Lira's Matlab Quick-Reference Fall 2012

This is a summary of commands that I find useful to keep handy. If you find something in error, please let me know.

; The semicolon at the end of a line suppresses the echo of the command result to the command window. It is optional.

Strings are specified with single quotes ' ', NOT double quotes " ".

(), [], {} have special meanings. Do not interchange them or you may get unexpected results. () are used for indicating arithmetic precedence. [] are used to indicate vectors and matrices. {} are used for cell arrays.

By default all math in Matlab is double precision.

Default number format is rather ugly. Make it prettier by starting each session with '`format short g`'. The 'g' stands for general format. Use '`help format`' for more info.

File naming: Avoid spaces in file names or periods. Underbars and hyphens are OK. Spaces or periods create cryptic error messages.

Keeping the desktop clean

`clear;` % deletes workspace variables. Include variable name(s) to delete selectively.
`clc;` % clear the command window.
The command history is generally helpful to keep. If you wish to clear it, use Edit>Clear Command History.

Creating vectors

Row Vectors
`X = [a b c];` (commas may be inserted as [a, b, c])
X = linspace(firstElement, lastElement, numElements);
numElements defaults to 100 if omitted,
`linspace(0,1)` results in [0 0.01 0.02 .... 0.99 1.0]
X = firstElement:increment:lastElementLimit;
increment defaults to 1 if omitted,
e.g. `x = 1:5` results in [1 2 3 4 5], 0:2:5 results in [0 2 4]

```
zerosRowVector = zeros(1,numElements);
onesRowVector = ones(1,numElements);
```

Column Vectors (end of row indicated by ;)
`xColumn = [a; b; c];` or
`x = [ a b c]; xColumn = x';` (where `x'` executes the transpose)

```
zerosColumnVector = zeros(numElements,1);
onesColumnVector = ones(numElements,1);
```

Creating Arrays

`Array = [1 2 3; 10 20 30];` results in two rows and three columns, where ';' denotes end of row, commas optional for the columns, e.g. [1, 2, 3; 10, 20, 30];

```
zerosArray = zeros(numColumns, numRows);
onesArray = ones(numColumns, numRows);
```

Accessing Vector Elements

(indexing starts with 1)

`x(1), x(3), x(end).  x([i j])` retrieves elements i and j. `x(i:j)` retrieves elements i through j.

`x(:)` - the colon alone is like a wildcard that retrieves all columns, and is used frequently with arrays.

Accessing Array Elements

general notation is array(i,j) where i is the row, j the column.

`array(i,j)` – retrieves element from row i, column j.

`array(i, [j k])` – retrieves two elements from array - row i, columns j and k.

`array([i j k], m)` – retrieves three elements from array, rows i, j, k, column m.

`array(i,:)` – here the wildcard is used to retrieve all columns in row i.

`array(:,j)` – here the wildcard is used to retrieve all rows in column j.

`array(2:7, j)` – here the wildcard is used to retrieve rows 2 through 7 of column j.

Vector Operations

`x + y` results in element by element [x(1)+y(1), x(2)+y(2), ...x(end)+y(end)]

`x + 3` results in element by element [x(1)+3, x(2)+3, ...x(end)+3], minus may be used also.

`x - y` results in element by element [x(1)-y(1), x(2)-y(2), ...x(end)-y(end)]

`x.*y` results in [x(1)*y(1), x(2)*y(2), ...x(end)*y(end)]

`x./y` results in [x(1)/y(1), x(2)/y(2), ...x(end)/y(end)]

`x.^2` results in [x(1)^2, x(2)^2, ...x(end)^2]

`dot(x,y)` results in the scalar x(1)*y(1)+x(2)*y(2)+....x(end)*y(end)

Example: Suppose we wish to evaluate 4*π*g(r)*r^2, where r = rdf(:,1) and g(r) = rdf(:,2). The formula to assign this to create column vector 'funct' with these values would be

```
funct = 4*pi*rdf(:,2).*( rdf(:,1).^2 )
```

where `pi` is a reserved name within Matlab.

Importing from a text file

1. Open the file in notepad.
2. In Notepad: drag the mouse over the rows that you wish to import. All columns will be selected.
3. In Notepad: Edit>Copy or use Ctrl-C.
4. In Matlab: Edit>Paste into workspace.
5. Use the array preview to specify the correct delimiter and the number of header rows in the dialog box. The variable will be assigned the name A_pastespecial.
6. After accepting the settings, look for the variable A_pastespecial in the workspace. Right-click and rename the variable to something meaningful.
7. You may 'open' the new variable in the array editor to verify the import process.

Importing Excel data

Note there is an Edit menu command specifically for pasting data copied to the windows clipboard from Excel. Generally the pasting method will be easier then importing from the file. Copy the Excels to the clipboard, then in Matlab use Edit > Paste to workspace.

Pasting into Matlab or importing with the wizard usually separates text and numerical data. To preserve the text and numbers in a cell array, first create and empty cell array and then paste into the cell array with the following procedure. (1) Create a new cell array variable, e.g. Mydata

= {}; (2) Open the array in the array editor; (3) Copy the excel data onto the clipboard; (4) Put the cursor in cell (1,1) and either: (a) right-click and select 'Paste Excel Data'; or (b) use 'Edit>Paste Excel Data.' Data can be accessed using rules for a cell array. Text values from the array will be strings. Numerical data and formulas become double precision scalars. Empty cells become empty numerical vectors. Values are obtained using cell array syntax, e.g. T = A{3,2} and T will be the same type as stored in A{3,2}. Using T = A(3,2) makes T into a cell array, and is usually messier.

Quick Numerical Integration of Functions

If an analytical expression is known as a function of the integration variable, a simple routine to use the the 'quad' function. Example: integrate ln((5-x)/x) from 1 to 2. A simple application of that might be the following:

```
h = @(x) log((5-x)./x) % must use dot operators, h is an anonymous function
quad(h,1,2) % note how limits are inserted.
```

Quick Numerical Integration of Vector Data

Suppose the integral to evaluate is

$$\int_{r(1)}^{r(j)} 4\pi g(r) r^2 dr$$

Suppose by preliminary calculations we have loaded the independent variable `r = rdf(:,1)` and integrand into `funct(:,1)` with the same number of rows.
Matlab offers trapezoidal integration of vector data using the function *cumtrapz*(*indepentVar,integrand*). So for the function at hand the formula would be

```
cumtrapz( rdf(1:j, 1), funct(1:j) )
```

where j is the upper limit of integration. The function `cumtrapz(intVarArray(1:j), functionArray(1:j))` returns a vector of length j, where the elements i are the cumulative integral between the lower integration limit element intVarArray(1), and the element intVarArray(i). The final element is the entire integral value.

Special note about integrating g(r)*u(r) where u(r) is the square well function.

The function funct(r)*u(r) has a value of zero when funct(r) is zero, or when u(r) is zero. The function has a value of $-\varepsilon$*funct(r) for $\sigma < r \le R\sigma$. Therefore the integral of funct(r)*u(r) can be determined by integrating funct(r) between 0 and $R\sigma$ and then multiplying the result by $-\varepsilon$.

Symbolic Algebra

```
% as typed into the command window…
% define the variable as a symbolic variable
>> syms xf
>> LHS = (5500*xf/0.1 + 0.1E6)*1/1E6*(.1E6*xf)/2.463/8.314
LHS =(50000000000*xf*((11*xf)/200 + 1/10))/10238691
% collect terms
>> LHS = collect(LHS)
LHS = (2750000000*xf^2)/10238691 + (5000000000*xf)/10238691
% set digits to 8,convert coefficients to variable precision arithmetic
>> digits(8); LHS = vpa(LHS)
LHS = 268.58902*xf^2 + 488.34368*xf
% solve for xf value
>> solve(LHS)
ans =
 -1.8181818
          0
```

```
>> subs('3*xf',ans(1)) % substitute a value for a symbolic variable.
ans = -5.4545455
```

Symbolic Integration

```
% provide the integral to evaluate
>>syms x ; int(13.43*x + 32.43*sqrt(x)) %integate with respect to x
ans = 1343/200*x^2+1081/50*x^(3/2)

% the constant of integration is not shown and also must be
% evaluated for finite integrals. See int help for finite integrals.

% Another example
% define the variables to be used.
>>syms g a t b
>>g = cos(a*t + b)
>>int(g,t) % second variable indicates variable of integration
ans = sin(b + a*t)/a

% the constant of integration must be added for a finite integral
```

Plotting x,y data

Plotting one or more data sets on the same plot and axes.

```
plot(xdata,y1data, 'linespec1'); % for one data set OR
plot(xdata',[y1data' y2data']); % pass data as columns for two sets OR
plot(xdata,y1data, 'linespec1', xdata, y2data, 'linespec2'); % for two sets
xlabel('text for x axis'); ylabel('text for y axis');
legend('y1 data','y2 data','Location','Southeast'); %legend in lower right
% Note that compass locations can be used to indicate the legend location.
```

The strings 'linespec1' and 'linespec2' are optional placeholders replaced with the line styles and colors specified by strings, e.g. '--r' for a red dashed line. Use 'help plot' in the command window for a quick summary of line style options. It is also possible to use

```
plot(xdata,y1data, 'linespec1');
hold on; % holds current plot so that second data set can be added.
plot(xdata,y2data, 'linespec2'); % add second data set to same plot axes.
```

The second method provides more flexibility to make further modifications to the lines by adding additional strings and paired specification values. See '`help plot`'.

Suppose an array of data exist in Matlab with 100 rows and seven columns in the Matlab array '`data`', with the first column being the independent variable (x values). Such a table will frequently result from a computer program or an experiment. To plot all the data simultaneously as functions of the independent variable, use the command

```
plot( data(:,1) , data(:, 2:7) )
```

To plot just the second column as a function of the independent variable

```
plot( data(:,1), data(:,2) )
```

The plot tools can be opened from the figure menu bar to add axis labels and more detail if you prefer to add labels and titles manually.

Example Plots – see below for plots

```
x1 = linspace(0,10,11);
y1 = x1.^2;
y1data = y1.*(1 + 0.1*(rand(1,11)-0.5)); %simulated random noise
x2 = 10*x1 + 10;
y2 = x2.^2 + x2;
```

%Example Plot1 - plot with 'data' and 'curve'---

```
figure; plot(x1, y1data,'ko'); hold on; plot(x1,y1,'k'); title('Plot1');
xlabel('x1label'); ylabel('x2label');
```

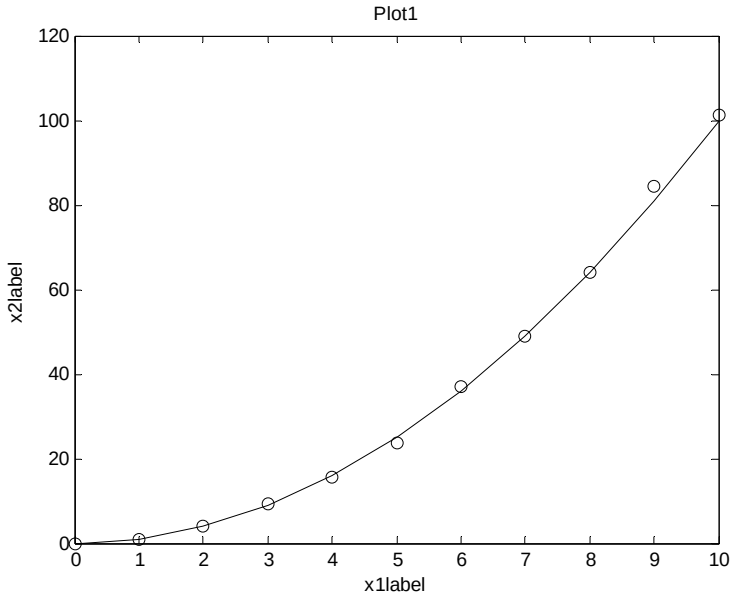
%Example Plot2 – new plot with two axes ----------

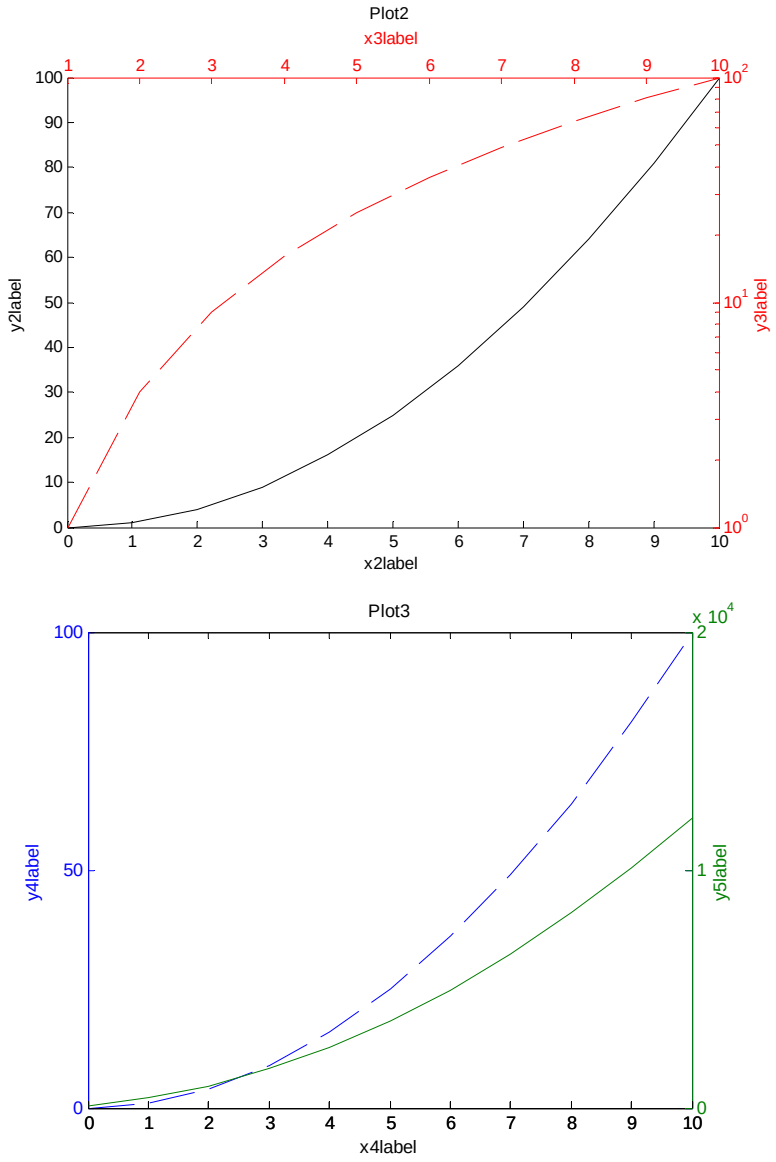
```
figure; plot(x1,y1,'k');
ax2 =gca; %get first axis handle
%turn off box in order to overlay second axis and
%make the size a little smaller to leave room for second axes
set(ax2,'Box','off', 'Position', [.1 .1 .8 .75]);
title('Plot2', 'Position', [5 110]); %use axis values to set position
%create a second axis on top of existing axis with transparent plot area and
%axis location at the top and right, log axis & labels red
ax3 = axes('Position',get(gca,'Position'),'Color','none',...
           'XAxisLocation','top','YAxisLocation','right',...
           'XColor','r','YColor','r','YScale','log');
hold on
plot(x1,y1,'r--'); %plot a red curve
xlabel(ax2, 'x2label'); xlabel(ax3, 'x3label');
ylabel(ax2, 'y2label'); ylabel(ax3, 'y3label');
```

%Example Plot3 --- new plot, 2y axes with a common x axis -------

```
figure; [ax4,h1,h2]=plotyy(x1,y1,x1,y2); title('Plot3');
%[ax4(1) ax4(2)] are handles for the axes
%yy plot labels and line styles are set using handles
xlabel(ax4(1),'x4label');
set(get(ax4(1),'Ylabel'),'String','y4label');
set(get(ax4(2),'Ylabel'),'String','y5label');
set(h1,'LineStyle','--')
```

The following figures were created with the script above and pasted in WORD with the emf format described below.





Creating multiple plots in the same figure

Suppose that we have the following vector data sets to plot: (x, y1), (x, y2), (z, y3). The following code will create two figures side-by-side with the first two data sets in the left plot and the last set in the right plot. Plots are laid in a grid using 'subplot (nrows, ncolumns, plotid)', where plots are numbers left to right in the same manner as reading rows of text.

```
% set up 1 row of 2 plots. First create the left plot.
subplot(1,2,1), plot(x,y1), xlabel('x values'), ylabel('y values')
title('behavior of y')
hold on   % add more data to the same plot
plot(x,y2, '--') % add a second line using dashed line.
% See Help LineSpec for more detail.
% add legend in the lower right corner…
legend('y1 data','y2 data,'Location','Southeast')
% The data on the plots are labeled in the order they were plotted.
hold off  % done with this plot
% now create the right plot
subplot(1,2,2), plot(z,y3), xlabel('z values'), ylabel('y3 values')
title('behavior of y')
```

Greek Characters for Labeling Plots
Matlab offers subsets of TeX and LaTeX to for using greek characters, subscripts and superscripts. Usually TeX is enabled by default. For Greek characters in a string, just use a back slash and the text name of the letter, e.g. \delta or \Delta for δ or Δ. Superscripts are ^, subscripts are _, and {} are delimiters, e.g. e^{-rt} results in e^{-rt}. Italics are indicated by \it, and bold by \bf. For more sophisticated equations LaTeX can be used. For more information search Help for 'Text Properites' (in quotes) and then scroll down to the 'String' property to find a description of the subset of TeX code that is permitted. To label the y axis with '$\Delta G_{mix}/(RT)$' use ylabel('\DeltaG_{mix}/(RT)').

Copy/Paste or Export of figures
This discussion assumes the use of a Windows platform. You can use Edit>Copy (See also Edit>Copy Options…and set to Metafile with Transparent Background). Usually the desired format is the 'emf' (enhanced metafile) which is a vector-based graphic that resizes nicely. You may also export as an .emf file (File>Export Setup...) and then import into another document.

Functions in Matlab

User-created functions are a powerful way to save a general set of statements that you will reuse. The general syntax of a function statement and of a function call is important. The syntax of a function definition is:

```
function [return array] = functionname(input1, input2, …, input n)
```

where `[return array]` is the list of variables to be passed back when the function completes. For example consider the function to calculate the volume and surface area of a box of dimensions {x, y, z}. A simple function called 'calcsize' performs the calculation:

```
function [volume area] = calcsize(x, y, z)
volume = x * y * z;
area = 2*x*y + 2*y*z + 2*x*z;
end % of function calcsize
```

To execute the function, one can use
`[V A] = calcsize(3, 2, 5);` % the variables/values will be substituted.
The function will return the calculated values in V and A. Another way to do this is
length = 3; height = 2; width = 5;
[Vol Area] = `calcsize(length, width, height);` % the variables/values will be substituted.
In the later function call, the calculated results are returned in variables Vol and Area.

Function calls can be made from the command window or from within another function. When a function call is made from the command window, the variables in the [return array] are saved in the 'workspace' so that further manipulations can be made.

Calling the Correct Function - Naming Functions and Files
Matlab **searches for functions by file name** and the function name in the file may or may not match the file name. For example the call

```
[outvar] = funct1(invar);
```

looks for FILE 'funct1.m' and passes it 'invar' even if the function name in funct1.m is something different. While the function name and file name do not need to be the same, Mathworks recommends that they are the same to avoid confusion. Be careful if you have multiple versions of the same function. You must be sure to call the correct version by filename. When you create a variation, it is best to rename the function and save the new variation with a matching file name.

Choosing variable names (structured variables)

If you use a period in a variable name, Matlab will create a 'structure'. Structures are very helpful for organizing data with properties. For example a tank may have a height 'h', surface area 'A', heat capacity 'Cp' and mass 'm'. We can organize these variables and make the variables easier to identify by using

```
tnk.h = 100; tnk.A = 300; tnk.Cp = 50.3; tnk.m = 345.3;
```

Arrays can be stored in structures, e.g. the mole fraction composition of a four component liquid in a tank could be represented by

```
tnk.x = [0.2 0.4 0.3 0.1]
```

It is also possible to create an array of structures for a series of tanks, e.g. tnk(1).h, tnk(2).h, etc. Empty structures can be defined using the 'struct' command and then the reserved memory locations can be filled during code execution. Matlab supports rich capabilities for object oriented programming also, see Help>Matlab>Getting Started>Programming.

Variable Scope

Variables are normally unique to the function in which they reside, and are usually passed using the function syntax described above. There are two exceptions. It is possible to declare global variables (see Help). The other important exception is when a function is nested into another function. Variables in nested functions are shared between the outer 'function' statement and the final 'end'. The nested functions must be placed before the 'end' statement of the outer function.

```
function [T P k] = main
%define some variables
P = 0.1; %MPa
Vbar = 1000; % cm^3
R = 8.314; % J/molK
n = 0.01; % mol
T = findT();
% The value of P is changed to 0.2.
% The value of k is also available to the
% outer function

      function T = findT()
      % can use vars from Main without passing.
      P = 0.2; % overrides the value in main.
      T = P * Vbar / n / R;
      k = 9;
      end % function findT

end % function main
```

Numerical Integration

Numerical integration is offered by several routines. A common routine to try first is ODE45. The syntax is

```
[t, y] = ode45(@odefun, tspan, y0)
```

Where t is the independent variable returned as an 1D array with the independent variable values used in the integration, and y is an array of dependent variables. The dimensions of y depend on the number of dependent variables; there is a column for each independent variable and the row can be matched to the corresponding row in t.

> odefun is a 'function handle' (name of a function where you will calculate differentials), which requires a '@' symbol before the function name. For example, a function handle might be @dydt. The function is defined separately and may be nested into the function that calls ode45.
>
> tspan is a vector of the initial and final values for the independent variable, e.g. [0, 10].
>
> y0 is a vector of initial values for each independent variable in y.

Matlab will 'automatically' decide on the step sizes to use to solve the differential equation. The 'ode45' statement tells Matlab to solve the equation and gives to Matlab the name of the function where Matlab can calculate the array of differential values ('odefun' in this example). Matlab will call the function as many times as necessary; you do not need to create the loop to call the function. Following the 'ode45' statement, the differential equation is solved and the results can be further manipulated or plotted. Matlab requires coupled differential equations to be defined/solved in the form [dy(1)/dt dy(2)/dt ..... dy(n)/dt] where y is the array of dependent variables, and t is the independent variable.

The manner to define the differential equations is most easily shown by looking at an example. Suppose that we wish to solve the differential equation for adiabatic tank depressurization, dT/dn = RT/n/Cv for an ideal gas where Cv = 20.9, starting from an initial temperature of T = 400K, and $n^{initial} = 500$. Suppose we wish to find T when n = 350. This example is easy to solve analytically, but will serve as an example of how to set up the functions.

```
function [n T P] = calctank( Tin, nin, nfin)
% Tin is the initial temperature, nin is initial moles,
% nfin is the final moles.
R = 8.3143 % J/mol-K
Cv = 20.9 % J/mol-K

% e bal is
% d(nU) = H dn ==> ndU = (H-U)dn ==> ndU = RTdn
% nCvdT = RTdn  ==>  dT/dn = RT/nCv
% analytical solution is (T/To)^(Cv/R) = (n/no), solution here should match.

% intermediate calculations can be inserted here if necessary.
% This simple problem does not require any. Next call the function
% to solve the differential equation. Matlab will calculate
% the differential(s) using function 'tankdiff' defined below
% and automatically pick an appropriate step size.

[n T] = ode45(@tankdiff, [nin nfin], Tin);
```

```
% the diffeq is completely solved. Arrays n and T are available for
% further processing.
Vtank = 10000; % cm^3
% use vector operators to generate P as a function of n and T.
P = n.*R.*T./Vtank;

          % define the differential equation
          function dTdn = tankdiff(n, T)
          % all variables from the main function can be used in this
          % nested function that provide Matlab the vector of
          % differential values. This example uses a single diffeq,
          % so a single differential is calculated.
          % Intermediate calculations can be inserted here if necessary
          % (none for this example.) In this case, there is a single
          % dependent variable stored in T(1). We need to use the array
          % notation to use the dependent variables in the calculation
          % of the differential values. Use T(1), not T.

          dTdn = [ R * T(1) / n / Cv ]
          end % of function dTdn

end % of function calctank
```

To solve the differential equation enter in the command window:
`[n T P] = calctank(400, 500, 350);`
The variables n, T, P will be placed in the workspace.

The analytical solution to this problem is T = Tin(n/nin)^(R/Cv). Check your answers found by numerical integration.

Function Handles and Anonymous Functions

Both function handles and anonymous functions use the `@` character, but in slightly different ways.

A function 'handle' is used to refer to objects. As humans, we use 'names', but the computer will use 'handles' to keep track of all kinds of objects. Each window has a handle for example, as used in the more complicated plotting examples above. When we create a plot window in Matlab, it is assigned a handle and we can manipulate the window or add objects if we keep know the window handle. Function names can be passed as handles. So when we use fzero, we usually pass it a function handle. However, it is also possible to pass it an anonymous function.

In the ODE45 example above, and in most solver calls, such as fzero, the name of the function is passed as a function handle. For a bubble calculation, I usually pass the name of the function, such as `@calcObj`, using a statement like `'bestT = fzero(@calcObj,guessT)'`. In this case `@calcObj` is a function handle.

It is also possible to pass an anonymous function to fzero. We do this in chap14/LLEflash.m. The code is:

```
  obj = @(ROF)sum(z.*(1-K)./(K+ROF*(1-K))); % configures objective function
  ROFnew = fzero(obj,ROFguess); %Solves objective function, start at ROFguess
```

Which could be combined to a rather complicated single statement:

```
ROFnew = fzero(@(ROF)sum(z.*(1-K)./(K+ROF*(1-K))),ROFguess);
```

This tells matlab to solve the function `sum(z.*(1-K)./(K+ROF*(1-K)))` by adjusting *only* `ROF`. The values of the other variables are fixed with the current values in memory. This could also be done by creating a function that calculates the sum and using a function handle:

```
ROFnew = fzero(@calcObj,ROFguess);
function err = calcObj(ROF) % syntax is important. ROF is the variable to
            % be adjusted, and err is the value that is go to zero.
     err = sum(z.*(1-K)./(K+ROF*(1-K)));  % all values of z and K will
        %be read from memory but not adjusted if this is nested in
        %another function.
end
```

Another place that you can see anonymous functions being used in a more advanced way is in chap08-09/PreosProps.m. Near line 200 find

```
[T,obj,exitflag] = fzero(@(T)FindMatch(Tref, Pref, T,P,match),T,options);
% Call optimizer
```

In this case, I am using user function FindMatch, but I am passing it as an anonymous function of *T*. This will call FindMatch repeatedly with other variables fixed, but vary only *T* until the conditions specified by '`match`' are met. You need to pass a function this way if it has multiple variables in the () because fzero can adjust only one of them. For another example, see near line 210,

```
[P,obj,exitflag] = fzero(@(P) FindMatch(Tref, Pref, T,P,match),P,options);
% Call optimizer
```

where you can know see that I am finding a match by adjusting *P* at fixed *T* by asking fzero to call the function repeatedly until a match is found.

Writing and Debugging Code

It is unlikely that your code will work the first time you write it. As you type in the editor window, note the highlighting on the code and any line highlights that appear on the right side of the editor window. Hover over the markers for suggestions. When your code is ready to test, you can click in the left margin to set a break point (grey dot until any file changes are saved and red dot otherwise). To run the code, click the green arrow in the top of the editor window. The code will run until the break point is reached. During the break you can hover over any variable to see the current value or you can type any variable name in the command window to get a report of a value. Also, you can type in the command window '`dbstop if error`' and a subsequent error will interrupt the code at the error point and give you an opportunity to click the line number to debug. (Use help dbstop to see many other options.) To continue, you can type `dbstep` or click the 'step', 'step in', 'step over', 'continue' buttons in the editor button bar. You will also find the following helpful to debug in the command window if running without the graphical interface: `who, whos, keyboard` (used within code). Use '`dbquit('all')`' to exit debug mode, or use the editor menu 'Debug>Exit Debug Mode'. See also the functions listed at the bottom of documentation for 'dbstop' and other commands linked to documentation for those functions.

Loop syntax

`for index = start:increment:end` % increment defaults to 1 if omitted
  statements
`end` % recommended to put a comment to identify the 'for' that mates with this 'end'

OR

`while` *expression* % expression is a logical expression, i.e. `n > 5`.
  statements
`end` % recommended to put a comment to identify the 'while' that mates with this 'end'

A loop can be exited at any time by using a '`break;`'.

Continuation lines

Long statements in .m files can be broken across multiple lines to improve readability. You can use the MATLAB sequence `. . .` to continue a line. The C language \ may be used sometimes.

```
array = [ 1 2 3 …
          4 5 6];
```

File I/O

Matlab can read files using some standard routines. However, much more flexibility is available using standard C I/O statements. For example, see fgets, fscanf for input, fputs, fprintf for output. A single long text line can be read with fgets and 'chopped' using strtok.

Making Output Compact and Pretty

Default formatting to the command window is configured for ease in reading at the computer screen. To make the output more compact, use '`format compact`' to suppress the blank lines, and '`format loose`' to restore. Use '`disp(A)`' for output of 'A' without a label. To provide documentation on an output line, use the '`sprintf`' function embedded in '`disp`', e.g. `disp(sprintf('The value of X is = %f  ', X))`, or more easily `fprintf('The value of X is = %f  \n', X))` when X is 5 will print '`The value of X is 5`'. In the fprintf command, the `\n` creates a new line. Type 'Formatting strings' into the online help dialog for more information on the format specifier. The formatting follows the ANSI C computer language standards.