Lab Session 1 Introduction to MATLAB Problem Sheet

Key aims of the Session:

- Familiarise with MATLAB
- Learn some essential commands that will be useful for the rest of the module and particularly the coursework
- Work with vectors and matrices
- Learn some commands commonly used in data science and machine learning

You can do the following exercises either in the Command Window or by writing a script. I would recommend writing this as a script so that you can save your work for later. For efficiency, it would be a good idea to write a separate script for each exercise. It is good practice to include annotations of what you're doing as comments. In Matlab, we include a comment using the `%` symbol.

To get more information on any of the commands, you can type **`help COMMANDNAME`** into Command Window. E.g. to get help with the *readtable* function, I would type **`help readtable`**.

Exercise 1: Smoothing Data

In this exercise, we will import some data from a csv file into a Matlab table. We will look at extracting a subset of the data, plotting it in a graph and creating a smoothed version of that data.

1. **Import the COVID Data from the "coronavirus-cases.csv" file into a table.**
   You will find the **`readtable`** command useful for this, using the filename (and path if necessary) as the function argument. E.g.
   `tab = readtable("FILENAME.csv")`
   will import the data from FILENAME.csv and store it in the variable called *tab*.

2. **Display the first 10 lines of the table.**
   To call the first *n* rows of the table, you can use parentheses. The command **`tab(1:n,:)`** would call the first *n* rows and *all* columns of the table *tab*. For this task, remember that when you enter a command into the Command Window, you can use a semi-colon `;` to suppress the output.

3. **Create a second table with only the data for the Area Name "England".**
   For this part, you think of it as extracting a subset of your table defined by the rows where "England" is mentioned in the AreaName field. This subset will include all columns but only select rows. Recall from point 2 above that you can use the command **`tab(rowIDs,:)`** to call the rows of tab given by **`rowIDs`**. **`rowIDs`** can be a list of row numbers, or it can be a logical array of the same height (number of rows) as your table, where 1 means "include the row" and 0 means "exclude the row".
   To find the rowIDs, you can use the **`strcmp`** command to compare strings. For single entries, e.g. **`strcmp("Scotland","England")`**, the command will give a binary result using the logical format: 1 if "Scotland"="England" and 0 otherwise. If one of your arguments is a column of entries (e.g. from your table), the command will return a logical

array, containing the binary result for each entry. You can call the data from a table's column using the **.** symbol; e.g. **`tab2.AreaName`** would call the array of data from the *AreaName* column of the table *tab2*.
You can create a copy of the table *tab* by simply using an assignment, e.g. **`tab2 = tab;`**

4. **Sort your new table by Specimen Date and display the first 10 rows to check that this has worked correctly.**
   The **`sortrows`** command will help you here. For tables, your first argument will be the table variable *tab2* (or whatever name you gave it) and the second argument is the name of the column that you want to sort by. Don't forget to output your sorted table to your variable using an assignment.

5. **Plot the Daily Cases against the Specimen Date as a blue line graph.**
   It is generally good practice to specify a figure number to act as an index for the figure window in which your plot will appear. We can do this with, e.g. **`figure(1)`**.
   To plot a line graph, we can use the **`plot`** command where the first argument is the array of values for the *x*-axis and the second is the array of values for the *y*-axis. Note that, in Matlab, arguments of a function are separated using a comma. You can use these directly from the table column; recall that in point 3 above we extracted data from a column.
   To specify that the graph should be blue, we just need to add a third argument: **`'b'`** where the apostrophes are part of the command and *b* means blue. If you instead wanted your graph to be red, you could instead use **`'r'`**, or if you wanted it to be cyan with dashed lines you could use **`'c--'`**. You can find more information about this in the Help Sheet.

6. **Label the x- and y-axes and give the graph a suitable title.**
   For the axis labels, we need the **`xlabel`** and **`ylabel`** commands where the axis label text will be the arguments of these functions. Don't forget to format them as text using apostrophes.
   To give the graph a title, you can use the **`title`** command.
   These commands will apply to the most recently used figure. If you have multiple figures and want to specify, you can include the **`figure`** command, specifying the figure's index, before you include the axis labels and title.

7. **Create an extra column in the table called "SevenDayAverage". Use a for loop to populate the new column with the average of the previous 7 days' Daily Cases, ignoring the first 6 days.**
   This sounds tricky but can be done in only four lines of code.
   We start by creating a new column, initialised with a column of zeros. To create a column of zeros, we can use the **`zeros`** command where the first argument is the number of rows and the second is the number of columns. You need your column to have the same number of rows as your table, which can be determined using the **`height`** command, e.g. **`height(tab2)`** will return the number of rows in the table *tab2*. You will only need 1 column. You can then create the new table column by simply assigning this column of zeros to the table, specifying the new column's name. E.g. if I want to add a column called *Col1*, populated with the variable *var1*, I would enter **`tab2.col1 = var1`**.
   For the next part, we want to use a *for* loop with the **`for`** command to loop over each row of the table (except the first six). E.g. if we want to loop over the variable *i* from 1 up to 20,

we would write **`for i = 1:20`**. Note that we don't need to use a colon, as with Python. We must terminate the *for* loop using the **`end`** command. All commands between **`for …`** and **`end`** will be carried out for each iteration.
Within the *for* loop, we want to: (i) extract the current and previous 6 entries from the *DailyCases* column. You will need to use your iteration variable, e.g. if I'm looping over *i* then *i-6* would refer to 6 rows higher, and the colon **:** to specify a range. (ii) use the **`mean`** function to calculate the mean value of your extracted data. (iii) store this value in the current (*i*th) row of your new column.

8. **Create a new figure. Plot both the Daily Cases and the 7 Day Average on the graph. Add x- and y- labels and a graph title. Add a legend to the graph.**
For most of this, we have already learned most of the commands. But now we want to include two plots on the same graph. Begin by specifying a new figure if you don't want to lose the previous one, e.g. **`figure(2)`**. To include more plots, you can simply add extra arguments to the **`plot`** function. E.g. if I enter **`plot(x1,y1,'b',x2,y2,'r')`** then I will get a single plot showing the graphs of *y1* against *x1* in blue and *y2* against *x2* in red.
To add a legend, you can use the **`legend`** command with the labels for your graphs entered as arguments in the same order as your plots. E.g. I might enter **`legend('Graph1','Graph2')`**.

EXTRA TASK: Try plotting the graph with a logarithmic scale on the y-axis using the **`semilogy`** command.

Exercise 2: Pooling

In this exercise, we will import an image into Matlab and implement a procedure called "pooling", which is a common task in computer vision machine learning frameworks.

If you are continuing from Exercise 1, you may want to close all figures with the **`close all`** command and clear the workspace with the **`clear all`** command.

1. **Load the image "InfoLab.jpg" into a variable called "im".**
For this, we can use the **`imread`** command. This will convert the image into a 3-dimensional array of values, which give the intensities of the red, green and blue colour channels of the image. As with the **`readtable`** command, **`imread`** takes the filename as the argument.

2. **Output the dimensions of the variable im.**
We can check the size of this matrix with the **`size`** command.

3. **Create a new variable "gim" containing the average of the three channels. This is a grayscale representation of the image.**
Here, we can use the **`mean`** function that we used in part 1.7 above. Since we want to make the average across the channels, we will need to add a second argument to specify the dimension which, in this case, is the third dimension.

4. **Create a new figure with a 1-row and 3-column array of subplots. Display the image in the 1st subplot window on the left and remove the axes. Display the grayscale image in the 2nd subplot window and remove the axes.**

First, we want to create a new figure using the **`figure`** command; this will hold our subfigures. We can add the first subfigure using the **`subplot(1,3,1)`** command, where

a. the first argument specifies the total number of rows in the array
b. the second argument specifies the total number of columns in the array
c. the third argument specifies the index of the subfigure in question

Once we have specified the subfigure, we can show the image using the **`imagesc`** command using the image variable name **`im`** as the argument. We can turn the axes off with the command **`axis off`**. We can then repeat these steps for the second subfigure showing the grayscale image. Finally, we can show the image with a grayscale colour map using the command **`colormap gray`**. For more information, you can enter **`help colormap`** into the Command Window.

5. **Use a "sliding window" approach to carry out max pooling on the grayscale image:**
   a. **Initialise a new matrix called "mp" to hold the result of your max pooling operation. This should be half the height and width of your image with only one channel.**
   We can initialise this as a matrix of zeros using the **`zeros`** command that we used in part 1.7. For the first two arguments of this command, we can use the **`size`** command from part 2.2 with an additional argument specifying the dimension. E.g. **`size(im,1)`** would give the number of rows in the matrix **`im`**.

   b. **Use for loops to slide a 2x2 window across your grayscale image with a stride of 2, finding the maximal value in each window and place this value in your new matrix "mp". I.e. the value of mp in the first row and column would be:**
   $$\boldsymbol{mp}(\mathbf{1},\mathbf{1}) = \max\left(\boldsymbol{gim}(\mathbf{1}\colon\mathbf{2},\mathbf{1}\colon\mathbf{2})\right)$$
   and the value for the second row and first column would be:
   $$\boldsymbol{mp}(\mathbf{2},\mathbf{1}) = \max(\boldsymbol{gim}(\mathbf{3}\colon\mathbf{4},\mathbf{1}\colon\mathbf{2}))$$
   At a general point in the matrix mp, we have:
   $$\boldsymbol{mp}(\boldsymbol{i},\boldsymbol{j}) = \max(\boldsymbol{gim}(\mathbf{2}\boldsymbol{i}-\mathbf{1}\colon\mathbf{2}\boldsymbol{i},\mathbf{2}\boldsymbol{j}-\mathbf{1}\colon\mathbf{2}\boldsymbol{j}))$$
   For this, you will need two nested *for* loops to iterate over the rows and columns of the matrix *mp*. We typically use the variables *i* and *j* to iterate over rows and columns respectively. For each *i* and *j*, we want to
      i. Extract the relevant sub-matrix of *gim* as specified above.
      ii. Calculate the maximum value within the sub-matrix. You can use the second and third arguments as: **`[],'all'`** , to specify that you want the maximum value among all elements of the sub-matrix. Without these additional arguments, the command would return the maximum values of each column. Try entering **`help max`** into the Command Window for more information.
      iii. Assign the maximum value to the *i*th row and *j*th column of the matrix *mp*.

6. **Plot the max pooling matrix "mp" in the third subplot.**
   Recall the figure that we created earlier using the **`figure`** command and the index. Add a third subplot, show the image with **`imagesc`** and turn the axes off as we did in part 2.4.

EXTRA TASK: Try carrying out pooling multiple times to see the effects.

Exercise 3: Blurring an image

In this exercise, we will import an image into Matlab and use a similar sliding window approach from Exercise 2 to create a blurred effect. This is a similar concept to the smoothing that we carried out in Exercise 1.

1. **Load the image "InfoLab.jpg". This image will be imported as an 8-bit data type. Convert it to a "double" type and divide the values by 255. Resize the image to 256x256x3.**
   Similar to part 2.1, we can import the image into a variable called *im* using the **`imread`** command. It can be converted to a double format using the **`double`** command. The **`imresize`** command can be used to rescale the image, with the double image variable name as the first argument and the new image size as the second argument in the format **`[256,256]`**. Note that we do not need to specify the size of the third dimension; this will be left unchanged.

2. **Create a new figure, display the resized image in the first subplot of a 1x3 subplot array.**
   We did this in part 2.3.

3. **Create a new matrix of zeros to hold the blurred image. It should be the same size as your image, i.e. 256x256x3.**
   We did using the **`zeros`** command in part 2.5.a.

4. **Use a sliding window approach to create a blurred version of the image with a 5x5 matrix of 1's, divided by 25. I.e. if your image is stored as a variable *I* and your new matrix is stored as *B*, you would calculate the first entry *B(3,3,1)* as:**

$$\boldsymbol{B(3,3,1) = \frac{1}{5 \times 5} \sum_{x=3-2}^{3+2} \sum_{y=3-2}^{3+2} I(x,y,1)}$$

   That is, the mean of the surrounding 5x5 matrix, with the entry B(3,3,1) at the centre of the matrix.

   For the *i*th row, *j*th column and *k*th channel, we would calculate the corresponding entry of *B* as

$$\boldsymbol{B(i,j,k) = \frac{1}{5 \times 5} \sum_{x=i-2}^{i+2} \sum_{y=j-2}^{j+2} I(x,y,k)}$$

   This is very similar to the sliding window approach that we used in 2.5.b. We'll need two nested *for* loops to iterate over the rows and columns. In this case, we have a colour image with 3 channels, so we will also need to iterate over the third dimension, i.e. *k=1:3*. You can ignore the two-pixel boundary of *B*. I.e. your for loops only need to consider from 3 up to 254. Consider why this is.
   For each *i*, *j* and *k*, we need to extract the submatrix of the image *I* as
   `I(i-2:i+2,j-2:j+2,k)`.
   Calculate the mean value of the submatrix using the **`mean`** command with **`'all'`** as a second argument and store it in **`B(i,j,k)`**.

5. **Display the matrix B in the second subplot of the figure and consider the difference between these.**
   This is referring to the same figure we created in part 3.2.

6. **Use a convolution operation to carry out the same blurring operation on the image using the conv2 function and display the result in the 3rd subplot. For this, you will need to specify a kernel matrix:**

$$ker = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 9$$

   We can start by creating the kernel function using the **`ones`** command to create a matrix of ones, divide the matrix by 9 (the sum of the matrix) and assign to a new variable called *ker*. Next, create a new matrix of zeros with the same size and number of channels (3rd dimension) of the image.
   Since we will use the **`conv2`** function, which only applies to a 2-dimensional matrix, we will need to use a *for* loop to iterate over the third dimension.
   At each point in the *for* loop, calculate the convolution using the **`conv2`** function where
   a. the first argument is a particular channel of your image. E.g. channel 1 would be **`im(:,:,1)`**
   b. the second argument is your kernel matrix
   c. we will use **`'same'`** for the third argument to specify that the output of the function will have the same number of rows and columns as the image.

   Finally, we just need to display the image in a third subplot of the figure that we created in part 3.2.

7. **Give titles to each of the subplots and an overall title to the figure.**
   To give each subplot a title, we need to specify the figure and subplot and then use the **`title`** command. To give a title to the overall figure, we can use the **`sgtitle`** command.

EXTRA TASK: Try using different blur kernels with the convolution approach. You can formulate these by changing the kernel matrix directly or create new ones with the **`fspecial`** command.

Exercise 4: Edge Detection

In this exercise, we will import an image into Matlab and use a similar approach to Exercise 3 to create an edge-detection effect using different kernels.

1. **Load the image "InfoLab.jpg", convert it to a "double" type and divide the values by 255, and convert the image to grayscale by taking the mean of the three channels. Create a 2x2 figure, display the grayscale image on the top-left.**
   We have covered these steps in parts 3.1, 2.3 and 3.2.

2. **Use convolution to perform edge detection, i.e. carry out a convolution of the image with the kernel k_x:**

$$G_x = k_x * I, \qquad \text{where} \quad k_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

   and *I* denotes the grayscale image. Plot the result on the top-right of the figure.
   For this, we can use the same **`conv2`** function that we used in part 3.6. We will need to

create the kernel matrix explicitly; e.g. the command **`kx = [1, 2; 3, 4]`** would create the matrix

$$kx = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

3. **Use convolution to perform edge detection with the kernel k_y:**

$$G_y = k_y * I, \qquad \text{where } k_y = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{-1} & \mathbf{-2} & \mathbf{-1} \end{bmatrix}$$

and plot the result on the bottom-left of the figure.
For this, we follow the same steps as part 4.2.

4. **Calculate the gradient magnitude as**

$$G = \sqrt{G_x^2 + G_y^2}$$

and plot the result on the bottom-right of the figure.
For this we will need to use the pointwise-square operator (e.g. **`Gx.^2`**) to calculate the square of each element of the matrices *Gx* and *Gy*. The square root can be found using the **`sqrt`** command.

5. **There are many established edge detection filters that can be implemented easily in Matlab, including Sobel, Prewitt, etc, via the edge function. Try these and compare with the edge detection that you carried out above.**
For the **`edge`** function, the first argument is the image matrix and the second is the name of the filter. You can find the names of the filters from the help page using **`help edge`**.

EXTRA TASKS:

1. In the 4th step in this exercise, the edge detection is displayed as a grayscale. Try showing the figure with a threshold and note how it changes as you vary the threshold.
2. Try plotting the images from parts 4.2 and 4.3 using the absolute values of the matrix values.