Self-Assessment: A1853059

Code Link:
https://github.com/GeraldFreis/Matlab-arithmetic-operations-solver-from-images/tree/allupdates

Areas of the rubric:

Conceptual Coverage

HD: All concepts covered mentioned in the practical are demonstrated and correct.

My work:

The Arithmetic equations recogniser and solver from images that I have produced utilises all of the key components mentioned in practicals. Such as, while loops, for loops, conditionals, functions, arrays, matrices, images and plotting. Examples of each are displayed beneath:

While Loops:

```
% logic for finding 1:
% 1 in this script will be proportional by some constant K
% we can take the top value of the 1, i.e the first 0
% and then move to the bottom. We can move left and then measure the distance we moved
% left we can then proportionally move right. We can then travel to the top
% and move at a 45 degree angle left

%% initialising the variables
test = false; % sentinal variable to control the loop
final_row = 0; % pre-initialising the variable that will contain the row where the code can iterate no further down-wards
increment_factor = 0; % this is the variable that will be increased in every loop of the while loop and will emulate a vector by changing the current
% row and or column or both for each iteration

%% iterating to the bottom of the number
while ~test
    if(matrix(row+increment_factor, column)==0 && row + increment_factor < 544) % if the pixel is no longer black and we have not exceeded the matrix
        % we have reached the bottom of the shape
        increment_factor = increment_factor + 1;

    elseif(matrix(row+increment_factor, column)~=0) % if the pixel is still black, continuing to increase the increment factor

        final_row = row + increment_factor-1;

        if(final_row - row < 50) % ensuring that the bottom of the shape is greater than 50 pixels away, as other shapes can be misrecognised if this buffer isn't her
            fprintf("Not a one\n");
            one = 0;
            return;
        end

        test = true; % ending the loop
    elseif(row + increment_factor == 544)
        fprintf("Not a one\n");
        one = 0;
        return;
    end
end

%% moving left and right at the bottom
increment_factor = 0; % resetting the increment factor
while test
    if(matrix(final_row, column-increment_factor)~=0)
        if(matrix(final_row, column+increment_factor-4)==0)  % iterating the same distance right as we did left and if this passes we have passed the test for the
            % bottom of the one
            % there is an offset here because the left side and the right side of the two
            % are not always the same distance, specifically for two digit operations
            test = false;
        else
            fprintf("No ones\n");
            one = 0;
            return;
        end
    else
        increment_factor = increment_factor + 1;
    end

end
```

For loops:

```
%% function to be universally implemented by all other finding shapes functions
% the function will take the direction to travel, the size of the vector, the current row, column and the
% matrix
% and it will return true if all pixels passed over by that vector are black
% and false if not

% possible directions are:
        % downleft, downright, upleft, upright, right, left, down, up
        % downleft = 45 degrees down and left (by increasing rows and decreasing columns in a 1:1 ratio
        % of increments)
        % downright = 45 degrees down and right (by increasing rows and columns by 1:1 increment ratio
        % upleft = 45 degrees up and left (by decreasing rows and columns by 1:1 increment ratio
        % upright = 45 degrees up and right (by decreasing rows and increasing columns by a 1:1 increment
        % ratio
        % right = increasing columns, left = decreasing columns, down = increasing rows, up = decreasing
        % rows


function boolval = findingnumbers(direction, length, current_row, current_column, matrix)
    boolval = false; % ensuring that the default value to return is false, as this will be returned if not every pixel iterated over was black

    try
        switch direction
            case "downleft" % if we need to move down and left, we increase row and decrease column
                for i = 1:length
                    if(matrix(current_row+i, current_column-i)~=0) % increasing the row and decreasing the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "downright" % if we need to move down and right we increase row and column
                for i = 1:length
                    if(matrix(current_row+i, current_column+i)~=0) % increasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upleft" % if we need to move up and left, we decrease row and column
                for i = 1:length
                    if(matrix(current_row-i, current_column-i)~=0) % decreasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upright" % if we need to move up and right we decrease row and increase column
                for i = 1:length
                    if(matrix(current_row-i, current_column+i)~=0) % decreasing the row and increasing the column
                        return;
                    end
```

Conditionals

- If else and switch statements

```
%% function to be universally implemented by all other finding shapes functions
% the function will take the direction to travel, the size of the vector, the current row, column and the
% matrix
% and it will return true if all pixels passed over by that vector are black
% and false if not

% possible directions are:
        % downleft, downright, upleft, upright, right, left, down, up
        % downleft = 45 degrees down and left (by increasing rows and decreasing columns in a 1:1 ratio
        % of increments)
        % downright = 45 degrees down and right (by increasing rows and columns by 1:1 increment ratio
        % upleft = 45 degrees up and left (by decreasing rows and columns by 1:1 increment ratio
        % upright = 45 degrees up and right (by decreasing rows and increasing columns by a 1:1 increment
        % ratio
        % right = increasing columns, left = decreasing columns, down = increasing rows, up = decreasing
        % rows


function boolval = findingnumbers(direction, length, current_row, current_column, matrix)
    boolval = false; % ensuring that the default value to return is false, as this will be returned if not every pixel iterated over was black

    try
        switch direction
            case "downleft" % if we need to move down and left, we increase row and decrease column
                for i = 1:length
                    if(matrix(current_row+i, current_column-i)~=0) % increasing the row and decreasing the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "downright" % if we need to move down and right we increase row and column
                for i = 1:length
                    if(matrix(current_row+i, current_column+i)~=0) % increasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upleft" % if we need to move up and left, we decrease row and column
                for i = 1:length
                    if(matrix(current_row-i, current_column-i)~=0) % decreasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upright" % if we need to move up and right we decrease row and increase column
                for i = 1:length
                    if(matrix(current_row-i, current_column+i)~=0) % decreasing the row and increasing the column
                        return;
                    end
```

Functions

- Functions just like conditionals were used throughout all of my code but here is an example


```
%% function to be universally implemented by all other finding shapes functions
% the function will take the direction to travel, the size of the vector, the current row, column and the
% matrix
% and it will return true if all pixels passed over by that vector are black
% and false if not

% possible directions are:
        % downleft, downright, upleft, upright, right, left, down, up
        % downleft = 45 degrees down and left (by increasing rows and decreasing columns in a 1:1 ratio
        % of increments)
        % downright = 45 degrees down and right (by increasing rows and columns by 1:1 increment ratio
        % upleft = 45 degrees up and left (by decreasing rows and columns by 1:1 increment ratio
        % upright = 45 degrees up and right (by decreasing rows and increasing columns by a 1:1 increment
        % ratio
        % right = increasing columns, left = decreasing columns, down = increasing rows, up = decreasing
        % rows


function boolval = findingnumbers(direction, length, current_row, current_column, matrix)
    boolval = false; % ensuring that the default value to return is false, as this will be returned if not every pixel iterated over was black

    try
        switch direction
            case "downleft" % if we need to move down and left, we increase row and decrease column
                for i = 1:length
                    if(matrix(current_row+i, current_column-i)~=0) % increasing the row and decreasing the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "downright" % if we need to move down and right we increase row and column
                for i = 1:length
                    if(matrix(current_row+i, current_column+i)~=0) % increasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upleft" % if we need to move up and left, we decrease row and column
                for i = 1:length
                    if(matrix(current_row-i, current_column-i)~=0) % decreasing the row and the column
                        return;
                    end
                end

                boolval = true; % as we have iterated the distance in the direction required and every pixel has been 0 or black
                % we know that we have passed this stage / vector test and we can return true to the calling
                % function
                return;

            case "upright" % if we need to move up and right we decrease row and increase column
                for i = 1:length
                    if(matrix(current_row-i, current_column+i)~=0) % decreasing the row and increasing the column
                        return;
                    end
```

Arrays:

```
%% function to retrieve the past results from pastresultsfile.txt and create two arrays
% one array is a list of indexs of the results and the other is the past results

function [index_array, past_result_array] = retrievingpastresults()

    % opening the file that contains the data
    file = fopen("pastresultsfile.txt","r+"); % opening as read only

    % taking the data from the file and adding it to the array of past results
    past_results = fscanf(file, "%f\n"); % this creates a n x 1 matrix, but I need a 1 x n matrix so we have to convert this
    fclose(file);

    %% converting the vertical matrix into a horizontal array
    past_result_array = zeros(length(past_results)); % initialising the returned array of past results as horizontal (1 x n)

    for i = 1:length(past_results)
        past_result_array(i) = past_results(i); % copying the current val of past results into our returnable vector
    end

    % initialising index array as the values from 1 to the length (amount of elements) of past result
    % array
    index_array = 1:length(past_result_array);
end
```

Matrices:

```
%% function to produce sub matrices for the FindingOperationsOneDigit and FindingOperationsTwoDigit classes
% the function will take the original matrix, the left column boundary and the right column boundary as
% parameters, and will iterate over every pixel in that range of the original matrix and return a new
% matrix

function submatrix = submatrices_maker(original_matrix, leftboundary, rightboundary)
    [totalrows, ~] = size(original_matrix); % finding the total rows and columns of the original matrix
    submatrix = zeros(totalrows, rightboundary-leftboundary); % initialising the submatrix

    try
        % iterating over every pixel in the range we need to iterate over
        for row = 1:totalrows
            for column = leftboundary:rightboundary
                submatrix(row, (column - leftboundary) + 1) = original_matrix(row, column); % setting each pixel as its counterpart of the original matrix
            end
        end
    catch % if the index of the matrix are negative or a non natural number we catch this errror
        fprintf("THE DIMENSIONS OF THE MATRICES PRODUCED FROM 'FindingOperations' ARE INCORRECT\n");
        return;
    end
end
```

Working with images:

```
%% function to convert the image to a cleaned matrix of its pixels (either black (0) or white (255))
function array_of_img = pixeltomatrix(pics)

    %% producing a matrix of gradients from the greyscaled image

    % initialising the image we want to read and producing the size of the
    % array we need
    img = imread(pics, 'png');
    [rows, columns] = size(img);
    array_of_img = zeros(rows, columns/3); % counteracting an error where the image is 3 times the size it needs to be

    % iterating through each pixel of the image and if it is black adding it to the matrix, otherwise
    % making making the element at (row, column) in the matrix white
    for row = 1:rows
        for column = 1:columns/3
            if(img(row, column) == 0)
                array_of_img(row, column) = img(row, column);
            else
                array_of_img(row, column) = 255;
            end

        end
    end

    fprintf("pixel to matrix compiled\n\n");
end
```

Plotting:

```
savingresult(solving_AO_operation(number_1, operator, number_2)); % solving the operation, printing the result and returning that to be saved
[xaxis, yaxis] = retrievingpastresults(); % retrieving past results and their indexs

% plotting the past results (yaxis) against their indexes (xaxis)
plot(xaxis, yaxis);
ylabel("Past Solutions");
xlabel("Solution Number (index)");
```

Value Add:

HD: Excellent functionality. Interesting code behaviour with substantial personal contribution

- Demonstration of creativity in design of interaction with users and or functionality.
- At least 3 extensions beyond minimal functionality and evidence of self initiated learning
- Able to answer *all* questions about code including explaining how to modify

My work:

For the extensions above what we were taught I utilised OOP, by building classes and using encapsulation. I also used asynchronous programming and error handling. These can once more be observed beneath:
I utilised vector based number recognition for recognising the numbers, which relied upon set vector lengths and directions. I outsourced this to the findingnumbers.m function in the findingnumbers directory, where I used a switch statement to check what direction the vector needed to iterate towards and the length the vector needed to be. If every pixel iterated over in the required direction of the required length, we had passed this test, and we could move on to the next vector to test.

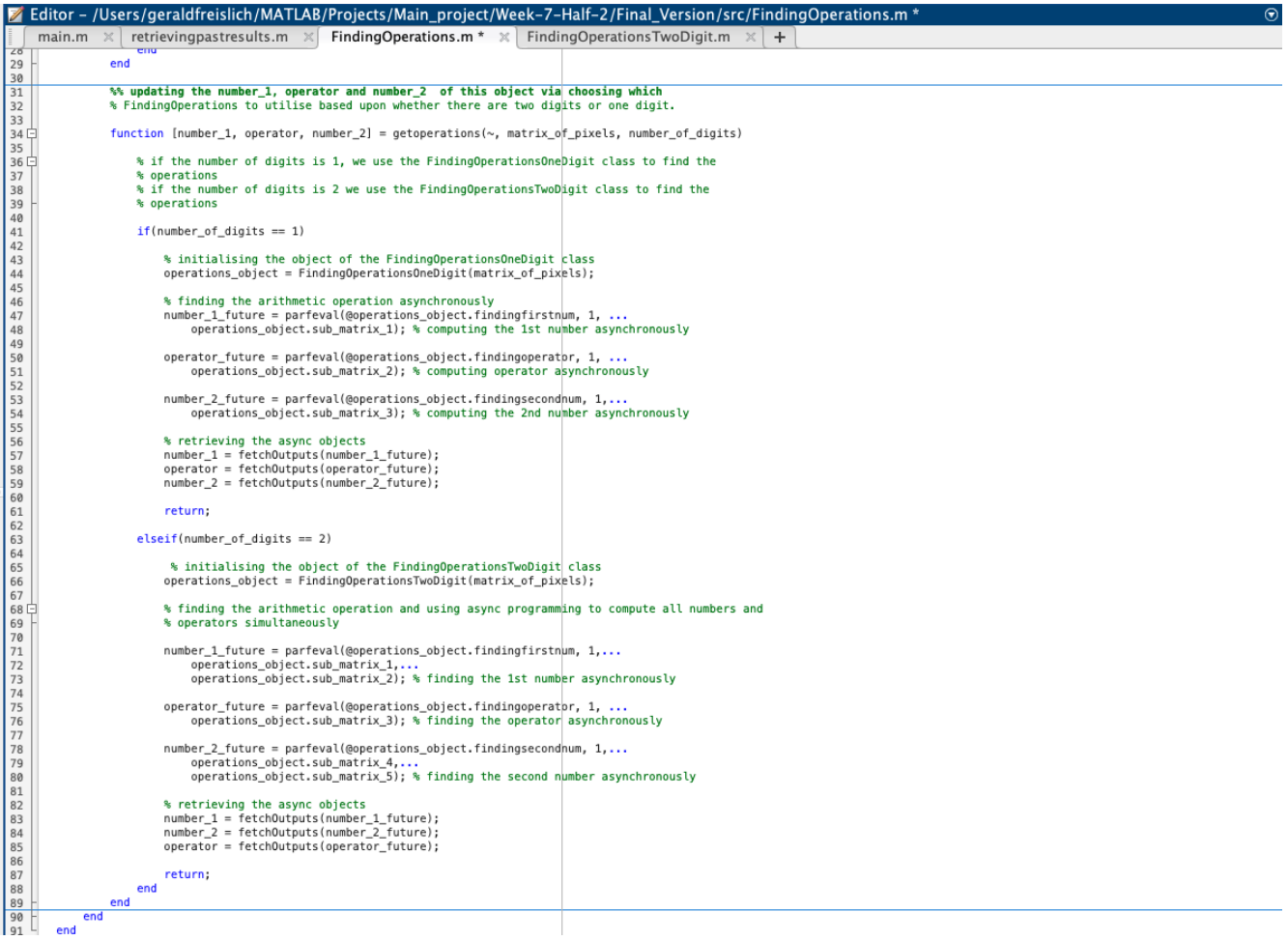
OOP and classes:

Editor - /Users/geraldfreislich/MATLAB/Projects/Main_project/Week-7-Half-2/Final_Version/src/FindingOperations.m

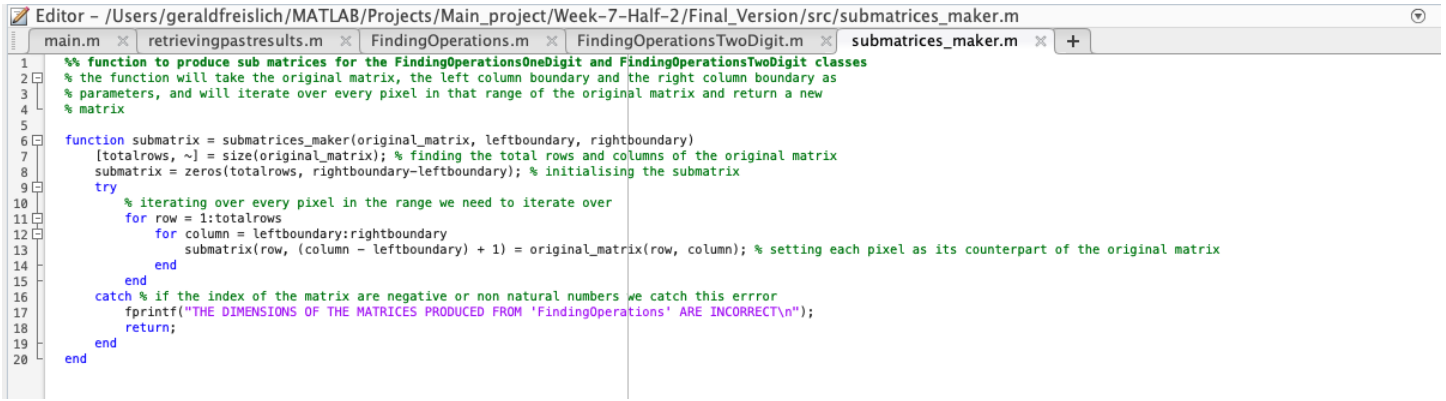
main.m | retrievingpastresults.m | FindingOperations.m

```
%% class that will handle finding the numbers / calling other classes which will find the individual numbers
% based upon whether it is a two digit or one digit operation
classdef FindingOperations

    properties
        image_location
        matrix_of_pixels
        number_of_digits
    end

    methods
        function obj = FindingOperations(image_filename, num_of_digits) %% constructors
            addpath("testing_images/"); % enabling image files to be accessed

            if (nargin == 0) % if no args are provided / default constructor
                % initialising variables

                obj.image_location = 'testing_images/88_times_88.png';
                obj.matrix_of_pixels = zeros(544, 900);

            elseif(nargin > 0) % if arguments are provided / parameterized constructor

                % creating / assigning the properties of the class
                obj.matrix_of_pixels= pixeltomatrix(image_filename); % constructing the matrix of pixels
                obj.image_location = image_filename;
                obj.number_of_digits = num_of_digits;

            end
        end

        %% updating the number_1, operator and number_2  of this object via choosing which
        % FindingOperations to utilise based upon whether there are two digits or one digit.

        function [number_1, operator, number_2] = getoperations(~, matrix_of_pixels, number_of_digits)

            % if the number of digits is 1, we use the FindingOperationsOneDigit class to find the
            % operations
            % if the number of digits is 2 we use the FindingOperationsTwoDigit class to find the
            % operations

            if(number_of_digits == 1)

                % initialising the object of the FindingOperationsOneDigit class
                operations_object = FindingOperationsOneDigit(matrix_of_pixels);

                % finding the arithmetic operation asynchronously
                number_1_future = parfeval(@operations_object.findingfirstnum, 1, ...
                    operations_object.sub_matrix_1); % computing the 1st number asynchronously

                operator_future = parfeval(@operations_object.findingoperator, 1, ...
                    operations_object.sub_matrix_2); % computing operator asynchronously

                number_2_future = parfeval(@operations_object.findingsecondnum, 1,...
                    operations_object.sub_matrix_3); % computing the 2nd number asynchronously

                % retrieving the async objects
                number_1 = fetchOutputs(number_1_future);
                operator = fetchOutputs(operator_future);
                number_2 = fetchOutputs(number_2_future);

                return;

            elseif(number_of_digits == 2)
```

Asynchronous Programming



Error handling:



Incremental development:

HD: Substantial evidence of development through well commented intermediate files embedding small increments of development. And, where-appropriate files implementing individual components of the solution

My work:

I kept logs on github of my clean and well documented code, aiming to have around 2-3 updates per week. I have readme's on each github directory for each update which includes what was changed. I broke functions into individual files from the beginning, and files with similar ideas or functions into directories as is good practice.

Testing:

HD: Evidence of careful stages of each stage of program development and testing of individual program components

My Work:
Throughout this project I separated individual stages of the project into weeks. Regarding week 3, for example, my aim was simply to take an image, grayscale and deblur the image, and then convert it to a matrix. Which I did through the production of the scripts takingpics.m, takingpictures.m and convertingtomatrix.m. In Week 4, I created the findingshapes.m file which took the original matrix of pixels and ensured that every pixel was either black or white. This made sure that any script I made to find the numbers, would be able to take the black pixels and iterate across them. It also made it easier for me to read the matrix in matlab and determine the general proportions of each shape. In Week 5, I first made the script to find plus symbols, and I used a constant image called “plus_symbol.png” to reference the algorithm. I did the same for findingones.m, where I used ‘number_1.png’ to test against. I continued this process of having a constant image to refer to in Week-5. This can be seen in the Week-5-x-update directories in the github branch allupdates. When my code was able to recognise individual numbers in the constant images number_x.png in “Week-5-3rd update directory” I moved onto getting the code to recognise arithmetic operations. In Week-5-end I created a callerfunction.m which tested thirds of the image to find the number. In the coming updates we can see that each findingnumbers function and callerfunction slowly changed so that it would match the image of the arithmetic processes. Again I used constant images to test against, and with every image I made I would test if the code worked and would tweak the vectors I used to test if it was a specific number or operator. When I reached the end of week-6 my code recognised all one digit number arithmetic operations and could solve it. In Week-6-Weekend directory, I instituted two different caller functions, one that solved one digit operations and one that solved two digit operations. To test two digit operations I used constant images which had usually two or more different integers in the operation, alongside a different operator. I would use these to test my code, and again I would tweak my vectors so that all numbers were recognised. After the Week-6 weekend / beginning of Week-7 in Week-7-2nd-update, I created two classes for handling the different types of arithmetic operations, and I developed these through lots of testing, and we can observe the development of the classes over the coming weeks. Likewise, I tested submatrices_maker.m that whole time through simply trying different pre-existing or new images and seeing if there was a problem in how the matrices were set up. I refined the boundaries for each matrice / third or fifth of the image, and we can again see this in the github directories. This process of testing against the images was again used when I implemented asynchronous programming to handle the finding of each number or operator in the image.

Comments and Style
HD: Code consistently exhibits all elements of good code style consistently through all versions

My Work:
Throughout my github, it is observable that I have maintained all elements of good code style throughout all versions. Here are some images beneath.

- Week 3:

```
function convertingimage = pixeltomatrix
    %% initialising an object from the class in takingpictures
    pics = takingpictures;
    pics.takingpic  % calling the function takingpic --> which takes a picture and converts it to greyscale, and then saves it

    %% producing a matrix of gradients from the greyscaled image

    % initialising the image we want to read and producing the size of the
    % array we need
    img = imread("tempimg.png", 'png');
    array_of_img = [];


    % iterating through each pixel of the image and adding its greyscale  colour to the
    % matrix
    % we are going to do this by creating a new page of dimension 257*257
    % for each iteration and append this to the array_of_img vector

    for i = 1:257
        sub_array = [];  % subarray to hold the gradients of the pixel in row i column j,
        % this array will be concatenated to the final array
        for j = 1:257  % j = 1: 3: 257
            sub_array = [sub_array, impixel(img, i, j)];
        end
        array_of_img = cat(257, array_of_img, sub_array);
    end

    % ensuring that the program works
    fprintf("Ended\n");
    % returning the multi-dimensional array created
    return array_of_img;
end
```

Week-5:

```
%% checking if the number is 1
function  one = findingones(matrix)
    % finding the rows and columns of the image / matrix of pixels
    [rows, columns] = size(matrix);

    % iterating over every pixel in the image and checking, via a conditional, whether the pixel being iterated
    % over is black, and if so implementing our logic to test if it is a one.
    for row = 1:rows
        for column = 1:columns
            if(matrix(row, column)==0 && matrix(row, column+2)==0)
%                 fprintf("%d, %d", row, column)
                % logic for finding 1:
                % 1 in this script will be proportional by some constant K
                % we can take the top value of the 1, i.e the first 0
                % and then move to the bottom. We can move left and then measure the distance we moved
                % left we can then proportionally move right. We can then travel to the top
                % and move at a 45 degree angle left

                % initialising the variables
                test = false;  % sentinal variable to control our while loop
                final_row = 0;  % initialising the end row that will be travelled to
                decrement_counter = 0; % initialising the decrement variable that will be increased for every iteration of the loop
                 % this variable will test if the xth value beneath the row is still black, and if so continuing to iterate down
                 % if not we are at the bottom of the one and then can move proceed with our logic.

                while ~test % iterating to the bottom of the shape
                    if(matrix(row+decrement_counter, column)~=0)
                        final_row = row + decrement_counter-1;
                        test = true;
%                           fprintf("%d, %d\n", row+decrement_counter, column)

                    else
                        decrement_counter = decrement_counter + 1;
                    end
                end

                % moving left and right
                test = false;
                decrement_counter = 0;
                while ~test % moving left and incrementing the movement counter
                    if(matrix(final_row, column-decrement_counter)~=0) % decreasing the columns so that we iterate leftwards and checking if the pixel is black
                        % if the pixel is not black we want to take the distance we moved left, and if we can move that distance right and
                        % the pixel we land on when we move right is still black then we have passed this test.

                        if(matrix(final_row, column+decrement_counter)==0) % moving the distance we moved left right and che
%                             fprintf("A one was found\n");
%                             one = 1;
                           test = true;
```

- Week-6

```
function fives = findingfives(matrix)
    [rows, columns] = size(matrix); % finding the rows and columns of the matrix so that we can iterate over every pixel
     % logic for five:
    % fives, from the highest and most leftmost point have a right branch
        % for font 72 calibri we move 30 pixels right initially
        % if all of the pixels we iterate across are black then we have passed the vector test for the first stage of the 5
        % as in we know that the top of the number we are analysing complies with the top right branch of a 5
    % and then move down and around that curve
        % We will then move down 33 rows and this will bring us to the bottom of the vertical branch from the top left of the 5.
        % If all of the pixels in that movement down are black, we will continue with the rest of our tests which involve moving down and around the
        % curve of the 5.
        % we will then move 23 pixels to the right (23 columns) and this allows us to use a smaller number of diagonal vectors for testing
        % If all of the pixels in the previous test are black we can then move 16 rows down and 16 columns to the right.
        % just like the previous tests, we will move 18 down and to the left, which takes us inwards from the curve
        % if all of these pixels are still black we will move 27 to the left for the tail of the 5, and if these tests pass we have a 5.

    % iterating over every pixel in the matrix and checking if the pixel is black. If so, we implement our tests to see if the number attached
    % to the pixel is black
    for row = 1:rows
        for column = 1:columns
            if(matrix(row, column)==0) % checking if the pixel is black (0)

                % initialising variables
                test = false;  % sentinel variable to control the while loop
                xcrement_var = 0; % variable to contain the increment factor to increment over pixels by changing the x and y (column and row),
                % which will emulate a vector as the terminal point of the vector is changing with every change of the xcrement_var.
                stage = 1;  % this is going to be the variable that contains the sta
                % of the loop, i.e what vector to initialise

                while ~test
                    if(stage==1)% moving 30 columns to the right as this should take us to the end of the top branch of the 5. If this passes
                        % we have passed the first stage of our tests to check if the number is a 5.
                        if(matrix(row, column+xcrement_var)==0 ...
                        && xcrement_var < 30) % if the current pixel iterated over is still black and the length of the current vector is less than that requir
                            xcrement_var =  xcrement_var + 1;
                        elseif(xcrement_var == 30) % if every pixel iterated over was black we have made it to the end of this vector and hence can move to the
                            stage = stage + 1;
                            xcrement_var = 0; % resetting the increment variable
                        else % if a pixel we iterate over is not black returning to the main file as the vector test has not been passed
                            fprintf("Not a five\n");
                            fives = 0;
                            return;
                        end

                    elseif(stage==2) % moving 33 down to start moving right to reach the curve
                        if(matrix(row+xcrement_var, column)==0 ...
                        && xcrement_var < 33) % if the current pixel iterated over is still black and the length of the current vector is less than that requir
                            xcrement_var =  xcrement_var + 1;
```

- Week-7

```
function matrices = callerfunction_two_digit(matrix)
    [rows, columns] =  size(matrix);
    addpath("findingnumbers_functions/");  % ensuring I can access the findingnumbers functions as they are in
    % their own directory

    % initialising the bound for each number / character in the image
    column_bound_1 = 389; % representes the end column for the first number
    column_bound_2 = 435; % represents the end column for the second number
    column_bound_3 = 474; % represents the end column for the operator
    column_bound_4 = 518; % represents the end column for the 1st number in the second pair of digits
    column_bound_5 = columns; % the end column for the second number in the second pair of digits is the final column of the image

    % iterating over every pixel in each fifth and adding it to the matrix that represents each fifth
    first_fifth_pixels = zeros(rows, column_bound_1); % initialising the matrix to hold the pixels
    for row = 1:rows
        for column = 1:column_bound_1
            first_fifth_pixels(row, column) = matrix(row, column);
        end
    end

    % iterating over every pixel in the second fifth and creating the new matrix to hold this
    second_fifth_pixels = zeros(rows, abs(column_bound_2-column_bound_1));  % initialising the matrix to hold the pixels
    for row = 1:rows
        for column = column_bound_1:column_bound_2
            second_fifth_pixels(row, column-column_bound_1+1) = matrix(row, column);
        end
    end

    % iterating over every pixel in the third fifth and created a new matrix to hold these
    third_fifth_pixels = zeros(rows, abs(column_bound_3-column_bound_2));  % initialising the matrix to hold the pixels
    for row = 1:rows
        for column = column_bound_2: column_bound_3
            third_fifth_pixels(row, column-column_bound_2+1) = matrix(row, column);
        end
    end

    % iterating over every pixel in the fourth fifth and creating a new matrix to hold these pixels
    fourth_fifth_pixels = zeros(rows, (column_bound_4-column_bound_3));  % initialising the matrix to hold the pixels
    for row = 1:rows
        for column = column_bound_3: column_bound_4
            fourth_fifth_pixels(row, column-column_bound_3+1) = matrix(row, column);
        end
    end

    % iterating over every pixel in the fourth fifth and creating a new matrix to hold these pixels
    final_fifth_pixels = zeros(rows, abs(column_bound_5-column_bound_4));  % initialising the matrix to hold the pixels
    for row = 1:rows
```

- Week-8

```
%% class that will handle finding the numbers / calling other classes which will find the individual numbers
% based upon whether it is a two digit or one digit operation
classdef FindingOperations

    properties
        image_location
        matrix_of_pixels
        number_of_digits
    end

    methods
        function obj = FindingOperations(image_filename, num_of_digits) %% constructors
            addpath("testing_images/"); % enabling image files to be accessed

            if (nargin == 0) % if no args are provided / default constructor
                % initialising variables

                obj.image_location = 'testing_images/88_times_88.png';
                obj.matrix_of_pixels = zeros(544, 900);

            elseif(nargin > 0) % if arguments are provided / parameterized constructor

                % creating / assigning the properties of the class
                obj.matrix_of_pixels= pixeltomatrix(image_filename); % constructing the matrix of pixels
                obj.image_location = image_filename;
                obj.number_of_digits = num_of_digits;

            end
        end

        %% updating the number_1, operator and number_2  of this object via choosing which
        % FindingOperations to utilise based upon whether there are two digits or one digit.

        function [number_1, operator, number_2] = getoperations(~, matrix_of_pixels, number_of_digits)

            % if the number of digits is 1, we use the FindingOperationsOneDigit class to find the
            % operations
            % if the number of digits is 2 we use the FindingOperationsTwoDigit class to find the
            % operations

            if(number_of_digits == 1)

                % initialising the object of the FindingOperationsOneDigit class
                operations_object = FindingOperationsOneDigit(matrix_of_pixels);

                % finding the arithmetic operation asynchronously
```