

48540 Signals and Systems

MATLAB[®] Tutorial

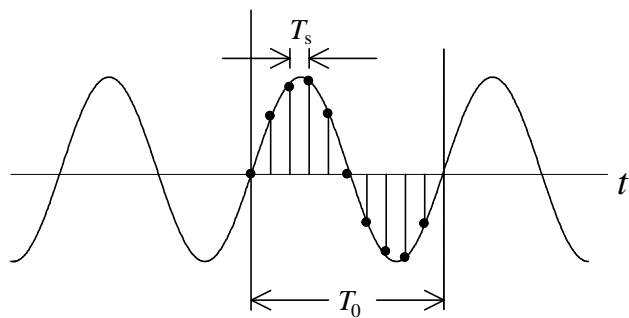
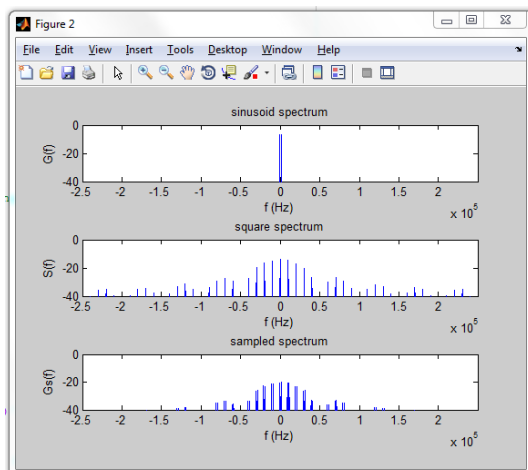
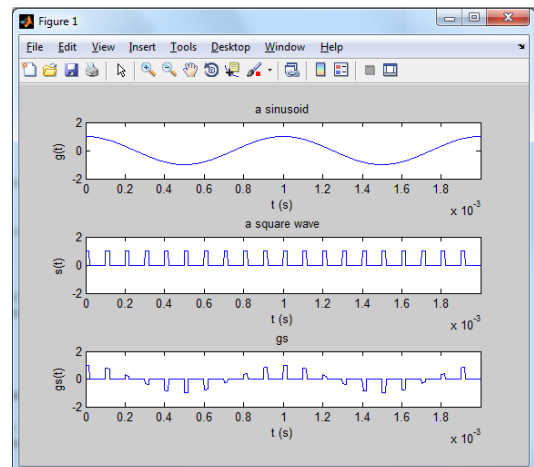
2015

```
% sample time & sample rate
Ts=2e-6;
fs=1/Ts;

% number of samples
N=1000;

% time window & FFT sample spacing
To=N*Ts;
fo=1/To;

% time and frequency vectors
t=0:Ts:To-Ts;
f=-fs/2:fo:fs/2-fo;
```



PMcL

MATLAB[®] – A Tutorial

Contents

Introduction	M.3
1.1 Invoking MATLAB	M.3
1.2 Entering Simple Statements	M.4
1.3 Formatting Output	M.5
1.4 Entering Simple Vectors	M.6
1.5 Entering Simple Matrices	M.7
1.6 Generating Vectors	M.8
1.7 M-Files: Scripts and Functions	M.9
1.8 Scripts	M.9
1.9 Functions	M.9
1.10 Creating a Script	M.10
1.11 Debugging your Script	M.11
1.12 Clearing the Workspace	M.12
1.13 Using Vectors in Functions	M.12
1.14 Plotting	M.13
1.15 Help	M.14
1.16 Plotting on a New Figure	M.14
1.17 Changing Figure Numbers	M.15
1.18 Clearing Old Figures	M.15
1.19 Script Comments	M.15
1.20 Changing the Plot Axes	M.17
1.21 Adding Axis Labels	M.17
1.22 Adding a Plot Title	M.17
1.23 Creating Subplots	M.17
1.24 Functions	M.18
1.25 Creating a Graph Function	M.20
1.26 Vector Operations	M.22
1.27 The FFT	M.26
1.28 FFT Samples	M.27

M.2

1.29 Time-Domain Samples	M.28
1.30 Frequency-Domain Samples	M.29
1.31 Plotting a Magnitude Spectrum	M.30
1.32 Finishing Your MATLAB Session	M.33

Introduction

MATLAB is a technical computing environment for high-performance numeric computation and visualization. MATLAB integrates numerical analysis, matrix computation, signal processing and graphics in an easy-to-use environment where problems and solutions are expressed just as they are written mathematically – without traditional programming.

The name MATLAB stands for *matrix laboratory*. It is an interactive system whose basic data element is a matrix that does not require dimensioning. This allows you to solve many numerical problems in a fraction of the time it would take to write a program in a language such as C.

In university environments, it has become the standard instructional tool for engineering courses that involve numeric computation, algorithm prototyping, and special purpose problem solving with matrix formulations that arise in disciplines such as automatic control theory and digital signal processing. In industrial settings, MATLAB is used for research and to solve practical engineering and mathematical problems.

MATLAB is now a standard package at universities and in industry

MATLAB also features a family of application-specific collections of functions called *toolboxes*. These extend the MATLAB environment in order to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems design, dynamic systems simulation, systems identification, neural networks, and others.

1.1 Invoking MATLAB

Start MATLAB like any other Windows[®] program. You will see a Command Window appear with the MATLAB prompt `>>`. At this point the MATLAB interpreter is awaiting instructions from you.

1.2 Entering Simple Statements

If you just type an expression (and press the `Enter` key), such as:

```
1900/81
```

then MATLAB assigns the answer to the `ans` variable and displays:

```
ans =
    23.4568
```

To assign a result to your own variable, you enter a statement such as:

```
Ts=2
```

MATLAB will respond with:

```
Ts =
     2
```

Note in the right pane titled `Workspace` that `Ts` has appeared, with a value and a range. MATLAB has “evaluated” the statement you typed of the form:

```
variable = expression
```

and returned the result to you on the screen. It has also created an internal variable called `Ts` that it now “knows” about which can be used in later expressions.

MATLAB supports
complex numbers

MATLAB allows complex numbers, indicated by the special functions `i` and `j`, in all its operations and functions. Engineers prefer:

```
z=3+4j
```

whilst mathematicians prefer:

```
z=3+4i
```

Note that `i` and `j` appear *after* a constant. Another example is:

```
w=5*exp(j*0.9273)
```

Note that the mathematical quantity e is represented by `exp`.

MATLAB will always display the output using `i`.

```
w =
    3.0000 + 4.0000i
```

1.3 Formatting Output

You can use the `format` command to control the numeric display format. The `format` command affects only how matrices display, not how they are computed or saved. The default format, called the `short` format, shows approximately five significant digits. The other formats show more significant digits or use scientific notation.

Push the \uparrow (up-arrow) to recall the previous command and edit it so that it says:

```
Ts=0.002
```

and press `Enter`.

Now enter:

```
format long
Ts
```

Changing the output
format

The resulting output is:

```
Ts =
    0.0020000000000000
```

Type:

```
format short e
Ts
```

The resulting output is:

```
Ts =
    2.0000e-003
```

Type:

```
format short
```

to return the formatting to the default state.

To suppress output to the screen append a semicolon, `;`, to your statement.

The semicolon, `;`,
suppresses output

```
Ts=0.002;
```

produces no output.

1.4 Entering Simple Vectors

Enter simple row vectors by separating values with spaces

To enter a row vector, surround the elements by brackets, []. For example, entering the statement:

```
t = [0 1 2]
```

results in the output:

```
t =  
    0    1    2
```

To make a column vector explicitly, separate elements with a semicolon, ;.

Enter simple column vectors by separating values with semicolons

Entering:

```
t = [0; 1; 2;]
```

results in the output:

```
t =  
    0  
    1  
    2
```

Use a single quote, ', to transpose a vector

You can also use the single quote, ', to transpose a vector:

```
t = [0 1 2]'
```

results in the output:

```
t =  
    0  
    1  
    2
```

Matrix elements can be any expression, for example:

```
x = [-1.3 sqrt(3) (1+2+3)*4/5]
```

results in:

```
x =  
-1.3000    1.7321    4.8000
```

1.5 Entering Simple Matrices

To enter a matrix, surround the elements by brackets, [], separate columns by spaces, and rows by semicolons. For example, entering the statement:

```
A=[1 2 3; 4 5 6; 7 8 9]
```

results in:

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

Enter simple matrices by separating values with spaces and semicolons

Individual matrix elements can be referenced with indices inside parentheses, (). Continuing the example:

```
A(3,4)=10
```

produces:

```
A =
     1     2     3     0
     4     5     6     0
     7     8     9    10
```

Notice the size of A is automatically increased to accommodate the new element and that the undefined intervening elements are set to zero. The indexing is of the form (row, column).

Two convenient ways to enter complex matrices are:

```
A=[1 2; 3 4] + j*[5 6; 7 8]
```

and

```
A=[1+5j 2+6j; 3+7j 4+8j]
```

When you enter complex numbers as matrix elements within brackets, it is important to avoid any blank spaces. An expression like $1 + 5j$, with blanks surrounding the + sign, represents two separate numbers. This is also true for real numbers; a blank before the exponent, as in $1.23 e-4$, causes an error.

Numbers should not have spaces when entering values

1.6 Generating Vectors

Generate a vector with start and end values using a colon, :, ...

The colon, :, is an important character in MATLAB. The statement:

```
t=1:5
```

generates a row vector containing the numbers from 1 to 5 with unit increments. It produces:

```
t =
     1     2     3     4     5
```

... and also specify the increment

You can use increments other than one:

```
y=0:pi/4:pi
```

results in:

```
y =
     0    0.7854    1.5708    2.3562    3.1416
```

Negative increments are possible.

```
z=6:-1:1
```

gives:

```
z =
     6     5     4     3     2     1
```

You can form large vectors from small vectors by surrounding the small vectors with brackets, []. For example, entering the statement:

```
x=[t y z]
```

produces a vector which is just the concatenation of the three smaller vectors t, y and z.

Now lets create a simple time vector:

```
Ts=2e-3
To=2
t=0:Ts:To
```

The output is quite large since we have created a vector with 1001 elements. Press the ↑ (up-arrow) and append a semicolon to the last line to suppress the output. Note that MATLAB is case-sensitive and T0, T0 and T0 are different.

1.7 M-Files: Scripts and Functions

MATLAB starts in a command-driven mode; when you enter single-line commands, MATLAB processes them immediately and displays the results. MATLAB can also execute sequences of commands that are stored in files. MATLAB is thus an interpretive environment.

Files that contain MATLAB statements are called *M-files* because they have a `.m` file extension. An M-file consists of a sequence of normal MATLAB statements, which possibly include references to other M-files. You can create M-files using a text editor. M-files defined

Two types of M-files can be used: *scripts* and *functions*. *Scripts* automate long sequences of commands. *Functions* provide extensibility to MATLAB. They allow you to add new functions to the existing functions.


1.8 Scripts

When a *script* is invoked, MATLAB simply executes the commands found in the file. The statements in a script operate globally on the data in the workspace. Scripts are useful for performing analyses, solving problems, or designing long sequences of commands that become cumbersome to do interactively. Scripts defined

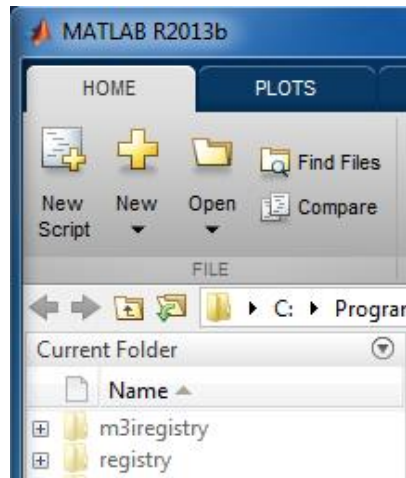
1.9 Functions

An M-file that contains the word `function` at the beginning of the first line is a *function*. A *function* differs from a *script* in that arguments may be passed, and variables defined and manipulated inside the file are local to the function and do not operate globally on the workspace. *Functions* are useful for extending MATLAB, that is, creating new MATLAB functions using the MATLAB language. Functions defined

1.10 Creating a Script

Click on the New Script icon () in the toolbar of the HOME ribbon:

Creating a new
M-file




MATLAB's internal M-file ASCII text editor will open. It has line numbers on the left and supports colour syntax highlighting.

Now type in:

```
Ts=2e-3;
To=2;
t=0:Ts:To;
```

M-file filename
restrictions

Click on the save icon () and give your M-file the name Lab2.m. When naming M-files, it is important to follow an 8.3 file notation, with no spaces, otherwise MATLAB cannot read them. You should also not use names which may be predefined functions; for example, do not save a script as `sin.m`.

Use F5 to Save and
Run a script

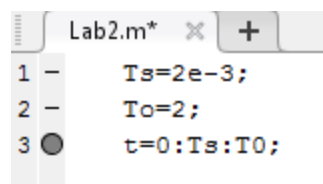
To execute your script, simply press the F5 key. This is the Debug | Save and Run shortcut key. If you Alt+Tab back to the MATLAB Command Window, you will see Lab2 on the command line showing that the Lab2 script was executed, but no other output – this is because we suppressed output using semicolons at the end of each line. To ensure that things are working, remove the last semicolon in your M-file and hit F5. You should now see the output. Add a semicolon back to the last line, as we are normally not interested in the output of such a trivial operation.

1.11 Debugging your Script

You can set breakpoints in your script to facilitate debugging. Just click on the dash (-) next to the line number where you want the breakpoint. You can toggle the breakpoint on and off by clicking. Change line 3 so that T_0 (T-small o) is replaced with $T0$ (T-zero):

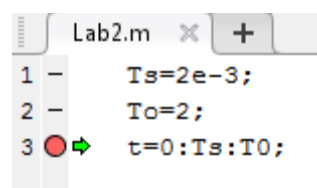
```
t=0:Ts:T0;
```

Click on the dash next to line 3:




Setting a breakpoint

to put a breakpoint on line 3. Press $F5$ and you should see a green arrow at the statement that is about to be executed:



Stepping through the script

You can use the icons in the toolbar or keyboard shortcuts to step into, over, etc. Press $F10$, which is the shortcut key for `Step`. MATLAB tries to execute line 3 (which results in an error) and shows a green down arrow indicating that there is a problem. You can click on the Quit Debugging icon () in the top right of the toolbar to terminate the debug session and see what is wrong.

The editor doesn't give you feedback on the error, but the Command Window does, in a red font:

```
Undefined function or variable 'T0'.
```

```
Error in Lab2 (line 3)
t=0:Ts:T0;
```

Errors appear in the Command Window

When things don't appear to be working, remember to check for any messages in the MATLAB Command Window.

Correct the error by reverting $T0$ back to T_0 , and toggle the breakpoint off.

1.12 Clearing the Workspace

Make a new line 1 (just type at the start of line 1) and add the statement:

Clearing the workspace

```
clear all;
```

This will clear the workspace of all variables and provide a “clean slate” for your script. Press F5 and observe the MATLAB Workspace pane. It should have cleared away all the previous variables you had defined and created just the variables that are defined in your script file.

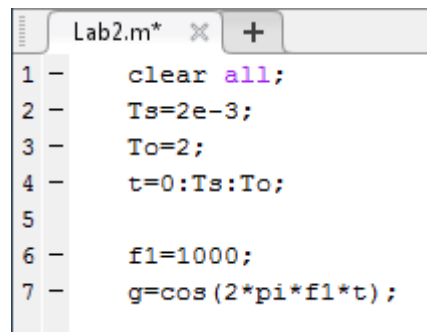
1.13 Using Vectors in Functions

Passing vectors as a single parameter to a function, and having functions return vectors, is a powerful feature of MATLAB. After line 4, type:

Functions take vectors and return vectors

```
f1=1000;
g=cos(2*pi*f1*t);
```

Leave a space so that your script looks like:



```
1 - clear all;
2 - Ts=2e-3;
3 - To=2;
4 - t=0:Ts:To;
5 -
6 - f1=1000;
7 - g=cos(2*pi*f1*t);
```

Notice that we are adhering to good programming practice. Blank lines are used to separate portions of code, and constants are given meaningful names (*f1* in this case) so they are easy to change later on. The constant *pi* is already pre-defined by MATLAB for us.

This script should have created a sinusoid in the variable *g*. To graphically see this, we will use MATLAB’s powerful graphing capability.

1.14 Plotting

Add line 9 (leave line 8 blank) with the statement:

```
plot(t,g);
```

Plotting is performed with the `plot` function

This statement will make a plot of one vector versus another vector (in this case a plot of g vs. t). MATLAB handles the rest for us. Press F5 and MATLAB opens up a new window titled Figure 1. We should see a sinusoid...

...but we don't, we only see a straight line. To check what is going on, we Alt+Tab to the Command Window and type g to look at the contents of the g vector. Sure enough, the `plot` command is working correctly, but our g vector only has elements of value 1.

Realizing that our mistake is setting up a time vector so that individual elements (or time samples) correspond exactly to the peaks of the 1000 Hz sinusoid, we change line 2 to read:

```
Ts=2e-4;
```

and press F5. We should see a sinusoid...

...but we don't, we see a filled-in rectangle. Thinking a bit more, we realise that the plot is working, but we are looking at 2000 cycles of a 1000 Hz waveform! We decide to graph just 2 cycles, so we change line 3 to read:

```
To=2e-3;
```

and press F5. We should see a sinusoid...

...but we don't, we see a piece-wise linear waveform. MATLAB just joins the dots, and it appears as though our time samples are not spaced "fine enough" to capture the smoothness of a real sinusoid. Change line 2 to read:

```
Ts=2e-5;
```

and press F5. We see a sinusoid at last!

Graph smoothness depends on the sample spacing

Remember that we need to set up our vectors carefully, and that to obtain smooth looking waveforms, we need to "sample" quite fast.

1.15 Help

You can press `F1` at any time to invoke MATLAB's help. Sometimes a more convenient way to get help is to use the `help` function, which is invoked from the Command Window. For example, type:

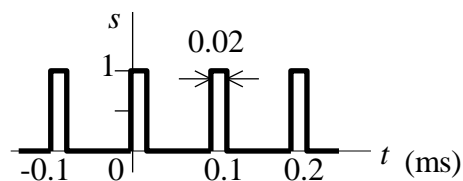
Getting help on a
MATLAB function

```
help square
```

and press `Enter`. MATLAB returns with help on the `square` function (which is really just an M-file function). Many functions can take a variable number of arguments, and it is important that you read about the various options of calling a function. Functions can also return numbers, vectors, arrays of vectors, and even plot things automatically.

1.16 Plotting on a New Figure

By appending code to your `Lab2.m` file, see if you can create a new vector called `s` which contains a 20% duty cycle, 10 kHz square wave that goes between 0 and 1 on the vertical scale:



You should read the help on the `square` function carefully. Conform to good programming practice by defining the frequency of the square wave as:

```
fc=10e3;
```

Plot the newly created `s` vector by adding the lines:

Plotting on a new
figure

```
figure;  
plot(t,s);
```

to the end of your script. The `figure` command will create a blank figure, which is then acted upon by the next `plot` command.

Make sure you “sample” fast enough to capture the edges of the square wave – you may have to change `Ts` to `2e-6`.

1.17 Changing Figure Numbers

If you run your script repeatedly by pressing F5 a few times, you will see that the `figure` function creates a new figure all the time, without closing old ones. We can pass an argument to the `figure` function to prevent this from happening. Change the line with `figure` so that it reads:

```
figure(2);
```

Changing figure numbers

Press F5 and see that the square wave is now graphed on Figure 2, and no new figures are created.

1.18 Clearing Old Figures

There are still some old figures open that we would like to close. MATLAB can do this for us, and it is good practice to close all figures before we begin running a script to avoid confusion – we like to start with a “clean slate”. On line 2, add the statement:

```
close all;
```

Closing old figures

and press F5. This closes all open figures. Add the line:

```
figure(1);
```

before your first plot statement. Although not necessary, since the first plot is put on Figure 1 by default, it makes your script more readable.

1.19 Script Comments

Conforming to good programming practice means that you should comment your script appropriately. This lets others (and you) understand what the script is intending to do.

MATLAB comments are lines that begin with the percent, `%`, sign.

Adding script comments

Go through your script file and add some simple but appropriate comments.

M.16

Your script should now look something like:

```
% =====  
% My first MATLAB M-file  
% by PMcL  
% =====  
  
% clear everything  
clear all;  
close all;  
  
% sample time  
Ts=2e-6;  
% time window  
To=2e-3;  
% time vector  
t=0:Ts:To;  
  
% a sinusoid  
f1=1000;  
g=cos(2*pi*f1*t);  
  
figure(1);  
plot(t,g);  
  
% a square wave  
fc=10e3;  
s=(square(2*pi*fc*t,20)+1)/2;  
  
figure(2);  
plot(t,s);
```

The script is looking good, but it's time to fix up those figures – we can't see the square wave particularly well, and we'd like to graph the two functions on the same figure for comparison purposes.

1.20 Changing the Plot Axes

Get help on the `axis` function (not the `axes` function, which is very different). The `axis` function is applied after the `plot` function. Add lines to your script so that each figure is plotted from 0 to $2e-3$ horizontally and -2 to 2 vertically. Don't forget that MATLAB can take vectors as arguments to functions. For example:

```
TimeAxes=[0 2e-3 -2 2];
...
axis(TimeAxes);
```

Changing the plot axes

1.21 Adding Axis Labels

All plots need appropriate axis labels. Use the following lines as a guide and add labels to each of your plots:

```
xlabel('t (s)');
ylabel('g(t)');
```

Adding labels to a plot

1.22 Adding a Plot Title

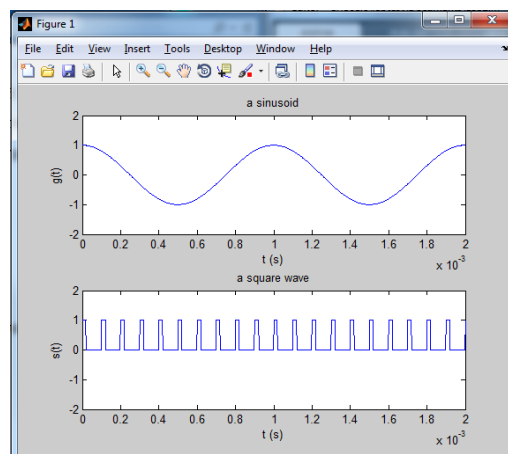
All plots need an appropriate title. Use the following line as a guide and add a title to each of your plots:

```
title('a sinusoid');
```

Adding a title to a plot

1.23 Creating Subplots

To get plots to lie on one figure, we use the `subplot` function. Type `help subplot` and make your plots appear on just Figure 1, with the sinusoid on top, as shown below:



Creating subplots

1.24 Functions

The best way to learn about functions is to look at an example. Suppose an M-file called `mean.m` contains the following statements:

An example
MATLAB function
showing the internal
structure

```
function y = mean(x)
% MEAN Average or mean value.
% For vectors, MEAN(x) returns the mean value.
% For matrices, MEAN(x) is a row vector
% containing the mean value of each column.

[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;
end
```

The existence of this file defines a new function called `mean`. The new function `mean` is used just like any other MATLAB function. For example, if `z` is a vector of the integers from 1 to 99:

```
z = 1:99;
```

the mean value is found by typing:

```
mean(z)
```

which results in:

```
ans =
    50
```

Here are some details of `mean.m`:

- The first line declares the function name, the input arguments, and the output arguments. Without this line, the file is a *script* file instead of a *function* file. The first line of a function M-file must declare the function...
- The first few lines document the M-file and display when you type `help mean`. ...then comes the help comments...
- The variables `m`, `n` and `y` are local to `mean` and do not exist in the workspace after `mean` has finished (or, if they previously existed, they remain unchanged). ...then comes the function body
- It was not necessary to put the integers from 1 to 99 in a variable with the name `x`. In fact, we used `mean` with a variable called `z`. The vector `z` that contained the integers from 1 to 99 was passed or copied into `mean` where it became a local variable named `x`. Function parameters are formal variables and are replaced with actual variables when called

You can create a slightly more complicated version of `mean`, called `stat`, that also calculates standard deviation:

```
function [mean,stdev] = stat(x)
    [m,n] = size(x);
    if m == 1
        m = n;
    end
    mean = sum(x)/m;
    stdev = sqrt(sum(x.^2)/m - mean.^2);
end
```

An example MATLAB function that returns a vector

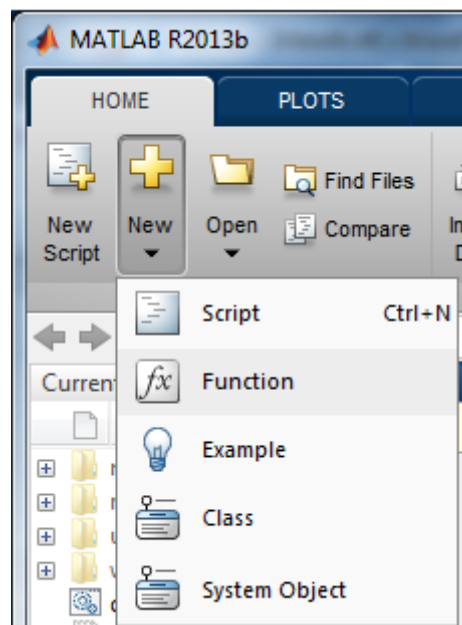
`stat` illustrates that it is possible to return multiple output arguments.

1.25 Creating a Graph Function

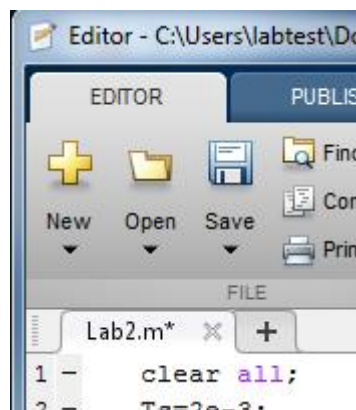
Create MATLAB functions to remove duplicate code

Observing good programming practice, we now look to remove duplicate code from our script and create a function instead. Looking at our script, we see that the plotting code appears to repeat, except with different parameters passed to the various plotting functions. Let's create a `graph` function that will do the plotting for us and make our code easier to read, use and maintain.

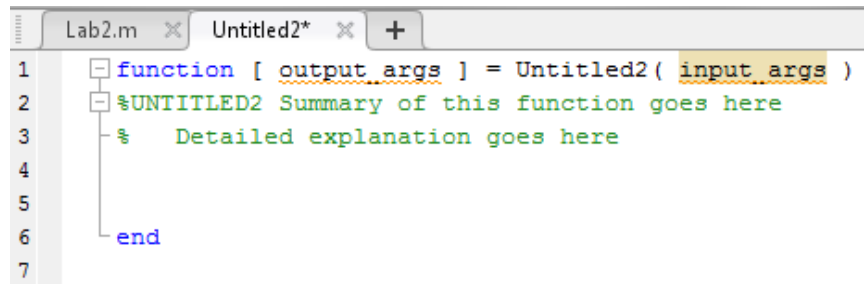
Functions are just M-files, but MATLAB will create a function template if you click on the New→Function menu item in the toolbar of the HOME ribbon:



MATLAB creates a function template in the editor, that you can now customise. A similar command exists in the toolbar of the EDITOR ribbon of the M-File editor:



You will get a function template that looks like:



```

1  function [ output_args ] = Untitled2( input_args )
2  %UNTITLED2 Summary of this function goes here
3  %   Detailed explanation goes here
4
5
6  end
7

```

Now edit it to create a new M-File function called `graph.m` that looks like:

```

function graph(Fg,Sub,x,y,Ax,XL,YL,TL)
%GRAPH Makes a subplot on a particular figure.
%   GRAPH(Fg,Sub,x,y,Ax,XL,YL,TL) makes a subplot
%   on a figure using the following parameters
%   Fg specifies the figure number.
%   Sub specifies the subplot coordinates.
%   x specifies the horizontal vector.
%   y specifies the vertical vector.
%   Ax specifies a vector containing the axes.
%   XL specifies the x-axis label.
%   YL specifies the y-axis label.
%   TL specifies the subplot title.

end

```

Notice that the function has no output arguments. Fill out the rest of the function to perform the desired action of plotting on a particular subplot (use your existing plotting code in your `Lab2.m` script as a guide). Make sure that you save the function M-file with the same name as the function, i.e. as `graph.m`. After your function is created, your `Lab2.m` script should be modified to use it, for example:

```
graph(1,211,t,g,TimeAxes,'t(s)','g(t)','a sinusoid');
```

In the Command Window, test the help for your function by typing `help graph`. Test your new function by running your modified `Lab2.m` script.

1.26 Vector Operations

Vectors operations
are intuitive and
easy

One of MATLAB's attractions is that it operates on vectors in a natural manner. For example, the line:

```
g=cos(2*pi*f1*t);
```

takes a linear input vector, t , and creates a sinusoidal output vector, g . We don't have to program any loops to iterate through each individual element of the t vector to create the g vector – “MATLAB does it behind the scenes”. This “abstraction” of operations into vector operations makes for very intuitive code, like the line above. But sometimes we have to be careful, as we will see next.

Let's create a new waveform, called gs which is to be obtained by multiplying the signals g and s . Add the following code to your `Lab2.m` script:

```
% a sampled sinusoid
gs=g*s;
graph(1,313,t,gs,TimeAxes,'t (s)','gs(t)','gs');
```

Also, change your other `graph` functions so they have subplot numbers of 311 and 312 (we now want to have 3 subplots, in one column). Press F5 to run your code.

The third subplot does not appear, and on investigating the Command Window, we see why – we have an error:

```
Error using *
Inner matrix dimensions must agree.

Error in Lab2 (line 32)
gs=g*s;
```

It appears as though the error is occurring in a hidden function called `'*'`. Type `help *` to see what this function does. It turns out that the `*` we have been using for multiplication is actually mapped to the MATLAB function that handles matrix multiplication. Since a vector is a special case of a matrix, `*` can handle vector multiplication too.

So what is the line:

```
gs=g*s;
```

actually asking MATLAB to do?

To illustrate vector multiplication, let's consider two row vectors:

$$\mathbf{x} = [a \ b \ c], \quad \mathbf{y} = [d \ e \ f]$$

There are two types of vector product, the inner product and the outer product. The inner product is familiar from physics, and is just the dot product, which results in a scalar:

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{xy}^T = \underset{1 \times 3}{[a \ b \ c]} \underset{3 \times 1}{\begin{bmatrix} d \\ e \\ f \end{bmatrix}} \underset{1 \times 1}{=} ad + be + cf$$

The vector inner product

The outer product creates a matrix:

$$\mathbf{x} \otimes \mathbf{y} = \underset{3 \times 1}{\begin{bmatrix} a \\ b \\ c \end{bmatrix}} \underset{1 \times 3}{[d \ e \ f]} = \underset{3 \times 3}{\begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}}$$

The vector outer product

In general, we need the number of columns of the first vector to match the number of rows of the second vector, otherwise the multiplication is undefined. We now understand the error “Inner matrix dimensions must agree.” We are asking MATLAB to perform:

$$\mathbf{x} * \mathbf{y} = [a \ b \ c][d \ e \ f] = \text{undefined}$$

What we really wanted to do was a multiplication on an element-by-element basis. That is, to multiply two “signals” together, we need to multiply corresponding values at particular time instants. We wanted:

$$\mathbf{x} * \mathbf{y} = [a \ b \ c][d \ e \ f] = [ad \ be \ cf]$$

The vector element-by-element product

M.24

MATLAB supports this operation, for all types of operator, and refers to them as array operations, which perform the operation on an element-by-element basis. To use an array operator, we prefix the normal operator with a period, `.`

Array operations are prefixed with a period

Change the multiplication line to:

```
gs=g.*s;
```

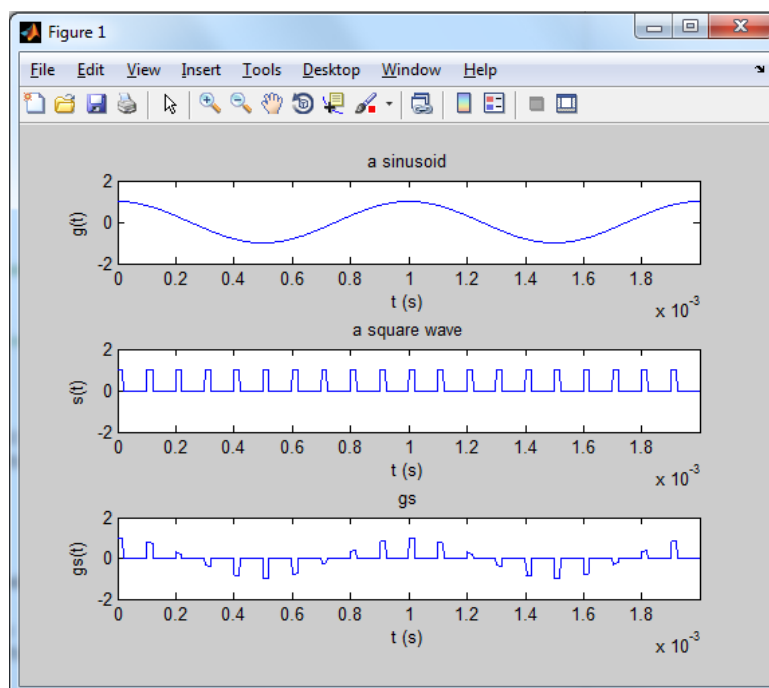
and press `F5`. We now see the result of multiplying the sinusoid with the square wave in the third subplot.

As another example of the `.` operator, suppose we wanted to change the original `g` signal to a sinusoid that has been squared. To raise a number to a power, we use the hat, `^`, operator. Since we want to raise a vector to a power, we need to use the `.^` operator:

```
g=cos(2*pi*f1*t);  
g=g.^2;
```

You can try this and press `F5` to see the result. Remove the line `g=g.^2` when you are satisfied that you understand the `.` operator.

The output of your script should now look like:



Your script should now look something like:

```

% =====
% My first MATLAB M-file
% by PMcL
% =====

% clear everything
clear all;
close all;

% sample time
Ts=2e-6;
% time window
To=2e-3;
% time vector
t=0:Ts:To;

TimeAxes=[0 2e-3 -2 2];

% a sinusoid
f1=1000;
g=cos(2*pi*f1*t);
    graph(1,311,t,g,TimeAxes,'t (s)', 'g(t)', 'a sinusoid');

% a square wave
fc=10e3;
s=(square(2*pi*fc*t,20)+1)/2;
    graph(1,312,t,s,TimeAxes,'t (s)', 's(t)', 'a square wave');

% a sampled sinusoid
gs=g.*s;
    graph(1,313,t,gs,TimeAxes,'t (s)', 'gs(t)', 'gs');

```

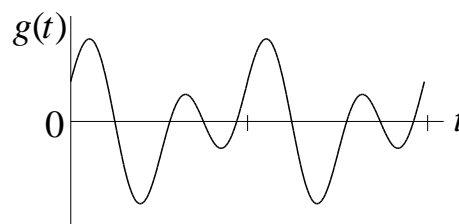
1.27 The FFT

The spectrum is a picture of the component sinusoids of a periodic waveform

We are now ready to embark on frequency-domain operations. We would like to see the spectrum of each of our three signals – i.e., we would like to see just exactly what sinusoids are used to make up each of our periodic signals. The spectrum is just a graphical way to illustrate this information – it graphs the amplitudes and phases of the component sinusoids versus the frequency.

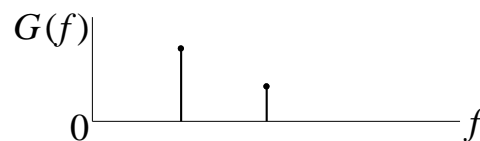
For example, a signal consisting of two sinusoids in the time-domain may appear as:

A time-domain waveform...



In the frequency-domain, it is represented as a spectrum:

... and its spectrum



The above spectrum tells us that the signal is composed of just two sinusoids. If the graph had a scale, we could read off the component frequencies and amplitudes (and phases).

To convert a time-domain vector into its frequency-domain representation, (that is, to get the spectrum of a signal), we use MATLAB's `fft` function.

Add the following line just after you plot the `g` vector:

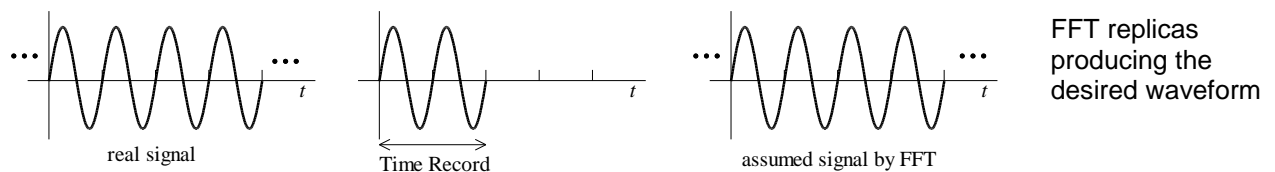
```
G = fft(g);
```

This creates a new vector, `G`, that holds the spectrum. By convention, we use capital letters to represent the frequency-domain. To graph `G`, we need to generate a special frequency vector, one that is “in line” with the samples generated by the `fft` function.

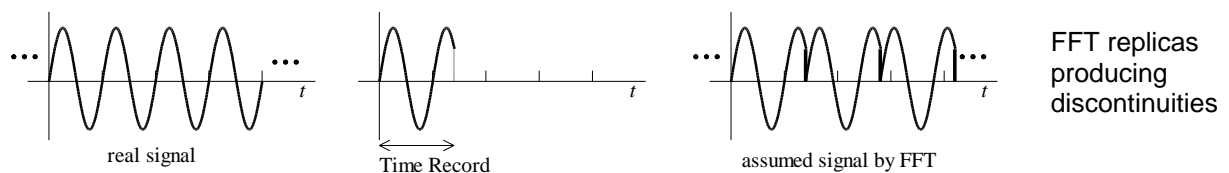
The `fft` function is used to get the spectrum in MATLAB

1.28 FFT Samples

The FFT operates on a finite length time record, but assumes that this time record is exactly one period of an infinitely long periodic signal. With the waveform shown below, where an integral number of periods fits **exactly** within the time record, the infinitely long signal assumed by the FFT is correct.



However, if we create a time record so that an integral number of periods of the waveform do **not** quite fit into it, then discontinuities are introduced by the replication of the time record by the FFT over all time:



This effect is known as *leakage*, and the effect in the frequency-domain is very apparent. For the case of a single sinusoid as shown, the normally thin spectral line will spread out in a peculiar pattern.

In a practical setting, such as measuring signals with a DSO, we get around this problem by *windowing* the time-domain waveform before taking the FFT.

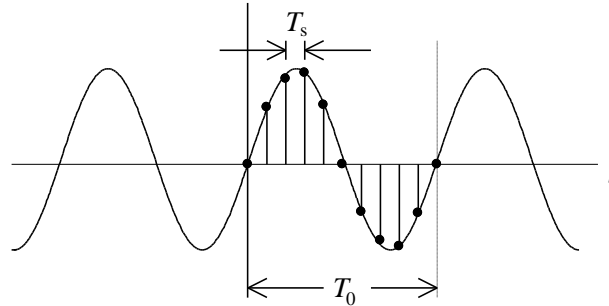
In a theoretical setting, we have complete control over the time record (or time window), so we should adjust our vectors to avoid spectral leakage. The simplest way is to ensure that our time window encapsulates **exactly** an integral number of periods of the time-domain signal. We can do this in a theoretical setting because the time-domain signal is known and generated by us – in a laboratory the time-domain signal is normally not known exactly.

For simulation using MATLAB, we need to ensure that an integral number of waveform periods fit inside the time window

1.29 Time-Domain Samples

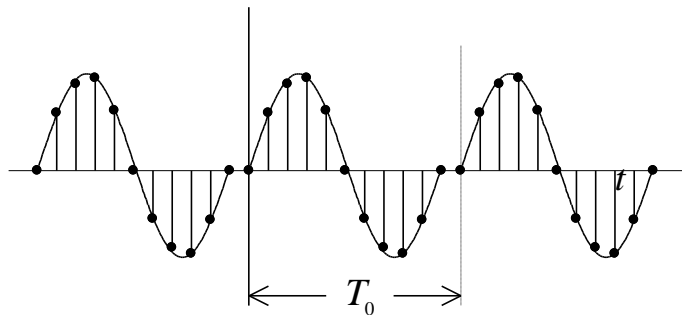
To avoid FFT problems we should generate signal vectors that contain exactly an integral number of periods. Consider taking samples of a simple sinusoid at a rate $f_s = 1/T_s$:

Samples of a sinusoid...



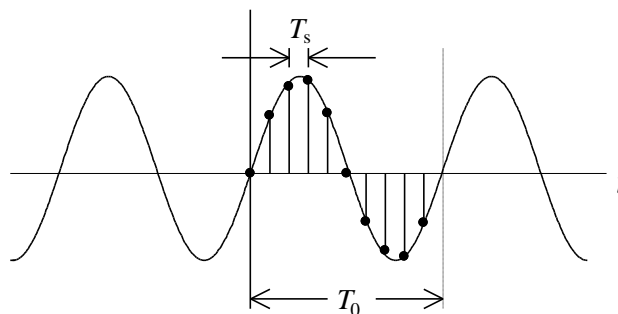
At first glance, it appears that we have sampled exactly one period of the waveform. But if we were to take these samples and repeat them, we would generate the following waveform:

... and the corresponding periodic extension



which is not quite right – the period has been extended and there is a discontinuity between one “sinusoid” and the next. On reflection, we realise that the *last* point in the original sampling scheme, is actually the *first* point of the next period. The way we need to sample is thus:

Correct sampling of a waveform over one period involves not repeating the first sample



That is, we need to take samples up to, but not including, T_0 . We also need to ensure that $NT_s = T_0$, with N an integer. The best way to do this is to modify the creation of our time vector:

```
% sample time
Ts=2e-6;
% number of samples
N=1000;
% time window
To=N*Ts;
% time vector
t=0:Ts:To-Ts;
```

Creation of a suitable time vector when using the `fft` function

1.30 Frequency-Domain Samples

The `FFT` function returns spectrum samples that are spaced in frequency by an amount equal to the “fundamental frequency”. With an FFT, in all cases, the “period” of the waveform is whatever the time window happens to be, regardless of whatever signal is actually in the time window. The “fundamental frequency” is therefore equal to the inverse of the time window. That is, the FFT function really knows nothing about the “fundamental frequency” of the actual signal, but it assumes that the samples provided in the time window are taken over exactly one period of the waveform. We therefore have:

The `fft` function assumes the time window is the period

```
f0=1/To;
```

The sample spacing of the `fft` output

This is the frequency spacing between FFT samples. Since the FFT returns N samples of the spectrum, the maximum frequency must be $(N-1)f_0$. This is similar to the consideration of time-domain samples – the “last” sample at Nf_0 is really the first sample of the next period of the FFT output (the FFT output is theoretically periodic and infinite in extent). Since $NT_s = T_0$ then $Nf_0 = f_s$. Therefore, the FFT output sample frequencies can be generated from:

```
fs=1/Ts;
f=0:f0:fs-f0;
```

The sample range of the `fft` output

Add these lines at appropriate locations in your `Lab2.m` script.

1.31 Plotting a Magnitude Spectrum

We are now in a position to plot the spectrum. Add the following line to your script, straight after you take the FFT:

```
figure(2);
plot(f,G);
```

Observe the output of Figure 2 and the Command Window. There is a warning:

```
Warning: Imaginary parts of complex X and/or Y
arguments ignored.
```

We have asked MATLAB to plot G , a vector of complex numbers (representing phasors), versus frequency. We can't graph a complex number versus a real number on a 2D graph. What we normally do is graph the *magnitude* spectrum separately to the *phase* spectrum. The fft output is complex...

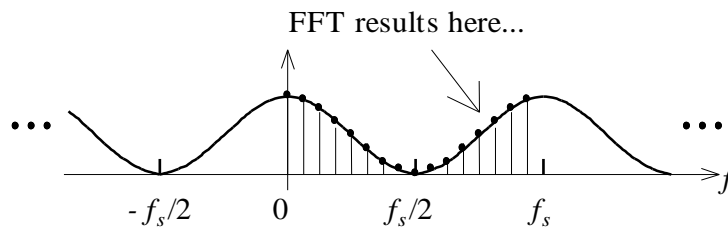
Change your code so that we plot the magnitude spectrum only:

... so we graph magnitude and phase separately

```
G=fft(g);
Gmag=abs(G);
figure(2);
plot(f,Gmag);
```

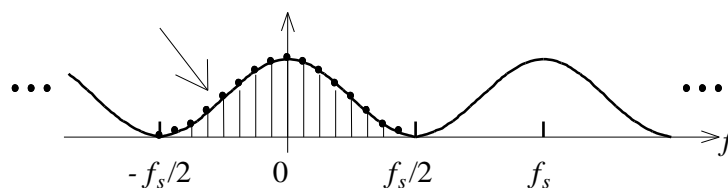
The resulting plot doesn't look quite right. We are expecting a double-sided spectrum (showing positive and negative rotating phasors), but we have generated a positive frequency vector and the FFT appears to return only positive frequencies. But since the FFT is periodic, the upper half is actually the same as the first part of the negative-side of the spectrum:

The FFT is periodic, so the upper half of the fft output...



...are the same as here.

... is the same as the first part of the negative-side of the spectrum



MATLAB knows that we would like to shift the upper half of the FFT output down to take the place of the lower half (i.e. swap the two halves) so there is a function which does this for us. Modify the `fft` line of code with the following:

```
G=fftshift(fft(g));
```

The `fftshift` function swaps the two halves of a vector

and modify the frequency vector so it reads:

```
f=-fs/2:fo:fs/2-fo;
```

and press F5 to run. The spectrum now looks right, but it has the wrong amplitude. This is a quirk of the FFT algorithm, and to understand it fully, you need to read Appendix A – The Fast Fourier Transform. Checking the FFT – Quick Reference Guide, we realize that we are dealing with Case 4, so to get correct amplitudes we need to divide the `fft` output by `N`. Change your `fft` line of code to the following:

```
G=fftshift(fft(g))/N;
```

The `fft` function needs to be scaled to get the correct amplitude

and press F5 to run. We now get the correct amplitude (in this case we know it is correct because the phasor magnitude is just $A/2 = 1/2 = 0.5$).

To make the plot look better, we could use the MATLAB `stem` function instead of `plot`:

```
stem(f,Gmag);
```

The `stem` function can be used in place of the `plot` function

To see the effect of leakage, change the number of samples in a “period” by changing:

```
% number of samples
N=700;
```

and pressing F5 to run. Try a few values and then return `N` to 1000.

Modify your script file so that it plots (using `plot`, not `stem`) the spectra of each of the three time-domain signals on Figure 2, with each subplot using the same axes.

Your script should now look something like:

```

% =====
% My first MATLAB M-file
% by PMcL
% =====

% clear everything
clear all;
close all;

% sample time & sample rate
Ts=2e-6;
fs=1/Ts;

% number of samples
N=1000;

% time window & FFT sample spacing
To=N*Ts;
fo=1/To;

% time and frequency vectors
t=0:Ts:To-Ts;
f=-fs/2:fo:fs/2-fo;

TimeAxes=[0 To-Ts -2 2];
FreqAxes=[-fs/2 fs/2-fo 0 0.5];

% a sinusoid
f1=1000;
g=cos(2*pi*f1*t);
    graph(1,311,t,g,TimeAxes,'t (s)','g(t)','a sinusoid');
G=fftshift(fft(g))/N;
Gmag=abs(G);
    graph(2,311,f,Gmag,FreqAxes,'f (Hz)','G(f)','sinusoid spectrum');

% a square wave
fc=10e3;
s=(square(2*pi*fc*t,20)+1)/2;
    graph(1,312,t,s,TimeAxes,'t (s)','s(t)','a square wave');
S=fftshift(fft(s))/N;
Smag=abs(S);
    graph(2,312,f,Smag,FreqAxes,'f (Hz)','S(f)','square spectrum');

% a sampled sinusoid
gs=g.*s;
    graph(1,313,t,gs,TimeAxes,'t (s)','gs(t)','gs');
Gs=fftshift(fft(gs))/N;
Gsmag=abs(Gs);
    graph(2,313,f,Gsmag,FreqAxes,'f (Hz)','Gs(f)','sampled spectrum');

```

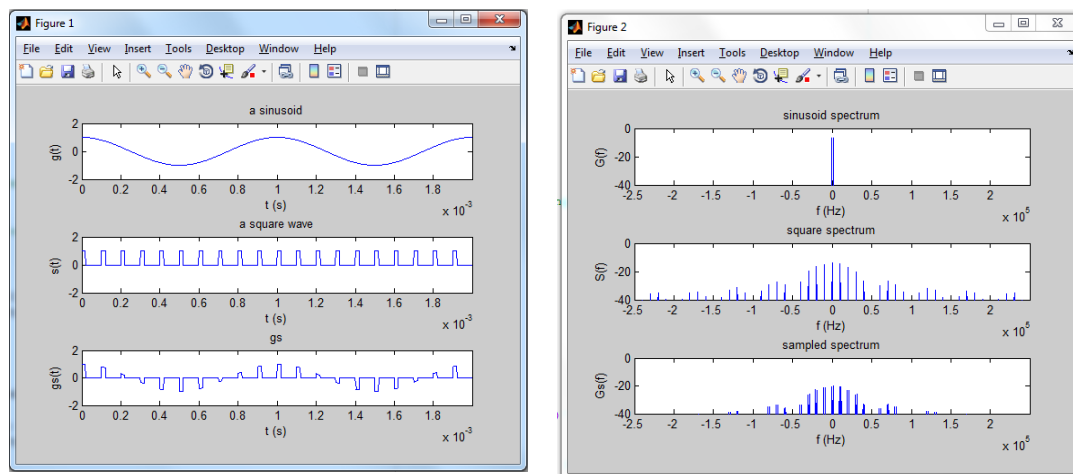
One last thing we can do with our magnitude spectra is to graph them on a log scale, i.e. on a dB scale. Engineers are normally interested in power, not amplitude, and it greatly extends the range of our vertical scale. Modify your script so that the spectrum magnitude is converted to decibels. For example:

```
Gmag=20*log10(abs(G));
```

Notice that MATLAB uses `log10` for the common logarithm (base 10), and uses `log` for the natural log (base e).

You will also need to change the vertical scale of the plots. Change the vertical range to -40 dB to 0 dB and run your script.

The output of your script file should look like:



1.32 Finishing Your MATLAB Session

Make sure you save all your M-files to either the network, a USB drive, or email them to your email account.

You are now aware of MATLAB's basic functionality, and are well on the way to completing the pre-lab work for Lab 2.

MATLAB is a very large software package and has many more advanced features that you will come to use later in the subject, as well as in other subjects and in industry. MATLAB has many more advanced features