# Math 8435
# Matlab Tutorial

**Purpose:** The purpose of this tutorial is to introduce some basic programming techniques in Matlab, as well as a few built-in Matlab tools. For further reference, take a look at the **Matlab Primer**. Furthermore, one can familiarize oneself with Matlab by using the on-line demonstration program. One accesses the demo program by typing *demo* at the Matlab prompt.

## 1 Creating Directories and Getting Files.

Each homework assignment should be performed in a different directory so that data files can be organized carefully. This will prevent much confusion as the quarter develops. To create a directory, one types the command *mkdir DIRECTORYNAME*. For example, to make the directory TUTORIAL, one types *mkdir TUTORIAL* at the prompt.

Typically, each homework assignment will require various specialized Matlab function routines. These will be obtained from the instructor's website in the following way.

1. From the TUTORIAL directory, type *netscape*3, do not click on the netscape icon as this gives version 2!
2. Go to Lowengrub's website $http://www.math.umn.edu/\sim lowengrb$
3. Click on *Class Resources for Math 8435*
4. Click on *New Window*
5. Click on *Tutorial Files*
6. Although this file will look very strange, save it as *tutor.tar.gz* in the TUTORIAL directory. You can find *save as* under the *file* icon in netscape.
7. Exit from netscape3 if desired.
8. At the prompt, in the TUTORIAL directory, type *gunzip tutor.tar*
9. Then, type *tar xvf tutor.tar*

This will create all the $*.m$ files that the tutorial will use. In the future, the other homework files will be obtained in an analogous way.

## 2 Starting up Matlab.

To start Matlab, one needs only to type *matlab* followed by *return*. One should make sure that one is working in an appropriate directory so that data files can be organized carefully. For example, in the TUTORIAL directory, type *matlab*. This will bring up the matlab program and give the prompt >>. All matlab commands will be given at this prompt. We note that Matlab is case sensitive so that upper and lower case letters are not equivalent.

If you have questions about any command in Matlab, there is excellent on-line help. Typing *help* at the prompt will give a list of all possible topics (there is also the command *help help*). To specialize, type *help topic*. If you need help with a specific function, type *help function*. As a test, type *help* cos and see the result.

## 3 Setting up Variables.

### 3.1 Scalar Variables.

Suppose we want to set the value of $x$ to be 7. Then, in Matlab, we simply type $x = 7$ at the prompt. Matlab returns to us

$$\begin{array}{ccc} x & = & \\ & & 7 \end{array}$$

which indicates that $x$ now equals 7. To prevent matlab from printing the value to the screen, which can be quite cumbersome, one simply appends the command with a semi-colon, (i.e. $x = 7;$). Other operations such as $+, -, *, /$ follow straightforwardly. Try these simple examples in Matlab:

1. Let $x = 25$, $y = 18$ then compute $x * y$, $y/x$, $x/y$.

2. Powers are performed using the caret operator, i.e. $x^2$ is found by typing $x$^2. Find $x^y$ using the values from the previous problem. Note that only 5 digits of the mantissa are shown. Actually, matlab calculates with many more digits. To see this, try the command *format long* and then recompute $x^y$. Note that you can store $x^y$ in a variable $z$ by typing $z = x^y$. To go back to the shorter mantissa, type *format short*.

3. Computing sines and cosines is straightforward also. Evalulate $\cos(x)$, $\sin(y)$, again using the values from before, by simply typing these commands into matlab. Again, use the format command to see the true value. The exponential function is *exp*. Compute *exp*$(y)$.

4. Type *help elfun* to see what available elementary functions that Matlab has.

## 3.2 Vector Variables.

The real power of Matlab is how it handles vector variables. To enter a row variable $\mathbf{x} = (0\ 2\ 4)$, one types the command $x = [0\ 2\ 4]$ at the matlab prompt (note brackets and not parentheses are used). Try this. To suppress the output, use the semi-colon (note that just typing the name of the vector without the semi-colon prints its value to the screen). This establishes $x$ as a length-3 row vector. Square brackets delineate the vector and spaces separate the components. To see that $x$ has length 3, type the command $length(x)$. There are *two* kinds of vectors, however. There are row vectors and column vectors. To enter a column vector

$$\mathbf{y} = \begin{pmatrix} 0 \\ 2 \\ 4 \end{pmatrix}$$

into Matlab, type $y = [0\ 2\ 4]'$ (or equivalently $y = [0;\ 2;\ 4;])$. Note that $\mathbf{y}$ has the same length as $\mathbf{x}$, although their "size" is different. $\mathbf{y}$ has 3 rows and 1 column while $\mathbf{x}$ has 1 row and 3 columns. Type $size(x)$ and $size(y)$ to see this distinction.

Now, let us use a vector to evaluate the function $f(x) = \sin(2\pi x)$ across the interval $[0, 1]$. First, we introduce a grid $0 = x_1 < x_2 < \ldots < x_n = 1$. Then, evaluate the function at each point $x_k$, i.e. $y_k = f(x_k)$. Let us suppose that we want our grid points to be spaced equally from each other. Then, the grid spacing should be $h = 1/n$ and so we should set $x_k = (k-1) * h$. One way to do this is to use the script file *setx.m*. Look at this file (in a different window if necessary) and understand its structure. Use the help command if necessary. One can view the file *setx.m* in Matlab by typing *type setx* at the prompt (no dot $m$ suffix). This function sets the components of the row vector $x$ to $x_k$ for $k = 1, \ldots, n$, where $n$ is set to 10. Note that % in Matlab specifies a comment and this line is not executed. The colon command 1:n produces the row vector [1 2 . . . n]. Type *help Colon* for further details.

Before running the Matlab routine *setx*, clear the Matlab workspace. There are two ways this can be done. The dangerous way to do this is to type *clear* . This clears all the variables held in memory. Another way to do this is to clear a single variable at a time. For example, to clear the $x$ variable, type *clear x*. Do this. Now, type *setx* at the Matlab prompt. After the prompt has returned, type $x$ to see what has happened.

There is an equivalent and simpler way to create such a vector. This is through the command *linspace*. Typing $z = linspace(a, b, n)$ creates a row vector $z$ with

$$z(k) = a + (k-1) * (b-a)/(n-1) \quad \text{for} \quad k = 1, \ldots, n$$

Try this. Clear the variable $z$ and type $z = linspace(0, 1, 11)$. Consider the difference $x - z$. Note that it is on the order of $10^{-15}$. To get more detail, try resetting the format by typing *format short e* and *format long e*.

Now that we have obtained a vector of grid points, we now evaluate the function $\sin(2\pi x)$ at the grid points. One way to do this is to write the scalar-level script:

$$\begin{array}{l} for\ k=1:n \\ \ y(k)=\sin(2*pi*x(k)) \\ end \end{array}$$

However, matlab supports vector-level operations and so these scalar-level operations can be accomplished by simply typing $y = \sin(2*pi*x)$ at the prompt. The act of replacing a loop in Matlab with a single vector-level operation will be referred to as *vectorization* and has the following three benefits

1. Speed. Many of the built-in Matlab functions execute faster when they operate on a vector of arguments.

2. Clarity. It is often easier to read a vectorized Matlab script rather than its scalar-level counterpart.

3. Education. Scientific computing on advanced machines requires one be able to think at the vector level.

## 4 Plotting.

Now, we plot the sine function we created in the previous section. To do this, type *plot(x,y)* at the prompt. This command produces a window with the function plotted in it. To make a title, type *title('The function y=sin(2*pi*x)')* at the prompt. To label the axes, type *xlabel('x (in radians)')* and *ylabel('y')*. The axes are automatically adjusted so that the range in $x$ is from the $min(x)$ to $max(x)$, the range in $y$ is analogous. To adjust the axes, one types $axis([xmin, xmax, ymin, ymax])$ where $[xmin, xmax, ymin, ymax]$ is the vector of desired minimum and maximum ranges. Test this command. Type $axis([-.5\ 1.5\ \ -1.5\ 1.5])$. Note that after this is done, the graph is not proportioned correctly. Type $axis$('image') to have the $x$ and $y$ axes proportioned correctly (i.e. length of $x$-axis is 2 while the length of the $y$-axis is 3).

The graph does not look very smooth to the eye. This is because the number of grid points is only $n = 10$. Redo the graph using $n = 20$. Use the *linspace* command to create the vector of grid points.

Now, we'll use the script file *SinePlot.m* to see the graphs of sine using different values of $n$. Look at the script file and understand what it does. The command *sprintf* prints a character string where the number $n$ is printed out using the command $\%3.0f$ which tells Matlab to print out at most 3 digits with 0 digits after the decimal point. Matlab prints to the screen using the command *disp*. Test the following printing commands

*disp(sprintf('One Degree = %5.3f Radians',pi/180))*

*disp(sprintf('One Degree = %5.3e Radians',pi/180))*

Finally the pause(1.5) tells Matlab to wait 1.5 seconds before finishing the loop. Now, run the script by typing *SinePlot* at the prompt.

Let's now plot a different function. Consider the function

$$f(x) = \left(\frac{1 - sin(2\pi x)^2/24}{1 + x/24 + x^2/384}\right)^8$$

on the interval $[0, 1]$. Suppose $x$ is already initialized to be a grid vector (i.e. by using *linspace*). Then, a scalar implementation of evaluating $f$ on a grid is

```
$for\ k = 1 : n$
  $y(k) = \Big((1 - \sin(2\pi x(k))^\wedge 2/24)/(1 + x(k)/24 + x(k)^2/384)\Big)^\wedge 8$
$end$
```

This routine is contained in the script *Compf.m* where $n = 1000$. Look at the file. The Matlab commands *tic* and *toc* give the elapsed time that it takes for the commands between them to be executed. Run the script by typing *Compf* at the prompt. Run it several times to see if the elapsed time changes.

A vectorized version of the same routine is given in the script *Compfv.m* where the $v$ stands for vectorized. This code relies on the command $q = w. * x$ which produces a vector $q$ with the property that each component is equal to the products of the corresponding components of $w$ and $x$, i.e. $q(k) = w(k) * x(k)$. For this command to work, the vectors $w$ and $x$ must have the same size. Matlab also supports scalar-vector multiplication. The command $z = x/24$ divides every component of $x$ by 24 and stores the result in $z$. Also, the command $z = 1 + x$ adds one to every component of $x$ and stores the result in $z$. Likewise, $z = w + x$ adds the components of $w$ and $x$ and stores them in $z$. Again, $w$ and $x$ must have the same size.

Now, run the script by typing *Compfv* at the prompt. Note the difference in elapsed time with the scalar version! This is why one always uses vectorized code in Matlab.

**PROBLEM 1:**

- Plot the graph of $f$ and give it a title that includes the name and the elapsed time. Use the *sprintf* command. Give the axes appropriate labels. To print out the graph, type *printgraphname.ps* at the prompt. This will produce the postscript file *graphname.ps* that you can then print out on the laser printer.

Before we move on, we note that multiple curves can be plotted on a single graph. An example of this is contained in the script *SineAndCosPlot.m*. Understand this script and run it. The command $plot(x1, y1,' specs1', x2, y2,' specs2', \ldots, xn, yn,' specsn')$ plots the $n$ curves $(xi, yi)$ on a single plot where *specsi* denote the specifications of the *i-th* curve. Type *help plot* to find out more about the specifications. Equivalently, one could type the sequence of commands

$$plot(x1, y1,' specs1')$$
$$hold$$
$$plot(x2, y2,' specs2')$$
$$\vdots$$
$$plot(xn, yn,' specsn')$$

The command *hold* freezes the plot and makes all the subsequent graphs appear on the same plot. One types *hold* again to remove this feature. One can clear the plot in the figure window by typing *clg* (for clear graph). After you have run *SineAndCos*, try plotting the same graph by using the *hold* command.

## 5 Loops and If Statements.

Consider the following example. Suppose $x_1$ is a given positive integer and that for $k \geq 1$ we define the sequence $x_1, x_2, \ldots$ as follows

$$x_{k+1} = \begin{cases} x_k/2, & \text{if } x_k \text{ is even} \\ 3x_k + 1 & \text{otherwise} \end{cases}$$

For example, if $x_1 = 5$, then the following sequence develops

$$5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

This is called the *up/down* sequence. Note that it cycles once the value 1 is reached. We will develop a script that simulates this sequence and at the same time, develop the use of *while* loops and *if* statements.

We will start with a script file that solicits a starting value and then generates the sequence, assembling the values in a vector $x$. Here is the script file:

```
x(1) = input('Enter initial positive integer :');
k = 1;
while (x(k) ~= 1)
 if rem(x(k), 2) == 0
   x(k + 1) = x(k)/2;
 else
   x(k + 1) = 3 * x(k) + 1;
 end
 k = k + 1
end
```

The *input* command is used to set up $x(1)$ and prompts for keyboard input, for example

Enter initial positive integer:

Whatever positive integer you type, it is assigned to $x(1)$. After $x(1)$ is initiallized, the *while* loop continues until $x(k) = 1$. The Matlab tool $rem(a, b)$ returns the remainder when $b$ is divided into $a$. The operators that can be used in an *if-then-else* or *while-end* are $<$ (less than), $<=$ (less than or equal), $==$ (equal), $>=$ (greater than or equal), $>$ (greater than), $\sim=$ (not equal).

One of the things one should guard against when iterating in this way is that an infinite number of iterations do not occur. This will produce an unacceptably large vector $x$. A way to guard against this is to replace the *while* condition with

```
while( (x(k) ~= 1)&(k < 500) )
```

which prevents more than 500 iterations from occuring. The operator & in the *while* condition means *and*. The and (&) or (|), not $\sim$ and exclusive or (*xor*) operations are valid in Matlab.

In the script file *UpDown.m* we give the Matlab code. Look at the file and understand what it does. The Matlab tool $zeros(m, 1)$ creates a column vector consisting of $m$ zeros. The two assignments just after the *while* loop act to reduce the size of $n$ and $x$. The command $subplot(mnp)$ breaks the figure plot into $m$ subplots and allocates the space for a single plot. The *plot* command then plots a graph in the allocated location. Type *help subplot* for further details. Finally, the command $sort(x)$ sorts the values of $x$ starting from smallest to largest. The command $-sort(-x)$ sorts the values of $x$ starting from largest to smallest.

**PROBLEM 2:**

- Run the script by typing *UpDown* at the prompt. Print out the graph corresponding to $x(1) = 17$.

## 6 Creating Functions

In this section, you will now gain experience with creating functions. They are very similar to the script files that we have dealt with earlier. Functions take the generic form

$$function[output\_values] = name\_of\_function[input\_values]$$

and unlike fortran, only one function can be contained in a $*.m$ file at a time. Like fortran, all variables used in the function are local and are not retained after the function call is finished. The only variables that are retained are those specified as output variables.

Consider the file *dydt.m*. Note the structure. This routine evaluates the function $f(t, y) = y + t$ for the input values $t$ and $y$, either (or both) of which could be a vector.

Test the evaluation by first setting $t = 0$ and $y = 1$ and typing $output = feval('dydt', t, y)$. Type *help feval* for further details on this function. Another way to evaluate the function *dydt.m* is to type $[output] = dydt(t, y)$. Try both approaches.

Now, take a look at the Runge-Kutta-Fehlberg 45 routine $rkf45DF.m$. Note its structure. Let's now use this routine to compute the solution to the initial value problem

$$\begin{aligned} y' &= y + t, \\ y(0) &= 1. \end{aligned}$$

Let us compute the solution over the time interval $0 \leq t \leq 10$. Be sure to read the comments associated with the routine. Now, set $a = 0$, $b = 10$, $y0 = 1$, $tol = 1.e - 6$, $dtmin = 1.e - 4$ and $dtmax = 0.1$. Run the routine by typing

$$[y, t, deltat] = rkf45DF('dydt', a, b, y0, tol, dtmin, dtmax)$$

note the single quotes. Plot the solution by typing $plot(t, y)$. The exact solution is $y(t) = e^t(y(0) + 1) - (t + 1)$. What is the absolute value of the error in the solution as a function of time? Plot the result. Note that the exponential is *exp* in matlab. Again, typing *help elfun* can tell you this.

Now, let us see how the time step varies in time. Plot the time step as a function of time by typing *plot(t,deltat)*. How does the graph look?

Now, rerun the code using a fixed time step. Use the *rk4DF.m* routine. Set $dt = 0.05$. Keep the old data, so instead type:

$$[y\_fixed, t\_fixed] = rk4DF('dydt', a, b, y0, dt)$$

for example. How many iterations were performed? Now, plot the absolute value of the error for the fixed time step. Note, the time at which the exact solution should be compared with the numerical result depends on the time step. What conclusions can you draw?