1. **ASSIGNMENT DESCRIPTION**

* Programming language: C++
* Build:
* Usage:

1. **FUNCTIONS IMPLEMENTED**

*Manipulation classes are named after its corresponding function described in the assignment content. Those files can be found in folder “manipulations”.*

1. **Convert the image to grayscale**

* Convert2GrayOpencv.(cpp/hpp): This class makes use of a OpenCV built-in function to convert the input image of 3 channels to a grayscale image.
* Convert2GrayManually.(cpp/hpp): This class, on the other hand, converts the input 3-channel input image to a grayscale one by applying Luma formula. More specifically, for each pixel, I used the following formula to calculate the corresponding gray value based on red, green and blue values

In the input image, blue, green and red channel index are 0, 1 and 2, respectively.

1. **Cycle through the color channel of the image**

* IterateChannels.(cpp/hpp): This class maintains a variable called to determine the next desired channel to which the input image will be converted. When the next channel is determined, is used to split the input image to 3 channels.

1. **Convert the image to grayscale and smooth it**

* Based on the value of the track bar, I compute sigma and kernel size as follows:

With the above kernel size, around 95% of the distribution has already been cover. Then, a Gaussian kernel to be used to smooth the input image is constructed according the computed sigma and kernel size as follow:

Finally, all G’s values are normalized so that its sum is equal to 1.

* GaussianBlurOpencv.(cpp/hpp): This class uses to smooth the input image using Gaussian Blur
* GaussianBlurManually.(cpp/hpp): Instead of using built-in OpenCV function as above, this one create a gaussian kernel using and then apply to perform convolution.

1. **Convert the image to grayscale and perform convolution to calculate its derivative with respect to x**

* ComputeDerivativeWrtX.(cpp/hpp): In order to take the derivative of the input image with respect to x, I use Sobel x derivative kernel  
  In order apply the above kernel to the input image, I make use of . Whilst doing convolution, value of elements may exceed the range of CV\_8U which is [0…255], the output matrix should be of a larger type. Then, I normalize the output matrix back to the range of [0…255] as follows

Thanks to the built-in function of OpenCV, I can normalize the matrix without doing any loop

1. **Convert the image to grayscale and perform convolution to calculate its derivative with respect to y**

* ComputeDerivativeWrtY.(cpp/hpp): This class is quite similar to the previous one which is supposed to compute derivate with respect to x. The y derivative kernel should be

1. **Show the magnitude of gradient of the image normalized to the range [0...255]**

* ComputeGradientMagnitude.(cpp/hpp): In order to compute magnitude of gradient of the input image, I, first, calculate normalized derivative with respect to x, y using the same technique as mentioned in section 4, 5 and combine them as follow   
  Also, to avoid overflow when computing gradient, I use .

1. **Convert the image to grayscale and plot the gradient vectors**

* DrawGradient.(cpp/hpp): In order to plot all gradient vectors, this class need to calculate the angle between each gradient vector and the x-axis. Given derivatives w.r.t x and y, the following formula will calculate the corresponding angle   
  These gradient vectors then can be easily plotted using

1. **Convert the image to grayscale and rotate it**

* RotateImage.(cpp/hpp): This class, firstly, computes a 2D rotation matrix which corresponds to the input angle. helps us to do so. In details, the following matrix will be created