# ENGN4528

# COMPUTER VISION

# CLAB-1 REPORT

Samyak Jain

u6734495

21/03/2020

# TABLE OF CONTENTS: Page

# 1. TASK-1 : MATLAB WARM UP:

## 1.1 a = np.array([[2, 4, 5],[5, 2, 200]]):

The above line of code generates a 2x3 matrix , where the first row comprises of [2,4,5] and the second row comprises of [5,2,200]

**Output:** [[2,4,5

5,2,200]]

## 1.2 b=a[0,:] :

The above code provides us with an output array b which contains the first row of the previously generated matrix a along with all the columns .

**Output:** b → [2,4,5]

## 1.3 f = np.random.randn(500,1)

The function np.random.randn creates an array of specified shape and fills it with random values as per standard normal distribution. In our case , f will be an array of shape 500x1 and all the numbers in that array are random values of the standard normal distribution. Shown below are the first 10 numbers of the array.

**Output:** [[-0.68874229] [ 0.90882744] [-1.72363179] [-0.28120206] [ 0.67792359] [-0.81745485] [-0.6959179 ] [ 0.385483 ] [-0.1375332 ] [-0.16182603]]

## 1.4 g=f[f<0] :

The above code will give all the values in the previously generated f array which are less than 0 and the numbers would be stored in the new array g . The numbers from the first 10 samples that remain after performing the code are as follows:

**Output:** [[-0.68874229] [-1.72363179] [-0.28120206] [-0.81745485] [-0.6959179 ] [-0.1375332 ] [-0.16182603]]

## 1.5 x = np.zeros (100) + 0.35:

The function np.zeros() creates an array of the specified shape or size with all elements in the array filled with 0 . In our case , it creates an array of size 100 with all zero values. Then it is adding 0.35 to all the elements in the array and making it an array of size 100 with all elements added to become 0.35.

**Output:** [ 0.35 0.35 0.35 …….. 0.35 0.35]→ 100 elements

## 1.6 y= 0.6 * np.ones([1, len(x)])

The function np.ones() creates an array of the specified shape or size with all elements initiated to 1. Here the function creates an array of shape (1,len(x)) , where len(x) is the length of the matrix x that was previously created. So, our array will be of size 1x100. Then ,the code is multiplying all the elements in the array with 0.6 , making all the elements in the array as 0.6 as all elements were initially 1.

**Output:** [[ 0.6 0.6 0.6 0.6 …….. 0.6 0.6]]

### 1.7 Z= X- Y :

The above code just subtracts the newly generated y matrix from the previously created x matrix and stores the difference in the new matrix z. The subtraction takes place element-wise , which means corresponding elements in the two matrices are subtracted.

**Output:** [[-0.25 -0.25 -0.25 …… -0.25 -0.25] ]

### 1.8 a = np.linspace(1,200) :

The np.linspace() function returns evenly spaced numbers over a specified interval. Meaning,that the difference between any two numbers generated from the function would be same. In our case, evenly spaced numbers between 1 and 200 would be generated with equal difference.

**Output :** [ 1. 5.06122449 9.12244898 13.18367347 17.24489796 21.30612245 25.36734694 . . . . . 175.63265306 179.69387755 183.75510204 187.81632653 191.87755102 195.93877551 200.]

### 1.9 b = a[: :-1] :

The slicing of any array [::-1] indicates that we are reversing the ordering of the elements . This slicing reverses the array and the values inside the array. In our case, this code will reverse the array a in which we previously stored the evenly spaced numbers and the store the values in the new array b.

**Output:** [200. 195.93877551 191.87755102 187.81632653 183.75510204 179.69387755 175.63265306. . . . . 25.36734694 21.30612245 17.24489796 13.18367347 9.12244898 5.06122449 1.]

### 1.10 b[b<=50]=0 :

The code above takes all the values in the array b that are less than or equal to 50 and sets all the corresponding values in the array to 0. So all the values in the array b that are less or equal to 50 are set to 0 .

**Output:** [200. 195.93877551 191.87755102 187.81632653 183.75510204 179.69387755 175.63265306. . . . . 0 0 0 0 0 0 ]

# 2. TASK-2 BASIC CODING PRACTICE

**2.1 Code Snippet:**

```
GRAYimg = Image.open('Lenna.png')
GRAYimg = GRAYimg.convert('L')

GRAYimg_inv = ImageOps.invert(GRAYimg)
```

In the above code, python's inbuilt function such as Pillow library were used to invert the picture. The .invert() functions takes in the image and outputs the negative image as shown is **fig.1(b).**

**Fig.1(a) Original Image**

**Fig.1(b) Negative Image**

**Fig.1 (b)** shows the negative image in which all the dark values of the image appear less darker and more brighter and vice-versa.

**2.2 Code Snippet :**

```
row,col= np.array(GRAYimg).shape
GRAYimg=np.array(GRAYimg)
GRAYimg_flip = GRAYimg[row-1: :-1]
```

The above code takes row,col as the number of columns and rows in the Image and then transforms the array such that all the rows and columns are reversed, which means that the first row and columns becomes the last and so on. This will lead to the image pixels getting reversed and as the result ,the image is being flipped as shown in **Fig.2(b).**

**Fig.2(a) Original Image**

**Fig.2(b) Flipped Image**

**2.3 Code Snippet:**

```
RGBimageo = Image.open('Lenna.png')
RGBimage=np.array(RGBimageo)
RGBimage_BGR=RGBimage[:,:,::-1]
```

In the above code RGBimageo is the original RGB image ,which is then convert into an array by using the numpy.array function. RGBimage_BGR is the new array in which the last and first column are interchanged which is shown by the slicing of the image in the 3rd line of the snippet , as

we have seen in Task-1 that [:,:,::-1] reverses the order of the columns and thereby swapping the R and B channels of the image as shown in **Fig.3(b)**

**Fig.3(a) Original RGB image**

**Fig.3(b) R and B swap**

**2.4 Code Snippet:**

```
f_avg =np.average([RGBimage,RGBinv],axis=0)
f_avg = f_avg.astype(int)
axes[2].imshow(f_avg.astype(np.uint8))
```

In the above task ,we are asked to average the original image and the flipped image and show the averaged image. In the above code, the **RGBimage** is the original picture and **RGBinv** the flipped image of the original RGB image**.** Using numpy's average function we passed the images into the function and got the average pixels of the two images as shown in **Fig.4(c)**. When averaging we notice that we might get float values , so the 2$^{nd}$ line converts all the elements of array into integer. When plotting the image we have taken into account the typecasting and displayed the image as type **uint8** so that the pixel values remain between 0-255.

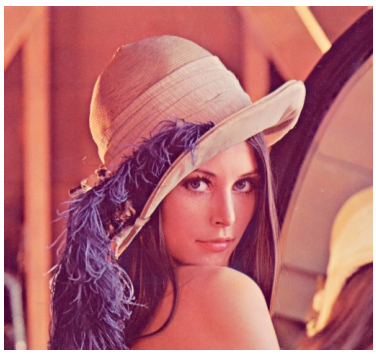

**Fig.4(a) Original Image**

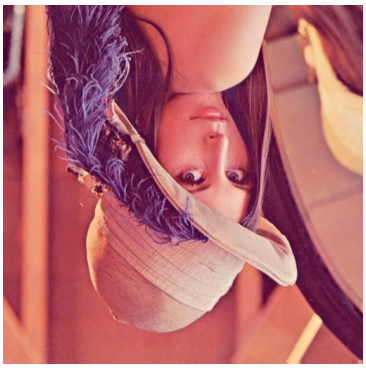

**Fig.4(b) Flipped Image**

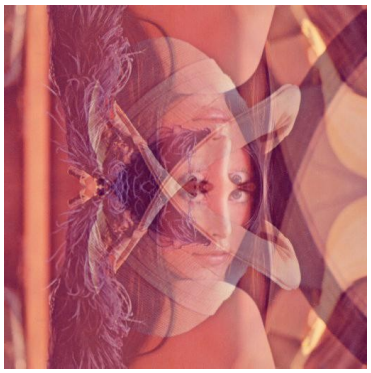

**Fig.4(c) Averaged Image**

**2.5 Code Snippet:**

```
GRAYimg_arr=np.array(GRAYimg)
r_noise=np.zeros((GRAYimg_arr.shape))
for i in range(GRAYimg_arr.shape[0]):
    for j in range(GRAYimg_arr.shape[1]):
        r_noise[i][j]=GRAYimg_arr[i][j]+np.random.randint(0,255)
output=np.clip(r_noise,0,255)
```

In the above code , firstly we are loading the image as an array. To add a random int value to all the pixels in the image we have to traverse through the array. So we used two for loops ,one for traversing along the row and one traversing along the columns. Next we made an empty array where the result of the sum of noise generated and input pixels could be stored. Now that we have accessed all the pixels we are adding a random int value to the existing pixel by using numpy's inbuilt random() functions and storing it into r_noise. Randint() generates a random integer value . Adding random values will make the image very noisy as random integer are being added to pixels which leads to image being as shown in **Fig5.(b).**

**Fig.5(a) Original Image**

**Fig.5(b) Random Noise Image**

# 3. TASK-3 BASIC IMAGE I/O

**3.1 The Frontal Face pictures with different lighting:**

**Fig.6(a) Pic 1**

**Fig.6(b) Pic 2**

**Fig.6(c) Pic 3**

### 3.2 Equalisation Task:

**(a)** To re- scale the image Python Image Library's "resize" function was used which scaled the picture down to 768 columns x 512 rows.

**(b)** For this task **Fig.6(c)** was used. To convert the RGB image into three separate greyscale channels, all the channels of the image were stored in different arrays which represented their pixel values in a greyscale image. For instance, to access the R-component of the image , all the pixels in the first column of its 3rd dimension was stored in an array and then displayed. Respectively for the other two separate greyscale channels. All the different channel outputs are displayed below.

Code Snippet:

```
rc=rchannel[:,:,0] ##Gives the R-component
rc=rchannel[:,:,1] ##Gives the G-component
rc=rchannel[:,:,2] ##Gives the B-component
```

where rchannel is the array form of the Original image.

**Fig.7(a) R-channel**

**Fig.7(b) G-Channel**

**Fig.7(c) B-Channel**

### 3.3 Histogram:

To create histograms of all the separate greyscale channels we have used pyhton's matplotlib() package . As the hist() function accepts 1-D array, we used numpy's flatten() function to convert the 2-D arrays into 1-D arrays. As the images has pixels ranging from 0 to 255 , we have provided the hist function with 256 bins as the frequency of each pixel is visible. X-axis shows the pixel and the Y-axis shows the frequency of the pixel in the image.

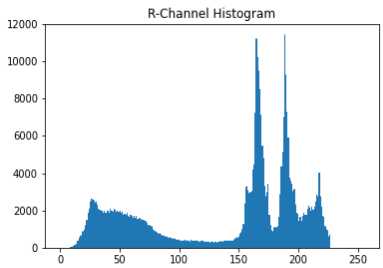


**Fig.8(a)R-Channel Histogram**

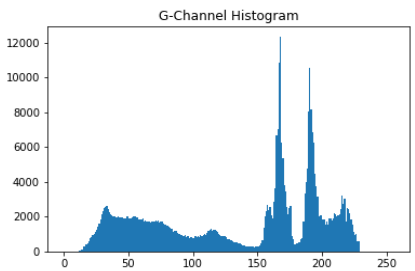


**Fig.8(b)G-Channel Histogram**

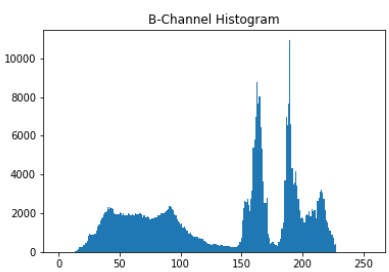


**Fig.8(c)B-Channel Histogram**

As we can see from the histogram plot of the all the channels that pixels between 100-150 are very less in numbers as compared to all other pixel values. Pixels values between 150-250 are densely populated whereas lighter pixels are also present in small numbers in the grey-scale images.

### 3.4 Histogram Equilization:

Python's ImageOps' Equilize() function was used to apply histogram equilization to the resized image and its grey-scale images.

**Fig.9(a) Image Equilization**

**Fig.9(b) R-Channel Equilization**

**Fig.9(c) G-Channel Equilization**

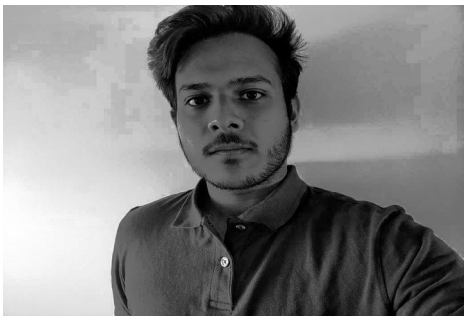

**Fig.9(d) B-Channel Equilization**

After Performing hist Equilization we can see that now that the pixel frequency are normalised and there is not a visible range where pixel frequency is close to zero , as it was the case earlier. We can see that in **Fig.8(a)** there are no pixels between the range 100-150 that is around 2000 or above but in **Fig10.(a)** after equilization we can observe that frequency of pixels between 100-150 are higher than before. And, same observations can be made for the G and B channels.

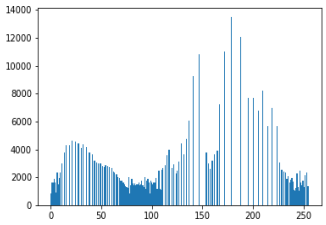


**Fig.10(a) R-Equilization Hist**

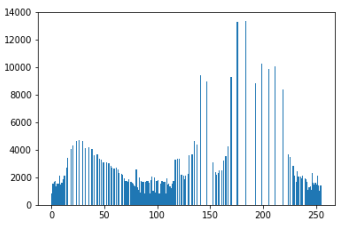


**Fig.10(a) G-Equilization Hist**

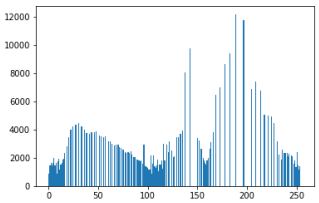


**Fig.10(a) B-Equilization Hist**

# 4. TASK-4 IMAGE DENOISING VIA A GAUSSIAN FILTER:

### 4.1 Image Cropping:

Cropping of the facial part was done by trial and error method and not using any inbuilt functions . The image was cropped out by using image slicing as shown in the code snippet to get the facial part of the picture.

**Code Snippet:**

```
crop=Q4image_arr[0:2000,1200:3200,:]
Q4_crop=Image.fromarray(crop)
Q4_crop=Q4_crop.resize((256,256),Image.ANTIALIAS)
Q4_crop.save('Q4_crop.jpg')
```

The first line takes the pixels in the range specified and hence crops the picture , as shown in Fig.11(b). The image is then resized by using the .resize() to the desired ratio and then saved as a grey-scale.

**Fig.11(a) Original image**

**Fig.11(a) Grey-Scale image**

### 4.2 Adding Gaussian Noise :

We have used Python's Open-CV package to add Gaussian noise to our image. Creating a new array with all values initiated to 0 where the noise that is generated can be stored,by using **np.random.normal()** function to generate noise with given mean and standard deviation. After the noise is generated it is simply added to the image by matrix addition. This new sum is then stored in an array and displayed as shown in **Fig.12(b).** The code written to add the noise is shown below:

**Code Snippet:**

```
def add_noise(img):
noise = np.zeros(img.shape,
dtype=np.uint8)

noise=np.random.normal(mean,std,
Q42image.shape)
                                    new_img
= img + noise
 return new_img
```

**Fig.12(a) Original Image**

**Fig.12(b) Noise Image**

### 4.3 Histogram :

Again with the help of **matplotlib** we create histograms for the original image and the image to which we added noise. Histogram for **Fig.12(a)** and **Fig.12(b)** are shown below:

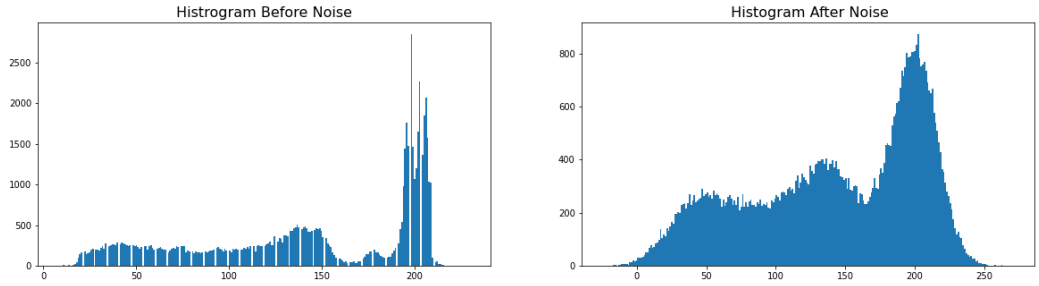


**Fig 12. Histograms Before and After Adding Noise**

### 4.4 Gaussian Filter:

Gaussian Blur is a type of image-filtering technique that uses Gaussian functions to reduce noise and calculate transformation to apply to each pixel in the image. The formula to produce a Gaussian function is G(x) = $(1/\sqrt{2\pi\sigma^2})e^{-\frac{(x)^2}{2\sigma^2}}$ . But since we are dealing with 2-D images , the product of two 1-D Gaussian would lead to producing a Gaussian kernel for 2-D array. Therefore, the formula becomes: $(1/2\pi\sigma^2)e^{-\frac{(x)^2+(y)^2}{2\sigma^2}}$

We first create a Gaussian filter using the above formula . Here is a Code Snippet :

```
def my_Gauss_filter(k_size,sigma):
    for x in range(-m, m+1):
        for y in range(-n, n+1):
        x1 = 2*np.pi*(sigma**2)
        x2 = np.exp(-(x**2 + y**2)/(2* sigma**2))
```

Now simply using the terms x1 and x2 we place the product in a new array and then we normalize the array such that the sum of the elements in the array add up to 1. Our 5*5 kernel produces the following output:

**[[0.02324684 0.03382395 0.03832756 0.03382395 0.02324684]**
**[0.03382395 0.04921356 0.05576627 0.04921356 0.03382395]**
**[0.03832756 0.05576627 0.06319147 0.05576627 0.03832756]**
**[0.03382395 0.04921356 0.05576627 0.04921356 0.03382395]**
**[0.02324684 0.03382395 0.03832756 0.03382395 0.02324684]]**

where we can see that the centre of the kernel has the largest value and thus the kernel would multiply the central pixels more and pay less attention to the corners of the image resulting in the central portion of the image to be blurred. Now we take our Gaussian kernel and apply it to our noise image. When applying the first thing we need to take care off is how do we deal with the corners. As,When multiplying only the centre pixel get changed as all the other pixel are summed to get the centre pixel, there might be areas where there are no corners as the centre pixel itself is a

corner. For those situation we have decided to ignore the corners on both the sides and just sum the surrounding pixels to get the centre. This is shown in the code snippet below:

```
def apply_gaussian(img):
    for i in range(height-4):
        for j in range(width-4):
            img_out[i][j]=np.sum(img[i:i+5,j:j+5]*gaussian_filter)
```

where Gaussian_filter is our 5*5 180 degree rotated (convoluted) Gaussian Kernel. Since we took range as height-4 and width -4 we have ignored the corners and thus adding only surrounding pixels. The output of our applied kernel with different sigmas are shown in **Fig.13:**

**Fig.13(a) Noise Image**

**Fig.13(b) Filter with sigma=1**

**Fig.13(c) Filter with sigma=2**

**Fig.13(d) Filter with sigma=5**

### 4.5 Comparing with inbuilt Gaussian Blur:

**Code Snippet:**

```
#Gaussian Blur with sigma=1
blur = cv2.GaussianBlur(np.array(gblur),
(5,5),1)
```

We can see that there is not much difference in **Fig.14(a)** and **Fig.14(b)** which emphasises the fact that our Gaussian filter is working correctly.

**Fig.14(a) Inbuilt Gaussian**

**Fig.14(b) My_Gaussian**

# 5. TASK-5 → 3x3 SOBEL FILTER:

The sobel filter uses two 3 x 3 kernels. One for changes in the horizontal direction, and one for changes in the vertical direction. The two kernels are convolved with the original image to calculate the approximations of the derivatives. If we define Gx and Gy as two images that contain the horizontal and vertical derivative approximations respectively, the computations are:

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -2 \end{pmatrix} *A \qquad \text{and} \qquad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} *A$$

Where A is the original source image.

To compute Gx and Gy we move the appropriate kernel (window) over the input image, computing the value for one pixel and then shifting one pixel to the right. Once the end of the row is reached, we move down to the beginning of the next row. In our kernel we have ignored the corners ,when the pixels present in the corners of the original image have no pixels in the window.

| | | | | |
|---|---|---|---|---|
| a11 | a12 | a13 | ... | … |
| a21 | a22 | a23 | ... | ... |
| a31 | a32 | a33 | ... | ... |
| ... | ... | ... | ... | ... |
| .. | ... | ... | ... | ... |

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

Kernel

| | | | | |
|---|---|---|---|---|
| b11 | b12 | b13 | ... | ... |
| b21 | b22 | b23 | ... | .... |
| b31 | b32 | b33 | ... | ... |
| …. | …. | ... | ... | ... |
| …. | ... | ... | ... | ... |

**output($G_x$)**

**where b22=** a11 – a13 + 2a21-2a23 + a31- a33

In our code we manually create two arrays F1 and F2 which contains the x-direction kernel and y-direction kernel respectively.

**Code Snippet:**

```
gx=np.sum(img[i:i+3,j:j+3]*np.flip(np.flip(F1, 1), 0) ) # 180 degree rotated
gy=np.sum(img[i:i+3,j:j+3]*np.flip(np.flip(F2, 1), 0) ) # 180 degree rotated
sobel_outcome[i][j] = np.sqrt(gx**2 + gy**2)
```

We ignore the corners and start summing over the multiplicands and store the sum in Gx and Gy which gives us the edge detection in vertical and horizontal edges respectively. From the code we can see that we have convolved the kernel onto the image by rotating F1 And F2 by 180 degrees.

At each pixel in the image, the gradient approximations given by Gx and Gy are combined to give the gradient magnitude, using:

**G=** $\sqrt{(Gx)^2+(Gy)^2}$

Next we store this **G** in sobel_outcome which contains the our sobel filtered image. Since high frequency noise creates false edges in the filtered image, it would be better if we apply sobel filter after we had reduced the noise by applying Gaussian filter. Since Gaussian filter reduces the high frequency noise, our sobel filter will perform better than just applying sobel filter to our noisy image. This difference is visible in the comparison of these two images in **Fig.16(b)** and **Fig.16(c)**.

**Results from our Sobel Filter:**

**Fig.15(a)Noisy Image**

**Fig.15(b) Gx**

**Fig.15(c) Gy**

We can see that **Fig.15(b)** and **Fig.15(c)** shows the edges in x-direction and y-direction respectively. Below is the output of the sobel filter when we combine our Gx and Gy to make it **G** as mentioned earlier.

**Fig16.(a) Noisy Image**

**Fig.16(b)Without Gaussian Blur**

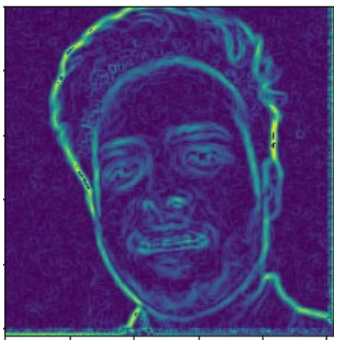

**Fig.16(c) With Gaussian Blur**

As mentioned above our sobel filter performs better when we have applied it after Gaussian blur filter.

**5.1 Testing with Inbuilt Sobel filter:**

**Code Snippet:**

```
sobelx = cv2.Sobel(np.array(blur),cv2.CV_64F,1,0,ksize=3)
sobely = cv2.Sobel(np.array(blur),cv2.CV_64F,0,1,ksize=3)
G=np.sqrt(sobelx**2 + sobely**2)
```

From the images below we can see that our sobel filter with Gaussian blur is almost giving us the same result as Open-CV's inbuilt sobel filter as represented by **Fig.17(b)**

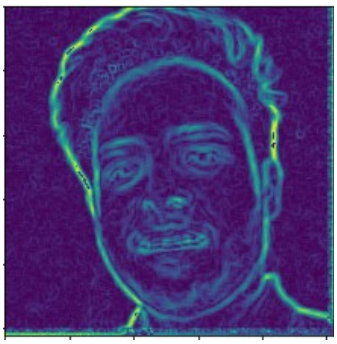

**Fig.17(a) My Sobel Filter**

**Fig.17(b) Inbuilt Sobel Filter**

**5.2 Testing on Other face Images:**

Here we have applied our sobel filter on the cropped and resized image of the remaining face images.

# 6. TASK-6 → IMAGE ROTATION:

For the my_rotation() function we have used the properties of affine transformations and rotation of a 2-D matrix around a point. We know that, if a point is located at (x,y), and we wanted to rotate that point around the origin, the coordinates of the new point would be located at (x',y'). Where

x' and y' are given by :

$$\mathbf{R}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

But , in our rotation function we have rotated about the point (x,y) where x and y are mid points of the image and in this instance we have used **backward mapping** . For instance for a 512*512 image, we rotated the image about the point (255,255). For this we just have to increase the calculation further so that it doesn't get rotated about origin. So, the formula used was :

int xpos = (int)(xOff*cosθ - yOff*sinθ + x0);

int ypos = (int)(yOff*cosθ + xOff*sinθ + y0);

where xoff is the (input image pixel – x0) and yoff is (input image pixel -y0) are the coordinates of the input point and (x0,y0) is the point we are rotating about.

**Code Snippet:**

```
cos=np.cos(math.radians(angle))
   sin=np.sin(math.radians(angle))
   for i in range(img_arr.shape[0]):
      for j in range(img_arr.shape[1]):
         xout=int((i-x)*cos - (j-y)*sin + x)
         yout=int((j-y)*cos + (i-x)*sin + y)
     if xout < 512 and  xout> 0  and yout < 512 and yout >0:
            B[i,j]=img_arr[xout,yout]
 return B
```

The code can rotate any image to (-90,90) degrees about the centre point , where xout,yout are the coordinates of the output point i.e the rotated image. The results from the code are shown below:

**Results from Rotation Function:**

**Fig.18(a) Original Image**

**Fig.18(b) -15° Image**

**Fig.18(c) -45° Image**

**Fig.18(d) -90° Image**

**Fig.18(e) 45° Image**

**Fig.18(f) 90° Image**

## 6.2 FORWARD AND BACKWARD MAPPING:

The **forward mapping** iterates over each pixel of the input image, computes new coordinates for it, and copies its value to the new location. To convert our code into forward mapping , only the coordinates of the input and output image needs to be interchanged. When mapping the coordinate might not be an integer , so why we assigned the nearest integer to the result which could result in many pixels may be addressed several times or some might not all, leaving holes in the image with no value assigned to the pixel.

The **backward mapping** is just when we set our output coordinates as the coordinates of the input and the input coordinates to our output array. The differences are shown below by the two images.

**Fig.19(a)Backward Mapping**

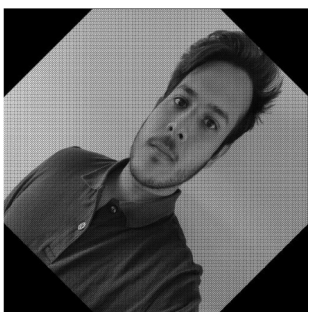

**Fig.19(b) Forward Mapping**

### 6.3 INTERPOLATION METHODS :

- **Nearest Neighbour:** The most basic method of interpolating. Requires the least processing time. Considers only one pixel.
- **Bilinear :** This method considers the closest 2x2 neighbourhood of known values surrounding the unknown pixels. Takes a weighted average of these 4 pixels to arrive at the final interpolated values. Results in better looking images than nearest neighbourhood and less processing time too.

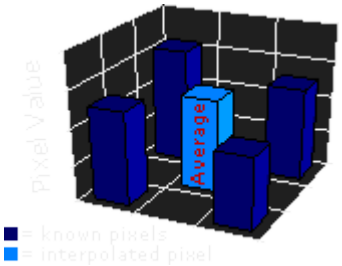


**Fig.19(a) Bilinear**

- **Cubic:** This method uses the closest 4x4 neighbourhood pixels surrounding the unknown pixel. Nearest neighbours are given more weightage when averaging. Results in Sharper Image than both.

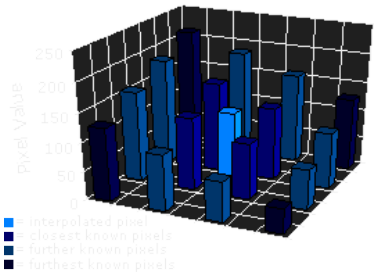


**Fig.19(b) Cubic**

**Outputs with different Interpolation methods:**

**Nearest:**
Output array for the image (considers only the nearest interpolated point):

```
[[[167 171 ... 170]
  [181 187 ... 185]
  [180 182 ... 179]
  [176 180 ... 179]]]
```

**Linear :**
Output array for the image (considers the nearest 2x2 neighbourhood pixels):

```
[[[173 177 ... 176]
  [176 178 ... 177]
  [178 182 ... 181]
  [180 182 ... 181]]]
```

**Cubic:**
Output array for the image (considers the nearest 4x4 neighbourhood pixels):

```
[[[173 177 ... 176]
  [176 179 ... 177]
  [178 181 ... 181]
  [180 182 ... 181]]]
```

From the above output arrays we can observe that the pixel values are changing according to the interpolation method defined.

# REFERENCES:

https://www.sciencedirect.com/topics/computer-science/image-rotation

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic2.htm

https://www.youtube.com/watch?v=uihBwtPIBxM

https://www.sciencedirect.com/topics/engineering/gaussian-blur

https://cs.appstate.edu/~rt/ImgProcessing/labs/lab4/lab4.html

**Fig.19(a) and Fig.19(b)** https://www.cambridgeincolour.com/tutorials/image-interpolation.htm