ENGR101 Lab 3

Submission Requirements:

All labs are due at 11:59pm the day after your lab section. See Part 3 of this lab on how to submit these files.

- Exercise 1: Save your work as lab3_1_recolor.m and lab3_1_test.m
- Exercise 2: Save your work as lab3_2.m

Part 1: Writing User-Defined Functions

Reminder from Lab 2

Recall from Lab 2 that functions allow the programmer to write an algorithm once and use it many times. Just like functions in math, they often take in an input(s) and generate an output(s).

MATLAB has some built-in functions for you to use. For instance, so far you have seen sqrt(), max(), zeros(), etc. In MATLAB, you can also write your own functions, called user-defined functions. In this lab, you will be learning how to do this.

Function Syntax

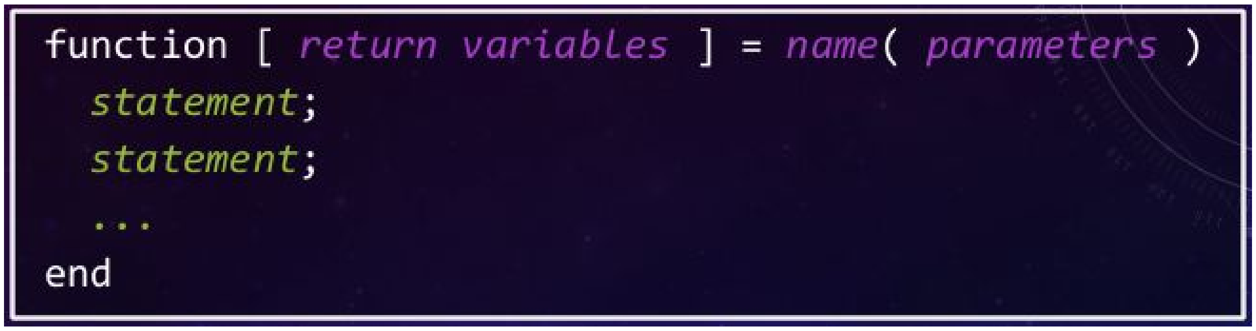
User-defined functions can be used in the command window, in a script, or within another function. A function in MATLAB consists of the following:

- The function header in the first line consists of the word “function”, a set of output arguments enclosed in square brackets, the assignment operator (“=”), the function name, and the input arguments enclosed in parentheses. **Note the function name must correspond to the file name.** In this class, each file will only contain one function.
  ```
  function [output_args] = function_name(input_args)
  ```
  Example:
  ```
  function [c] = pythagoras(a, b)
  ```
- The next line should be a comment describing what the function does. This comment will be displayed when the user types help followed by the function name in the command window.

- The next lines comprise the body of this function, which consists of all statements, and **must assign values to the output variables at some point**.
- The last line should be the word "end", denoting the completion of the function. Note that this is optional in the scope of this class.
- The function should be saved in a file called <name of function>.m. **If the file name does not exactly match the function name, the function cannot be called correctly.**

  Example:
  For a function with the header `function [c] = pythagoras(a, b)`
  The file name would be pythagoras.m

Below is a summary of the key parts of a function.



```
function [ return variables ] = name( parameters )
  statement;
  statement;
  ...
end
```

Example

Let's use this information to create our new square root function, `mysqrt`. We'll start with the function header. When we use `sqrt` we give it one input and expect one output. Thus for our function we will do the same.

Now, in this function we are expected to assign something to our output variable. For this function, the output variable should be the square root of `in`. We can create any number of variables we want inside this function to accomplish this task. However, those variables will not be provided to the Workspace. Remember that the Workspace is where we can find all of the variables available for use in the command window and in scripts. Only the value of `out` will pop out of this. Thus the body of the function could look like this:

```
function [out] = mysqrt(in)
      half = 1/2;
      out = in .^ half;
end
```

Now we can use our function. Note that since `half` and `out` are declared within the function, we can **not** use them outside of the function (for example, in a script that calls the function `mysqrt`. Download **mysqrt.m (note that the name of the file is exactly the name of the function)** from Google Drive and run it in your command window by typing the following:

```
>> three = mysqrt(9);
```

Functions vs. Scripts

Program files (.m files) can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output. Both scripts and functions contain MATLAB code, and both are stored in text files with a .m extension.

We've been using scripts in lab so far. Scripts are collections of commands that execute in order when you run the script. In contrast, the first line in a function file is the function header, sometimes called the function definition (see above). Scripts also affect your workspace: variables created in your script will be added to your workspace and will stay there unless you clear the workspace (type *clear* into your command window). This means that script variables have "global" scope. In contrast, any variables created inside functions will never be added to the workspace, meaning that function variables have "local" scope, that is, local to the function.

Built-in Function Example: Displaying values

There are three ways to display values in MATLAB. Try the following commands in your command window. One should show something different than the others.

```
>> x = 5
>> disp(x)
>> display(x)
```

You can also use these to display vectors:

```
>> disp([4 3 7 8])
```

Part 2: Logical Arrays

Logical Expressions

In MATLAB, you use logical expressions to compare statements from left to right. The possible operators that are used in logical expressions are listed in the following table:

Symbol	**Operation**
< , <=	less than, less than or equal to
> , >=	greater than, greater than or equal to
==	equivalent to
~=	not equivalent to
& , |	AND, OR
~	NOT

We evaluate logical expressions by determining the truth of the statement. A true statement will evaluate to logical 1, a false statement to logical 0. In this situation, the 1 and 0 are logical numbers meaning “true” and “false”, and are treated differently than scalar 1s and 0s we use for math. The result of a logical expression has the “logical” type while an arithmetic expression will likely have the “double” type (Try using `whos` after doing a logical comparison). For example, if you were comparing the scalars 1 and 2, you would get:

```
>> 1 > 2
ans =
     0
>> 1 < 2
ans =
     1
```

Logical Arrays

We can use logical expressions to create logical arrays. When you use these operators on matrices, the effect is to perform element-by-element comparisons:

```
>> m = [ 1 5 3; 6 8 2 ];

>> m > 4

ans =

     0     1     0

     1     1     0
```

Logical Indexing

Recall in Lab2, we learned how to index matrices using parentheses:

```
>> m = [2 3 4 5; 6 7 8 9]
>> m(2, 1:2:end)          % Select second row elements
                      % with odd-valued column indices
ans =
     6     8
```

In this example, we are using the element positions in the array to select certain elements. Another way that we can select certain elements in an array is through logical indexing.

Logical indexing is selecting a subarray based on some *condition on the values of the elements*. When you use logical operators to compare a matrix to a scalar value, or to another matrix of the same size, the result is a “logical matrix,” which consists of logical 1s and 0s.

Once we have generated our logical matrix, we use it as an index in whatever matrix we want to change. Only locations that have a 1 will be affected by any changes we apply. Logical indexing is thus extremely useful when the elements we wish to select all have something in common but there is no easily detectable pattern in their subscript indices.

```
>> mat = [1:4; 5:8; 9:12; 13:16]
mat =
    1    2    3    4
    5    6    7    8
    9   10   11   12
   13   14   15   16

>> mults_of_3_logic = (mod(mat, 3) == 0)
mults_of_3_logic =
    0    0    1    0
    0    1    0    0
    1    0    0    1
```

Note how the above statement includes both the logical operator (==) and the assignment operator (=). While it looks strange, this is the valid way to use logical indexing. Of course, we can apply any other logical operators we know, too. When two arrays of the same size appear in a compound (and/or) statement, each element can be compared individually using the elementwise operators `&` and `|`. Since `mults_of_3_logic` was created using logical operators, it is a logical matrix (not a matrix of mathematical 1s and 0s). You can verify this by hovering over the variable in the workspace or by typing "`whos`" into the command window.

Modifying Matrices

So far, we've talked about selecting particular elements in a matrix by various forms of indexing. Using this information to modify your matrix takes only a little more work if you follow one simple rule:

The dimensions on the right must match the dimensions on the left.

Let's start with an example of **what not to do**. Say you are trying to add 1 to every element greater than 3. The issue with the code below is that `mat(mat>3)` consists of the 13 elements larger than 3, while `mat` holds all 16 elements.

```
>> mat = [1:4; 5:8; 9:12; 13:16]
mat =
    1    2    3    4
    5    6    7    8
    9   10   11   12
   13   14   15   16
```

```
>> mat(mat > 3) = mat + 1
In an assignment A(:)=B, the number of elements in A and B must be
the same.
```

In order to fix this code, we must ensure that the right side of the assignment has the same number of elements:

```
>> mat(mat > 3) = mat(mat > 3) + 1
mat =
    1    2    3    5
    6    7    8    9
   10   11   12   13
   14   15   16   17
```

Next let's study the following code. In this case, we get an error because the dimensions on the left (2 rows, 2 columns) do not match the dimensions on the right (1 row, 4 columns). Instead we must ensure that the dimensions on the right are the same (2 rows, 2 columns).

```
>> mat = [1:4; 5:8; 9:12; 13:16]
mat =
    1    2    3    4
    5    6    7    8
    9   10   11   12
   13   14   15   16

>> mat(1:2, 1:2)
ans =
    1    2
    5    6

>> mat(3,:)
ans =
    9 10 11 12

>> mat(1:2, 1:2) = mat(3,:)
Subscripted assignment dimension mismatch

>> [mat(3,1:2); mat(3,3:4)]
ans =
    9   10
   11   12

>> mat(1:2, 1:2) = [mat(3,1:2); mat(3,3:4)]
```

```
mat =
     9    10     3     4
    11    12     7     8
     9    10    11    12
    13    14    15    16
```

There is one exception to this rule, and it involves scalar values. Just like you can add 1 to a matrix just by `mat + 1`, you can set a collection of values equal to a scalar value:

```
>> mat = [1:4; 5:8; 9:12; 13:16]
mat =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16

>> mat(mat >= 12) = 0
mat =
     1     2     3     4
     5     6     7     8
     9    10    11     0
     0     0     0     0
```

Part 3: Turning in Your Assignments

Labs must be turned in through the relevant assignment in Canvas. Be sure to submit all files listed under Submission Requirements.

The name of your files must be exactly as specified in order to receive credit.

Exercise 1: Finding Function Errors

A group of Ohio State students developed a malicious algorithm that destroys the elegant coloring of pictures containing maize and blue! They have corrupted some University of Michigan images.

Fortunately for the University of Michigan administration, an ENGR 101 GSI devised a cunning algorithm to reverse the evil coloring. Unfortunately, the GSI is busy helping students in office hours: it is your task to help the GSI by fixing the errors in the code they threw together late in the night.

Your Task

Download `lab3_1_recolor.m`, `lab3_1_test.m`, `evil.jpg`, and `evil2.jpg`. The `lab3_1_recolor.m` is the function, and the `lab3_1_test.m` is the test script that calls your function. Between the two code files, there are **5 errors** you need to find and document. Fix each error and add a short comment describing what you did to fix the problem.

Hint: You haven't learned images yet, so the errors are likely regarding function syntax and logical arrays

Testing and Submission

Run the test script on `evil.jpg` and `evil2.jpg` to test that the function works. If fixed correctly, running the test script will run without errors and show a maize and blue image!

After fixing the function and the test script, submit `lab3_1_recolor.m` and `lab3_1_test.m`.

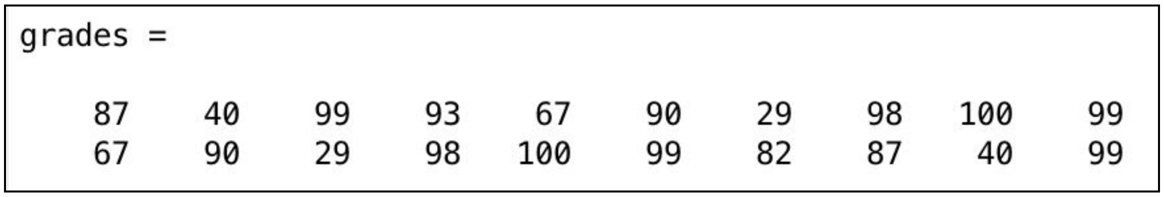
Exercise 2: Writing Your Own Function

It is the end of the semester, and the ENGR 101 GSIs are calculating final grades. Unfortunately, it looks like some grading errors have occurred, which need to be fixed! Fortunately, with all of the MATLAB matrix manipulation that you have learned, the instructors are confident that you will be able to make the adjustments quickly!

Your task is to write a function to perform adjustments and write a script to test the function. There is not a starter file for the function, so you must create `lab3_2.m`. The `lab3_2_test.m` script is provided for you, but you will have to update it to correctly call your function.

Function Inputs

We will write a function called `lab3_2` that takes **two matrices as input**. The first matrix is a 10x10 matrix called `grades` containing the lab grades for 100 students. The first two lines of an example are shown below.



```
grades =

    87    40    99    93    67    90    29    98   100    99
    67    90    29    98   100    99    82    87    40    99
```

We cannot use the `grades` matrix to tell which GSI a student is assigned to. Therefore, we also have a 10x10 matrix called `gsi` containing the GSI for the 100 students. This matrix only contains the numbers 0 through 8, where

0 == Amit	1 == Alex	2 == Barrett
3 == Josh	4 == Bethany	5 == Lizzy
6 == Nate	7 == Eric	8 == Jason

This `gsi` matrix can be thought of as a key for the `grades` matrix. For instance, `gsi(1, 1)` and `grades(1, 1)` are the same student (this student has a GSI of 7 and is in Eric's section; this student has a lab score of 1.5%).

The function will take a third input variable: the `curve`. Every student grade will receive the bonus of the curve.

Function Outputs

The `lab3_2` function will have only one output variable, a matrix called `new_grade`.

Test Script

Before implementing your function, it is important to complete the test script to understand how our function will be used. Download `lab3_2_test.m` add a new line to call our `lab3_2` function. Use a value of 10 for the `curve`.

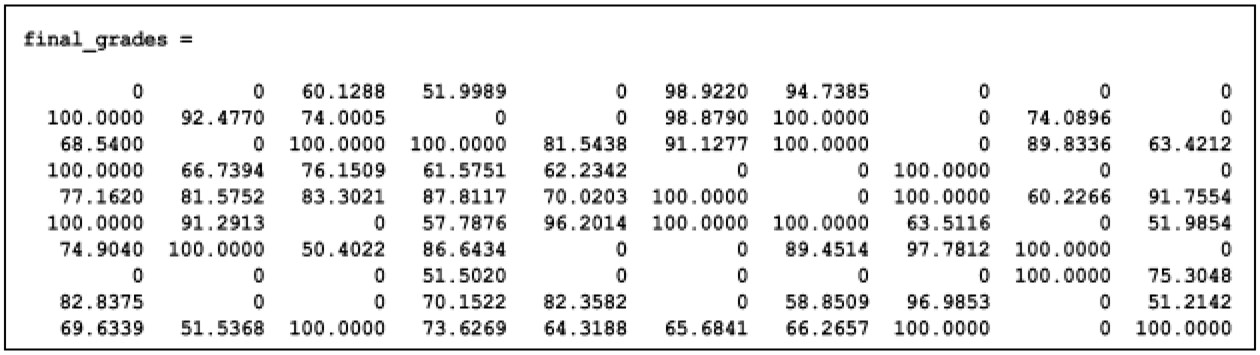
Function Requirements

Apply the following rules to output matrix `new_grades`:

1. Amit went a bit overboard, and gave all of his students three extra points on every lab assignment. To fix this, deduct three points from every student in one of Amit's sections
2. All students in Lizzy's and Jason's labs receive 10 extra points since Lizzy and Jason are awesome
3. All remaining students (not including Amit's, Lizzy's or Jason's students) receive 5 extra points (since actually all of the other GSIs are awesome)
4. Add the curve to every grade
5. Set all grades that are below 50 to 0
6. It is not possible to achieve a grade higher than 100% in ENGR 101. Set all scores greater than 100% to 100

Results

Once you have completed the function and the script, the updated grades matrix should match the following:



```
final_grades =

         0         0   60.1288   51.9989         0   98.9220   94.7385         0         0         0
  100.0000   92.4770   74.0005         0         0   98.8790  100.0000         0   74.0896         0
   68.5400         0  100.0000  100.0000   81.5438   91.1277  100.0000         0   89.8336   63.4212
  100.0000   66.7394   76.1509   61.5751   62.2342         0         0  100.0000         0         0
   77.1620   81.5752   83.3021   87.8117   70.0203  100.0000         0  100.0000   60.2266   91.7554
  100.0000   91.2913         0   57.7876   96.2014  100.0000  100.0000   63.5116         0   51.9854
   74.9040  100.0000   50.4022   86.6434         0         0   89.4514   97.7812  100.0000         0
         0         0         0   51.5020         0         0         0         0  100.0000   75.3048
   82.8375         0         0   70.1522   82.3582         0   58.8509   96.9853         0   51.2142
   69.6339   51.5368  100.0000   73.6269   64.3188   65.6841   66.2657  100.0000         0  100.0000
```

Submission

Submit the `lab3_2.m` function.