Introduction to MATLAB and MATLAB Programming

Leigh J. Little SUNY Brockport September 18, 2001

Contents

1 Introduction

MATLAB is a powerful software program that allows you to do very complex numerical simulations and view the results using a variety of graphical tools. MATLAB is very easy to learn. It has its own programming language which allows you to create detailed functions and operates much like an interactive FORTRAN or C/C++ compiler. The purpose of this handout is to illustrate the basic operation of MATLAB, both as a simple calculator and a powerful programming tool.

This handout assumes that you have read the handout on MATLAB Linear Algebra and are familiar with common programming concepts.

1.1 Starting MATLAB

Waiting for details on new computer network in Dailey

1.2 The command line

When you start MATLAB, you will see a window that contains 3 frames. The main frame in this window is the `command window` and the `>>` is the `command prompt`. This is main way to interact with the MATLAB engine.

Everything in MATLAB is *case sensitive*, as in C/C++. Thus, the variable `AA` is different from the variable `Aa`. Also, one very important note: Although the commands in the `help` files are usually given in upper case letters (for emphasis), you should *always* reference built-in functions with lower case letters. On computers running Windows, the case type of built-in functions doesn't matter, but it WILL be significant if you should happen to transfer your MATLAB files to another operating system (UNIX, Mac).

1.3 Getting Help

MATLAB has an extensive online help facility. If you know the command name that you need help with (say, the `sin` function), the easiest way to get help is to enter:

```
>> help sin
SIN    Sine.
  SIN(X) is the sine of the elements of X.
```

A menu driven help system that briefly describes all commands (organized by function) can be observed by entering

```
>> helpwin
```

If you are not sure what command you want, use the `Help` menu pull down from the title bar. This allows you to search the entire manual by keyword or topic.

1.4 The diary Command

You can use the `diary` command to keep a record of all the commands you have entered. You can also use the history window in the lower left portion of the screen. The commands

```
>> diary on
>> diary off
```

turns the diary on and off respectively. By default, the diary will be stored in a file called `diary` in the current directory. You can give a specific file to use for the diary be entering

```
>> diary('filename')
```

1.5 Starting Over; The clear Command

When you want to completely remove one or more variables from the workspace use the clear command:

`clear` - clears all variables from the workspace

`clear A b X` - clears only the variables `A`, `b` and `X`.

1.6 Formatting the Output

MATLAB can display the result of calculations in several different precisions. The standard format is 5 digit open formatting. This is not always the best way to display the results of numerical calculations, hence, this can be changed to one of several alternate formats using the `FORMAT` command.

`format` - default 5 digit open formatting

`format short` - same as above

`format short e` - 5 digit exponential formatting

`format long` - 15 digit open formatting

`format long e` - 15 digit exponential formatting

See `help format` for more options.

1.7 Variables

MATLAB was originally designed to do calculations in linear algebra. It has since evolved into much more, but due to its origins, all variables in MATLAB are treated as n-dimensional matrices with the elements stored in double precision. You need to feel comfortable with linear algebra concepts (vectors and matrices) in order to use it, but fortunately, you only need to know a little to do a lot.

See the handout on MATLAB Linear Algebra for more information.

2 Programming Constructs

One of the nice things about MATLAB is that you can jump right into using it as a programming language. This is because

1) All variables are, by default, assumed to be double precision matrices, they do not need to be declared (there are occasional exceptions to this).

2) Only a few basic constructs are required to write a great many programs.

This section will outline the use of these constructs.

Relational (or logical) operators are used to make comparisons between variables. The result of a logical comparison is always either TRUE (= 1) or FALSE (= 0). The relational operators available in MATLAB are shown in Table 1.

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	logical equivalence
~ =	logical inequivalence
&	logical AND
|	logical OR
~	logical NOT (negation)

Table 1: Relational Operators in MATLAB

2.1 for loops

The basic looping operation in MATLAB is the `for` loop. It most closely resembles a for loop in BASIC with a little FORTRAN thrown in.

```
for i = start : step : end
   |
   |   Body of Loop
   |
end
```

Here, `i` is the loop index, `start` is the starting value `i`, `step` is the step increment and `end` is the terminal value of `i`.

As in other languages, loops can be nested, but each must have it's own `end` statement.

```
for i =  start_i : step_i : end_i
   for j = start_j : step_j : end_j
      |
      |   Body of Loop
      |
   end
end
```

Here is a simple example of a loop that will take the values of a vector `x` and sum them up.

```
total = 0;
for i = 1 : length(x)
   total = total + x(i);
end
```

There are some important things to note about this example:

1) The length of `x` does not need to be known. We can use the `length` inquiry function to obtain this information.

2) The line in the body of the loop ends in a semicolon. This is to prevent the value of `total` from being displayed on the screen at every step.

3) This example is for demonstration purposes only. A much better way to obtain the sum of the components of a vector would be to use the `sum` command

```
total = sum(x);
```

The reason for this is that the `sum` command will run much faster. Also, it makes the code more compact and easier to analyze.

2.2 while loops

An alternative to a `for` loop is the `while` loop. This is used in cases where the `end` value in a `for` loop is unknown.

```
while (conditional expression is true)
   |
   |       Body of Loop
   |
end
```

A simple example of a `while` loop is given below:

```
i = 0;
while (i < 10)
   i = i + 1;
end
```

This loop will add 1 to `i` until it reaches 10, then terminate. Also, note that the command to end a `while` loop is the same as in a `for` loop.

2.3 if-then-else Structures

The `if-then-else` construct is the same in MATLAB as in many other languages, with one important difference: there is no `then` actually present in the structure. Its presence is implied. The basic outline for the `if-then-else` is:

```
if(conditional expression 1)
   |
   |  Code for conditional expression 1 = TRUE
   |
else
   |
   |  Code for conditional expression 1 = FALSE
   |
end
```

As always, the `else` block is optional.

There is also an `if-then-elseif` contruct:

```
if (conditional expression 1)
   |
   |  Code for conditional expression 1 = TRUE
   |
elseif  (conditional expression 2)
   |
```

```
   |  Code for conditional expression 2 = TRUE
   |
else
   |
   |  Code for conditional expression 1,2 = FALSE
   |
end
```

Again, the command to end an `if` block is the same as with `for` and +while+ loops.

Here is a simple example of the `if-then-elseif` contruct:

```
i = 17;
if (i < 17)
   disp('i is less than 17')
elseif (i == 17)
   disp('i is equal to 17')
else
   disp('i is greater than 17')
end
```

2.4 switch construct

The `switch` contruct is a more refined version of the `if-then-else`. It is used mainly when you have several possibilities that can be expressed as a function of one variable. Although it's behavior can be imitated with `if-then-elseif` statements, the `switch` tends to be easier to analyze.

```
switch (j)
  case(1)
     |
     | Code to execute for j == 1
     |
  case(2)
     |
     | Code to execute for j == 2
     |
  case(8)
     |
     | Code to execute for j == 8
     |
  otherwise
     |
     | Code to execute for j not 1,2 or 8
     |
  end
```

2.5 Breaking a loop

Sometimes you need to break out of a loop prior to it's proper termination. This can be accomplished with the `break` command. This command will terminate a `for` or `while` loop. If a `break` appears inside several nested loops, the `break` applies only to the innermost loop. See `help break` for more information.

2.6 comments

It is always a good idea to add comments to your code. It makes it easy for yourself and other people to understand your program. In MATLAB, the comment character is the percent sign (%). Any statements after the comment symbol (and on the same line) will be ignored.

```
% Set i to an initial value
i = 0;
```

```
% Start while loop
while (i < 10)      % Stop when i = 10
   i = i + 1;       % Increment i
end
```

As in the above example, comments can be placed anywhere on a line.

3 Script Files

Script files are collections of MATLAB statements designed to accomplish a specific task and is similar to a `MAIN` program in other languages.

As a simple example of a script file, create a file called `testscript.m` in the current directory. You can do this by clicking on the `New M-file` on the MATLAB toolbar. This will open the MATLAB editor. Select `Save As` from the `File` pulldown and name the file `testscript`. The program will automatically give this file the `.m` extension.

Enter the following into the file:

```
n = 10;
A = rand(n);
b = rand(n,1);
x = A\b
```

Save this file and then enter `testscript` at the command prompt:

```
>> testscript
```

This script sets `n` to a value of 10. It then generates a random 10×10 matrix, A and a column vector of length 10, b. It then solves the linear system $Ax = b$ for x and displays the result.

The output will vary each time you execute the script since the values of A and b are random.

Script files are convenient for quick programming or for use as driver programs, but for more complicated programming situations, you should use functions. These are explained in the next section

4 User Defined Functions

As with any program language, the ability to create your own functions is a powerful tool. The general syntax for a MATLAB function is:

```
function [output argument list] = fname(input argument list)
%
%  Comments (not required, but recommended)
%
 |
 |
 | Body of function
 |
```

Note that there is no `end` statement for the function as a whole. In order for MATLAB to be able to find the function, it needs to be entered into a text file called (for the generic example above) `fname.m`. A few examples will help explain this concept more clearly.

4.1 Example 1 - Quadratic Equation

Enter the following text into a file called `solvquad.m`:

```
function [x] = solvquad(a,b,c)
%
%  This function solves the quadratic equation
%
```

```
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in a
%  vector of length 2.

x(1) = (-b + sqrt(b^2-4*a*c)) / (2*a);
x(2) = (-b - sqrt(b^2-4*a*c)) / (2*a);
```

This function then can be called from either the command line, a script file or another function. Also, the variables `x`, `a`, `b` and `c` are dummy variables and their names inside the function have no relation to the variables actually used in a formal function call.

```
>> a = 1; b = 0; c = 1;
>> x = solvquad(a,b,c)
x =
    1    -1
>> x(1)
ans
    1
>> x(2)
ans
   -1
```

Here, the equation $x^2 - 1 = 0$ has been solved and as expected, the 2 solutions are $x_1 = 1$ and $x_2 = -1$. The values of `a`, `b` and `c` can also be sent directly to the function as in:

```
>> x = solvquad(1,0,-1)
x =
    1    -1
```

Another way of coding the `solvquad` function is to return the 2 solutions separately. Enter the following into a file called `solvquad2.m`:

```
function [x1,x2] = solvquad2(a,b,c)
%
%  This function solves the quadratic equation
%
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in
%  the variables x1 and x2

x1 = (-b + sqrt(b^2-4*a*c)) / (2*a);
x2 = (-b - sqrt(b^2-4*a*c)) / (2*a);
```

This modified function can then be called as:

```
>> [x1, x2] = solvquad2(1,0,-1)
x1 =
    1
x2 =
   -1
```

Although this latter method will produce the same answers, the original version of the `solvquad` function is preferred. The original version keeps the output data in the same variable, hence, it is easier to keep track of and send to other functions if needed. Also, the sequence of output arguments is shorter.

4.2 Example 2 - Sorting

As a more complex example, consider the task of sorting the elements of a vector in increasing order. Create a file called `mysort.m` and enter the following text:

```
function [x] = mysort(y)
%
% This function sorts a vector in increasing order
% using a standard bubble sort.
%

x = y;
n = length(x);
for i = 1 : n-1
   for j = i+1 : n
      if (x(i) > x(j))
         t = x(j);
         x(j) = x(i);
         x(i) = t;
      end
   end
end
```

This function can now be used. Enter the commands below:

```
>> x = rand(10,1)
>> y = mysort(x)
```

The output should consist of 2 column vectors, the second one containing the sorted elements of the first.

It is possible for a variable to appear in both the input and output lists of a function. For example, the commands

```
>> x = rand(10,1)
>> x = mysort(x)
```

would sort a vector and overwrite the original vector with the sorted version.

MATLAB already has a built-in sort function called `sort` that will duplicate the above example. You should always use this function rather than the example above because it will execute much faster.

4.3 Example 3 - Plotting Eigenvalues

A user-defined function also has full access to the plotting capabilities of MATLAB. This is an example of using MATLAB graphics for investigating the eigenvalues of random matrices

```
function [e] = plotev(n)
%
%  This function creates a random matrix of square
%  dimension (n).  It computes the eigenvalues (e) of
%  the matrix and plots them in the complex plane.
%

A = rand(n);
e = eig(A);

close all    % Closes all currently open figures.
figure(1)
  plot(real(e),imag(e),'r*')
  xlabel('Real')
  ylabel('Imaginary')
  t1 = ['Eigenvalues of a random matrix of dimension ' num2str(n)];
  title(t1)
```

This example can the be called from the command line:

```
>> plotev(50)
```

4.4 Example 4 - Plotting matrix elements

This last example is one that builds on the previous two. Suppose we want a function that will do the following:

1) Generate a random vector `x` of specified length, `n`.

2) Generate a plot of the elements of `x` versus their index location.

3) Sort the elements of `x`.

4) Generate a plot of the sorted vector elements versus their index location.

5) The plots in 2) and 4) should appear in the same window (stacked on top of one another).

The only tricky part about this example is step 5), which requires the use of the `subplot` command. Here is a function that will accomplish the task:

```
function [x,y] = plsort(n)
%
%  This function carries out steps 1-5 as described above.
%  The sorting function created in Example 2 will be used.
%
%  The output variables are:
%     x = random column vector of length x
%     y = sorted version of y.

x = rand(n,1);
y = mysort(x);

figure(1)
  subplot(2,1,1)
  i = (1:n);
  plot(i,x)
  xlabel('Index')
  ylabel('x')
  t1 = ['Elements of a random vector of length ' num2str(n)];
  title(t1);

  subplot(2,2,2)
  plot(i,y)
  xlabel('Index')
  ylabel('y')
  t1 = ['Sorted elements of the vector'];
  title(t1);
```

Try out this example for several values of `n` and verify that it works.

Some notes regarding the above example:

1) You should always use the MATLAB `sort` function instead of one of your own.

2) The vector `i` was created for use as the index location of the vector elements. This does not have to be done. The following plot command would accomplish the same task:

```
plot(x)
```

3) Note the arrangement of the `subplot` arguments. The first 2 numbers give the number of rows and column in the *plot matrix*. Since we want the graphs to be stacked, we want a 2×1 plot matrix. If the command had been

```
subplot(1,2,1)
```

the graphs would have appeared next to each other.

5 Lab work

1) Modify Example 1 so that the coefficients of the quadratic equation `a`, `b` and `c` are input as a vector `d`. Be sure to add comments your MATLAB file that explain what the elements of the vector `d` are.

2) Modify Example 2 so that the elements are sorted in decreasing order instead. Also, explain how you can use the MATLAB `sort` function to do this.

3) Modify Example 3 to plot the singular values of the matrix `A` in addition to the eigenvalues. See `help svd` for information on how to obtain the singular values. Use the `subplot` command to place these plots next to each other in the same plot window.

4) Modify Example 4 so that a symbol is plotted at each point (in addition to connecting the points with lines). Also, change the program so that each figure is drawn in its own window. See `help figure` for more information.