Clab – 1 Report

ENGN4528

NAME -> Tanmay Negi

Uid -> u6741351

Date : 06/04/2020

Task 1 : Basic Warmup

1. **a = np.array([[2,4,5] , [5,2,200]])** , forms a 2-dimensional array of shape 2*3 where row1 = [2 , 4 , 5] and row2 = [5,2,200]. Column1 = [2,5] , column2 = [4,2] , column3 = [5,200] . Resultant array is of type numpy.ndarray
2. **b = a[0 , :] ,** b will contain all the column elements at 0^{th} row from a , i.e b <- [2,4,5] and b.type is numpy.ndarray
3. **f = np.random.randn(500 , 1) ,** creates a numpy.ndarray of shape 500*1 and populate it with random floats sampled from a univariate gaussian distribution of mean = 0 , variance = 1
4. **g = f[f<0] ,** g will contain all the -ve elements from f.
5. **x = np.zeros(100) + 0.35** , creates a 1 d array of 100 elements each filled with zero and add 0.35 to each element via vectorize addition.
6. **y = 0.6 * np.ones(1, len(x))** , creates a numpy.ndarray of shape 1*100 <length(x) = 100> where each element is 1 , then perform a vectorize multiplication of 0.6 to each elements.
7. **z = x-y** , performs a vectorize subtraction of x by y
8. **A = np.linspace(1,200)** , return a 1d array of linearly separated 50 (default) elements with first element 1.0 and last being 200.0
9. **B = A[::-1] ,** returns elements from A , starting from the last index towards first index , in a step of 1 , in a nutshell B will contain A reversed
10. **B[B<=50] = 0** , replace all the elements in B, that are less than equal to 50 , by 0.

Task 2 : Basic Coding Practice

This task mainly aims on getting familiar with basic image processing libraries functionalities that we will be performing throughout this lab.

1. First will read the RGB image Lenna.png using cv2.imread('Lenna.png' , 0). Notice the flag 0 , this specifies that the image will be read in grayscale mode. Now to map the negative of this image we will just subtract each pixel value of our image by 255. Here's the result.

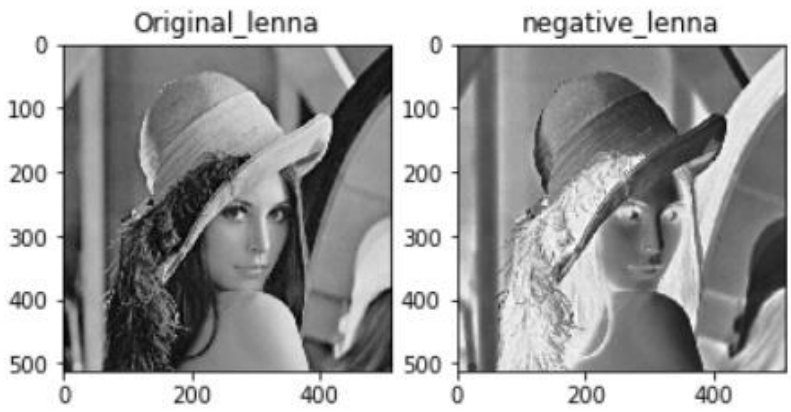


Fig 2.1

2. To flip the image we use similar concept as of array flipping as we saw in task 1.9 i.e: flipped = lenna[::-1] , here's the result

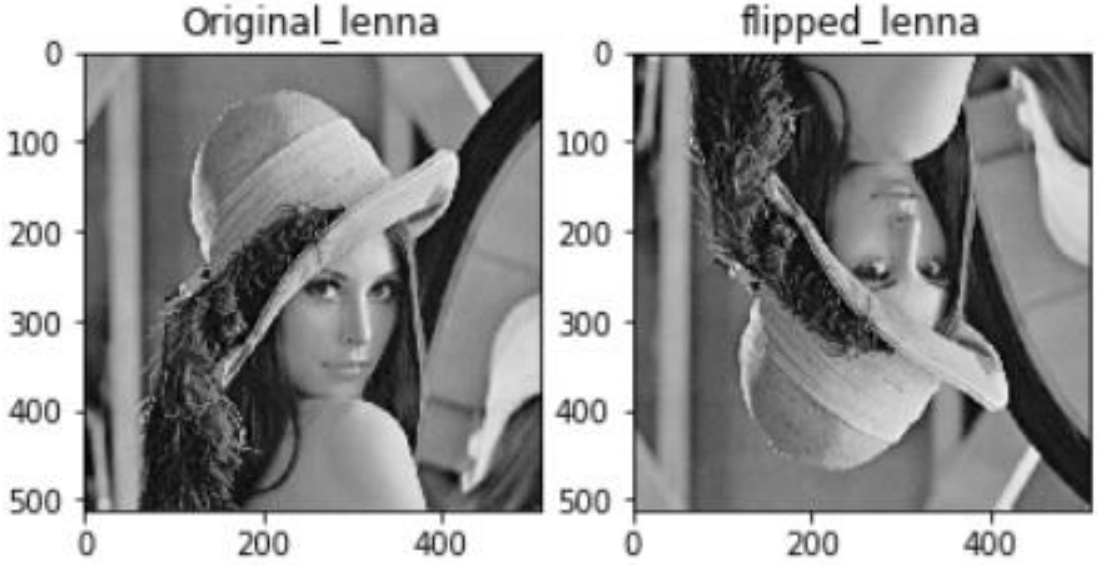


Fig 2.1

3. Now to load a colored image we again will use cv2.imread , but without any flag. The colored image that we read has a shape of n*m*3. Every colored image comprises of 3 grayscale channels red, green and blue as specified by the last element of it's shape. For example Lenna[: , : , 1] will give us the green channel of RGB lenna.png. thus to swap the red(0^{th}) and blue(2^{nd}) channel we will just swap Lenna[:,:,0] and Lenna[:,:,2] with each other. Here's the result

Fig 2.2

4. Here we will calculate the pixel wise average between grayscale original Lenna and its vertically flipped image.

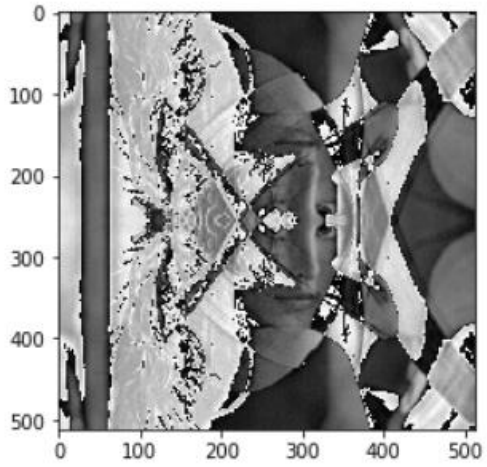


Fig 2.3 : Average between original lenna and flipped lenna

5. Now to add random noise between [0 , 255] to each pixel of our gray scale image we will use np.random.randint(low = 0 , high = 256 , size = lenna.size , dtype = np.uint8) to create a noise matrix.

Then we will add this to our grayscale lenna image. Here's the result:

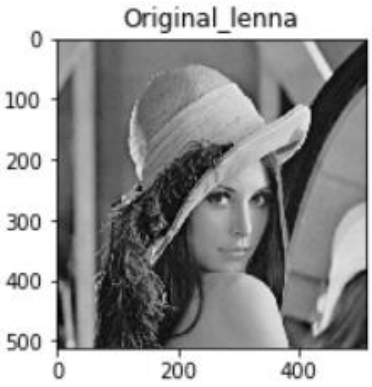


Fig 2.4 lenna

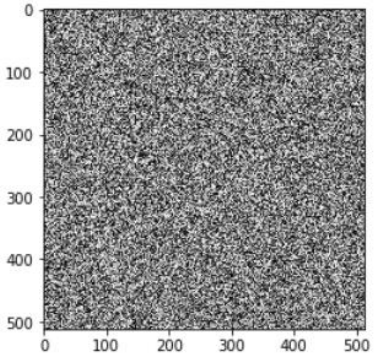


fig 2.5 noised_matrix

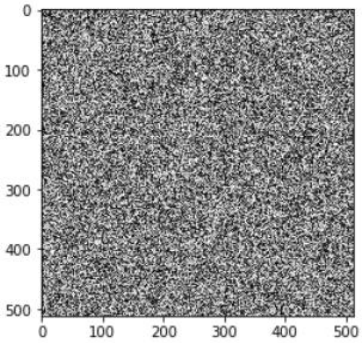


fig 2.6 uint8[lenna + noised_matrix]

Task 3

Overview:

This task is divided into three parts. **Part 1** and **2** demanded the student to take 3 portrait photo of their face and resize it to 1024*720 (columns*rows). Following are the photos I'll be working with, and which are provided in the zip folder.

Fig 3.1 face1

Fig 3.2 face2

Fig 3.2 face 3

Notice that all the photos are almost similar and only differ in lightning conditions. This is done so pixel matrix in all the three images are almost same and differ only slightly, due to variation in brightness.

It may be helpful later to see how different filters respond to this brightness alterations (if any), when applied to different images.

Documentation:

For **part 3** of this task I have built a short python function that'll perform tasks as specified.

Function:=> **operation(Image)**

@param Image Input RGB image

@output: > Displays 3 histogram for each R, G, B channel

> Displays 3 gray scale image after performing histogram equalization to each R , G, B channel

> Displays one RGB image after performing histogram equalization to input image.

Pseudo Code:

1. Resize input image to 768*512 {column*rows} using cv2.resize().
2. Split the image to individual channels
3. Calculate the histogram <cv2.calcHist()> for individual channels and display it.

4. Apply histogram equalization<cv2.equalizeHist()> on the individual channels and the original image and display it.

Note : <cv2.equalizeHist> takes only a gray image as input. To apply histogram equalization on RGB image, apply on individual grayscale scale channels and then merge them to form an RGB image.

Results:

Fig 3.4 input image (face 1)

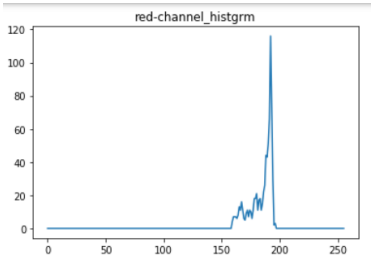


Fig 3.5 red_channel_histgrm

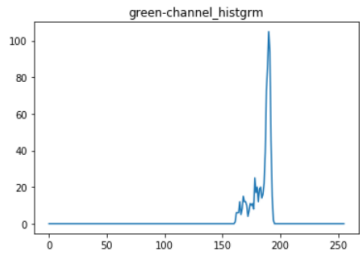


Fig 3.6 green_channel_histgrm

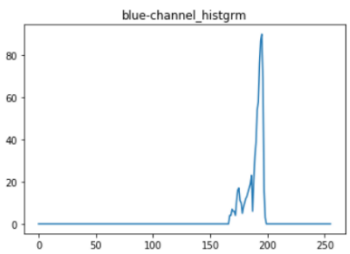


Fig 3.7 blue_channel_histgrm

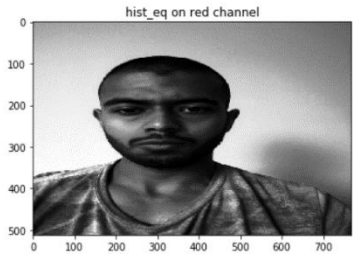


Fig 3.8 hstgrm equ on red_channel

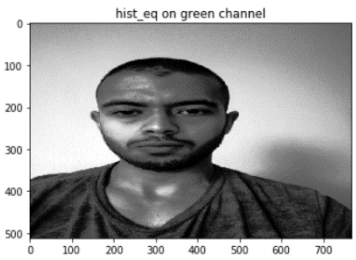


Fig 3.9 hstgrm equ on green_channel

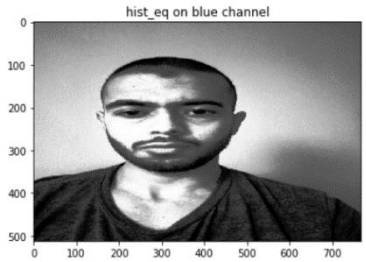


Fig 3.10 hstgrm equ on blue_channel

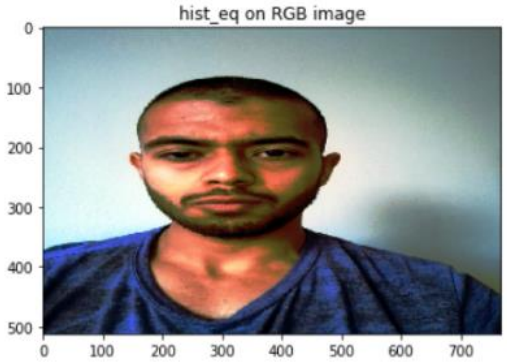


Fig 3.11 hstgrm equilization on RGB image

Task 4: Image Denoising via a Gaussian Filter

Overview:

The main aim for this task is to built a function that'll convolute a gaussian filter over an image to which noise has been added.

Implementation:

Following is the image we will use and adjacent to it is the image we get after adding a gaussian noise to this image.

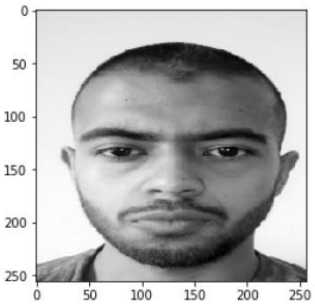


Fig 4.1 Input Image

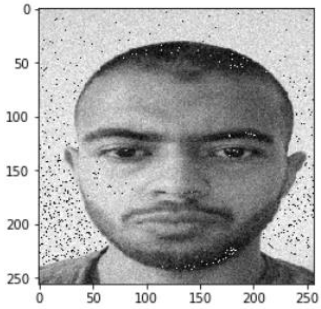
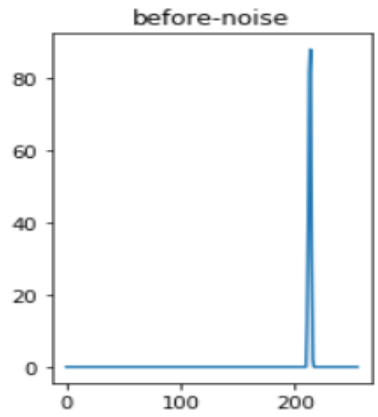
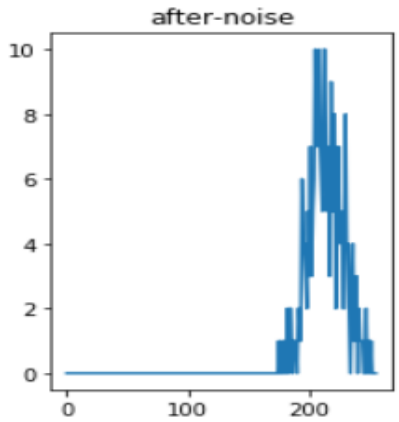


Fig 4.2 Input Image + Gaussian Noise

Following are the histograms of the corresponding images.





Now we will implement a function to convolute our guassian kernel over the noised image.

We will implement a self gaussian kernel according to the formula

$$G_{2D}(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Here σ -> standard deviation of the gaussian filter

x , y -> displacement coordinates of the current cell from the center

as we are using 5*5 gaussian kernel, center will be the cell (2,2)

a general insertion in a 5*5 gaussian kernel at i row and j column will look like:

```
y = i - 2
x = j - 2
ker[i , j] = (1 / (2*math.pi*math.pow(sig , 2))) * math.exp(-(x*x + y*y)/2*sig*sig)
```

Now for the next step we have defined a convolution function which will work as follows:

Documentation of function:

def convul (image , kernel)

…

return Image

@param image : input grayscale image over which kernel has to be applied

@param kernel : desired kernel

@return Image : convolution resultant image of @param kernel over @param image

Pseudo code:

1. Declare empty image1 of shape = @param image.shape
2. Map the @param kernel over @param image , element wise multiplying corresponding mapped cells.
3. Calculate the sum of this element-wise multiplication matrix and copy the sum to image1 at destination whose row and column number is the row and column number of the center cell of the mapped matrix in the image.
4. Slide the kernel to right by 1 stride and repeat steps 2-4
5. If @param kernel reached at the end of @param image width , move the kernel at the starting of row 2 , repeat 2-4
6. When kernel reached at bottom left end of the image , end the iteration and return image1

Result and analysis:

Here's our result when we apply convolution to input image with gaussian kernel of **sigma = 1**

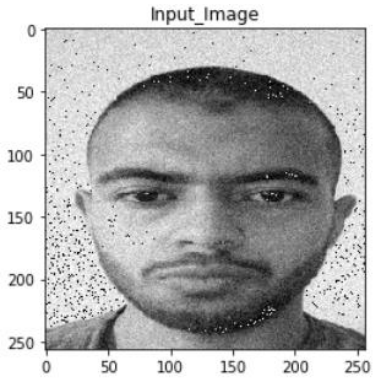


Fig 4.5 : Input Noised Image

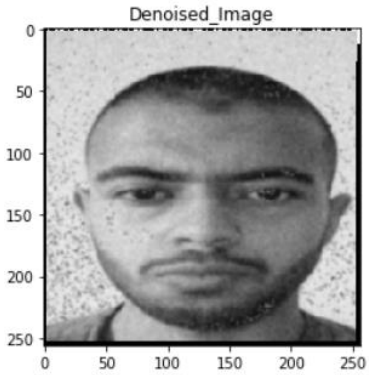


Fig 4.6 : Output Image from our convolution function

And to check our performance we will compare our function with python inbuilt function [cv2.GaussianBlur]

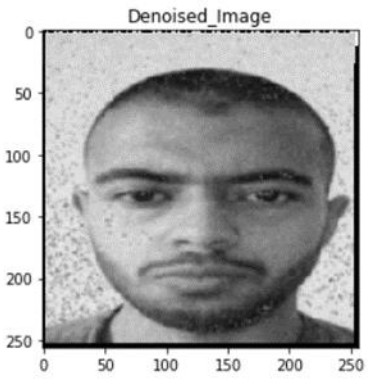


Fig 4.7 output image from custom function

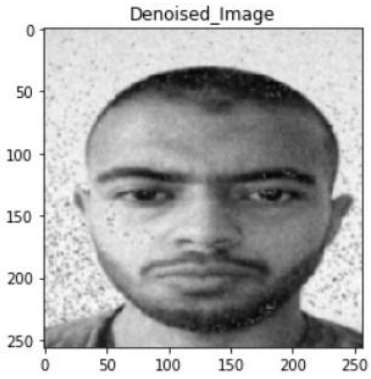


Fig 4.8 output from inbuilt python function

As we can see both images almost similar. We can further improve our function by padding our output image, i.e. removing those black lines you can see at borders of fig 1.

Note that the value of sigma (standard deviation) will alter our result. Here's some comparison of our output images when we change the value of sigma in our @param kernel in our function.

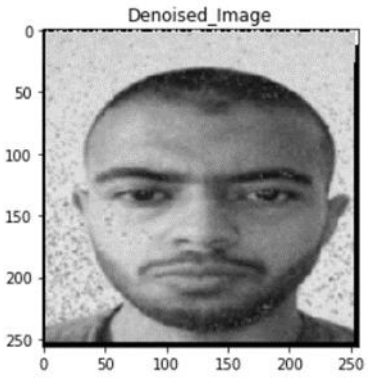


Fig 4.9 : Sigma = 1.0

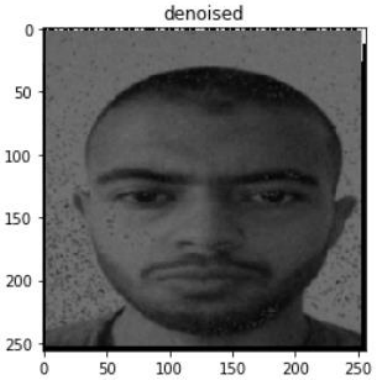


Fig 4.10 : Sigma = 1.2

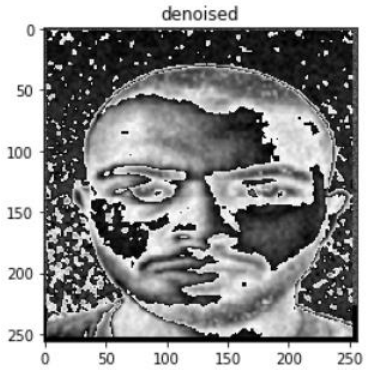


Fig 4.12 : Sigma = 0.78

Task 5:

Overview:

The main aim for this task is to implement 3*3 sobel filter for edge detection.

As discussed here, first we will convolute our x-gradient kernel $\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$ and then our y-gradient kernel $\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$ over our target image and we will approximate our gradient $\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$ to produce final output.

The convolution function is similar to what we used in task 4.

Implementation 1

Documentation of function:

def sobel (image , kernel.shape)

…

return Image

@param image input grayscale image

@param kernel.shape shape of kernel you want to apply

@return Image output image where edges are calculated

Pseudo code:

1. First, we will calculate x-gradient of our image by convoluting kernel Gx over our image

   i.e. sobel_x = convul (image , Gx)

2. Then we will calculate y-gradient of our image

   i.e. sobel_y = convul(image , Gy)

3. Then we will apply pixel wise gradient approximation on sobel_x and sobel_y to get our output image, i.e. sobel = math.sqrt (sobel_x * sobel_x + sobel_y * sobel_y)

Result :

Here's our result when we apply our function to following images.

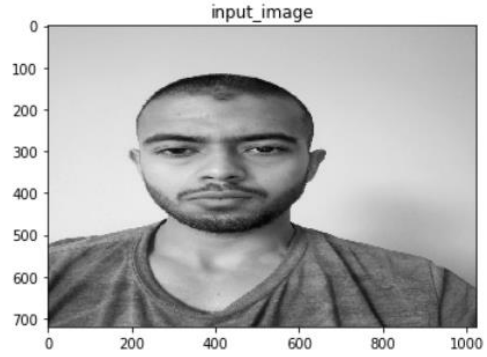


Fig 5.1 : Input Image 1

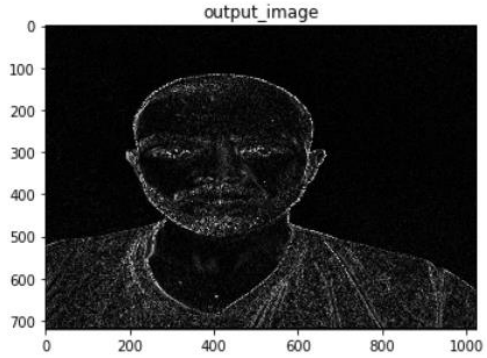


Fig 5.2: Output Image 1

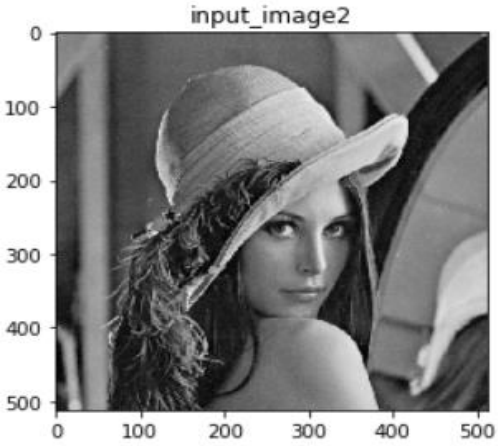


Fig 5.3. : input image 2

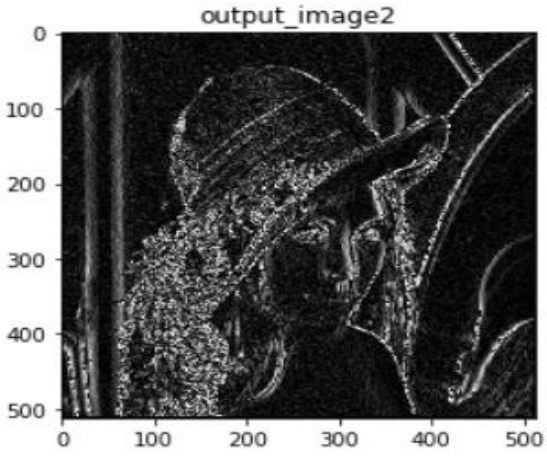


fig 5.4 : output image 2

Implementation 2:

We can further improve our sobel filter by reducing extra noises to just focus on edges. Everything in our function is same but just instead of passing raw image as input to our custom sobel function now we will pass the image after applying inbuilt gaussian blur to it. Here's the result

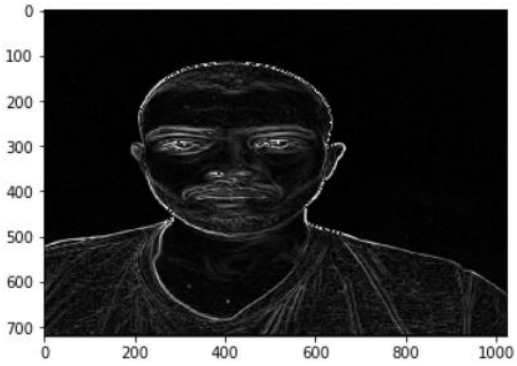


Fig 5.5: output of u_id_1.png with gaussian blur

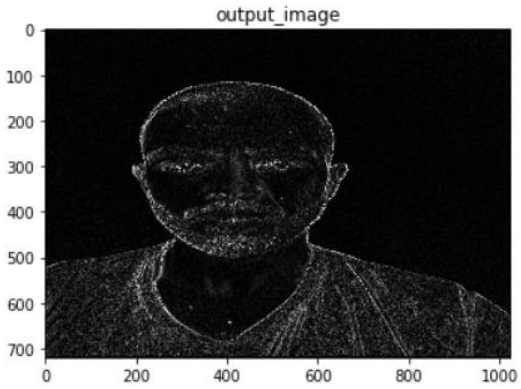


Fig 5.6 : output of u_id_1.png without gaussian blur

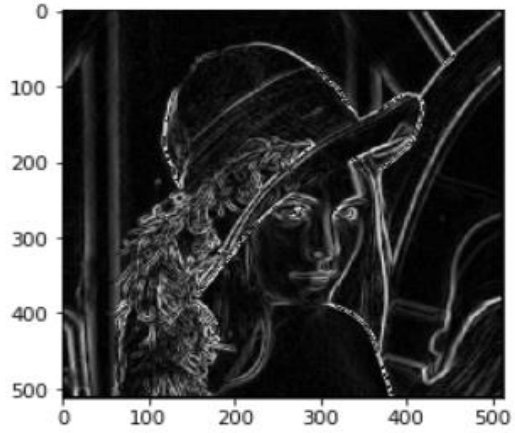


Fig 5.7 : output of Lenna.png with gaussian blur

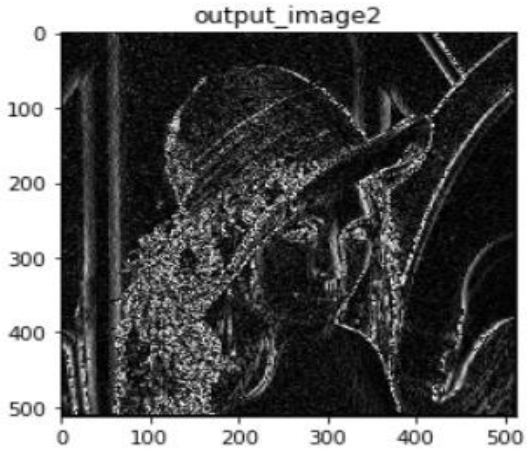
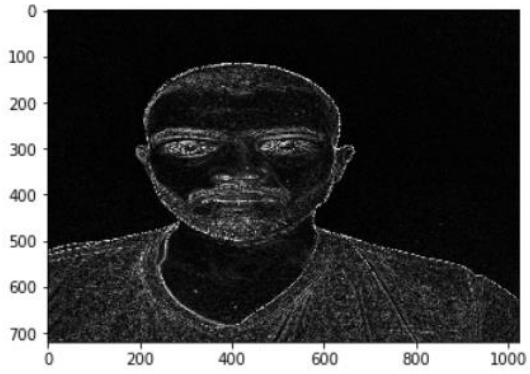


fig 5.8 : output of Lenna.png without gaussian blur

Analysis:

Now to check our custom sobel function we will compare its result with python inbuilt function cv2.Sobel



5.9 : inbuilt sobel on face 2

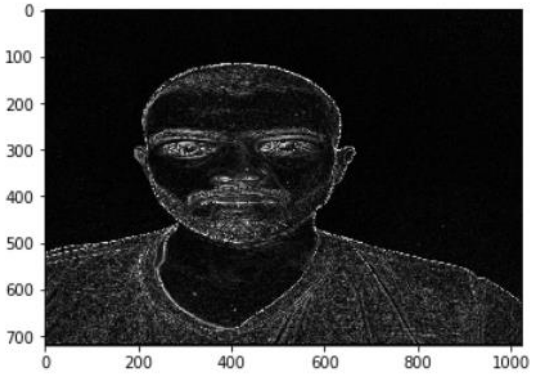


Fig 5.10 : custom sobel on face 2

Fig

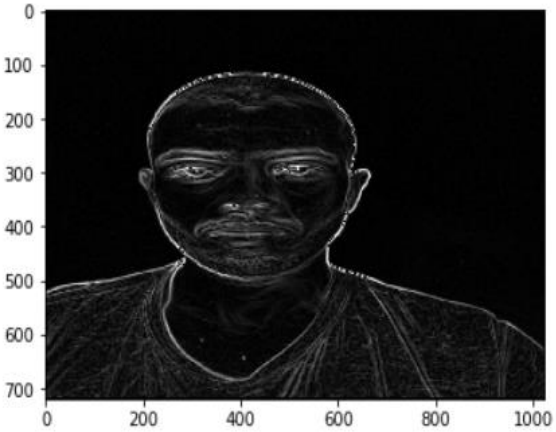


Fig 5.11 : custom sobel on face 2 + gaussian Blur

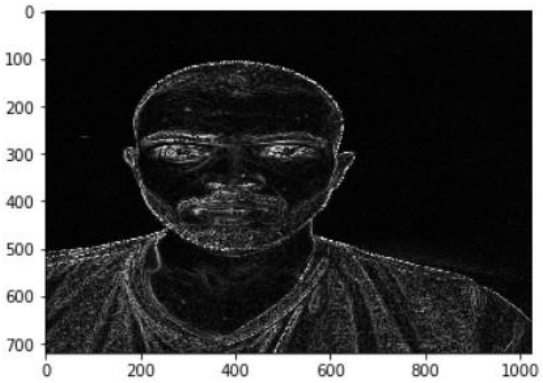


Fig 5.12 : inbuilt sobel on face 1

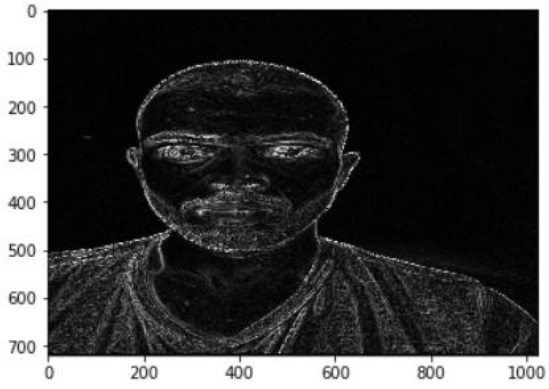


Fig 5.13 : custom sobel on face2

Thus we can see our custom sobel function is at par with python inbuilt sobel function.

A brief insight on why sobel works the way it works:

Consider the x-gradient of sobel $\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$. In a light overview you can see that this kernel is briefly calculating the difference between intensities of 0^{th} and 2^{nd} column pixels. Now when we have smooth region i.e no sudden variations in intensities, the difference between these intensities will be close to 0 , hence these region are marked by black(in grayscale black has intensity of zero), but when there is a sudden fluctuation in intensity of pixel while traversing along the columns the difference of intensity values between adjacent column pixels will be high, thus the result of convolution fn this kernel in that region will be in higher range (close to 255) , hence these areas will be marked by white pixels.

Similarly, we use y-gradient kernel to calculate sudden change of intensity along the rows. In the end we approximate these gradients by root sum squares.

Task 6: Image rotation

In this task our main goal was to achieve image rotations over different angles. While there are many ways to achieve this, the implementation of this task closely follow the mathematics of this article.

I can document the whole understanding of this code but then I'll just be paraphrasing the above article.

The mathematics explained in above article was *neque amplius neque minus* in the understanding of this concept.

Result and analysis:

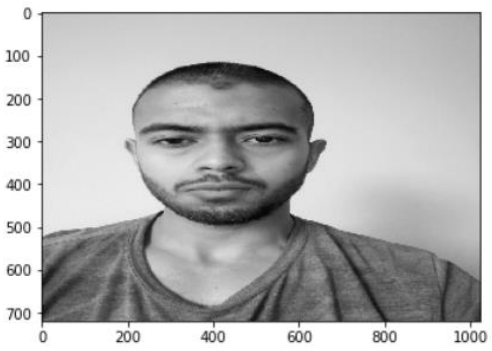


Fig 6.1 Input Image face 1

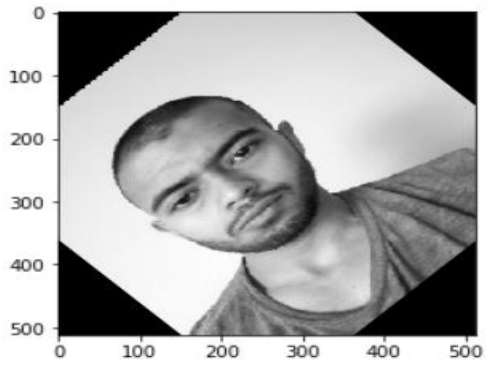


Fig 6.2 rotated at -45 degree

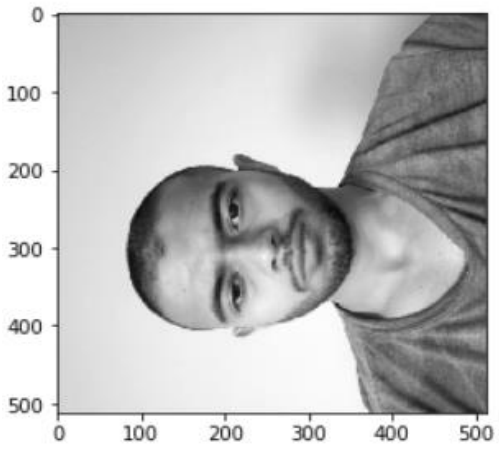


Fig 6.3 Rotated at -90 degree

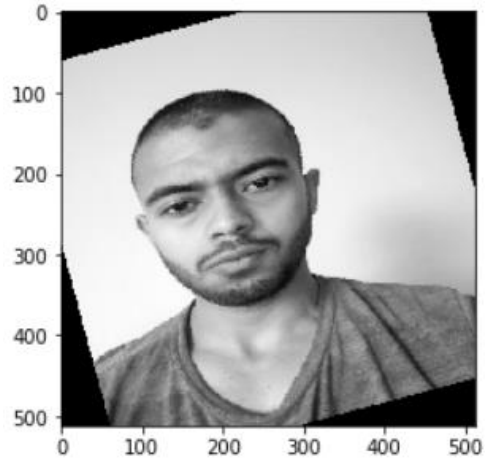


Fig 6.4 Rotated at -15 degree

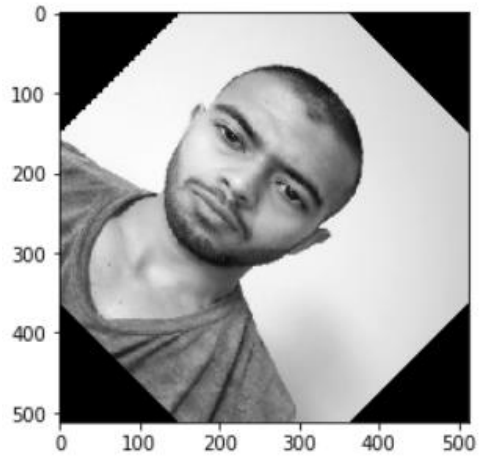


Fig 6.5 Rotated at +45 degree

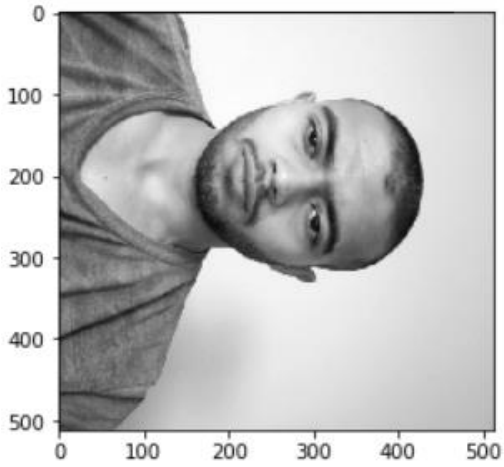


Fig 6.6 Rotated at +90 degree

(algorithm used -> backward mapping)

Forward Mapping v/s Backward Mapping:

In any geometric or affine transformation , new coordinates of a pixel are some trigonometric functions of its initial coordinate in input image. But many times these functions will not return an integer value, as required by a matrix to address a specific cell to be mapped in the output image.

Forward mapping computes these new coordinates by assigning a nearest integer to the value computed by these functions. Though it solves the initial problem but a new

situation arises. The problem is that while assigning nearest integer coordinates , some cells in the output image are addressed several times and some not at all , leading to “holes” in output image where no value was assigned to a pixel/cell.

The **backward mapping** solves this problem by iterating over each pixel of the output image and uses the *inverse* transformation to determine the position in the input image from which a value must be sampled. Thus no cells in the output image is left unassigned.

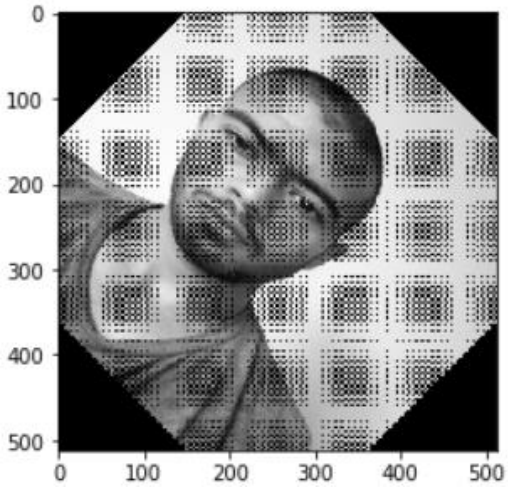
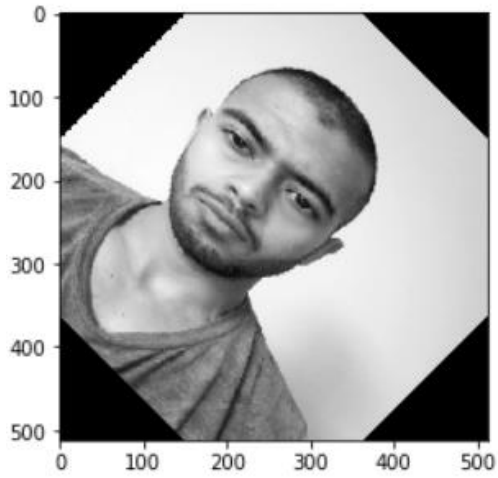


Fig 6.7 rotated face2 by 45 degree (forward mapping)



F ig 6.8 rotated face2 by 45 degree (backward mapping)