logical expressions containing scalar values. They are short-circuit operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
&&	Returns logical 1 (true) if both inputs evaluate to true, and logical 0 (false) if they do not.
||	Returns logical 1 (true) if either input, or both, evaluate to true, and logical 0 (false) if they do not.

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

If A equals zero, then the entire expression will evaluate to logical 0 (false), regardless of the value of B. Under these circumstances, there is no need to evaluate B because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is true. Again, regardless of the value of B, the statement will evaluate to true. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute an M-file function only if the M-file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, myfun.m, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids divide-by-zero errors when b equals zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the && and || operators in if and while statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

SIM

Simulate your Simulink model using all simulation parameter dialog settings.

SWITCH

Switch among several cases based on expression.

Example:

```
color = input('color: ', 's')

swith color
case 'red'
      c = [1 0 0]
case 'green'
      c = [0 1 0]
otherwise
      error('Invalid choice of color')
end
```

Reference

ANY

Determine if any array elements are nonzero.

FIND

Find indices of nonzero elements.

I = FIND(X) returns the linear indices of the array X that are nonzero. X may be a logical expression. Use IND2SUB(I,SIZE(X)) to calculate multiple subscripts from the linear indices I.

Example:

```
A = magic(3)
find(A > 5)
```

finds the linear indices of the 4 entries of the matrix A that are greater than 5.

ISFINITE

True for finite elements.

ISINF

True for infinite elements.

ISKEYWORD

Check if input is a keyword. ISKEYWORD(S) returns one if S is a MATLAB keyword, and 0 otherwise. MATLAB keywords cannot be used as variable names.

ISNAN

True for Not-a-Number.

NNZ

Number of nonzero matrix elements

NUMEL

Number of elements in an array or subscripted array expression.

Toolboxes

Compiler

MATLAB® Compiler Version 4 takes M-files as input and generates redistributable, stand-alone applications or software components. These resulting applications and components are platform specific.

The MATLAB Compiler can generate the following kinds of applications or components. None of these requires MATLAB on the end-user's system:

- Stand-alone applications
- C and C++ shared libraries (dynamically linked libraries, or DLLs, on Microsoft Windows)
- Excel add-ins; requires MATLAB Builder for Excel
- COM objects; requires MATLAB Builder for COM

The MATLAB Compiler supports all the functionality of MATLAB, including objects. In addition, no special considerations are necessary for private and method functions; they are handled by the Compiler.

All the libraries for deployment on PCs are located in:

toolbox/compiler/deploy/win32/MCRInstaller.exe

MCC

Invoke MATLAB to C/C++ Compiler (Version 4.3).

Example:
```
mcc –m file
```

Vectorized examples

- How to replace elements of a vector, given certain criteria:

```
b = a; b(a>2.0) = 2
```

- How many positive elements in vector a?

```
b = length(find(a > 0))
b = sum(a > 0)
```

- Number of elements in a matrix:

```
prod(size(s))
```

- Terminate a while loop if any difference is 0:

```
while any(~(diff))
```

- Highlight a single data point in a figure:

```
plot(x,y,4,y(4), 'r*')
```

- Mental model for eval: evaluating functions:

```
function y = halfcircle(fh,n)
if nargin < 2
n = 20;
end
y = fh(0:pi/n:pi);
```

And you would call this function like this:

```
ys = halfcircle(@sin);
yc = halfcircle(@cos);
```

It is more direct however to call the function itself instead of passing it to feval:

```
y = sin(0:pi/3:3);
```

- Filtering data. Removing negative values from a microarray experiment:

```
cy3 = file(:,1);
cy5 = file(:,2);
positivevals = (cy3 > 0) and (cy5 > 0);
cy3(~=positivevals) = [];
cy5(~=positivevals) = [];
```

- Remove elements from a matrix:

```
a = rand(100,100)
c = a(find(a ~= 0 & a ~= 1))
```

Then the user can use the reshape function to create another matrix.

- Reverse a vector x:

```
x = x(end:-1:1)
```

- Shuffle an array x:

```
y = x(randperm(length(x)))
```

- Change the dimensions of a matrix:

```
a = rand(50,50)
a(:,:) = rand(20,20)
```

- Substract an specific value from each element of x which is greater than 3:

```
x(x > 3) = x(x > 3) - 3;
```

- Find the min or max of a matrix m in terms of the location in terms of the row (r) and column (c). It can handle multiple minimums or maximums (same values):

```
[r,c]=find(m==max(m(:)));
[r,c]=find(m==min(m(:)));
```

- Compute the moving average of a vector *y* with a window of size n:

```
filter(ones(1,n)/n,1,y(:))
```

- Create matrices of 1s and 0s with the same size than matrix m:

```
mones = ~m;
mzeros = ~~m;
```

- Monotic increasing (i.e. non-decreasing) over rows:

```
~any(diff(A,[],2)<0,2)
```

Selected problems and how to solve them

Problem Description:
How do I save or load my variables to or from files with similar names?

For example, within a FOR loop, I calculate values for my variables 'a', 'b', and 'c', ten times. I want to save these variables for each iteration to MAT-files with names test01.mat, test02.mat....test10.mat.

Solution:
You can use string concatenation to create the strings for your filenames:

```
a = 1;
b = 2;
c = 3;

filename = 'test';

for i = 1:10
a = a + 1;
b = b + 2;
c = c + 3;
str_counter = num2str(i);

if i < 10
str_counter = ['0' str_counter]; %create '01' - '09' strings for
'test01.mat' - 'test09.mat'
end

new_filename = [filename str_counter]; %concatenate 'test' to the test
number
save(new_filename,'a','b','c')
end
```

For more information on string concatenation, type *help horzcat* and *help strcat* at the MATLAB command prompt.

Problem Description:
I have a set of about 50 data files which I want import to MATLAB. Each file has several columns.

I have found that the command

```
load -ascii filename.txt
```

works rather well for my purpose on a single file. However, I want to load about 50 different files. I would like some direction on how best to load multilple files into separate arrays.

Solution:
You can use the LOAD command in a loop to load in all the files. If you place all of the files you would like to load into a single directory, you can use the DIR command to capture the file names. Here is an example:

```
files = dir('*.txt');
for i=1:length(files)
      eval(['load ' files(i).name ' -ascii']);
end
```

This code will load in all of the files in the directory (assuming they are .txt files) and save them as arrays with the file name as the array name.

Problem Description:
I need to use greek symbols in my plots. How can I insert them?

Solution:
The following table lists these characters and the character sequences used to define Tex commands.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	Υ	\circ	°
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∋	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	ℵ
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	⊘
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in	∈	\o	o
\rfloor	⌋	\lceil	⌈	\nabla	∇

\lfloor	⌊	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	′
\wedge	∧	\times	x	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	⟨	\rangle	⟩		

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use \fontname in combination with one of the other modifiers:

- \bf -- Bold font
- \it -- Italic font
- \sl -- Oblique font (rarely available)
- \rm -- Normal font
- \fontname{fontname} -- Specify the name of the font family to use.
- \fontsize{fontsize} -- Specify the font size in FontUnits.
- \color(colorSpec) -- Specify color for succeeding characters

Example:

```
legend('\alpha')
```

Subscripts or superscript?

X^2 or X_2 → `X^{2} or X_2`

For more in the topic check:

http://www.mathworks.com/access/helpdesk/help/techdoc/ref/text_props.html

Problem Description:
How can I process a sequence of files?

Solution:
If you can generate the filename using an incrementing counter, use code like this:

```
for k=1:20
  fname=sprintf('/path-name/m%d.dat',k);
  data=load(fname);
  % or
  data=imread(fname);
  % or
  fid=fopen(fname, 'rb');
  fread(fid, ...);
end
```

If instead you want to process all the files in a directory, you might instead wish to use dir:

```
d =dir('*.jpg');
for k=1:length(d)
  fname=d(k).name;
  % ...
end
```

Selected articles from www.mathworks.com

Dynamic function creation with anonymous and nested functions

When you are developing algorithms to solve technical computing problems, it is often useful to create functions on-the-fly so that you can customize them at run-time without having to define them in files beforehand. For example:

- You may want to create a function based on a user's input and then evaluate or plot it over a specific range.
- You may want to define a function and then pass it as an argument to an optimization, numerical integration, or ODE solving routine (also known as a *function function* in MATLAB).
- In curve fitting, you need to create a function that best fits some chosen data and subsequently need to evaluate it at other data points or perform further operations on it, such as integration.
- Sometimes it is necessary to create a function that needs to retain dynamic state and which must be initialized the first time it is called and updated at each subsequent call.

Other uses for dynamically created functions include callbacks in MATLAB Handle Graphics, data acquisition objects and timers, and, generally, any time you need to pass a function as a parameter to another routine.

Earlier versions of MATLAB already provided a number of features for managing functions, but there were limitations, for example:

- The MATLAB language has function handles that let you pass functions as parameters to routines, but they needed to be associated with a function definition in an M-file and evaluated with the feval command.
- You can specify arbitrary functions as a string with the inline command and pass them to the function function routines, but this command is difficult to use, particularly to separate the many different types of parameters, including the arguments to the inlined function and the function function itself.
- The keyword persistent provides state in functions, but you can have only one instance of a traditional MATLAB function. Also, the function must carry out initialization the first time it is called.

Two new language features in MATLAB 7, anonymous functions and nested functions, address these issues and requirements for creating and managing functions.

Anonymous and Nested Functions in MATLAB 7

Anonymous functions let you define a function on-the-fly at the command line or in M code, without an associated file. For example, to create a polynomial function,

$y(x) = 3x^2 + 2x + 1$

type

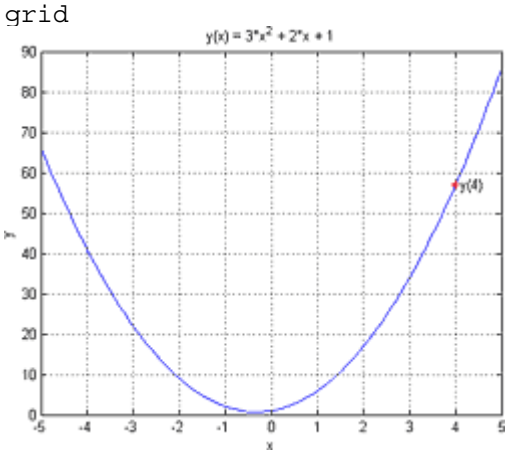
```
y = @(x) 3*x.^2 + 2*x + 1;
```

This returns function handle `y` representing a function of one variable defined by the expression on the right. The variables in parentheses following the @ symbol (in this case, only `x`) specify variables of which `y` is a function. The function `y` can now be evaluated by calling it like any other function (another new feature in MATLAB 7).

```
y(3) ans =
34
```

You could evaluate this polynomial over a range of points and plot the results together with the previously evaluated point.

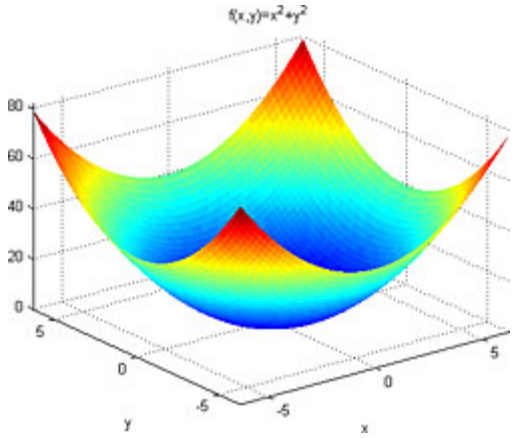
```
x=-5:.1:5;
plot(x,y(x),'-b',4,y(4),'r*');
title('y(x) = 3*x^2 + 2*x + 1');
xlabel('x');
ylabel('y');
text(4,y(4),'y(4)');
grid
```



Click on image to see enlarged view.

We can create handles to functions of multiple variables.

```
g=@(x,y) (x.^2 + y.^2);
ezsurf(g);
shading flat;
title('f(x,y)=x^2+y^2');
```



Click on image to see enlarged view.

Nested functions are functions in M-files that are nested inside other functions and which can see the parent function's workspace. The following function `taxDemo.m` contains a nested function.

```
function y = taxDemo(income)
% Calculate the tax on income.

AdjustedIncome = income - 6000; % Calculate adjusted income

% Call 'computeTax' without passing 'AdjustedIncome' as a parameter.

y = computeTax; function y = computeTax % This function can see the
variable 'AdjustedIncome'
% in the calling function's workspace
y = 0.28 * AdjustedIncome; end end
```

The nested function `computeTax` can see the variables in the parent function's workspace, in this case `AdjustedIncome` and `income`. This makes sharing of data between multiple nested functions easy. We can call the function in the usual way.

```
% What is the tax on income of 80,000?
tax=taxDemo(80e3) tax =
2.0720e+004
```

The ability of nested functions to see into their parent's workspace enables you to control the scope of your variables, letting them be accessed by multiple functions, while avoiding the side effects of us ing a global variable. At the same time, you can reduce your memory requirements by sharing large data sets in a controlled way.

An `end` statement is required at the end of all nested functions, making them different from traditional local subfunctions. However, all functions in a file containing nested functions, including traditional subfunctions, require termination with an `end` statement. Functions can be nested to any level.

Customizing Functions

You can create different variants or customized versions of the same function with anonymous functions or nested functions. For example, if we want to create a function like

$y = ax^2 + bx + c$

and customize `a`, `b`, or `c` at runtime, we write

```
a=3; b=2; c=-10;
y1=@(x) a*x.^2 + b*x + c;
```

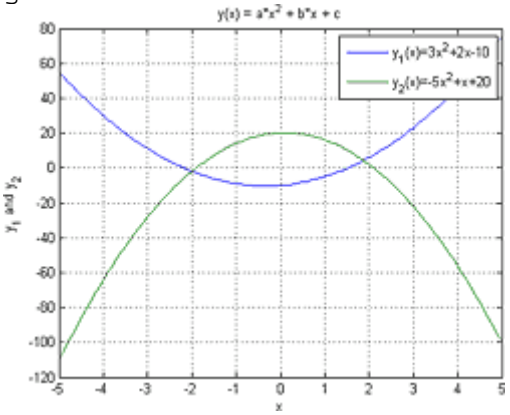
Any variables in the function that are not listed in parentheses are taken from the calling workspace at the time of definition.

Now let's change `a`, `b`, or `c` and generate other customized functions. For example,

```
a=-5; b=1; c=20;
y2=@(x) a*x.^2 + b*x + c;
```

Evaluate or plot these as before.

```
x=-5:.1:5;
plot(x, y1(x), x, y2(x));
legend('y_1(x)=3x^2+2x-10','y_2(x)=-5x^2+x+20');
xlabel('x');
ylabel('y_1 and y_2');
title('y(x) = a*x^2 + b*x + c');
grid
```



Click on image to see enlarged view.

Since the definitions of `y1` and `y2` included the values of `a`, `b`, and `c` at the time the function handles are created, they are robust to variations in the workspace such as changed or deleted values of `a`, `b`, or `c`.

```
a=100;
y1(3) ans =
23

a=5000;
```

```
y1(3) ans =
23
```

Let's see how we can carry out a similar task with nested functions by looking at `makefcn.m`, which contains a nested function.

```
function fcn = makefcn(a,b,c)
% This function returns a handle to a customized version of 'parabola'.
% a,b,c specifies the coefficients of the function.

fcn = @parabola; % Return handle to nested function function y =
parabola(x)
% This nested function can see the variables 'a','b', and 'c'
y = a*x.^2 + b.*x + c;
end
end
```

When you call `makefcn`, it returns a function handle to the internal nested function, which has been customized according to the parameters passed to the parent function. For example,

```
f1 = makefcn(3,2,10);
f2 = makefcn(0,5,25);
```

We can evaluate these two different functions as before.

```
f1(2) ans =
26

f2(2) ans =
35
```

In general, anonymous functions are better for quick, on-the-fly simple function creation and customization. Nested functions are more suited to more complex function customization and management.

Working with Function Functions

You can pass nested and anonymous functions to optimization or integration routines, known as function functions in MATLAB. For example, to find the area under the curve `f1` between 0 and 4, use

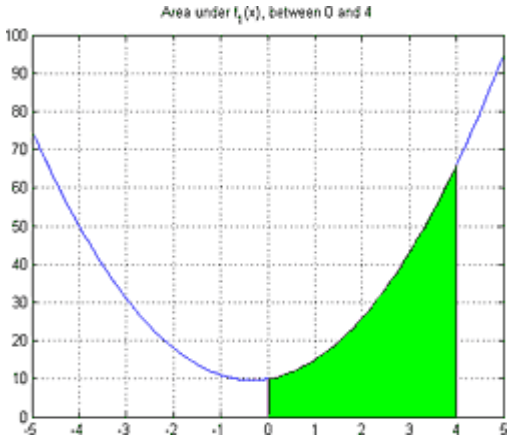
```
areaunder=quad(f1,0,4) areaunder =
120.0000
```

By having the integrand `f1` defined beforehand, you keep its arguments (such as the polynomial coefficients) separate from those of the integrator quad (such as the limits of integration). This can eliminate potential confusion.

We will plot `f1` again together with a plot of the area just calculated.

```
x=-5:.1:5;
plot(x,f1(x));
```

```
hold on
x=0:.1:4;
area(x,f1(x),'Facecolor','g'); % You can evaluate f without feval
hold off
title('Area under f_1(x), between 0 and 4');
grid
```



Click on image to see enlarged view.

Handling Dynamic Function State

If you need to create a function that retains dynamic state, you can use a nested function and store its state in the parent's workspace. Functions with dynamic state can represent algorithms that range from simple counters to components of a large system.

Simple Function State

Let's create a counter function that returns a number which increments each time it is called. Let's look at the function `makecounter.m`.

```
function countfcn = makecounter(initvalue)
% This function returns a handle to a customized nested function
'getCounter'.
% initvalue specifies the initial value of the counter whose handle is
% returned.

currentCount = initvalue; % Initial value
countfcn = @getCounter; % Return handle to getCounter

function count = getCounter
% This function increments the variable 'currentCount', when it
% gets called (using its function handle).
currentCount = currentCount + 1;
count = currentCount; end end
```

When you call `makecounter`, it returns a handle to its nested function `getCounter`. `getCounter` is customized by the value of `initvalue`, a variable it can see via nesting within the workspace of `makecounter`. Let's make a couple of counter functions.

```
counter1 = makecounter(0); % Define counter initialized to 0
counter2 = makecounter(10); % Define counter initialized to 10
```

Here we have created two customized counters: one that starts at 0 and one that starts at 10. Each handle is a separate instance of the nested function and its calling workspace. Note `counter1` does not take any parameters. We need to use the parentheses to invoke the function instead of looking at the function handle variable itself.

```
counter1Value=counter1() counter1Value =
1
```

We can call the two functions independently as there are two separate workspaces kept for the parent functions. They remain in memory while the handles to their nested functions exist. Each time the counter functions are called, the associated variable `currentCount` is incremented.

```
counter1Value=counter1() counter1Value =
2

counter2Value=counter2() counter2Value =
11
```

Complex Function State

You can use nested functions to store a more complex dynamic state such as that of a filter. Let's look at the function `makeFilter.m`.

```
function filterhandle = makeFilter(b,a)

% Initialize State
state=zeros(max(length(a),length(b))-1,1);

% Return handle to filter function
filterhandle=@callFilter;
function output = callFilter(input)
% Calculate output and update state
[output,state] = filter(b,a,input,state); end end
```

The function `makeFilter` returns a handle to a nested function (`callFilter`) that performs a filtering operation and maintains its state in the parent's workspace. The filter state is initialized when the function handle is created, then gets updated each time the nested function is called. The nested function also has to calculate the output of the filtering operation and return it as its output argument. You can call this function in a for-loop many times, as shown in `simplesystem.m`.

```
%% System Parameters
frameSize = 1000;
sampleTime = 1e-3;

%% Initialize System Components and Component Parameters
filter1 = makeFilter([0.02,0,-0.23,0,0.49,0,-0.23,0,0.02],1);
filter2 = makeFilter([-0.05 0.13 0.83 0.13 -0.05],1);
randSource = makeRandSource(frameSize);
scope = makeScope(sampleTime,[-2 2],'My Scope');
```

```
%% Simulation Loop
for k = 1:100 signal1 = randSource(); signal2 = filter1(signal1);
signal3 = filter2(signal2); scope(signal3) end
```

The main for-loop in the simulation now becomes very clean and simple where the only variables passed between the functions (`signal1`, `signal2`, and `signal3`) are pure data, without being mixed up with other function parameters (such as filter coefficients) and state.

The `makeFilter` routine could be expanded to return more than one function, such as one to return the filter state, or even one to reset the filter. These function handles could be returned as multiple output arguments or as a structure, where each field is a function handle. The functions can then be called in a manner similar to evaluating the methods of an object.

Other Uses of Nested and Anonymous Functions

The function handling capabilities of nested and anonymous functions have many other uses.

Function Composition

You can create handles to functions of functions, allowing a very natural mathematical notation for function composition, such as

```
f=@(x) x.^2;
g=@(x) 3*x;
h=@(x) g(f(x)); % Equivalent to 3*(x^2)
h(3) ans =
27
```

Memoization

We can store the previously computed return values of a function in a "table" for later use instead of recalculating values. This is helpful if the function is computationally intensive. Let's look at a function that stores computed values and makes them available for reuse.

```
function f = memoize(F)
% One argument F, inputs testable with ==
% Scaler input to F

x = [];
y = [];
f = @inner; function out = inner(in) ind = find(in == x);
if isempty(ind) out = F(in);
x(end+1) = in;
y(end+1) = out; else out = y(ind); end end end
```

Here's how to use this function. First you create a "memoized" version of the function for which you want to remember the return values, for example, `sin`.

```
f = memoize(@sin) f =
@memoize/inner
```

Let's call the function a few times.

```
f(pi/2) ans =
1

f(pi/4) ans =
0.7071

f(pi/8) ans =
0.3827
```

The returned values, in this case `sin(pi/2)`, `sin(pi/4)`, and `sin(pi/8)` are stored. To see how this is working, let's use the `functions` command to inspect the state of the function handle `f`.

```
functionInfo = functions(f) functionInfo =
function:   'memoize/inner'
ype:        'nested'
file:       [1x76 char]
workspace:{[1x1 struct]}

functionInfo.workspace{1} ans =
f: @memoize/inner
F: @sin
x: [1.5708 0.7854 0.3927]
y: [1 0.7071 0.3827]
```

Now if you request a previously computed result, such as for `sin(pi/4)`, the value is taken from the table without a new value being added. Here you can see the stored workspace doesn't change.

```
f(pi/4) ans =
0.7071

functionInfo = functions(f);
functionInfo.workspace{1} ans =
f: @memoize/inner
F: @sin
x: [1.5708 0.7854 0.3927]
y: [1 0.7071 0.3827]
```

Data Structures

Nested functions can be used to create data structures such as lists and trees. Find Sturla Molden's example in the comp.soft-sys.matlab Usenet newsgroup.

The new dynamic function creation and handling features of nested and anonymous functions have made functions fully-fledged members of the MATLAB language and opened up a whole new set of programming patterns. For more information on functions and function handles in MATLAB, see the following resources.

Code Vectorization Guide

This technical note provides an introduction to vectorization techniques. In order to understand some of the possible techniques, an introduction to MATLAB referencing is provided. Then several vectorization examples are discussed.

This technical note examines how to identify situations where vectorized techniques would yield a quicker or cleaner algorithm. Vectorization is often a smooth process; however, in many application-specific cases, it can be difficult to construct a vectorized routine. Understanding the tools and examples presented in this technical note is a good introduction to the topic, but it is by no means an exhaustive resource of all the vectorization techniques available.

Tools and Techniques

Extended Examples

More Resources

Tools and Techniques

Section 1: MATLAB Indexing or Referencing

MATLAB enables you to select subsets of an array or matrix. There are three basic types of indexing available in MATLAB: subscripted, linear, and logical. These methods are explained in the MATLAB Digest article, Matrix Indexing in MATLAB. While this article goes into detail on MATLAB indexing,

the remainder of this section is a good introduction. If you are not familiar with MATLAB indexing, you should read the MATLAB Digest article, and/or the documentation on colon, paren (round(), square[], and curly{} braces) and end.

Subscripted Indexing

In subscripted indexing, the values of the subscripts are the indices of the matrix where the matrix's elements are desired. That is, elements = matrix(subscripts). Thus, if `A = 6:10`, then `A([3,5])` denotes the third and fifth elements of matrix A:

```
A = 6:10;


A([3,5])
```

```
ans=
8 10
```

For multidimensional arrays, multiple index parameters are used for subscripted indexing:

```
A = [11 14 17; ...
     12 15 18; ...
     13 16 19];


A(2:3,2)
```

```
ans=
15
16
```

Linear Indexing

In linear, or absolute indexing, every element of a matrix can be referenced by its subscripted indices (as described above), or by its single, columnwise, linear index:

```
A = [11 14 17; ...
     12 15 18; ...
     13 16 19];
```

```
A(6)
```

```
ans=
16
```

```
A([3,1,8])
```

```
ans=
13 11 18
```

See the functions SUB2IND and IND2SUB for more information.

Also, notice how the returned elements of the indexed matrix preserve the shape that was specified by the index variable. In the previous example, the index variable was a vector of size 1-by-3, and so the result was also of size 1-by-3. If an index variable of size 3-by-1 was used, a preserved shape would have been seen in the results:

```
A([3;1;8])
```

```
ans=
13
11
18
```

Logical Indexing

With logical, or Boolean, indexing, the index parameter is a logical matrix that is the same size as A and contains only 0's and 1's.

The elements of A that are selected have a '1' in the corresponding position of the logical indexing matrix. For example, if `A = 6:10`, then `A(logical([0 0 1 0 1]))` denotes the third and fifth elements of A:

```
A = 6:10;


A(logical([0 0 1 0 1]))
```

```
ans=
8 10
```

For more information on matrix indexing and matrix manipulation, see the resources listed under Section 10 of this technical note.

Section 2: Array Operations vs. Matrix Operations

`y(i) = fcn(x1(i), x2(i), ...)`

The simplest type of vector operations, array operations, can be thought of as bulk processing. In this approach, the same operation is performed for each corresponding element in a data set, which may include more than one matrix. The operation is performed on an element-by-element basis.

Matrix operations, or linear algebra operations, operate according to the rules of linear algebra. This type of operation is sometimes useful for vectorization as illustrated in Section 6 and Section 8 of this technical note.

For example, you may have data measurements from an experiment that measures the volume of a cone, (`V = 1/12 * pi * D^2 * H`), by recording the diameter (`D`) of the end, and the height (`H`). If you had run the experiment once, you would just have one value for each of the two observables; however, `D` and `H` are scalars. Here is the calculation you will need to run in MATLAB:

```
V = 1/12*pi*(D^2)*H;
```

Now, suppose that you actually run the experiment 100 times. Now `D` and `H` are vectors of length 100, and you want to calculate the corresponding vector `V`, which represents the volume for each run.

In most programming languages, you would set up a loop, the equivalent of the following MATLAB code:

```
for n = 1:100
    V(n) = 1/12*pi*(D(n)^2)*H(n);
end
```

With MATLAB, you can ignore the fact that you ran 100 experiments. You can perform the calculation element-by-element over each vector, with (almost) the same syntax as the first equation, above.

```
% Provide data so that the code can be executed
% Use of negative values will be apparent later
```

```
D = [-0.2 1.0 1.5 3.0 -1.0 4.2 3.1];

H = [ 2.1 2.4 1.8 2.6 2.6 2.2 1.8];



% Perform the vectorized calculation

V = 1/12*pi*(D.^2).*H;
```

The only difference is the use of the `.*` and `./` operators. These differentiate array operators (element-by-element operators) from the matrix operators (linear algebra operators), `*` and `/`.

For more information, refer to the Matrices and Linear Algebra section of the MATLAB Mathematics documentation.

Section 3: Boolean Array Operations

`y = bool(x1, x2, ...)`

A logical extension of the bulk processing technique is to vectorize comparisons and decision making. MATLAB comparison operators, like the array operators above, accept vector inputs and produce vector outputs.

Suppose that after running the above experiment, you find that negative numbers have been measured for the diameter. As these values are clearly erroneous, you decide that those runs for the experiment are invalid.

You can find out which runs are valid using the >= operator on the vector `D`:

```
D = [-0.2 1.0 1.5 3.0 -1.0 4.2 3.14];

D >= 0
```

```
ans=

0 1 1 1 0 1 1
```

Now you can exploit the logical indexing power of MATLAB to remove the erroneous values:

```
Vgood = V(D>=0);
```

This selects the subset of V for which the corresponding elements of D are nonnegative.

Suppose that if all values for masses are negative, you want to display a warning message and exit the routine. You need a way to condense the vector of Boolean values into a single value. There are two vectorized Boolean operators, ANY and ALL, which perform Boolean AND and OR functions over a vector. Thus you can perform the following test:

```
if all(D < 0)
   warning('All values of diameter are negative.');
   return;
end
```

You can also compare two vectors of the same size using the Boolean operators, resulting in expressions such as:

```
(V >= 0) & (D > H)
```

```
ans=
0 0 0 1 0 1 1
```

The result is a vector of the same size as the inputs.

Additionally, since MATLAB uses IEEE arithmetic, there are special values to denote overflow, underflow, and undefined operations: INF , -INF , and NaN , respectively.INF and -INF are supported by relational operators (i.e., Inf==Inf returns true, and Inf<1 returns false). However, by the IEEE standard, NaN is never equal to anything, even itself (NaN==NaN returns false). Therefore, there are special Boolean operators, ISINF and ISNAN, to perform logical tests for these values. For example, in some applications, it's useful to exclude NaNs in order to make valid computations:

```
x = [2 -1 0 3 NaN 2 NaN 11 4 Inf];


xvalid = x(~isnan(x))
```

```
xvalid =
2 -1 0 3 2 11 4 Inf
```

Also, if you try to compare two matrices of different sizes, an error will occur. You need to check sizes explicitly if the operands might not be the same size. See the documentation on SIZE, ISEQUAL, and ISEQUALWITHEQUALNANS(R13 and later) for more information.

Section 4: Constructing Matrices from Vectors

`y(:,i) = fcn(x(:))`

Creating simple matrices, such as constant matrices, is easy in MATLAB. The following code creates a size 5-by-5 matrix of all 10s:

```
A = ones(5,5)*10
```

The multiplication here is not necessary; you can achieve the same result using the indexing technique described in the next example.

Another common application involves selecting specified elements of a vector to create a new matrix. This is often a simple application of the indexing power of MATLAB. The next example creates a matrix corresponding to elements `[2, 3; 5, 6; 8, 9; 11, 12; ...]` of a vector `x`:

```
% Create a vector from 1 to 51

x = 1:51;


% Reshape it such that it wraps through 3 rows

x = reshape(x, 3, length(x)/3);


% Select the 2nd and 3rd rows, and transpose it

A = x(2:3,:)';
```

In this example, indexing is used to exploit the pattern of the desired matrix. There are several tools available to exploit different types of patterns and symmetry. Relevant functions are summarized in Section 5 of this technical note.

There is a well-known technique for duplicating a vector of size `M-by-1` `n` times, to create a matrix of size`M-by-N`. In this method, known as Tony's Trick, the first column of the vector is indexed (or referenced) `n` times:

```
v = (1:5)' n = 3;


M = v(:,ones(n,1))
```

```
M =
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
```

Now it is apparent how the first matrix, size 5-by-5 constant matrix of all 10's, can be created without the array multiplication operation:

```
A = 10;



A = A(ones(5,5))
```

```
A =
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
```

The same technique can be applied to row vectors by switching the subscripts. It can also duplicate specific rows or columns of a matrix. For example,

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

```
B =
8 1 6
3 5 7
4 9 2
```

```
N = 2; B(:,N(ones(1,3)))
```

```
ans=
1 1 1
5 5 5
9 9 9
```

While it is good to understand how this technique works, usually you do not need to use it explicitly. Instead, you can use the functions REPMAT and MESHGRID, which implement this technique. Refer to the documentation and Section 6 for more information and examples.

Section 5: Utility Functions

The following table lists functions that are useful for creating patterned matrices and symmetric matrices. Some have already been used in this technical note. Clicking on a function will take you to the documentation for more information.

CUMPROD	Provide a cumulative product of elements
CUMSUM	Provide a cumulative sum of elements
DIAG	Create diagonal matrices and diagonals of a matrix
EYE	Produce an identity matrix
FILTER	Create one-dimensional digital filter (good for vectors where each element depends on the previous element)
GALLERY	Provides a gallery of different types of matrices
HANKEL	Create a Hankel matrix
MESHGRID	Create X and Y arrays for 3-D plots
NDGRID	Generate of arrays for N-D functions and interpolation. (Note: uses different coordinate system than MESHGRID)
ONES	Create an array of ones
PASCAL	Create a Pascal matrix
REPMAT	Replicate and tile an array
RESHAPE	Change size of array
SPDIAGS	Create a sparse matrix formed from diagonals
TOEPLITZ	Create a Toeplitz matrix
ZEROs	Create an array of zeros

Extended Examples

Section 6: Matrix Functions of Two Vectors

```
y(i,j) = fcn(x1(i), x2(j))
```

Suppose you want to evaluate a function F of two variables:

$$F(x,y) = x*\exp(-x^2 - y^2)$$

You need to evaluate the function at every point in vector x, and for each point in x, at every point in vector y. In other words, you need to define a grid of values for F, given vectors x and y.

You can duplicate x and y to create an output vector of the desired size using MESHGRID. This allows you to use the techniques from Section 2 to compute the function.

```
x = (-2:.2:2);

y = (-1.5:.2:1.5)';

[X,Y] = meshgrid(x, y);

F = X .* exp(-X.^2 - Y.^2);
```

In some cases, you can use the matrix multiplication operator in order to avoid creating the intermediate matrices. For example, if

```
F(x,y) = x*y
```

where x and y are vectors, then you can simply take the outer product of x and y:

```
x = (-2:2);

y = (-1.5:.5:1.5);


x'*y
```

```
     3.0000    2.0000    1.0000         0   -1.0000   -2.0000   -3.0000
     1.5000    1.0000    0.5000         0   -0.5000   -1.0000   -1.5000
          0         0         0         0         0         0         0
    -1.5000   -1.0000   -0.5000         0    0.5000    1.0000    1.5000
    -3.0000   -2.0000   -1.0000         0    1.0000    2.0000    3.0000
```

When creating matrices from two vectors, there are also cases where sparse matrices make use of more efficient use of storage space, and also make use of very efficient algorithms. An example is discussed further in Section 8.

Section 7: Ordering, Setting, and Counting Operations

In the examples discussed so far, any calculations done on one element of a vector have been independent of other elements in the same vector. However, in many applications, the calculation that you are trying to do depends heavily on these other values. For example, suppose you are

working with a vector x which represents a set. You do not know without looking at the rest of the vector whether a particular element is redundant and should be removed. It is not obvious at first how to remove these redundant values without resorting to loops.

This area of vectorization requires a fair amount of ingenuity. There are many functions available that you can use to vectorize code:

DIFF	Acts as difference operator: diff(X), for a vector X, is: `[X(2) - X(1), X(3) - X(2), ... X(n) - X(n-1)]`
FIND	Finds indices of the nonzero, non-NaN elements
INTERSECT	Finds the set intersection
MAX	Find largest component
MIN	Find smallest component
SETDIFF	Finds the set difference
SETXOR	Finds the set exclusive OR
SORT	Sort in ascending order
UNION	Finds the set union
UNIQUE	Find unique elements of a set

Now, proceed with this example, eliminating redundant elements of a vector. Note that once a vector is sorted, any redundant elements are adjacent. In addition, any equal adjacent elements in a vector create a zero entry in the DIFF of that vector. This suggests the following implementation for the operation. You are attempting to select the elements of the sorted vector that correspond to nonzero differences.

```
% First try. NOT QUITE RIGHT!!

x = sort(x(:));

difference = diff(x);

y = x(difference~=0);
```

This is almost correct, but you have forgotten to take into account the fact that the DIFF function returns a vector that has one fewer element than the input vector. In your first algorithm, the last unique element is not accounted for. You can fix this by adding one element to the vector x before taking the difference. You need to make sure that the added element is always different than the previous element. One way to do this is to add a NaN.

```
% Final version.
```

```
x = sort(x(:));

difference = diff([x;NaN]);

y = x(difference~=0);
```

The (:) operation was used to ensure that x is a vector. The ~=0 operation was used rather than the FIND function because the FIND function does not return indices for NaN elements, and the last element of difference, as defined, is a NaN. Alternatively, this example can be accomplished with the code

```
y=unique(x);
```

but the point of the above exercise is to exploit vectorization and demonstrate writing your own code for efficiency. Additionally, the UNIQUE function provides more functionality than necessary, and this can slow down the execution of the code.

Suppose that you do not just want to return the set x. You want to know how many instances of each element in the set occurred in the original matrix. Once you have the sorted x, you can use teh FIND function to determine the indices where the distribution changes. The difference between subsequent indices will indicate the number of occurrences for that element. This is the "diff of find of diff" trick. Building on the above example:

```
% Find the redundancy in a vector x

x = sort(x(:));

difference = diff([x;max(x)+1]);

count = diff(find([1;difference]));

y = x(find(difference));

plot(y,count)
```

This plots the number of occurrences of each element of x. Note that we avoided using NaN here, because FIND does not return indices for NaN elements. However, the number of occurrences of NaNs and Infs are easily computed as special cases:

```
count_nans = sum(isnan(x(:)));

count_infs = sum(isinf(x(:)));
```

Another trick for vectorizing summing and counting operations is to exploit the way sparse matrices are created. This is discussed in greater detail in an example in Section 9.

Section 8: Sparse Matrix Structures

You can use sparse matrices to increase efficiency in some cases. Often vectorization is easier if you construct a large intermediate matrix. In some cases, you can take advantage of a sparse matrix structure to vectorize code without requiring large amounts of storage space for this intermediate matrix.

Suppose that in the last example, you knew beforehand that the domain of the set `y` is a subset of the integers, `{k+1, k+2, ..., k+n}`; that is,

```
y = 1:n + k
```

These might represent indices in a colormap. You can then count the occurrences of each element of the set. This is an alternative method to the "diff of find of diff" trick from the last Section.

Construct a large `m`-by-`n` matrix A, where `m` is the number of elements in the original vector `x`, and `n` is the number of elements in the set `y`.

```
if x(i) = y(j)
   A(i,j) = 1
else A(i,j) = 0
```

Looking back at Sections 3 and 4, you might suspect that you need to construct matrices from `x` and `y`. This would work, but it would require a lot of storage space. You can do better by exploiting the fact that most of the matrix A consists of 0's, with only one `1` value per element of `x`.

Here is how to construct the matrix (note that `y(j) = k+j`, from the above formula):

```
x = sort(x(:));
A = sparse(1:length(x), x+k, 1, length(x), n);
```

Now you can perform a sum over the columns of A to get the number of occurrences.

```
count = sum(A);
```

In this case you do not have to form the sorted vector y explicitly, since you know beforehand that `y = 1:n + k`.

The key here is to use the data (i.e., x) to control the structure of the matrix A. Since x takes on integer values in a known range, you are able to construct the matrix more efficiently.

Suppose you want to multiply each column of a very large matrix by the same vector. There is a way to do this using sparse matrices that saves space and can also be faster than matrix construction using the tools outlined in Section 5. Here's how it works:

```
F = rand(1024,1024);

x = rand(1024,1);   % Point-wise multiplication of each row of F .

Y = F * diag(sparse(x)); % Point-wise multiplication of each column of F.

Y = diag(sparse(x)) * F;
```

The matrix multiplication operator handles the bulk of the work, while sparse matrices prevent the temporary variable from being too large.

You can also use sparse matrices for counting and binning operations. This involves manipulating the way sparse matrices are created. This is discussed in the next section of examples.

Section 9: Additional Examples

The following examples use several techniques discussed in this technical note, as well as a few other relevant techniques. Try using tic and toc (or t=cputime and cputime-t) around the examples to see how they speed up the code.

Vectorizing a double FOR loop that creates a matrix by computation:

Tools:

array multiplication

Before:

```
A = magic(100);

B = pascal(100);

for j = 1:100
```

```
   for k = 1:100;
     X(j,k) = sqrt(A(j,k)) * (B(j,k) - 1);
   end
end
```

After:

```
A = magic(100);
B = pascal(100);
X = sqrt(A).*(B-1);
```

Vectorizing a loop that creates a vector whose elements depend on the previous element:

Tools:

FILTER, CUMSUM, CUMPROD

Before:

```
A = 1;
L = 1000;
for i = 1:L
  A(i+1) = 2*A(i)+1;
end
```

After:

```
L = 1000;
A = filter([1],[1 -2],ones(1,L+1));
```

In addition, if your vector construction only uses addition or multiplication, you can use the CUMSUM or CUMPROD function.

Before:

```
n=10000;
```

```
V_B=100*ones(1,n);

V_B2=100*ones(1,n);

ScaleFactor=rand(1,n-1);

for i = 2:n

  V_B(i) = V_B(i-1)*(1+ScaleFactor(i-1));

end

for i=2:n

  V_B2(i) = V_B2(i-1)+3;

end
```

After:

```
n=10000;

V_A=100*ones(1,n);

V_A2 = 100*ones(1,n);

ScaleFactor=rand(1,n-1);

V_A=cumprod([100 1+ScaleFactor]);

V_A2=cumsum([100 3*ones(1,n-1)]);
```

Note that if you run the two examples one after another, V_B and V_A will differ. This is because you recreate it before computing V_A. If you use the same ScaleFactor matrix for both examples, V_B and V_A will agree.

Vectorizing code that finds the cumulative sum of a vector at every fifth element:

Tools:

CUMSUM, vector indexing

Before:

```
% Use an arbitrary vector, x

x = 1:10000;
```

```
y = [];
for n = 5:5:length(x)
  y = [y sum(x(1:n))];
end
```

After, using preallocation:

```
x = 1:10000;
ylength = (length(x) - mod(length(x),5))/5;
% Avoid using ZEROS function during preallocation
y(1:ylength) = 0;
for n = 5:5:length(x)
  y(n/5) = sum(x(1:n));
end
```

After, using vectorization (preallocation is no longer needed):

```
x = 1:10000;
cums = cumsum(x);
y = cums(5:5:length(x));
```

Vectorizing code that repeats a vector value when the following value is zero:

Tools:

FIND, CUMSUM, DIFF

Ideally, you want to replace the zeros within a vector with values and zeros by repeating the previous value. For example, convert this:

```
a=2;
b=3;
c=5;
d=15;
```

```
e=11;

x = [a 0 0 0 b 0 0 c 0 0 0 0 d 0 e 0 0 0 0 0];
```

into this:

```
x = [a a a a b b b c c c c c d d e e e e e e];
```

Solution:

```
valind = find(x);

x(valind(2:end)) = diff(x(valind));

x = cumsum(x);
```

Vectorizing code that accumulates a sum at designated indices:

Tools:

SPARSE

Before:

```
% The values are summed at designated indices

values = [20 15 45 50 75 10 15 15 35 40 10];

% The indices associated with the values are summed cumulatively

indices = [2 4 4 1 3 4 2 1 3 3 1];

totals = zeros(max(indices),1);

for n = 1:length(indices)

  totals(indices(n)) = totals(indices(n)) + values(n);

end
```

After:

```
% The values are summed at designated indices

values = [20 15 45 50 75 10 15 15 35 40 10];

% The indices associated with the values are summed cumulatively
```

```
indices = [2 4 4 1 3 4 2 1 3 3 1];

totals = full(sparse(indices,1,values));
```

This method exploits the way sparse matrices are created. When using the SPARSE function to create a sparse matrix, any values that are assigned to the same index are summed rather than replacing the existing value. This is called "vector accumulation," and is the way that MATLAB handles sparse matrices.

More Resources

Section 10: Matrix Indexing and Manipulation

The MATLAB Digest article Matrix Indexing in MATLAB provides much more detail on indexing than Section 1 of this technical note.

The Getting Started documentation link will guide you through matrix manipulation in MATLAB. It includes matrix creation, indexing, manipulation, array operation, matrix operations, as well as other topics. (*Click on the **Manipulating Matrices** link)

Peter Acklam has built a Web page devoted to MATLAB array manipulation tips and tricks.

Section 11: Matrix Memory Preallocation

For more information on preallocation to speed up computations, refer to the following online resolution:

26623: How do I preallocate memory when using MATLAB?

The following technical note is a comprehensive guide for memory management in MATLAB:

The Technical Support Guide to Memory Management

Section 12: Maximizing MATLAB Performance

While this technical note generally discusses how to vectorize code, often times the motivation is to speed up the performance. Therefore, this section provides some additional resources for maximizing the performance of your code:

General tips on speeding up code:

How do I increase the speed or performance of MATLAB?

Conclusion

There are conventional methods for vectorizing code and reducing run-time as introduced throughout this technical note, but there are also techniques (such as the last example in Section 9) that exploit the functionality of the SPARSE function. The above discussion is by no means an

exhaustive list of all the useful techniques available in MATLAB. It is meant to be an introduction to ideas and theories that can be applied to vectorizing code.