Matlab Introduction

Matlab is both a powerful computational environment and a programming language that easily handles matrix and complex arithmetic. It is a large software package that has many advanced features built-in, and it has become a standard tool for many working in science or engineering disciplines. Among other things, it allows easy plotting in both two and three dimensions.

Matlab has two different methods for executing commands: *interactive mode* and *batch mode*. In interactive mode, commands are typed (or cut-and-pasted) into the 'command window'. In batch mode, a series of commands are saved in a text file (either using Matlab's built-in editor, or another text editor such as Emacs) with a '.m' extension. The batch commands in a file are then executed by typing the name of the file at the Matlab command prompt. The advantage to using a '.m' file is that you can make small changes to your code (even in different Matlab sessions) without having to remember and retype the entire set of commands. Also, when using Matlab's built-in editor, there are simple debugging tools that can come in handy when your programs start getting large and complicated. More on writing .m files later.

Scalar Variables and Arithmetic Operators

Scalar variables are assigned in the obvious way:

```
>> x = 7
x =
       7
```

The `>>` is Matlab's command prompt. Notice that Matlab echos back to you the value of the assignment you have made. This can be helpful occasionally, but will generally be unwieldy when working with vectors. By placing a semicolon at the end of a statement, you can suppress this verbose behavior.

```
>> x = 7;
```

Variables in Matlab follow the usual naming conventions. Any combination of letters, numbers and underscore symbols ('_') can be used, as long as the first character is a letter. Note also that variable names are case sensitive ('X' is different from 'x').

All of the expected scalar arithmetic operators are available:

```
>> 2*x
ans =
    14

>> x^2
ans =
    49
```

Notice that the multiply operation is not implied as it is in some other computational environments and the '*' operator has to be specified (typing '>>2x' will result in an error). This is also a good time to point out that Matlab remembers the commands that you are entering and you can use the up-arrow button to scroll through them and edit them for re-entry. This is very handy, especially when you are repeatedly making minor changes to a long command line.

There are a few predefined variables in Matlab that you will use often: `pi`, `i`, and `j`. Both `i` and `j` are defined to be the square-root of -1 and `pi` is defined as the usual 3.1416... . These variables can be assigned other values, so the statement

```
>> pi = 4;
```

is valid. Care is needed to avoid changing a predefined variable and then forgetting about that change later. That may seem unreasonable with `pi`, but `i` and `j` are both natural variables to use as indices and they are often changed. A useful command is the '`clear`', or '`clear all`' command. Typing either of these at the command prompt will remove all current variables from Matlab's memory and reset predefined variables to their original values. The `clear` command can also remove one specific variable from memory. The command

```
>> clear x
```

would only remove the variable named 'x' from Matlab's memory.

Matrix Variables and Arithmetic Operators

Another useful Matlab command is '`whos`', which will report the names and dimensions of all variables currently in Matlab's memory. After typing the command at the prompt we see:

```
>> whos
  Name      Size         Bytes  Class

  x         1x1              8  double array

Grand total is 1 elements using 8 bytes
```

The important thing to note right now is that the size is given as '`1x1`'. Matlab is really designed to work with vectors and matrices, and a scalar variable is just a special case of a 1x1 dimensional vector. To assign a vector containing the first 5 integers to the variable `x`, we could type this command:

```
>> x = [1 2 3 2^2 2*3-1]
x =
     1     2     3     4     5
```

We won't have much occasion to operate on matrices that are higher dimension, but if you wanted to create a 2-D matrix you could use a command something like:

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

To create larger vectors than the toy examples above (say, the integers up to 100), we would need to type a lot of numbers. Not surprisingly, there are easier ways built in to create vectors. To create the same 5-element vector we did above, we could also type:

```
>> x = [1:5]
x =
     1     2     3     4     5
```

The colon operator creates vectors of equally spaced elements given a beginning point, and maximum ending point and the step size in between elements. Specifically, `[b:s:e]` creates the vector `[b b+s b+2*s ... e]`. If no step size is specified (as in the example above), a step of 1 is assumed. So, the command `[1:2:10]` would create the vector of odd integers less than 10, `[1 3 5 7 9]` and the command `[1:3:10]` would create the vector of elements `[1 4 7 10]`.

Let's create the vector of odd elements mentioned above:

```
>> x_odd = [1:2:10]
x_odd =
     1     3     5     7     9
```

You can access any element by indexing the vector name with parenthesis (indexing starts from one, not from zero as in many other programming languages). For instance, to access the 3rd element, we would type:

```
>> x_odd(3)
ans =
     5
```

We can also access a range of elements (a subset of the original vector) by using the colon operator and giving a starting and ending index.

```
>> x_odd(2:4)
ans =
     3     5     7
```

If we want to do simple arithmetic on a vector and a scalar, the expected things happen,

```
>> 3+[1 2 3]
ans =
     4     5     6

>> 3*[1 2 3]
ans =
     3     6     9
```

and the addition or subtraction of matrices is possible as long as they are the same size:

```
>> [1 2 3]+[4 5 6]
ans =
     5     7     9
```

The operators '`*`' and '`/`' actually represent matrix multiplication and division which is not typically what we will need in this course. However, a common task will be to form a new vector by multiplying (or dividing) the elements of two vectors together, and the special operators '`.*`' and '`./`' serve that purpose.

```
>> [1 2 3].*[4 5 6]
ans =
     4    10    18

>> [1 2 3]./[4 5 6]
ans =
    0.2500    0.4000    0.5000
```

Beware that the operator '`^`' is a shortcut for repeated *matrix* multiplications ('`*`'), whereas the operator '`.^`' is the shortcut for repeated *element-by-element* multiplications ('`.*`'). So, to square all of the elements in a vector, we would use

```
>> [1 2 3].^2
ans =
     1     4     9
```

Built-in Commands

Matlab has many built-in commands to do both elementary mathematical operations and also complex scientific calculations. The '`help`' command will be useful in learning about built-in commands. By typing `help` at the command prompt, you are given a list of the different categories that Matlab commands fall into (e.g., general, elementary matrix operations, elementary math functions, graphics, etc.). Notice that there are probably even specific toolboxes included with your Matlab package for performing computations for disciplines such as signal processing. To see all of the commands under a certain topic, type '`help topic`'. To get a description of a specific command, type '`help command`'.

We'll look at a couple of example commands. First, the square-root command, `sqrt`. To calculate the square root of a number, type:

```
>> sqrt(2)
ans =
    1.4142
```

Again, to illustrate that Matlab understands complex numbers, we can have it calculate the root of a negative number:

```
>> sqrt(-9)
ans =
        0 + 3.0000i
```

Many Matlab commands that operate on scalars will also operate element-by-element if you give it a vector. For instance:

```
>> sqrt([1 2 4 6])
ans =
    1.0000    1.4142    2.0000    2.4495

>> sqrt([1:8])
ans =
    1.0000    1.4142    1.7321    2.0000    2.2361    2.4495    2.6458    2.8
284
```

Another useful command is `sin` function, which also operates on a vector.

```
>> sin([pi/4 pi/2 pi])
ans =
    0.7071    1.0000    0.0000
```

It is important to realize that digital signals are really just a collection of points and can be represented by a vector. Because Matlab allows functions like `sin` and `sqrt` (as well as many others) to operate on vectors, many of the signal definitions and calculation we would like to perform are very straight-forward. We will illustrate this a little more explicitly in the next section.

Signals, Plotting and Batch Operation

We will now use the elementary tools in Matlab to build a basic signal we have talked about in class. All of the commands given in this section are already written in the file *sinplot.m*. They can be executed in batch by typing *sinplot* at the command prompt. If the file is not found, use the 'current directory' browser at the top of the Matlab window to make the current directory the one where the file lives.

First, we will define a signal which is a 2 Hz sinewave over the interval [0,1] seconds:

```
>> t   = [0:.01:1];                % independent (time) variable
>> A = 8;                          % amplitude

>> f_1 = 2;                        % create a 2 Hz sine wave lasting 1 sec
>> s_1 = A*sin(2*pi*f_1*t);
```

Anything following a '%' symbol is a comment and will be ignored by Matlab. A 4 Hz sinewave with the same amplitude will also be defined:

```
>> f_2 = 4;                        % create a 4 Hz sine wave lasting 1 sec
>> s_2 = A*sin(2*pi*f_2*t);
```

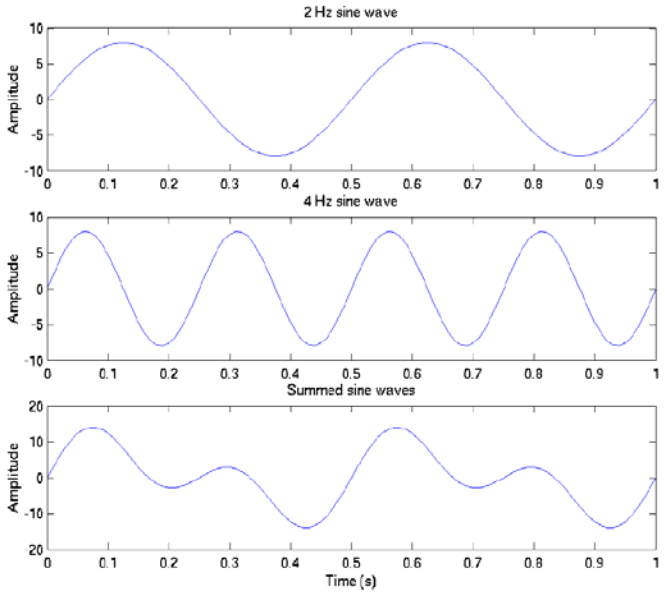
There are a few commands necessary to do basic plotting of these signals. The `figure` command will bring up a new figure window and the `close` command closes a specific window ('`close all`' closes all open windows). The `subplot(r, c, p)` command allows many plots to be 'tiled' into `r` rows and `c` columns on one figure window. After this command is issued, any succeeding commands will affect the p^{th} tile. The `plot(x,y)` command will then plot the equal-length vectors `x` and `y` on the appropriate axis and in the obvious way. The two sinewaves created above are then plotted along with the sum of the two signals in separate tiles with the commands:

```
%plot the 2 Hz sine wave in the top panel
figure
subplot(3,1,1)
plot(t, s_1)
title('2 Hz sine wave')
ylabel('Amplitude')

%plot the 4 Hz sine wave in the middle panel
subplot(3,1,2)
plot(t, s_2)
title('4 Hz sine wave')
ylabel('Amplitude')

%plot the summed sine waves in the bottom panel
subplot(3,1,3)
plot(t, s_1+s_2)
title('Summed sine waves')
ylabel('Amplitude')
xlabel('Time (s)')
```

The commands `title`, `xlabel`, and `ylabel` all take strings as arguments and apply the appropriate label to the top, x-axis or y-axis of the current subplot, respectively.

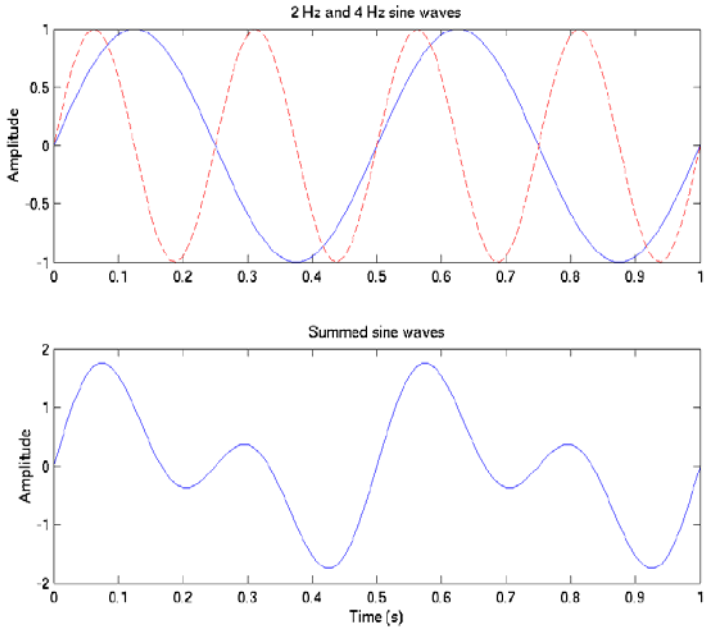


Normally, whenever you have plotted a function on a subplot and you try to plot another function, Matlab will erase the subplot and start over. Often you will want to plot two signals on top of each other for comparison. This can be done by using the '`hold`', which toggles the hold property of a subplot on or off. In this next example, the 2 original sinewaves are created again, this time using the built-in exponential function `exp`, the built-in function `imag` and Euler's relation. The `imag` function just returns the imaginary part of a complex number. These statements accomplish the same thing that the `sin` function did earlier, and are only used to illustrate again that Matlab handles all of the complex arithmetic we need very naturally. In this example, the two sinewaves are plotted on top of each other, and the sum is plotted again in its own subplot. Notice that the `plot` command can also take a third argument that specifies the color and linestyle of the resulting plot. There are many options here, and they can be viewed by typing `help plot` at the prompt.

```
% create the same sine waves using Euler's relations
s_3 = (exp(j*2*pi*f_1*t) - exp(-j*2*pi*f_1*t))/(2*j);
s_4 = imag(exp(j*2*pi*f_2*t));

% plot the 2 and 4 Hz waves together in the same panel
figure
subplot(2,1,1)
plot(t,s_3, 'b-')
hold on
plot(t, s_4, 'r--')
ylabel('Amplitude')

% again plot the sum in the bottom panel
subplot(2,1,2)
plot(t, s_3+s_4)
ylabel('Amplitude')
xlabel('Time (s)')
```



Basic Programming (For Loops)

Though we haven't mentioned much of this, Matlab is also actually a programming language with all of the usual iterative and conditional execution abilities. Because it is matrix based, many operations that you would normally use a loop for (say, if you were programming in C) can be done much faster using Matlab's built-in matrix operations. But, loops are sometimes unavoidable when you have repeated operations to perform on a list of parameters, and we will illustrate a simple one with an example here. All of the commands given in this section can be found in the file *sumsinplot.m*, and can be executed in batch by typing *sumsineplot* at the command prompt.

By far, the most common looping apparatus (in Matlab, at least) is the `for` loop. The basic `for` loop has the following structure:

`for` *variable* = *values*
  statements
`end`

where *variable* in this case is generally some sort of counter or indexing variable. This is probably best explained through an illustration. In this example, we will generate sinwaves with odd integers for frequencies (1, 3, 5, 7, 9, 11) and cumulatively sum them together. After each new sinewave is added to the others, the result is plotted in its own subplot. To be explicit, a 1 Hz sinewave is generated and plotted. Then a 3 Hz sinewave is generated, added to the 1 Hz sinewave and plotted. A 5 Hz sinewave is generated, added to the sum of the 1st two signals and plotted, etc. The only new things in this example are the use of the `for` loop, and the use of the zeros command. The `zeros(n,m)` command returns a vector of dimension `(n x m)`, initialized to all zeros.

```
%plot cumulatively summed sine waves with odd integer frequencies scaled by
1/f

>> t = 0:.001:1;        % independent (time) variable
>> A = 1;              % amplitude
>> f = 1:2:11;         % odd frequencies
>> s = zeros(1,101);   % empty 'signal' vector to start with

>> figure

% for each frequency, create the signal and add it into s(t)
>> for count=1:6
>>     s = s + A*sin(2*pi*f(count)*t)/f(count);
>>     subplot(6,1,count)
>>     plot(t,s);
>>     ylabel('Amplitude')
>> end

>> subplot(6,1,1)
>> title('Cumulatively summing sine waves')
```

```
>> subplot(6,1,6)
>> xlabel('Time (s)')
```