# Lab 1

# Introduction to MATLAB and Moving Average

## 0. Preface

The first part of this first laboratory exercise aims to familiarize you with MATLAB commands, syntax and programming. You do not need to submit anything for section 1.

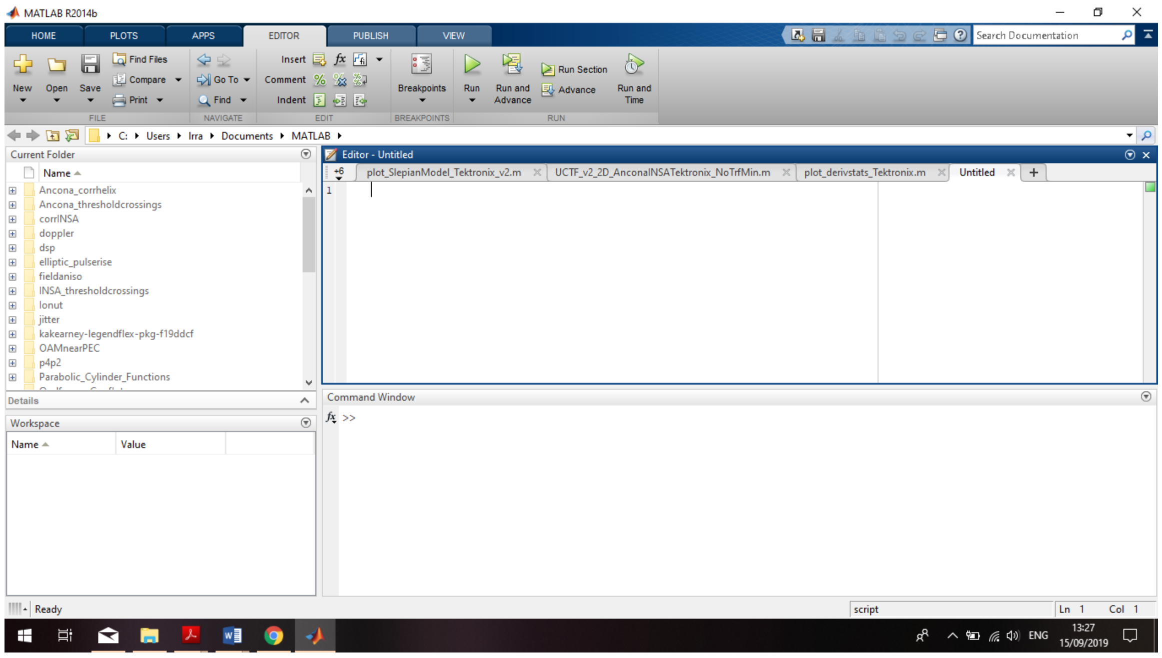
In the second and third parts of this lab, you will study the moving average filter and convolutions. Your lab report should answer ALL questions in EITHER section 2 OR section 3. This mean you choose your own (mis)adventure.

## 1. MATLAB Introduction

MATLAB (develop by MathWorks Inc.) is a powerful high-level programming language that takes the pain away from working with data of various kinds. It is a command-line driven environment, meaning that there is no need to wait for your program to compile, no need to worry about addressing the wrong areas of memory, no need to stress over cross-platform incompatibilities. You write and run as you go along. However, you can also place the commands in a file (called a MATLAB script, with file extension .m) to run from the command line, so that you do not need to retype them in case of longer sets of commands forming a MATLAB program.

MATLAB has become a de facto industry standard for prototyping algorithms and much else. Because of its strong graphical user interface and specialist toolboxed developed for several application areas in science and engineering. A new updated version is released twice a year, e.g., R2019a and R2019b. So fire up MATLAB and enjoy the wonderful little numbers below!

After you start up MATLAB (e.g., by double-clicking the icon on the Desktop, or from invoking from the command line), you should see something similar to the following screen shot (since MATLAB gets updated, details of the layout may have become different; also, the menus may look different depending on the set of packages or modules installed); student or professional license, etc.:



For now, we will focus on command-line instructions, which you type in the bottom-right window, suprisingly and ingeneously called “Command Window”.

1.1 MATLAB is quite user friendly. If you need help at any time, you can type the command “help” at the MATLAB prompt (>>), e.g. “>> help print” will give explanation, within the MATLAB window, about generting output files for a graph. There are other useful functions as well, e.g. “doc”, with more background and details, which opens in your browser and requires internet connection. Follow the code below by copying each command, shown here without >> or starting with % (which indicates a comment line), into the MATLAB command-line window.

```
% <--- This is a comment in MATLAB
% to get help just type:
help
```

```
help help
help win

% This will show you all the toolboxes available for your use.
% A toolbox contains functions like cos, fft, filter.  To see
% what functions are in
% a toolbox type 'help <toolbox>'.
help elfun
help signal

% To get help on a particular function type 'help <function>'.
help angle
help wavread
help fft

% You will find all functions in MATLAB written in uppercase
% letters. This is only to make it easier to read, so don't
% try calling the function SIN(X), or FFT(X).

% To answer why you must go on:
why

% MATLAB includes the search function 'lookfor'.
% The following returns functions that have something to do with
% 'average'. You can only search on one string.
lookfor average
help lookfor
```

1.2 Variables can be assigned on the fly, anywhere and anytime, so you don’t need to declare them first such as in languages like C. However, it is still good practice to declare their initial values at the top of the program, for easy reference. Furthermore, they don't have to be typecast as an integer, float, or char. They can even change type whenever you fancy (although that is not really recommended if you can avoid it). The only rule is that a variable name can not start with a number, e.g., '5a' is not allowed.

```
a = pi/6    % assign the number pi/6 to variable a
b = sin(a)  % compute the sin of a and put it in b

% Note that sin(30) != sin(pi/6). MATLAB works in radians
% so you must convert all degrees to radians!!
% sin(30 degrees) = sin(30*pi/180) = sin(pi/6)
% Even though a and b are assigned I can make them anything
% I want.
a = 'hello'

% Let's say we have a triangle and the two shortest sides
% are lengths 3 and 4. What is the length of the hypotenuse?
% Pythagoras: h^2 = a^2 + b^2
side_1 = 3
side_2 = 4
```

```
hypot = sqrt(side_1^2 + side_2^2)

% Notice how after every command you enter MATLAB responds.
% This can get annoying when your future programs are large.
% To keep MATLAB from writing anything to the screen end the
% lines you don't want to see with a semicolon:
side_1 = 3;
side_2 = 4;
hypot = sqrt(side_1^2 + side_2^2)
```

MATLAB does not distinguish between user-defined constants and variables.

1.3 The real power of MATLAB is its ability to calculate with vectors and matrices. Indeed, "MATLAB" is short for "MATrix LABoratory." This means everything is discretized – just the way you like it in digital signal processing. If you want a continuous or analog function, it means you have to represent it in a discrete sampled and quantized format. Some examples:

```
b = [1; 2; 3]      % make a column vector
b = [1, 2, 3]      % make a row vector
b = [1 2 3]        % the commas can be left out too, for a row
                   % vector
c = b*2            % multiply each element of a vector by 2

% A vector can be multiplied by another vector:
d = [2, 3, 4]
b.*d

% The dot here is very important! It tells MATLAB to multiply the
% components element-by-element. So, b.*d = [1*2, 2*3, 3*4]

% We can also multiply the vectors to obtain a matrix:
b'*d

% This tells MATLAB to compute the outer (or dyadic) product
%              [1]             [1*2   1*3   1*4]
%      b'*d = |2|*[2,3,4] = |2*2   2*3   2*4|
%              [3]             [3*2   3*3   3*4]
% The apostrophe (') says transpose-and-complex-conjugate the
% preceding vector, which means
% b goes from a 1 row by 3 column vector to a 3 row by 1 column.
% Multiplying a 3x1 * 1x3 = 3x3.

% If we instead do
b*d'

% This means the inner (or scalar) product
%                      [2]
%      b*d' = [1,2,3]*|3| = 1*2 + 2*3 + 3*4 = 20
%                      [4]
% Multiplying a 1x3 * 3x1 = 1x1, as you know from matrix theory.
```

```
      % Row vectors are 1xn matrices, having one row and n columns.
      % An nxm matrix has n rows and m columns.
      b = [1,2,3,4; 5,6,7,8; 9,10,11,12]  % Create a 3x4 matrix

      % Now b is a 3x4 matrix, where each element in a row is separated
      % by a comma (,) and each column is separated by a semicolon (;).

      % The semicolon for separating rows can be omitted by using a
      % carriage return instead, which treats the elements as being on
      % separate rows.

      % A constant can be used to be added or multiplying an entire
      % matrix:
      b*5                  % multiply every element by 5
      d = b+5              % add 5 to every element; assign the result to
d

      % Some quick ways to create uniform vectors and matrices:
      t = [0:9] % create a 10-element integer vector 0,1,2,…, 9
      t1 = [0:0.1:9.9] % 1x100 vector from 0 to 9.9, incremented by 0.1
      t2 = linspace(0,9.9,100) %does the same thing as t1
      t = [1:1:10;1:2:20] % a 2x10 matrix, second row incremented by 2,
                          % which overwrites the previous format and
                          % values of t defined earlier

      % You can check the value of a specific element of t by using
      % the indices of that element:
      t(1,4)       % Value of t in first row, fourth column
      t(4,1)       % Oops… t only has 2 rows, not 4.
      t(0,1)       % Oops… Indices in MATLAB always start at 1, not 0!
                   % Remember this when you use counters that start at 0
                   % as indices of a matrix (e.g., in loops): you may
                   % have to add 1 before you can use them as such.

      % To find the values in an entire column use the colon (:).
      t(:,4)       % Return values for all rows in the fourth column
      t(2,:)       % Return values of second row for all columns

      % Remember that MATLAB uses two rules for matrices:
      % 1. Indexing starts at 1, not 0.
      % 2. Matrix sizes are specified as rows by columns.

      % Other helpful functions for matricies can be found from
      help elmat
```

1.4 A keen thing about MATLAB is that it is a high-level scripting language that makes debugging fast and easy. As you create and assign variables, MATLAB keeps those variables in memory until you explicitly clear them or exit the program. At this point, if you have run all the code above, you should have several variables in memory. You can see them in the window

"Workspace" in the screenshot above, or by typing from the command line to get an output list:

```
      % To see what you have in memory:
      who          % List variables in memory.

      % You should have a list like:
      %Your variables are:
      %a        b        d        hypot    side_2
      %ans      c        h        side_1   t

      % ans contains your last mathematical result

      % To get more detailed information you can use the command
'whos':
      whos

      %  Name         Size          Bytes Class            Attributes
      %  a            1x5           10    char array
      %  ans          1x1            8    double array
      %  b            3x4           96    double array
      %  c            1x3           24    double array
      %  d            3x4           96    double array
      %  h            1x1            8    double array
      %  hypot        1x1            8    double array
      %  side_1       1x1            8    double array
      %  side_2       1x1            8    double array
      %  t            2x10         160    double array
      %  t1           1x100        800    double array
      %  t2           1x100        800    double array

      % Grand total is 257 elements using 2026 bytes

      % This whos command includes size and memory information.
      % Here we can see that t is a 2x10 matrix of 160 bytes, of class
      % double; a is a 1x5 matrix of class char -- which means that it
      % contains letters. At the bottom we see that all the variables
      % in memory take up 2026 bytes.

      % To clear the workspace of specific variables, use 'clear':
      clear a h side_1  % Clear variables a, h, side_1

      % To clear the entire workspace, use 'clear' without arguments:
      clear

      % When working with sound and images, memory can become severely
      % taxed because MATLAB itself is a memory hog. Then it may become
      % necessary to clear variables (matrices) that are no longer
      % needed.
      % In extreme cases, and when you don't want these data to go
      lost,
      % you can write them to disk, before clearing them and reloading
      % them again later. Here is an example using complex numbers:
      %     >> a = [1+i 2-i 3]';
      %     >> save ws_today
```

```
>> clear
>> load ws_today
>> a


a =


        1.0000 - 1.0000i
        2.0000 + 1.0000i
        3.0000 + 0.0000i
% This saves your entire session (all variables), called the
% workspace, in a file named ws_today, then reloads it after
% clearing

% When terminating a line with a semicolon (;); it will suppress
% the anwer or echo of the input. It is usually a good idea to do
% this as a default when you write longer sequences of commands,
% if you do not want to be slaughtered by reams and reams of
% numbers that you are not looking for as output, as it can be
% hard to detect later where you have forgotten to place a ;.
```

1.5 One of the coolest features of MATLAB is its ability to quickly and painlessly visualize data. Let us start with a simple example.

```
% Let's plot the sine function for values between [-pi,pi].
% First, create a vector t that has increments of size 0.1:
t = [-pi:0.1:pi];

% Now let's take the sine of this array t and store it in a
% variable called sin_t
sin_t = sin(t);

% If you want to see all the values of sin_t just type the
% variable name without semicolon
sin_t

% To plot sine_t, use the 'plot' command.
plot(sin_t)

% A figure should pop up (default figure 1) and display one
% sampled period of a sinusoid.
% The y-axis shows that it goes from +1 to -1, but the x-axis
% goes from 1 to 63. This is because if there is only one
variable
% passed to the plot function, 'plot' uses the index numbers as
% the x-axis. Therefore, to make the plot have accurate axes, we
% need to use both t and sin_t:
plot(t, sin_t)    % Now the axes are as intended

% One important note is that both variables given to plot() must
% have the same length. The first matrix gives the x-position,
% the second matrix gives the y-values. They have to come in
% pairs.

% Let's add to this plot the cosine function for the same values.
cos_t = cos(t);
plot(t, cos_t)

% Now we see one period of the cos function, but we have drawn
% over and lost the previous plot…
% To keep what was already in the plot, use the command 'hold'
% with the switch ‘on’ addded to keep the previous plot:
hold on
plot(t, sin_t)

% Both plots are now shown; but how can we distinguish the two
% without knowing what the functions are? The 'plot' command
% allows several datasets to be drawn with different line
% types, symbols and characteristics.
clf            % clear current figure
plot(t,sin_t,t,cos_t)

% This will plot both sin_t and cos_t and give them different
% colors. We can specify the colors we want in the command plot.
plot(t,sin_t,'go-',t,cos_t,'r:')

% This says to plot sin_t in green, with symbol 'o' at each data
% point and using a solid line to connect them;
% next, in the same diagram
% plot cos_t in red with a dotted line but no symbols.
% Now we can apply a legend to distinguish them:
legend('Sin(t)','Cos(t)')

% You can click and drag the legend to any position you want, or
% specify them in the command:
plot(t,sin_t,'go-',t,cos_t,'r:');
legend(‘Sin(t),’Cos(t)’,‘Location’,’Northwest’);
legend boxoff
% The relocation and removal of the legend box in the last line
is
% also handy when the box of the legend obscures part of the
% plotted data (NB: the legend box is white fill, not
transparent)

% Now we can label the axes and give the plot an optional title.
xlabel('t'); ylabel('Amplitude'); title('Sin(t) and Cos(t)');

% A grid can also be added to the graph for easy vizualization
grid
% Additional options are grid on; grid off; grid minor;.
% Its axes can be made tight
axis tight
% or
axis equal  % to make scales on x and y axes the same
% To maximize information and visibility in the graph, rescale
the
% axes:
axis([-pi/2 pi -0.5 1]); %to restrict the plotted x- and y-ranges
```

```
% Much more can be learned about these functions by using 'help'
help plot
help hold
help legend
help axis
```

1.6 Sounds: there are several ways to make and play sound in MATLAB. Just do a 'help audio' to get the basics. Here we shall create a one-second sound, play it back, and write it to a .wav file (in 1.9).

In the "File" menu, select "New Script". A script or "m-file" (short for M(ATLAB)-file) is a list of commands that you can type and then run. Copy the code below into the script window and save it using the menu as "soundplay.m". Then run it by pressing "F5", selecting "Run">"Run soundplay", or "Run and Time" showing the Profiler for this sound file, clicking the appropriate GUI button on the toolbar (a document with an arrow pointing downwards), or by typing "soundplay" in the MATLAB command window.

```
% This script will play a couple of cool sounds
duration = 1;                 % Duration in seconds
Fs = 44100;                   % Sampling rate in Hz

T = 1/Fs;                     % Sampling period in seconds
t = [0:T:duration-T];         % Time vector

% Create a sinewave at an amplitude of 0.25 and frequency 440 Hz.
% In MATLAB keep audio amplitudes in the range +-1.0 otherwise
% they will clip if not normalized.
signal = 0.25*sin(2*pi*440*t);

% To hear it in MATLAB:
sound(signal, Fs);      % Use headphones

% Let's add another tone to the signal:
signal = signal + 0.25*sin(2*pi*440*2*t);
sound(signal, Fs);

% Let's add another tone to the signal:
signal = signal + 0.15*sin(2*pi*440*3*t);
sound(signal, Fs);

% For a religious sound experience try the following:
load handel;
sound(y);
```

1.7 A handy general command is “pause(<n>)”, which inserts a pause of <n> seconds before continuing with the rest of the program. “pause” without argument waits for user input to press any key before the program continues.

1.8 Images: whereas sound is one-dimensional (it is a function of time or frequency), an image has two dimensions, x and y, showing its intensity or specific colour content as a function of space coordinates. Images can be created, read, and written like arrays and files. Download the image .zip file from the QM+ and unzip it.

```
% Read an image into imagedata. Make sure you are in the right
% directory by typing ‘pwd’ (present working directory).
% You can change to the intended image directory
% by using the folder button, to the top right of the directory
% bar in the MATLAB window, or using the ‘cd’ command you may be
% familiar with from UNIX, with option ‘..’ to go up one level of
% subdirectory.
% Several image formats are supported; for a detailed list,
% type 'imformats' at the command prompt.

imagedata = imread('knit.jpg');

% The 'whos' command shows the following:
whos
imagedata         416x313                 130208  uint8 array
% This signifies that the type of data used to store information
% for imagedata is an 8-bit insigned integer, i.e., a value
% ranging from 0 to 255.
% Information about the image file can be found too.

imfinfo('knit.jpg')

% In this case, it says 'BitDepth: 8'.

% Before we display the image data, we must create a window
% of the same size as the image. We do this using 'figure()':

figure('Position',[100 120 size(imagedata,2) size(imagedata,1)]);

% This command creates a window at the starting position
% 100 pixels from the left, 120 pixels from the bottom, has a
% width of size(imagedata,2), and has a height of
% size(imagedata,1).
% Recall that for matrices the format is rows x columns.

% To display this image, we can use either 'image' or 'imagesc'.
% Use a help command to see the difference.
imagesc(imagedata);

% Now we get rid of the axes because we don't need them. This
% expands the image to the maximum extent of the figure.
set(gca,'Position',[0 0 1 1])

% Finally, set its colormap to gray scale.
colormap('gray');
```

1.9 Processes can be automated using “for” loops. Let's automate the synthesis of a sound using several sine waves. Save this program into an m-file for easier programming and execution.

```
% Our variables
fundamental = 200;              % Fundamental frequency
number_partials = 10;           % number of sinewaves to add
duration = 3;                   % Duration in seconds
Fs = 44100;                     % Sampling rate in Hz
T = 1/Fs;                       % Sampling period in seconds
t = [0:T:duration-T];           % Time vector

% Even though MATLAB has the ability of working with dynamic
% variables, like vectors that are constantly changing size and
% for which no space needs to be reserved beforehand,
% preallocating memory for variables will make your algorithms
% much faster. Let’s make a 1-D zero vector to store the signal.
signal = zeros(1,duration*Fs);

% Let's make the sound more interesting by applying an amplitude
% envelope.
ampenv = zeros(1,duration*Fs);
attack_duration = 1;
release_duration = 1;
ampenv = [linspace(0,1,attack_duration*Fs) ...
            ones(1,(duration-attack_duration-
            release_duration)*Fs) ...
            linspace(1,0, release_duration*Fs)];
% Forgot what linspace does? Type “help linspace”!

% Notice two things about the above command. First, it is
% broken across several lines. As long as the ellipses (...) is
% used MATLAB will continue to read the lines as if they were
one.
% Second, ampenv is a vector created by concatenating several
% vectors together. This is like the following:
a = [1 2 3]; b = [4 5 6];
c = [a b]

% Let’s now plot the amplitude envelope.
plot(t,ampenv);

% Now create the signal using a ‘for’ loop.
for i=1:number_partials,
  signal = signal + ampenv.*sin(2*pi*fundamental*i*t);
end
soundsc(signal,Fs);      % Play sound and automatically scale.

% Before writing to a sound file, make sure that there are no
% data points above 1; and if there are let's divide the entire
% signal by the largest amplitude plus a small amount (just to
% be sure).
if (max(signal) > 1.0)
  signal = signal/(max(signal)+0.1);
end
wavwrite(signal, Fs, 'test.wav');
```

1.10 MATLAB is a scripting language. This means that compiling it is not necessary (although MATLAB compilation can be done, making the code run faster). You can create a script and save it as an m-file, then run it as a command in another script or at the command line. To do this, open a new file in MATLAB called "sinewave.m." Place the following content into the file:

```
duration = 1;
Fs = 44100;
T = 1/Fs;
t = [0:T:duration-T];
signal = 0.25*sin(2*pi*440*t);
sound(signal, Fs);

% Now save that file and in the MATLAB command window (marked
% by the '>>') type 'sinewave'. You should hear a sinewave at
% 440 Hz (standard ‘a’). If you don't and it says
%           "??? Undefined function or variable 'sinewave'.
% then MATLAB doesn't know where to find your m-file. In the
% command window (at the '>>') type 'pwd'. This will return the
% present working directory MATLAB is looking at. Copy sinewave.m
% there and try it again. If it still doesn't work, find a TA.

% M-files can segment your code into blocks. If you have a
% program that deals with simulating an ideal string, you don't
% want a single file that is 1000 lines of code. Instead you can
% create scripts that will take care of various parts of the
% algorithm, like creating the delay lines, sampling the string,
% and writing to a sound file or the speakers. Put simply,
“divide
% and conquer” as the Romans knew already.
```

1.11 A special format for an m-file is a *function*. A function is any bit of code that takes an argument and may return something. In contrast to a script, any variable that is created within a function stays in that function only. This is called a local (as opposed to global) variable. In other words, anything created in the function that is not passed on is unknown/destroyed when the function is finished. As an example, let's create a function that will take as arguments, frequency, amplitude, and duration, and it will return a vector of sound data.

```
% Create a new m-file called 'sinefun.m', and at the top of the
% file write:
% --- BEGIN FUNCTION --- %
function [signal] = sinefun(freq, amp, phase, dur)

% This says to MATLAB this is a function that takes four input
% variables and spits out one in the end. Now we can add what we
% had above, with some changes to the variables, as follows.

      Fs = 44100;
      T = 1/Fs;
      t = [0:T:dur-T];
      signal = amp*sin(2*pi*freq*t + phase);
      % --- END FUNCTION --- %
      % Save this. In the MATLAB command window ('>>'), type:
      snd = sinefun(200, 0.5, 0, 1);
      sound(snd, 44100)
      snd = sinefun(411, 0.5, 0, 2);
      sound(snd, 44100)

      % Or more tersely:
      sound(sinefun(1325, 0.5, 0, 0.5), 44100);

% There can be only one function per m-file, unless you want
% to create subfunction—functions called only by the first
% function in the file. Also, the name of the m-file has to be
% the same name as the name of the function. For more
information,
>> help function
```

1.12 The remarkable thing about MATLAB functions is that they can return several variables (multiple outputs). This is specified by the first line in the function m-file:

```
function [a2, b2, c2, d2, e2] = foo(a1, b1, c1, d1, e1)

% Another strange thing about MATLAB functions is that the
% input and output variables aren't all required. For instance
% I can call the function above in any of the following ways:

[a2, b2, c2, d2, e2] = foo(a1, b1, c1, d1, e1)
[a2, b2, c2] = foo(a1, b1, c1, d1, e1)
[a2, b2, c2] = foo(a1)
foo(a1, b1)

% When this happens only the items on the left side are returned,
% and the variables that are not specified in the input list are
% set to default values within the function.
```

```
        % Note that the following calls are not allowed:

        [a2, b2, c2] = foo(b1, c1, d1)
        [b2, c2] = foo(a1, b1, c1)

        % Can you see why?
        % MATLAB expects that both the input and output variables will
        % be in the EXACT same order as defined in the function
        % definition. In the second call, b2 will actually be a2, and c2
        % will be b2.

        % So, when we write functions how can we detect the format the
        % function has been called in, without needing an elephant's
brain
        % to remember this for everey function we have defined and used?
        % With each function call, there are two hidden variables sent to
        % the function: nargin and nargout.
        % These are simply the number of arguments put into the function,
        % and the number of arguments that the user expects to be
returned
        % from the function.

        % The following function definition allows for two input
arguments
        % and for four output arguments. Let's make it return an error or
        % help message when the user tries to divide two numbers without
        % supplying a second number, and then exits from the function.

        function [add, sub, mul, div] = math(a, b)
              if (nargin ~= 2)
                    disp('Usage: [add, sub, mul, div] = math(a,b)');
                    return;
              end
              % Otherwise continue
              if (nargout >= 1)
                    add = a + b;
              end
              if (nargout >= 2)
                    sub = a - b;
              end
              if (nargout >= 3)
                    mul = a*b;
              end
              if (nargout >= 4)
                    div = a/b;
              end
              % If nargout == 0 then just display results!
              if (nargout == 0)
                    add = a + b;
                    sub = a - b;
                    mul = a*b;
                    div = a/b;
                    fprintf('add = %4.2f\n', add);
                    fprintf('sub = %4.2f\n', sub);
                    fprintf('mul = %4.2f\n', mul);
                    fprintf('div = %4.2f\n', div);
              end
```

```
            return;

      % Once this function is placed in a file named math.m (and put it
      % in a directory that can be called from MATLAB) it can be
called:
      >> math
      Usage: [add, sub, mult, div] = math(a,b)
      >> math(2,1)
      add = 3.00
      sub = 1.00
      mul = 2.00
      div = 2.00
      ans =

             3

      >> math(2,1);
      add = 3.00
      sub = 1.00
      mul = 2.00
      div = 2.00

      >> add = math(2,0)
      add =

             2
      >> [add, sub, mult] = math(2,0)
      add =

             2

      sub =

             2

      mult =

             0
```

Having the power to create scripts and functions in MATLAB, thereby extending its palette of built-in tools, is of great benefit. However, as with any programming language, especially high-level scripting languages, there are words reserved by the system that cannot be used as names for variables, scripts, or functions. To get a list of reserved keywords, type `iskeyword`.

Caution should also be used when naming your variables, functions and m-files. As mentioned before for variables, do not start a filename with a number! Also, be careful not to use a function name that is already used or pre-defined. For instance, don't create an m-file called 'fft.m' because that already exists.[1] Don't create a variable named 'sound' because that is the name of a built-in function; if you want to call the function later, you will instead get the variable that has overwritten it (during this session).

If you are unsure about the name you are choosing for a variable, script, or function, type: `which <word-you-are-unsure-about>`. If this word-you-are-unsure-about is not found, then it is free to use.

There are plenty of cool demos that come with MATLAB. To survey them, type `demo` or `help demo`.

Congratulations! Now that you have experience in MATLAB you can honestly place "MATLAB" under the "Computer Skills" section of your resume. You have increased your net worth by 16%, and doubled the probability of you passing this course. You may now proceed to the next part of this lab, and seal your fate.

[1] This also means great fun can be had by writing a rude script named "fft.m" in your friend's MATLAB directory.

## 2. Moving Averages of Time-Series

The convolution of two discrete sequences of finite length, $x[n]$ and $h[n]$, is defined by the expression

$$y[n] = x[n] * h[n] = \sum_{k=0}^{M-1} h[k]x[n-k] = \sum_{k=0}^{M-1} x[k]h[n-k]. \quad (1)$$

where the notation "$\Sigma_{k=0}^{M-1}$ $h[k]x[n-k]$" means a summation of $M$ terms that are based on the expression that follows the "Σ" sign, for values of the index $k$ that run from 0 to $M$-1, i.e.,

$$\Sigma_{k=0}^{M-1} h[k]x[n-k] = h[0]x[n] + h[1]x[n-1] + \ldots + h[M-1]x[n-(M-1)].$$

The length of $y[n]$ is governed by the lengths of $x[n]$ and $h[n]$ (see lecture notes). In particular, when $h[n]$ is a length-$M$ sequence of constant values $1/M$, then $y[n]$ is called the $M$-point (or $M$-order) moving average of $x[n]$. In other words, the $M$-point moving average of $x[n]$ is

$$y[n] = \frac{1}{M}\sum_{k=0}^{M-1} x[n-k]. \quad (2)$$

This says for every output point in $y[n]$, we average the latest $M$ points of input $x[n]$.

*Figure 1: Buoys will be buoys*

To get a feel for what this does in practice, download the dataset 045200603.txt from QM+. This is ocean buoy data as measured off the coast of Oceanside, California, during March 2006.[2] Basically the buoy (Figure 1) goes up and down, like all good buoys do, and records these motions at a sampling rate of 1.25 Hz. Look into this data and perform some analyses, which will either impress or alienate your peers.

2.1 Before you can use these data, you need to either enter it in by hand, or be smart like the rest of the kids and create a function to read in the data and return a data structure. You can either write your own, or use what I have created.

```
function [data, count] = readbuoydata(datafile)

fid = fopen(datafile,'r');
tline = fgetl(fid);
tline = fgetl(fid);

[A,count] = fscanf(fid,'%d %d %d %d %d %f %f %d %f %f',[10 inf]);

data.date = datenum([A(1:5,:); zeros(1,size(A,2))]')';
data.Hs = A(6,:);  % significant wave height
data.Tp = A(7,:);  % peak period
data.Dp = A(8,:);  % peak period direction
data.Ta = A(9,:);  % average period
data.SST = A(10,:); % sea surface temperature

fclose(fid);
```

[For interest, also look up the MATLAB functions dlmread() and dlrmwrite(). These are handy and fast functions to use with I/O of large data sets in a variety of formats. Note that these functions use (0,0) as the first row-column element.] In your lab report, include this code and add comments on what each line does. This means, for each line of the function write sweet nothings about what it is

[2] Data acquired from http://www.cdip.ucsd.edu

doing and how it does what it does so well, in a technical and succint yet informative and precise manner.

2.2 Write a script to plot the peak period (Tp), and the significant wave height (Hs), as functions of time. You can access this data from the data structure by using the “period”: `data.Hs`, etc. Make one plot for Tp, and one plot for Hs. Label the x-axis “Time” and use `datetick` to place date labels. Label the y-axis as “Peak Period (s)” and “Significant Wave Height (m)”, respectively. Include your code and the figures in your report. Your plots should look similar to Figure 2 as a guide, although admittedly not as impressive. Do not spend too much time on the labeling of the time-axis, at the detriment to the rest of the lab!

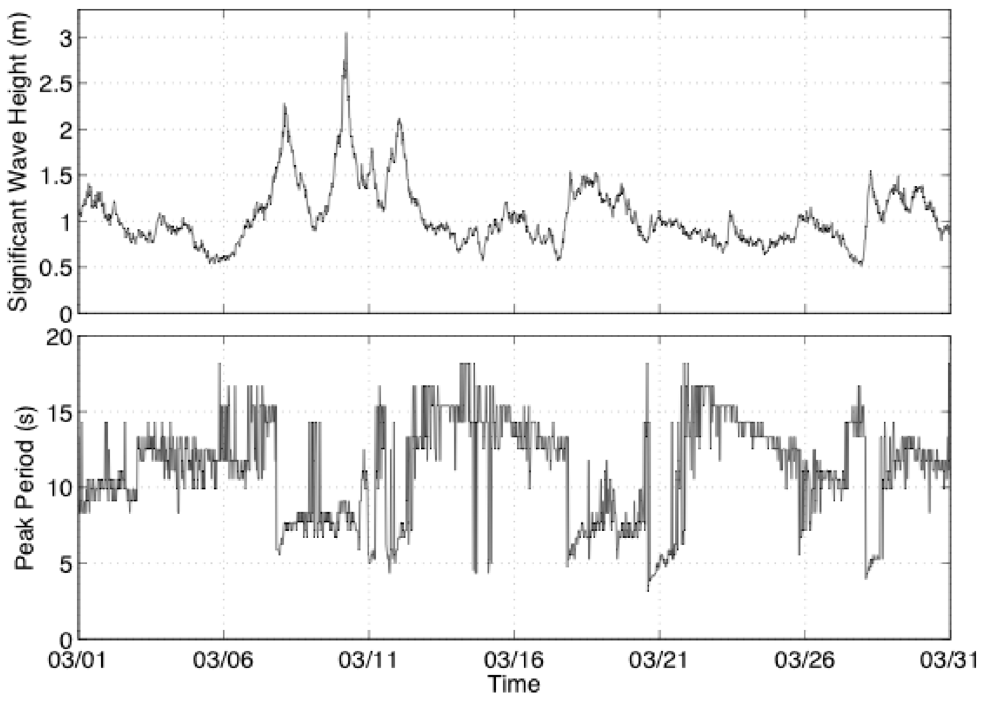


*Figure 2: Significant wave height (top) and peak wave period (bottom) of the Oceanside offshore buoy for the month of March 2006. Spot the good times for surfin’!*

Now a brief discussion on the meaning of these ocean buoy data. In raw format, the Oceanside buoy records its displacement in the z-direction (height) at a rate of 1.25 Hz. These data were collected over intervals of about 27-minute (2,048 samples), and then processed and archived. The data you are working with are the processed data, not the raw data.

The “significant wave height” (Hs) observed by the Oceanside buoy is shown in the top plot of Figure 2. This number is the average of the 20% highest displacements that were measured.[3] Hence when the number grows, the waves are becoming larger. However, this is not the only important statistic for surfers.

The “Peak Period” (Tp) observed by the buoy is shown in the bottom plot of Figure 2. The frequencies present in the water are found by spectral analysis of the z-displacement data. The peak period is the inverse of the frequency with the greatest energy. When the peak period is large, the waves are longer and cleaner (i.e., less noisy). When the peak period is long, and the significant wave height is large, then the surf is “going off” (Figure 3).

*Figure 3: Here a physical oceanographer is “in the field” measuring significant wave stoke*

In the plot of Tp something interesting happens during the middle and end of the month. There are sudden increases and then gradual decreases of the peak period. These are actually signatures of far-away storms. Ocean waves’ travel speeds are in proportion to their period. Low frequencies —and hence long

[3] Of the sampled data; some local maxima may have been missed.

periods— travel much faster than high frequencies that correspond to short periods. When a storm transfers energy into the ocean through strong winds, the long period energy outruns the short period energy, creating "wave trains" (packets). In these trains, long periods are in the front and short periods are in the caboose. These trains travel across the ocean and eventually run aground and abuse innocent buoys minding their own business. So what you are seeing in the peak period plot are actually signatures of far-away storms.

2.3 Write a function that performs the moving average (given in equation (2)) of any 1-dimensional input data for any non-zero and positive order $M$. Make sure to add $M - 1$ zeros at the beginning of the data so that your output data will be the same length as your input data. You must not use built-in functions like movmean() or conv(). Include your all-purpose function in your lab report.

2.4 Using this function, perform a moving average on the peak period data (Tp) using $M$=5, 21, and 51. Produce a plot for each $M$, labeled in the same way as the bottom plot in Figure 2. On the same plot, show the original data in the background. To do this, first use `plot(origdata,'Color',[0.8 0.8 0.8],'LineWidth',2);` then `hold on;` then `plot(avedata,'k');`. Include your MATLAB code and these three plots in your report. Ideally, your plots should look like those in Figure 4, though, again, not as singularly spectacular.

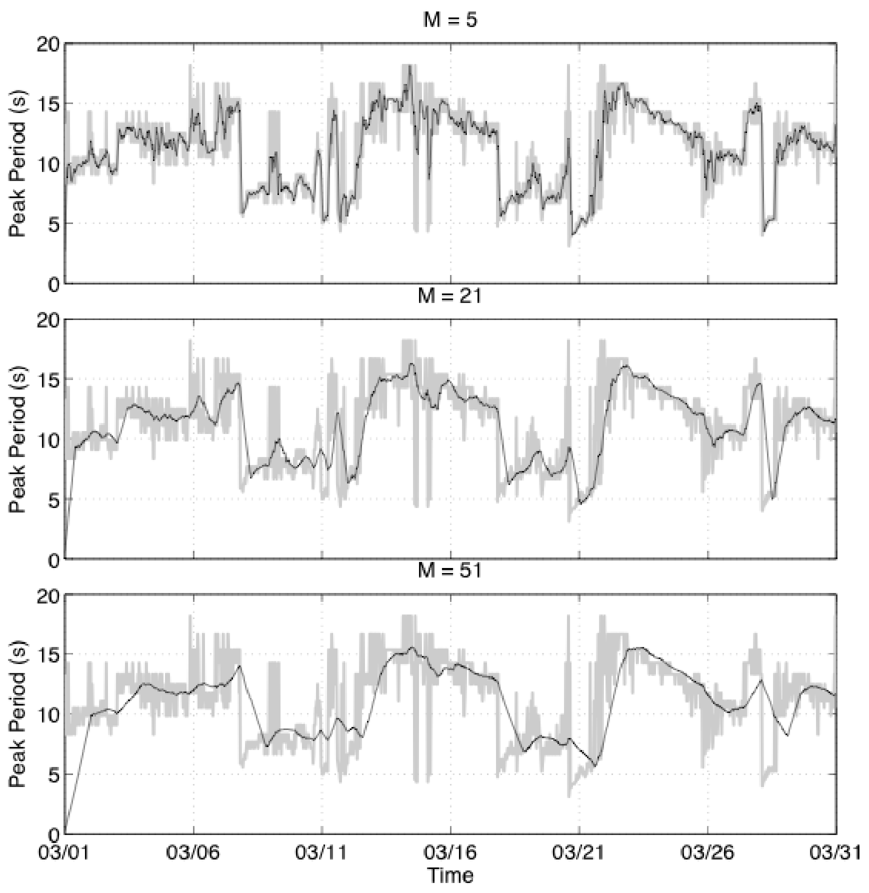


*Figure 4: Moving averages of peak period data for different M.*

2.5 In full sentences, maybe even a well-formed and logical paragraph, answer all of these questions:

2.5.1 What do you observe in the plots when $M$ increases?

2.5.2 Why do you think you observe this “thing”?

2.5.3 What is happening at the beginning of the averaged data set, and why does this happen?

2.5.4 What happens to the running average when the peak period suddenly drops?

2.5.5 Are these drops preserved?

2.5.6 Are the wave trains more clear?

2.6 Repeat 2.4 for the Hs data using the same $M$. Include your plots, but this time not your code.

2.7 Do you observe anything different in the plots from 2.6 to those you had in 2.4? When $M$ increases, what happens to the peaks in the data? How does the size of this effect relate to $M$? Explain why the peaks move.

This effect can be corrected for by several methods. One of the simplest ways is to convert equation (2) into a non-causal system —meaning that the system uses future input data samples to compute the present output. Since we have all the data that we need in off-line computation (i.e., these data are not processed in real-time), this is not  a problem. Instead of performing equation (2) as shown, we will revise it to

$$y[n] = \frac{1}{M} \sum_{k=-(M-1)/2}^{(M-1)/2} x[n-k], \tag{3}$$

where $M$ is an odd integer.

2.8 Now redo 2.4 and 2.6 using equation (3) so that the time effect is corrected. Include your figure and code.

So you may be getting it by now: the moving average smoothens the data. Essentially the moving average is similar to a lowpass filter in that it attenuates high frequencies.

## 3. Moving Averages of Images

The moving average of a 2-D sequence, $x[m,n]$, is obtained by

$$y[m,n] = \frac{1}{M^2} \sum_{l=0}^{M-1} \sum_{k=0}^{M-1} x[m-l, n-k]. \tag{4}$$

This says for every point in $y[m,n]$, we average $M^2$ points in the vicinity of $x[m,n]$. The rather complex looking equation (4) can be implemented by performing two moving averages as in equation (2): one for the rows of the image, and the other one for the columns of the output of the moving average of the rows. (Technically speaking, the two averaging operations are said to be "separable.") To get a feel for the result, use one of the images "bridge.gif," "bird.gif," "boat.gif," or "cameraman.tif" from the image .zip file on QM+. Choose one that best reflects your personality or mood. (Hint: probably not the bird.)

3.1 Write a function that implements a 2-D moving average process using your zero-delay 1-D moving average function for any odd $M$, created in 2.8. Include your function in your lab report.

3.2 Based on your acquired experience with moving average filters (see previous section), predict the effects of a moving average when performed on images.

3.3 For your selected image, run your 2-D moving average function, created in 3.1, using $M$ = 3, 5, and 7. Include your code and plots.

3.4 What do you observe when $M$ increases? How would you best describe the effects of the moving average on emotionally moving imagery? Was your prediction in 3.2 correct? If not, you can modify your answer in 3.2 to the correct answer; but you may never be able to sleep at night.