M219 Principles and Applications of MRI (Winter 2022)
Homework Assignment #0 (zero point)
This assignment is not graded nor due.

This assignment is meant to provide an introduction to basic Matlab coding and functions that can form the basis for subsequent assignments in M219. If you have used Matlab before, then this assignment will likely prove relatively straightforward. If you have no familiarity with Matlab, then take this assignment seriously and work through the solutions. It is designed to give you some skills that you will need later in the course.

1. M-files

   Matlab enables saving the code you write as either a *script* or a *function* in the form of an *m-file* with a name like [filename].m. A script is simply a list of commands that are run in sequence when the file is *called* from the Matlab command prompt. A script does not accept *input* variables, nor does it produce *output* variables. A function requires a function declaration and can accept input variables and can produce output variables. To get going try the following after you open the Matlab application:

   ```
   >> edit M219_Homework00_mfile.m
   ```

   You'll be asked "Do you want to create it?" Yes! The Matlab editor will appear with an empty file. Type (or copy and paste) the following (not the line numbers):

   ```
   % This is a comment. This code is not executed.
   % I promise to comment all of my code.

   apple=1     % This defines a variable named 'apple' and sets it equal to one.
   orange=2;   % This defines a variable named 'orange' and sets it equal to two.
   %             The semi-colon suppresses output to the command prompt.
   ```

   Return to the Matlab command prompt and type the following, which will run your m-file script:

   ```
   >> M219_Homework00_mfile
   ```

   You should see the value of *apple*, but not *orange* returned to the command space. Note the importance of the ";" in Matlab. *Note*: To run this file you need to be either in the same directory as the saved location of the m-file or you need to add the path of the file to Matlab. Not sure what that means? Try,

   ```
   >>help path
   >>help cd
   ```

   In fact, for every function that is part of the Matlab language "help [function-name]" will provide useful information. Google is your friend.

2. Scalars, Vectors, and Matrices
   In the previous Matlab script we defined two scalar variables. We can also easily define vectors and matrices. Remember that the dimensionality of vectors and matrices is *really* important. A vector that is 1x3 is not equivalent to one that is 3x1. Furthermore, if we multiply vectors and matrices, then their inner dimensions must match. Try creating the following script:

```
% This function creates scalars, vectors, and matrices and performs some
% simple operations.

speed_1=1;                              % A simple scalar
speed_2=2;                              % A simple scalar

vel vec 1=[1 2 3]                       % A row vector
vel vec 2=[4; 5; 6]                     % A column vector

matrix_1=[1 2; 3 4; 5 6]                % A 3x2 (rows x columns) array

matrix_2=eye(3)                         % T e identity matrix.

new_vec1=speed_1.*velocity_1            % Performs dot−multiplication
new_vec2=speed_2.*velocity_1            % Performs dot−multiplication

vel_mat1=vel vec 1*matrix_1             % The inner dimensions must match [1 x 3]*[3 x 2]
% vel_mat2=vel vec 2*matrix_1           % This will not compute though...

vel_mat2=new_vec2*matrix_2              % This returns the new_vec2 vector...
                                        % This is just multiplying by "one"
```

3. Functions
   So far we have only used Matlab *scripts*, whereas we can also use *functions*. Functions are a specific kind of script that enable calling the code in the function from another function (or script) to obtain a new output given a provided input. Let’s try to create a very basic function that calculates the intersection of two lines using their slopes and y intercepts, and then plots the result. Pay special attention to the plotting component here. You’ll be plotting homework results a lot this quarter, and matlab plots take some tweaking to look nice.

```
function [intersection] = lin_intersect(m1, b1, m2, b2, varargin)
%Returns the intersection of two lines based on their slopes and intercepts
%     Inputs:
%      m1:[1x1]    double  −  the  slope of line 1
%       b1:[1x1]   double  −  the  y intercept of line 1
%      m2:[1x1]    double  −  the  slope of line 2
%       b2:[1x1]   double  −  the  y intercept of line 2
%
%      Optional arguments:
%            plot_flag: [1x1] double  − 1 if we want to plot outputs, 0 otherwise
```

```
%
%       Outputs:
%                 intersection: the x and  y coordinates of the intersection

% Below is an if statement.               If the conditions specified after the if/elseif
% statements are true, the lines of code following will be executed.
% Otherwise, the lines of code after "else" will be run. Notice we use '=='
% not '=' to compare if values are the same. We could also use
% ≥, ≤, <, or >, to test for other relationships, but we won't here.

if nargin == 4     % no optional argments are entered
   plot_flag = 1; % defaults to plotting the output
elseif nargin == 5 % one optional argument was entered
   plot_flag = varargin{1}; % takes the value of the fifth argument
else
   warning('Initializing with default values.')
   m1 = 2; m2 = -5; b1 = -10; b2 = 20; plot_flag = 1;
end

if m1 == m2 && b1 == b2 %check if both slopes and y intercepts are the same
   warning('These lines are the same! They always intersect.')
   intersection = [nan,nan];
elseif m1 == m2 %check if only the slopes are the same
   warning('These lines are the parallel! They never intersect.')
   intersection = [nan,nan];
else %Calculate the intersection!

   x_int = (b2-b1)./(m1-m2);
   y_int = m1.*(x_int)+b1;

   % This returns the
   intersection = [x_int, y_int];
end

if plot_flag == 1 % optional argument that allows plotting
   %% Preparing to plot the lines
   % create an array of x values +- 10 units from the intersection spaced
   % by .1
   if ~isnan(intersection(1)) % checks if there is a valid intersection
      xmin = intersection(1) - 10;
      xmax = intersection(1) + 10;
   else
      xmin = -20 ; xmax = 20;% if not, defaults to [-20, 20]
   end
   x_range = xmin:.1:xmax;
   % return the y value at each of the sampled x values
   y1 = m1.*x_range+b1;
   y2 = m2.*x_range+b2;

   %% Plotting the lines
```

```matlab
    figure % This creates a new figure
    plot(x_range,y1,'linewidth',3); % This plots our sampled x and y values
    %for line 1
    hold on % This keeps the next plot command from deleting the old graph
    plot(x_range,y2,'linewidth',3); % This plots our second line

    % Plot the intersection we found as a black square with MarkerSize = 10
    plot(intersection(1), intersection(2), 'ks', 'MarkerSize',10,...
      'MarkerFaceColor','k');

    % Set the limits of our graph
    xlim([min(x_range),max(x_range)]);
    ylim([min([y1 y2]),max([y1,y2])]);

    % Create a legend for our data
    legend('Line 1','Line 2','Intersection');

    % Now some labels for our axes
    xlabel('X Values (unitless)');
    ylabel('Y Values (unitless)');
    title('Intersection of Two Lines');

    % The graph could still look a little neater. Let's modify the plot
    % appearance (gcf is get current figure, gca is get current axis)
    set(gcf,'Color','w');
    set(gca,'Color','w','XColor','k','YColor','k','FontSize', 12, 'Box',...
      'on', 'LineWidth', 3.0);
    set(get(gca,'Title'),'Color','k','FontSize',18,'FontWeight','bold');
    set(get(gca,'Xlabel'),'FontSize',16,'FontWeight','bold');
    set(get(gca,'Ylabel'),'FontSize',16,'FontWeight','bold');
    %set(gcf,'Color','k');
    grid on

    hold off

else
end
end
```

We can call this function in a new matlab script.

```matlab
% This script runs the function lin_intersect. Try playing around with
% different input values, and running lin_intersect directly from the
% command line. You're highly encouraged to insert breakpoints (mentioned
% below) into the function lin_intersect to see how things progress line by
% line.

m1 = 2; %the      slope of our first line
m2 = 4; %the      slope of our second line
```

```
b1 = 2; %the y intercept of our first line
b2 = 8; %the y intercept of our second line
plot_flag = 1; % Plotting the results? 1 if yes, 0 if no.
intersection = lin_intersect(m1,b1,m2,b2,plot_flag);
% note, because we have commented lin_intersect well, we can type
% "help lin_intersect and see how to run it

%% Debugging

%If for any reason we ran into issues while running lin_intersect, we could
%use a "breakpoint". We can open lin_intersect in matlab, select the line
%before the code breaks, and select breakpoint−>Set. This will place a red
%dot on the line where the breakpoint is featured and stop the code
%mid−execution at that location so we can investigate the issue. We can
%remove the breakpoint and re−run the function later if we think we have it
%right.

%We can also put breakpoints anywhere in our code, run it, and then use the
%"step" button to go execute the code line by line as we check the outputs
%we are getting.

%% Printing our plot
% We've created another function that prints the output graph to a
% directory we specify. In this case it is the working directory.

pathname = [pwd,'/'];
print2desktop(pathname,'lin_intersect_output_graph');
```

To print the output graph to the current directory, try using this code:

```
function print2desktop(path,name,size)

if nargin<3
    size = [10,6];
end

if nargin<2
    name='newfig';
end

set(gcf,'InvertHardCopy','off'); % keep the colors as they are on screen
set(gcf,'Units','Inches'); % If this is left as 'normalized'
% 'OuterPosition' and 'Position' interfere

set(gcf,'PaperUnits','Inches');

set(gcf,'PaperSize',size); % Set the paper size to match the position
set(gcf,'PaperPosition',[0 0 size]); % Match the paper position to the position
set(gcf,'PaperOrientation','portrait');
```

```
set(gcf,'PaperPositionMode','auto');

print(gcf,'-dpng','-r300','-opengl',[path name '.png']);

disp([path name '.png"']);
```

4. Images

We can use matlab to load, manipulate, and view images. Let's try doing so below:

```
% We can load an image in matlab with the following commands

%We are loading a default image stored in matlab. Ordinarily we would have
%to provide the full path to the image, or if it was stored in our
%workspace, use ./filename
img_path = 'ngc6543a.jpg'; % specify the path of the image.

% %Or we could select our own image using this code. Uncomment it and try
% it for yourself on any image you have on your computer.
%
% [fname pathname] = uigetfile();
% img_path = fullfile(pathname,fname);

%reads the image
my_img = imread(img_path);
% note we now have a 650x650x3 uint8 variable in our workspace. This is the
% image size [650x600] and rgb values

%We can also now show the image
figure
imshow(my_img);

%However, for M219 , we'll frequently be using a different image
%format: dicom (.dcm) and a different loading command

% Again are loading a default image stored in matlab. Ordinarily we would
% have to provide the full path to the image, or if it was stored in our
%workspace, use ./filename
image_info = dicominfo('CT-MONO2-16-ankle.dcm'); % gets the header info
my_dicom = dicomread(image_info);
figure, imshow(my_dicom,[]);

% we could alternatively use a method that auto-scales our data using a
% colormap of our choice as the second argument, or a default one if none
% is specified.
figure, imagesc(my_dicom)

% We can perform an incredible variety of operations on the image that
% we've loaded in. The sky is the limit here, and the matlab help function
```

```
% and the mathworks website are your friends. Check out "basic for loops"
% to see a simple example.
```

5. For Loops
   Let's try using a for loop to manipulate images. Here we're going to load a matlab default dicom image and loop through to change all indices with values greater than 1000 to 0 (black).

```
% We might also decide to use for loops to investigate images.

%We are loading a default image stored in matlab. Ordinarily we would have
%to provide the full path to the image, or if it was stored in our
%workspace, use ./filename
image_info = dicominfo('CT-MONO2-16-ankle.dcm'); % gets the header info
my_dicom = dicomread(image_info);
figure, imshow(my_dicom,[]);

% Perhaps we want to find all elements of "my_dicom" that are greater than
% 1000, and replace them with zero (black)

% initialize a new image that is the same as the old one to start
my dicom new = my_dicom;

% We are going to use a nested for loop here. We loop through every element
% of the first dimension of "my dicom new" using "i"
for i = 1:size(my dicom new,1)
% We loop through each element of the second dimension of
    % "my dicom new" using "j"
  for j = 1:size(my dicom new,2)
if my dicom new(i,j) > 1000 % check if the current element is
            % larger than 1000
my dicom new(i,j) = 0; % if so, replace it with zero
        end
    end
end
figure, imshow(my dicom new,[]);

% However, we could have done this faster without for loops
my dicom new2 = my_dicom;
replacement_inds = find(my_dicom > 1000);
my dicom new2(replacement_inds) = 0;

figure, imshow(my dicom new,[]);
```

If you're new to matlab, hopefully you've picked up a few skills that will prove useful this quarter. Remember, there is a ton of documentation regarding using matlab (i.e. the "help" command and googling things on the mathworks website). If you're stuck on a coding problem, try that first.