COMPUTER VISION CLAB1 REPORT

ENGN6528

01/04/2020

u7043565
Tanya Dixit

CONTENTS

TASK – 1: PYTHON WARM-UP

1.1 a = np.array([[2, 4, 5], [5,2,200]])
We are declaring a matrix here. The input is two smaller arrays inside another array which tells us that it's a matrix. Each row is present as the smaller array which has 3 elements, and there are 2 such smaller arrays, that's why we get 2 rows and 3 columns - A 2x3 matrix.

Output:

```
array([[  2,   4,   5],
       [  5,   2, 200]])
```

1.2 b = a[0,:]
We are slicing the matrix a here. So here first we get the row number which is 0, i.e. we want the first row. And then ':' means we want all the columns.

Output:

```
array([2, 4, 5])
```

1.3 f = np.random.randn(500, 1)
This assigns a 500x1 matrix to f which is filled with random double values. These random values are normally distributed and are random.

1.4 g = f[f < 0]
Returns all the elements which satisfy the condition f<0 and assigns them to g. g is an array of shape (267,) but can really be any shape and value because our f is random.

1.5 x = np.zeros ((1,100)) + 0.35
np.zeros((1,100)) returns a 1x100 matrix of zeros. Adding 0.35 adds this scalar value to each entry of the matrix which is expected because it's actually the matrix plus scalar multiplied by an identity matrix of the same size as the (1x100) or (m,n) matrix.

Output:
It is a 1x100 matrix of all values as 0.35

1.6 y = 0.6 * np.ones([1,len(x.transpose()])
len(x.transpose()) returns the length of the column vector x (since we transposed it)
np.ones(m,n) returns a matrix of size **mxn** with all entries 1's. Then each entry is multiplied with 0.6 (or whatever scalar is written here)

Output:
It is a 1x100 matrix (or row vector of length 100) whose value is 0.6

1.7 z = x - y
Subtracts the two matrices and stores result in z

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

Returns a numpy ndarray of evenly spaced integers between 1 and 200. It returns 200 values as specified by the third argument.

1.9 b = a[: :-1]
b is assigned the reverse of array a

1.10 b[b <=50]=0
The indexes at which value of b is less than 50 are assigned the value 0.

Output:
All values from 0 to 50 index are zero, rest remain unchanged

TASK 2 – BASIC CODING PRACTICE

2.1 Mapping a grayscale image to its Negative Image

Documentation

To load a gray image in Python, we use cv2.imread with first argument as image path and second argument as 0 (1 for color and -1 for unchanged).
To get the negative image, we subtract each pixel of the image from 255. This operation gives us the Negative Image as *dark pixels become light and the light pixels become dark*.

Refer to Task2.py for the code.
Note: Please make sure the code and face images are in the same folder before running the Tasks.

Observations and Results

In the Fig 2.1, we display the original (left) and the inverted image (right) side by side for comparison.

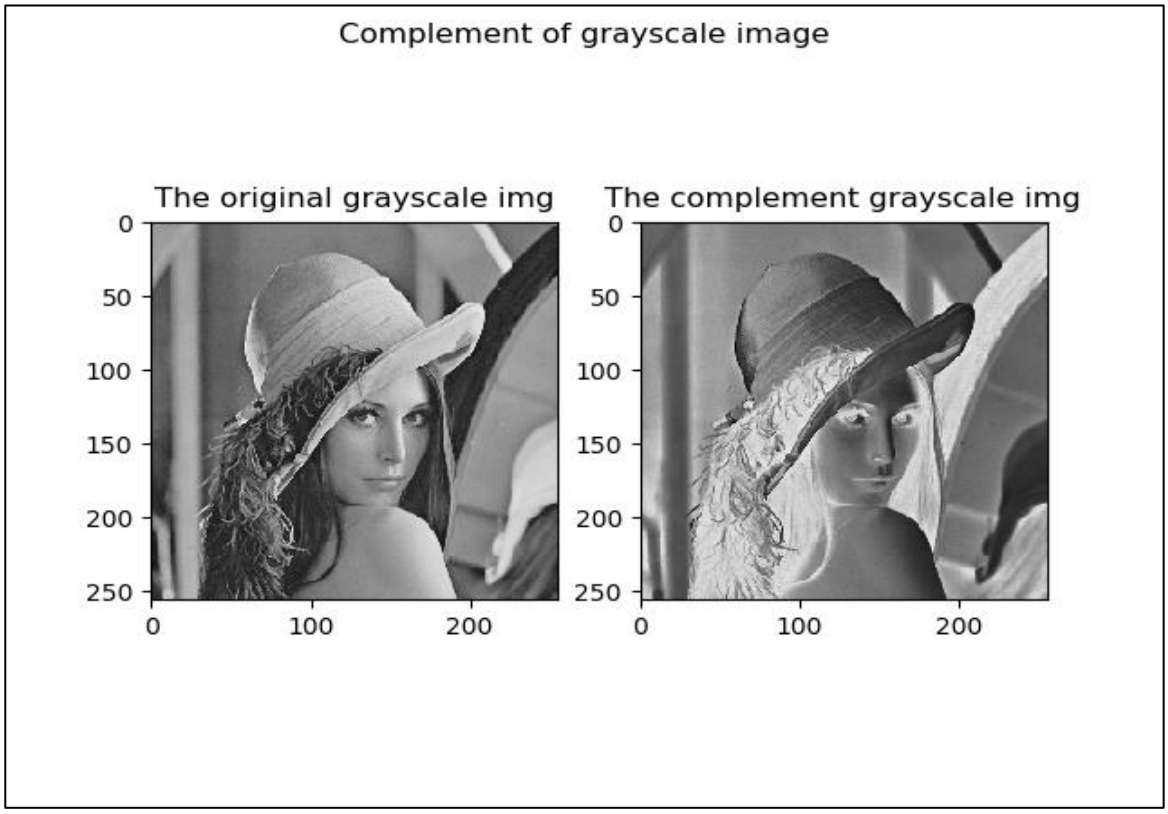


FIG 2.1 COMPLEMENT OF GRAYSCALE IMAGE

Analysis

It's visible from Fig 2.1 that the dark pixels – value near to 0 have turned very light, i.e. their values have come close to 255. And the opposite has happened for the lighter pixels which is what we expect. Subtracting pixel intensities (range 0 to 255) gives this desired effect giving output in range (255-0), just sort of complementing our image.

2.2 Flip the grayscale image vertically

Documentation

For this, we first create a new matrix of zeros having size same as the original gray image. Then keeping the x direction values same, we assign the reverse of the original y values to the new matrix. This ensures that the resulting image is flipped along the y-axis i.e. vertically. Please refer to Task2.py for the code.

Observations and Results

In the Fig 2.2, we display the original (left) and the flipped image (right) side by side for comparison.

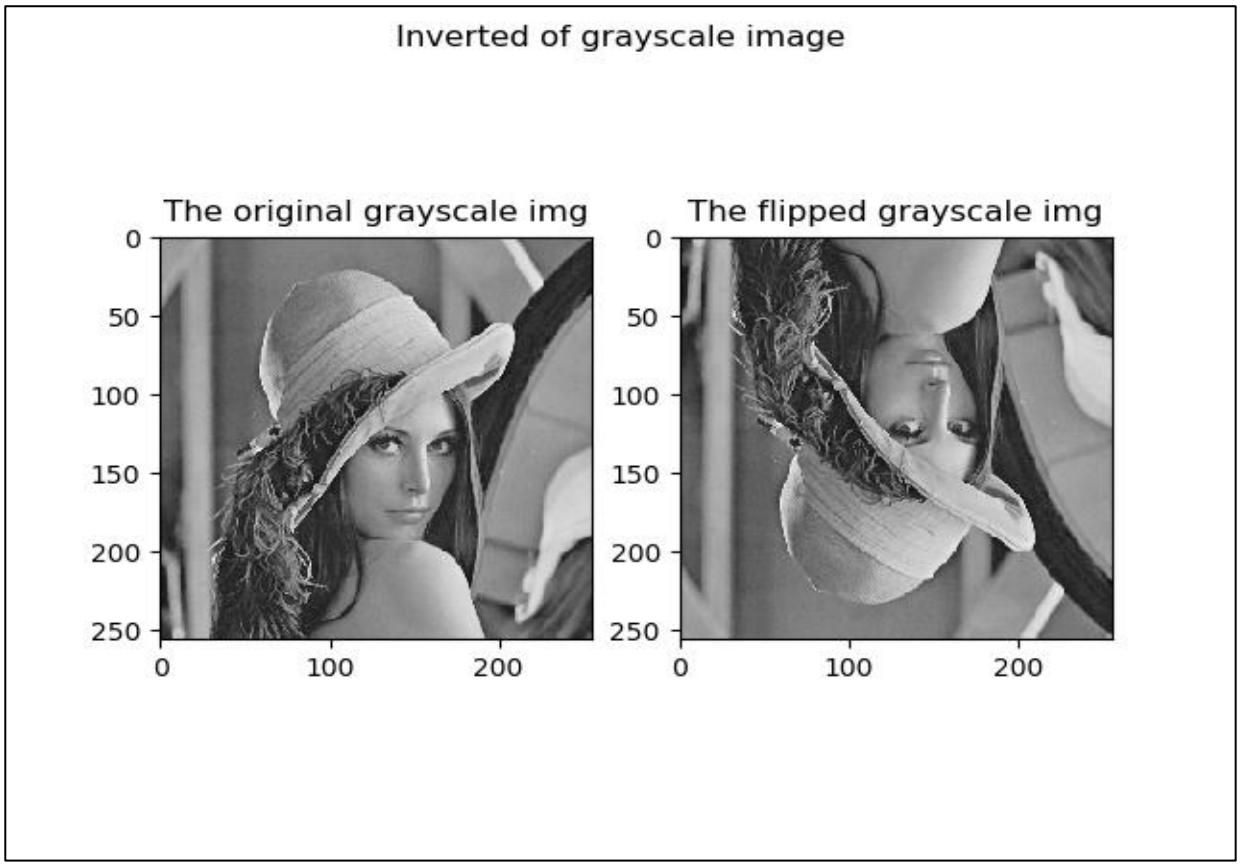


FIG 2.2 - GRAYSCALE IMAGE AND IT'S VERTICALLY FLIPPED VERSION

Analysis
As per the result in Fig 2.2, we can see that our approach works, and the image is a mirror of the original image along the y-axis. Reversing the y coordinate values while keeping x coordinates same gives us a flipped version of the grayscale image.

2.3 Swap Red and Blue Channels of a color image

Documentation
To load a color image in python, we use cv2.imread with image path as first argument and 1 as second argument (since color image).

Note: The function cv2.imread() of cv2 actually treats a color image as BGR (i.e. the first channel is Blue, second is Green and so on), whereas other functions like plt.imshow() treats an image as RGB. So, to avoid any issues, we convert our read image to RGB using the function cv2. cvtColor() taking first argument as the image and second as the conversion criteria - cv2.COLOR_BGR2RGB.

To swap the red and blue channels of the image, we create a copy of each of these in new variables (because if we directly swap, it won't work as x[:,:,1] is a view or reference, not a copy). Then we create a new matrix as a copy of the original color image and assign the red channel of original image to the blue channel of new image, and the blue channel of original image to the red channel of new image.

Observations and Results

In the Fig 2.3, we display the original (left) and the swapped channel image (right) side by side for comparison.

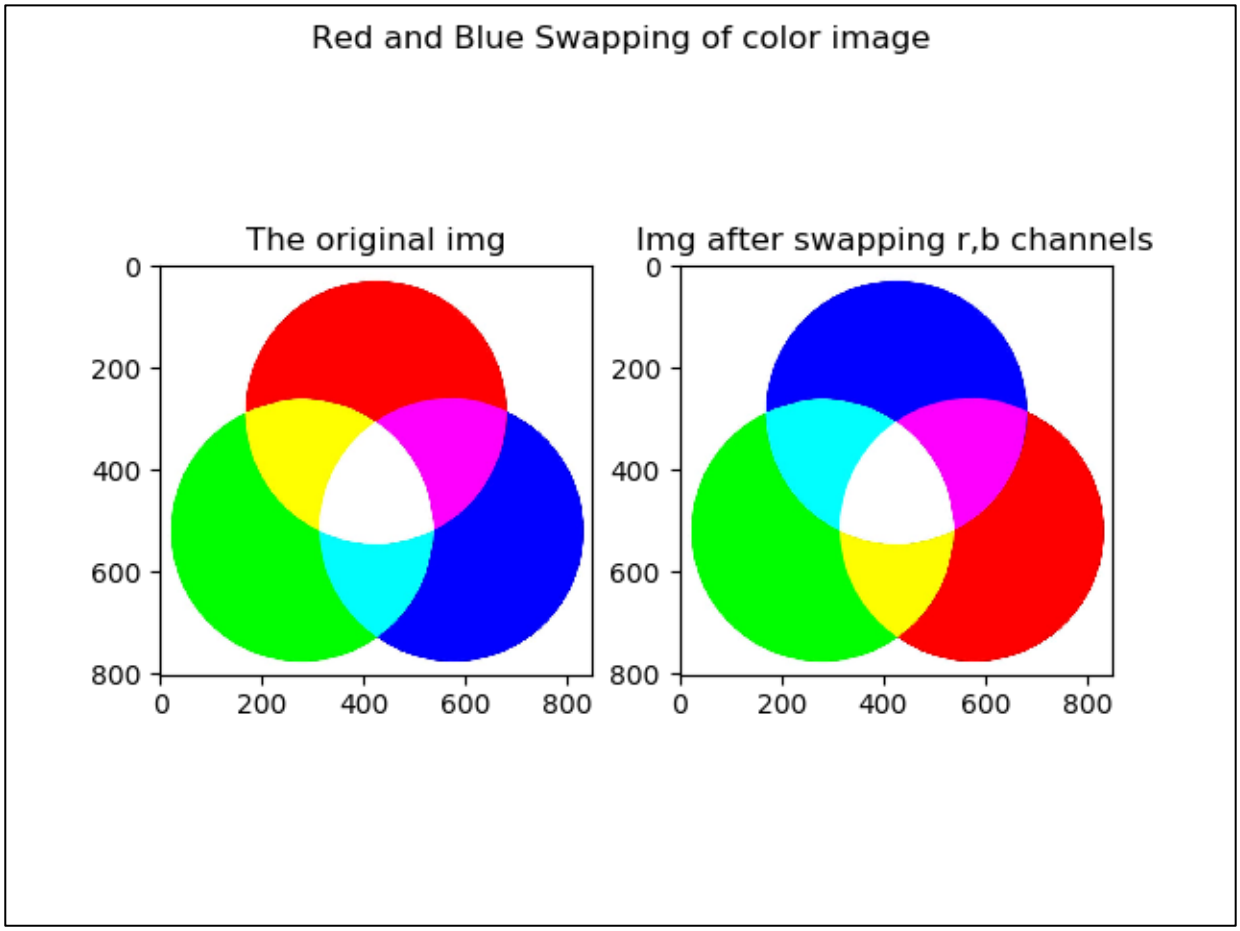


FIG 2.3 - COLOR IMAGE AND RESULT WHEN R AND B CHANNEL SWAPPED

Analysis

From Fig 2.3, it can be observed that the channels have been swapped. Where there was red color, there is now blue and vice versa. It can also be seen that the yellow has been swapped with light blue and vice versa. The reason being yellow is a mixture of red and green, and light blue is a mixture of blue and green. Before swapping the channels, the pixels showing yellow had ones in red and green channels, but after swapping, the red channel became filled with zero and blue channel values in those pixels became 1 – just like light blue, and that's why we observe this.

Note: It's handy to note that a color image has (height)x(width)x3 size. Since we have converted to RGB, our first channel i.e., Image[:,:,0] is red and so on.

2.4 Average the grayscale input with vertically flipped image

Documentation

We already have the grayscale image and it's flipped version. To take the average, we can directly assign a new variable to the sum of the two matrices and divide by 2. Then we typecast it to uint8 so that our image is unsigned int. Python allows us to do this averaging of images directly as if we are operating on scalars. Please refer to Task2.py for the python code.

Observations and Results

In the Fig 2.4, we display the original (left), flipped (middle) and averaged images (right) side by side for comparison.

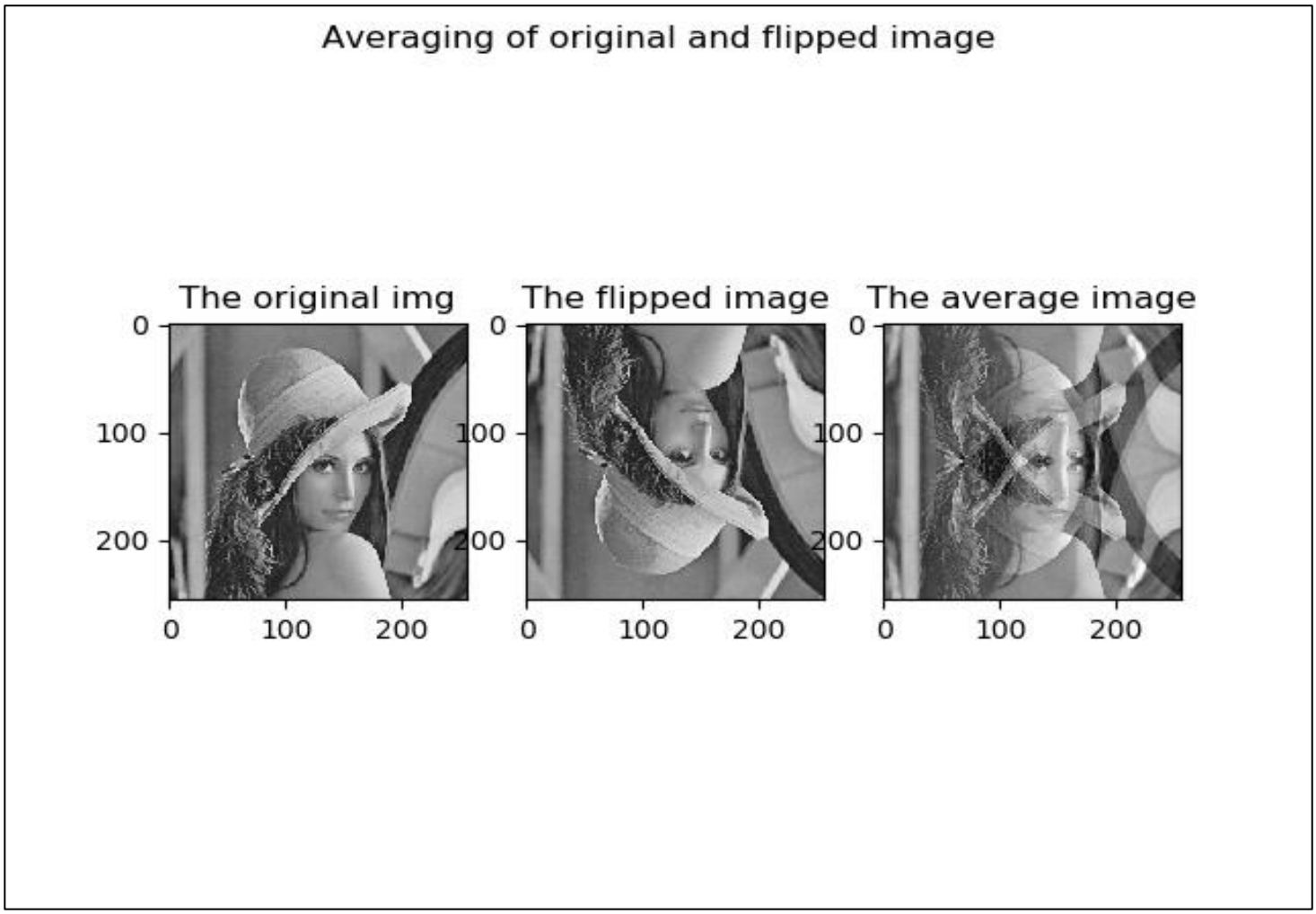


FIG 2.4 - AVERAGE OF GRAY IMG AND FLIPPED IMG

Analysis

From Fig 2.4, we can roughly see both images – the original one and the flipped one having effect on the final image. The reason is that we are taking an average and the pixel intensities get averaged at each position (x,y). Also, since the flipped version is a mirror image of the original one, this final image as well is vertically symmetrical.

2.5 Add Random Value, Then Clip

Documentation

We get a random value from np.random.randint(0,255), and then clip it between 0 and 255 using basic matrix operations. Please refer to code in Task2.py

Observations and Results

In the Fig 2.5, we display the original (left) and the modified image (right) side by side for comparison.

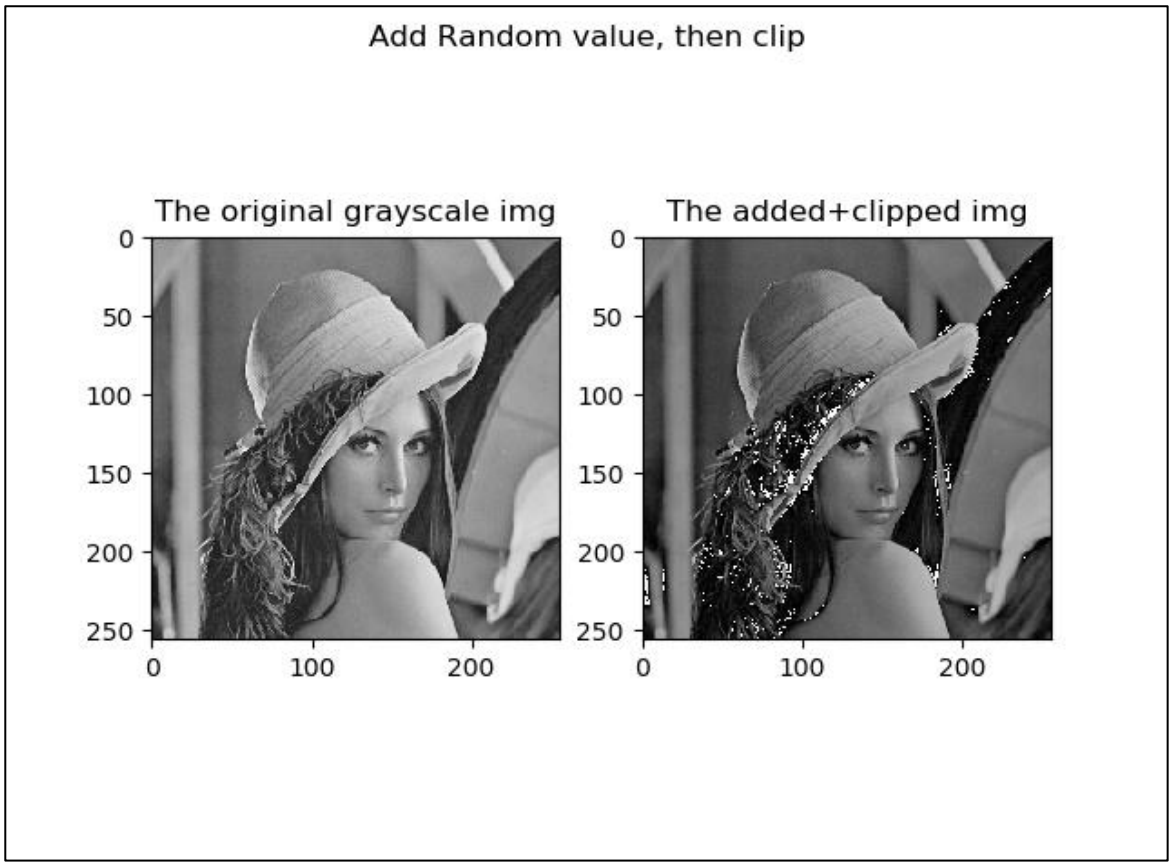


FIG 2.5 - CLIPPED IMAGE AFTER ADDING RANDOM VALUE BETWEEN 0 AND 255

Analysis
From Fig 2.5, we see a lot of the pixels being white and a lot of them being black (255) in the final image. This just tells us that our code works, and the dark pixels get cut-off at 255 whereas the bright ones at 0.

TASK 3 – BASIC IMAGE I/O

The rescaled images are in the accompanying files in the zip.

3. Following operations on one face image

3.a. Resize

Documentation

As before, use cv2.imread() and cv2.cvtcolor() for reading and getting RGB images in correct channel format. To resize the image, we use cv2.resize() which takes in arguments as the image, the size to be resized to and the interpolation method.

Observation and Results

The image is of the desired size i.e. 768x512

3.b. Display RGB channel images

Documentation
Get each of the channel images using matrix slicing (Task 2.3) and display using matplotlib.

Observations and Results

In the Fig 3.b, we display the original (upper-left), the red channel grayscale (upper-right), the green channel grayscale (lower-left) and the blue channel grayscale images (lower-right).

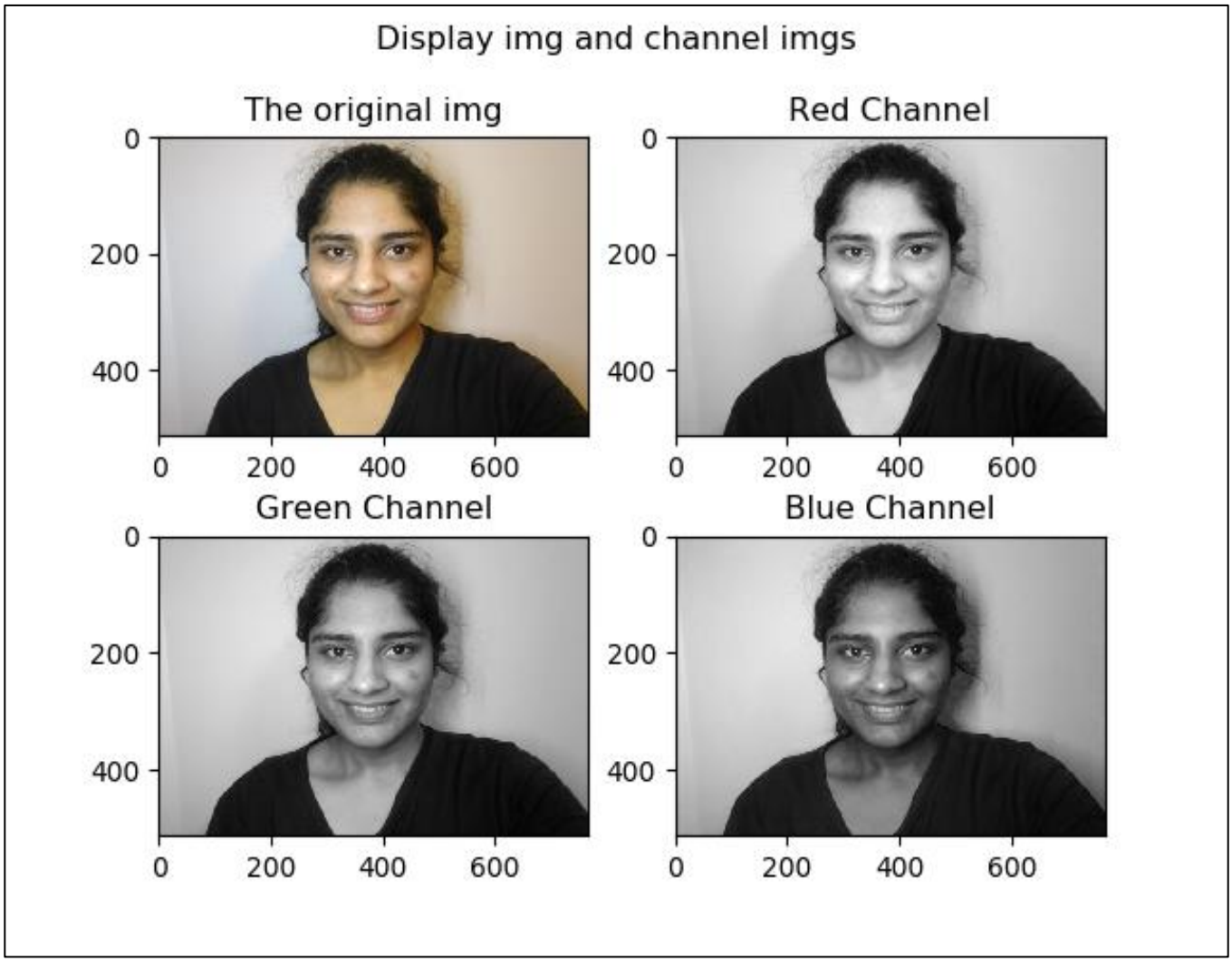


FIG 3.B - ORIGINAL AND RGB CHANNEL IMAGES

Analysis

As observed in Fig 3.b, the RGB channel images are grayscale and can be visually confirmed. We can infer the intensity of each of the colours by looking at their respective channel images.

3.c. Compute Histograms for RGB channel grayscale images

Documentation

We use cv2.calcHist() function to calculate histograms of each of the channel images. Then we plot using plt.plot()

Observations and Results

In the Fig 3.c, we display the histograms of pixel intensities of all three channels. The red line represents red channel histogram, blue line for blue channel and green line for the green channel.

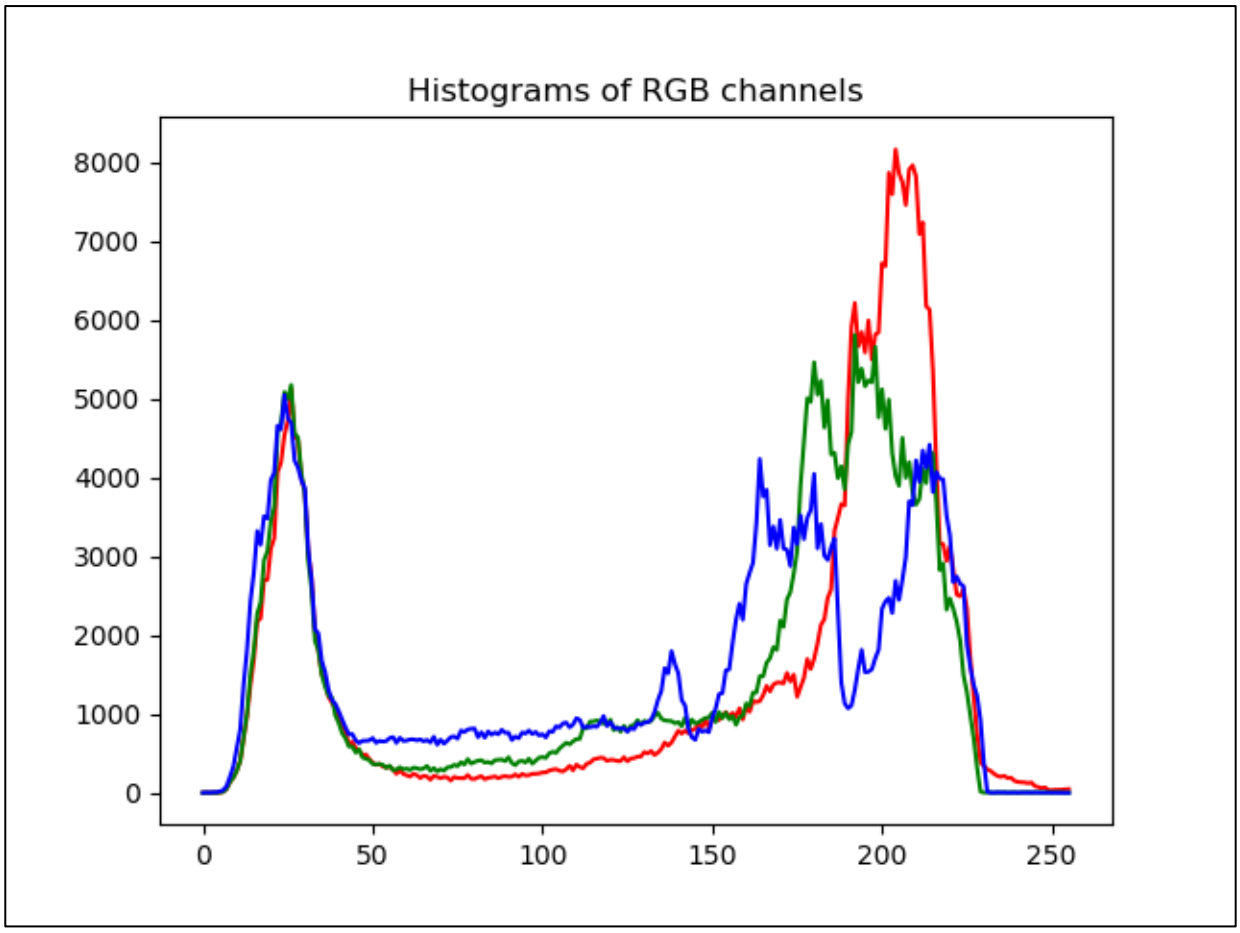


FIG 3.C -HISTOGRAMS OF PIXEL INTENSITIES RGB CHANNEL GRAYSCALE IMAGES

Analysis

From Fig 3.c, We can compare the number of pixels having a certain intensity. Like for red channel, a large number of pixels have intensity close to 255 which could mean that there is a lot of red color in our original image (can also be visually confirmed).

3.d. Equalize the image and the RGB channel grayscale images

Documentation

Using cv2.equalizeHist(), we equalize the RGB grayscale images. To get the equalized color image, we create a new image with the equalized RGB images as its RGB channels. Refer to Task3.py for details.

Observation and Results

In the Fig 3.d, we display the equalized original (upper-left), the equalized red channel grayscale (upper-right), the equalized green channel grayscale (lower-left) and the equalized blue channel grayscale image (lower-right).

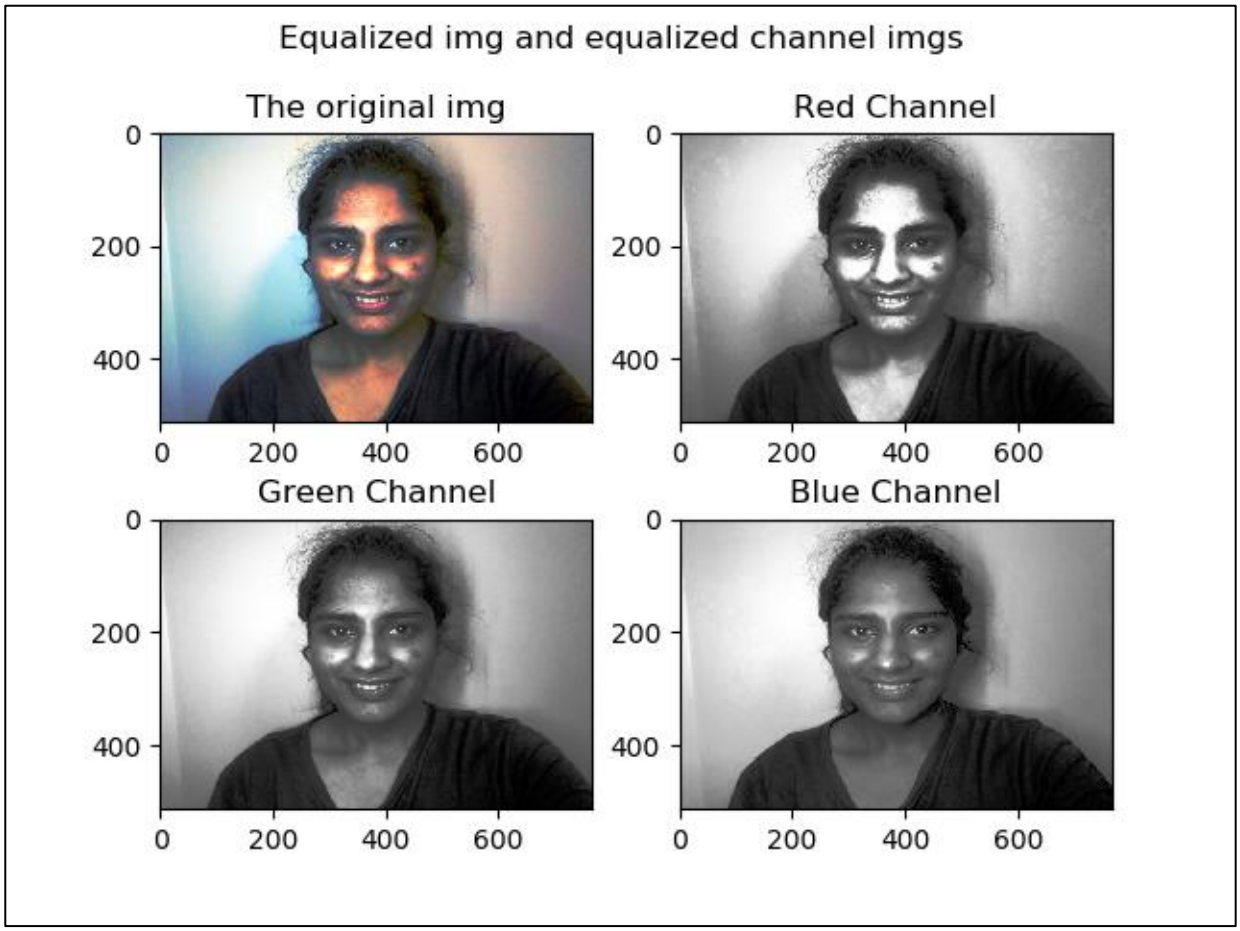


FIG 3.D - EQUALIZED IMG AND EQUALIZED RGB CHANNEL IMAGES

In the Fig 3.e, we display the histograms of pixel intensities of all three channels after equalization.

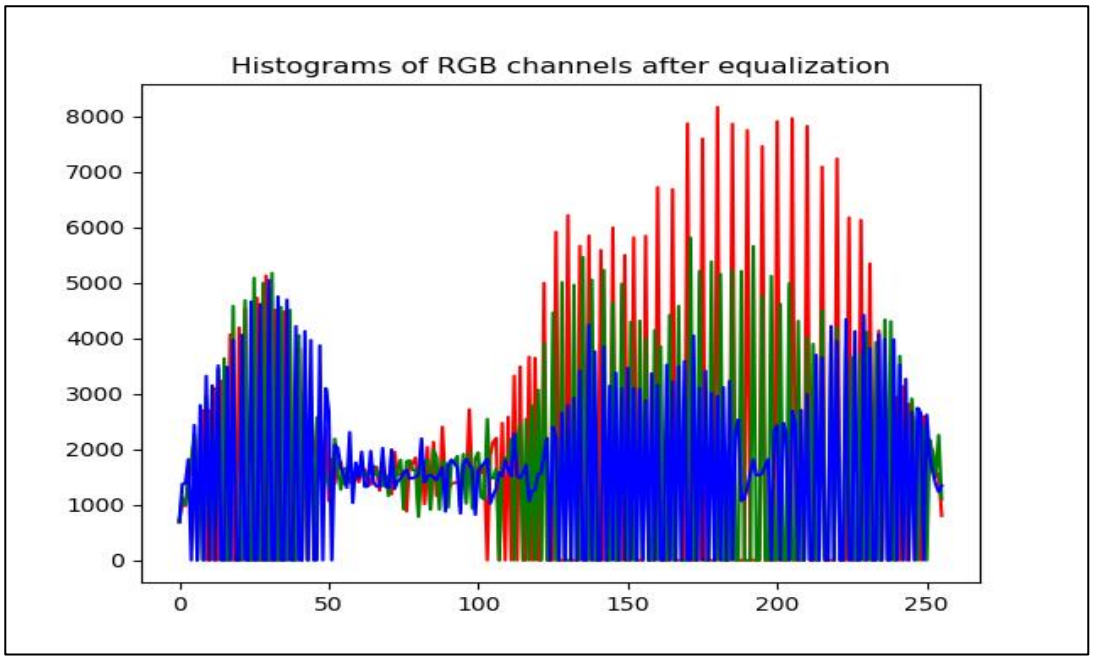


FIG 3.E - HISTOGRAMS OF EQUALIZED RGB CHANNEL IMAGES

Analysis

We use histogram equalization in order to improve the contrast of any image. From Fig 3.d, we observe that the contrast between lighter and darker parts of images has increased. This can help improve the lighting condition of our images.

Also, looking at Fig. 3.e, we can say that the histograms are more uniform now.

TASK 4 – IMAGE DENOISING VIA A GAUSSIAN FILTER

4.1 Crop and Resize

Documentation
Read in an image using cv2.imread(). To crop it, use *array slicing in python* and take from row 50 to 600, and from column 200 to 800, approximately giving a mid-section of the image. To convert the image to gray, we use cv2.cvtcolor() with the image as first argument and cv2.COLOR_RGB2GRAY as the second. Please refer to Task4and5.py for code.

Observations and Results

In the Fig 4.1, we display the original (left) and the cropped resized gray image (right) side by side for comparison.

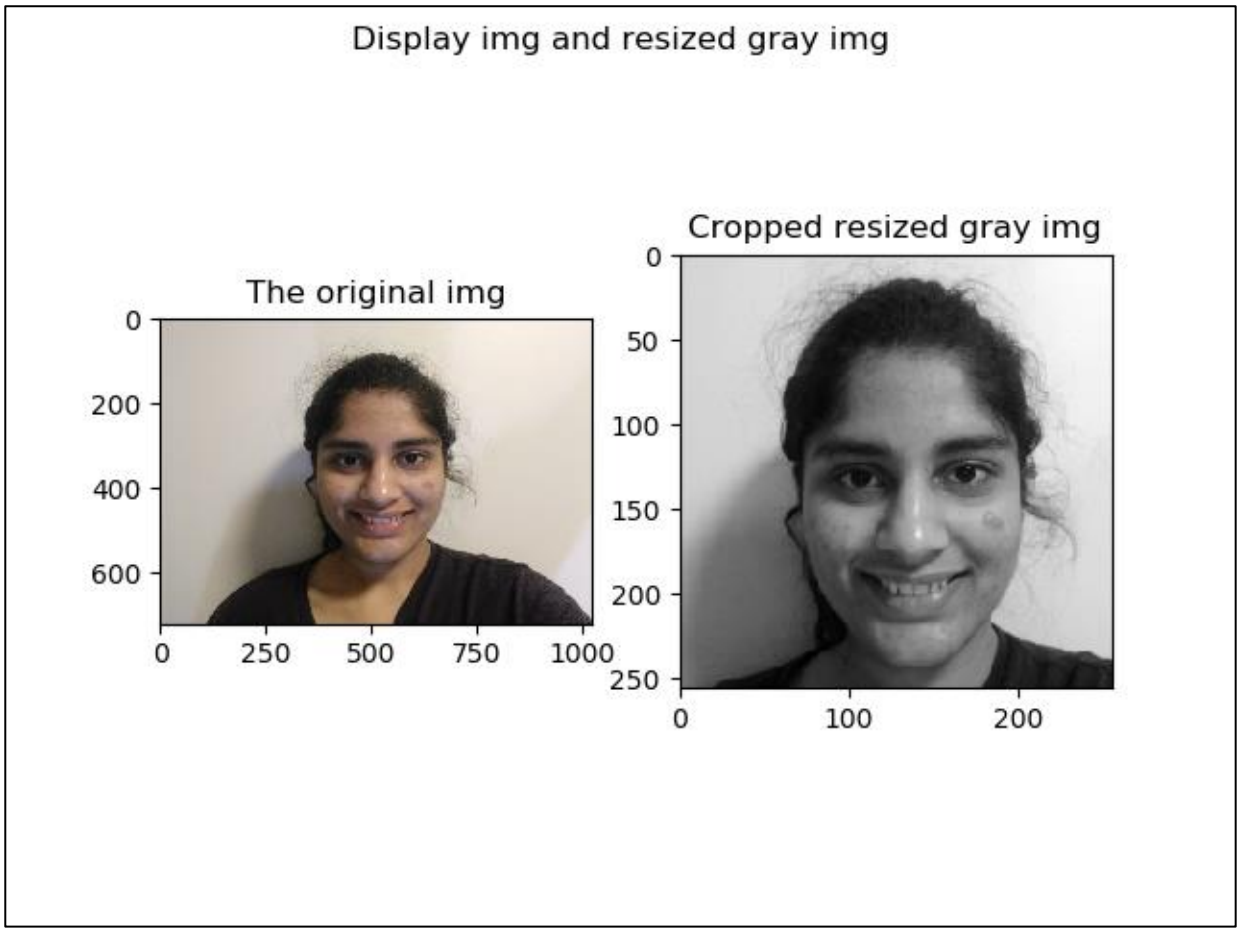


FIG 4.1 - THE ORIGINAL COLOR IMG AND THE CROPPED AND RESIZED 256X256 GRAY IMAGE

4.2 Add Gaussian Noise to the Image

Documentation

Using np.random.randn(256,256), we get a matrix of random values of same size as the image. Add it with the image to get the noisy image.

Observations and Analysis

In the Fig 4.2, we display the grayscale image with gaussian noise added.

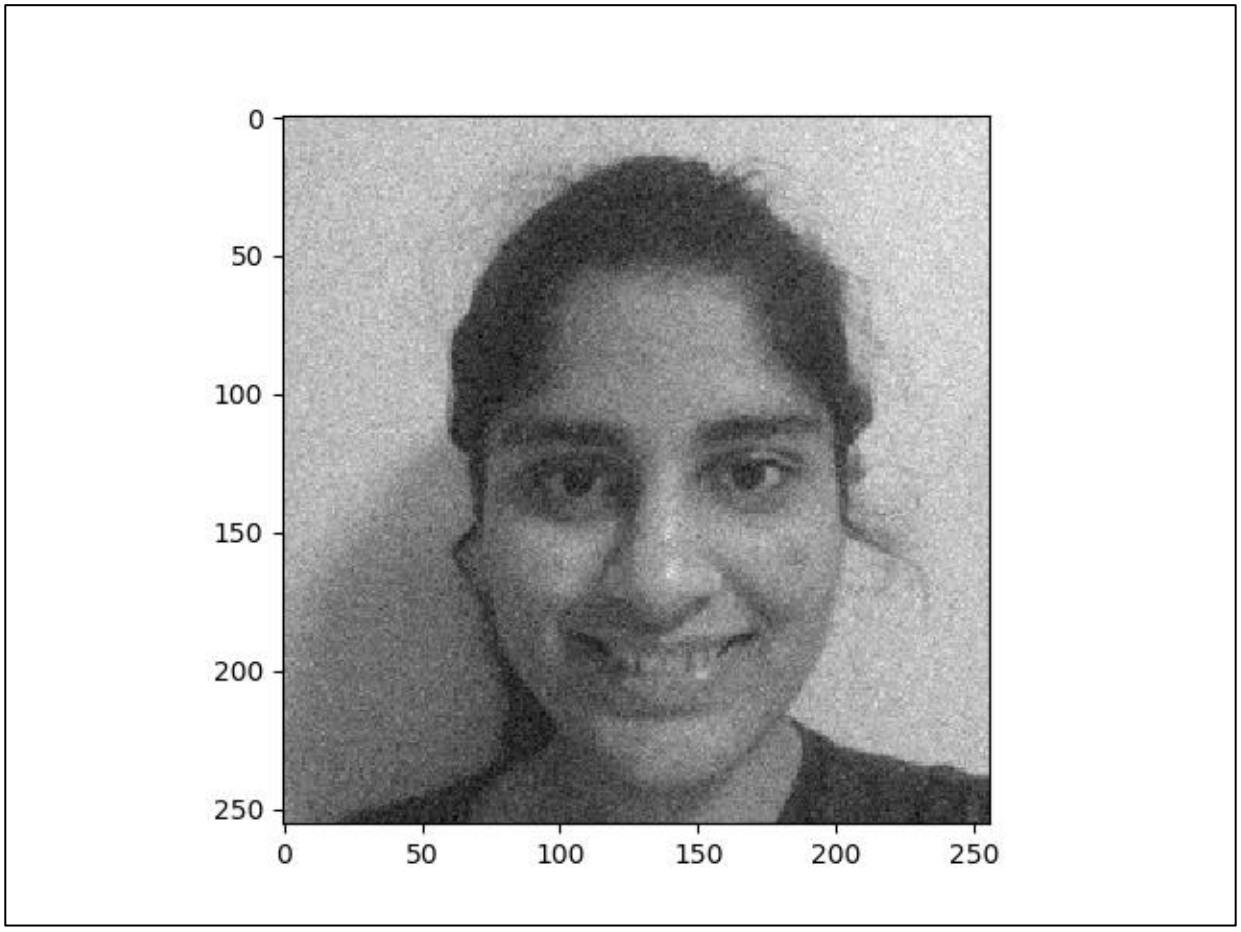
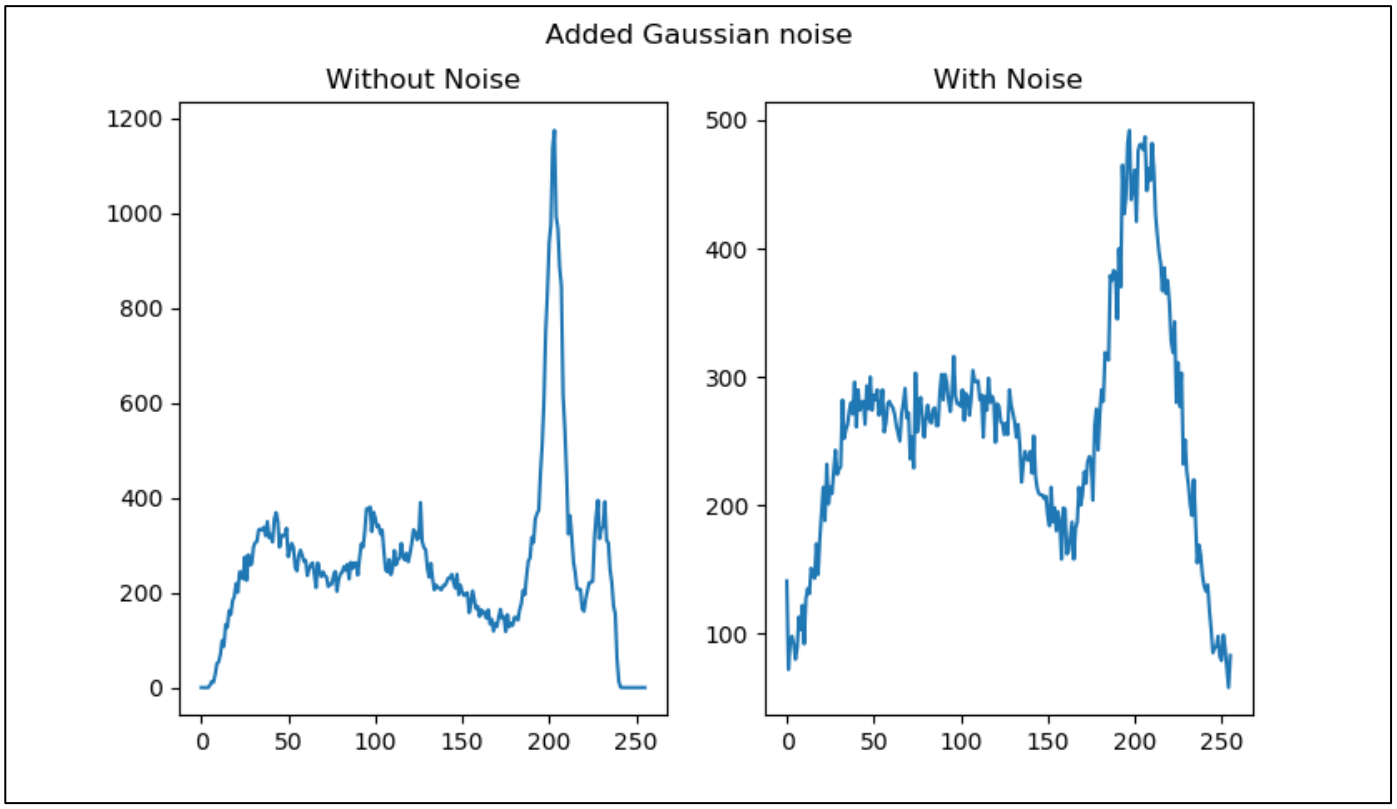


FIG 4.2 GRAY IMAGE WITH GAUSSIAN NOISE ADDED. WE CAN SEE THE NOISE VERY VISIBLY

4.3 Display Histograms

Observations and Results

In the Fig 4.3, we display the histograms of the gray image without noise (left) and the gray image with noise (right) side by side for comparison.



HISTOGRAM COMPARISON AFTER ADDING THE GAUSSIAN NOISE. OBSERVE HOW THERE IS MORE VARIATIONS IN THE HISTOGRAM WITH NOISE

Analysis

Adding noise makes our histogram values vary more. We can see the spikes originally in the image now have turned something like a gaussian-distribution centered on the spike. This is classic property of Gaussian which achieves its highest value at its mean. So if we are adding a gaussian noise with zero mean to some data, the mean of the resulting is just mean of the data and takes the highest value there.

4.4 Implement Gaussian filter

Documentation and Observations

To create a Gaussian kernel, we use cv2.getGaussianKernel(5, 1) which creates an array of size 5 and samples from a Gaussian Distribution of standard deviation 1 and mean 0. To convert it into a 5x5 kernel, we take the outer product of this vector with itself. Observing the kernel, it has highest value in the middle and goes low as we move away from the middle. Please refer to Task4and5.py for the complete code.

For the convolution function, we first pad the image with appropriate padding ((kernel_size-1)/2) so that out output image is of the same size as input. We perform the convolution operation by sort of superimposing the kernel starting at origin (x=0, y=0), doing element-wise multiplication and then adding all the values. This gives us one value of the output image. We move the filter over the whole image in strides of 1 in the x and y direction and keep saving the results to the new image. Then we return the new/final image.

4.5 Display Smoothed Images

Observations and Results

In the Fig 4.4, we display the gray image with noise (left) and the smoothed gray image (right) side by side for comparison. The smoothed gray image is obtained after convolving with our custom gaussian filter of standard deviation 1.

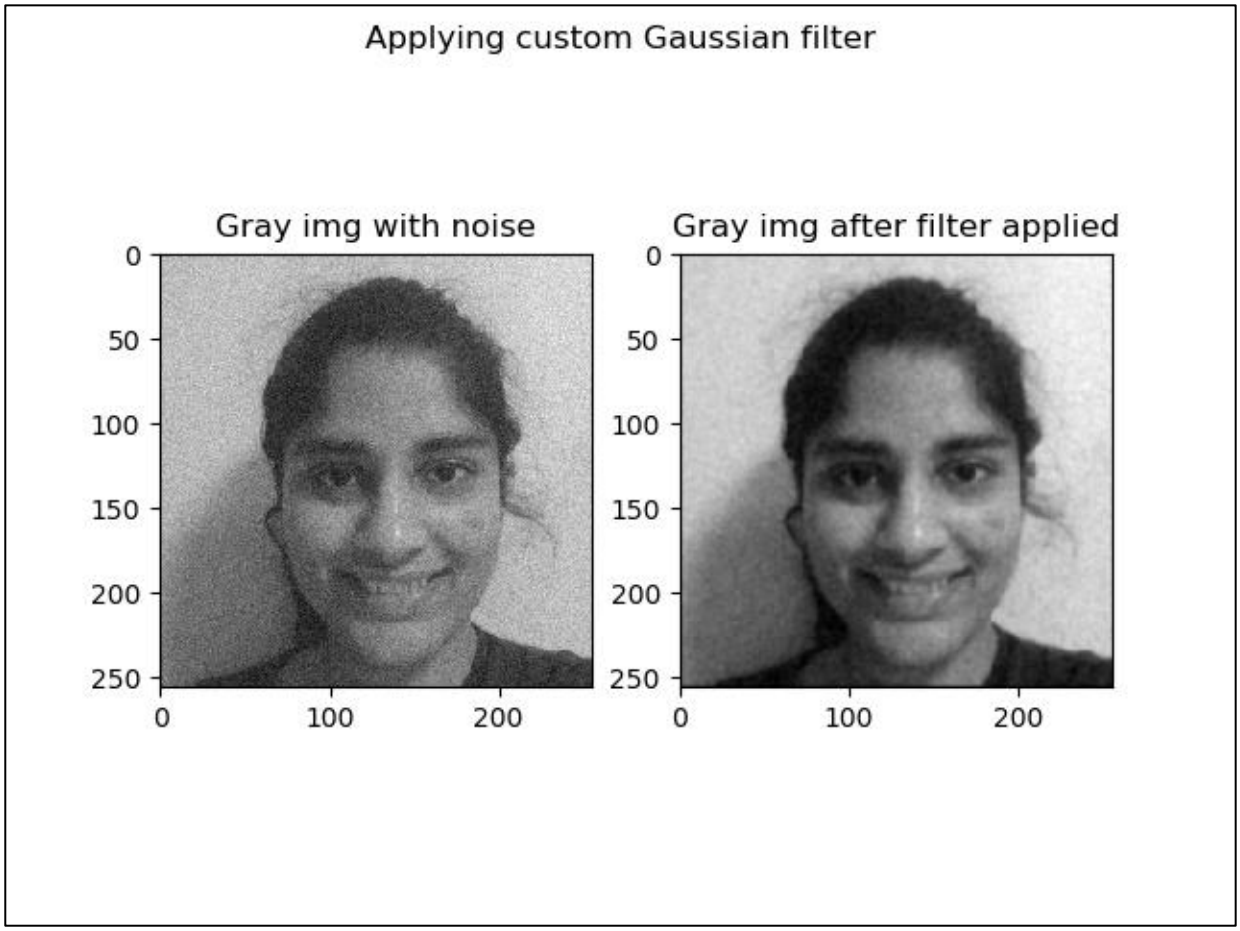


FIG 4.4 – ORIGINAL IMAGE WITH NOISE AND SMOOTHED IMAGE DISPLAYED. THIS SMOOTHED IMAGE IS OBTAINED AFTER CONVOLVING THE NOISY IMAGE WITH A GAUSSIAN FILTER OF STD=1

In the Fig 4.5, we display smoothed images with gaussian filters of different standard deviations.

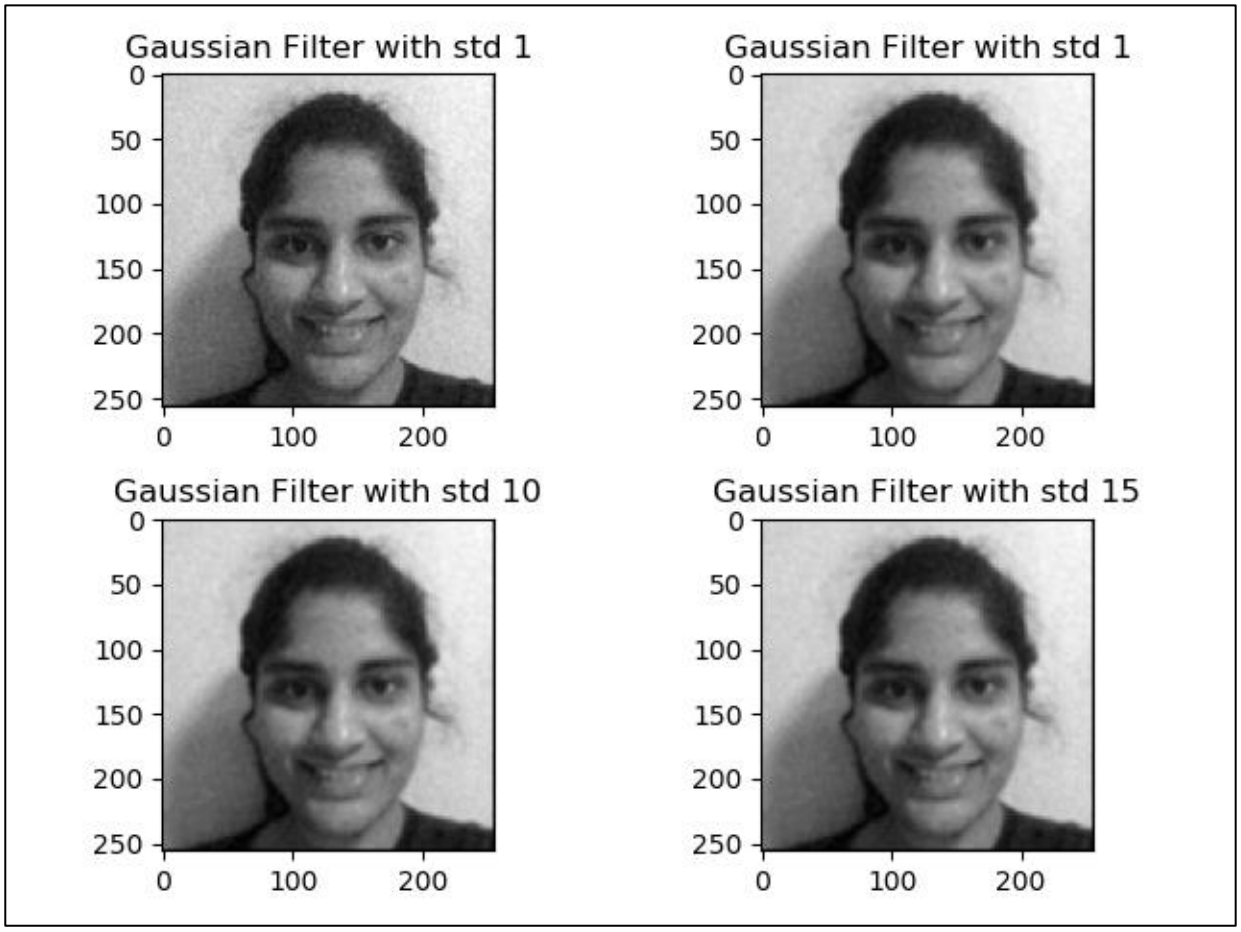


FIG 4.5 – SMOOTHED IMAGES AFTER APPLYING GAUSSIAN FILTERS.

Results

First observation – from Fig 4.4, we observe that applying a gaussian filter has removed some of the noise as expected. The gaussian filter has averaged out the variations introduced by gaussian noise. The mean stays the same (which we want) but it has smoothed out the variance.

Second observation – from Fig 4.5, we can see that as we increase the standard deviation of our filter, the resulting image becomes smoother. At roughly std=15, we get an image close to our original image. This is justified as our original noise was of std=15. To smooth out that noise, it makes sense that a filter with standard deviation of 15 would be a good option.

4.6 Comparison with Built-in filter

Documentation

We use cv2.GaussianBlur() to get smoothed image from built-in function.

Observations and Analysis

In the Fig 4.6, we display smoothed images with custom gaussian filter (left) and built-in gaussian filter(right).

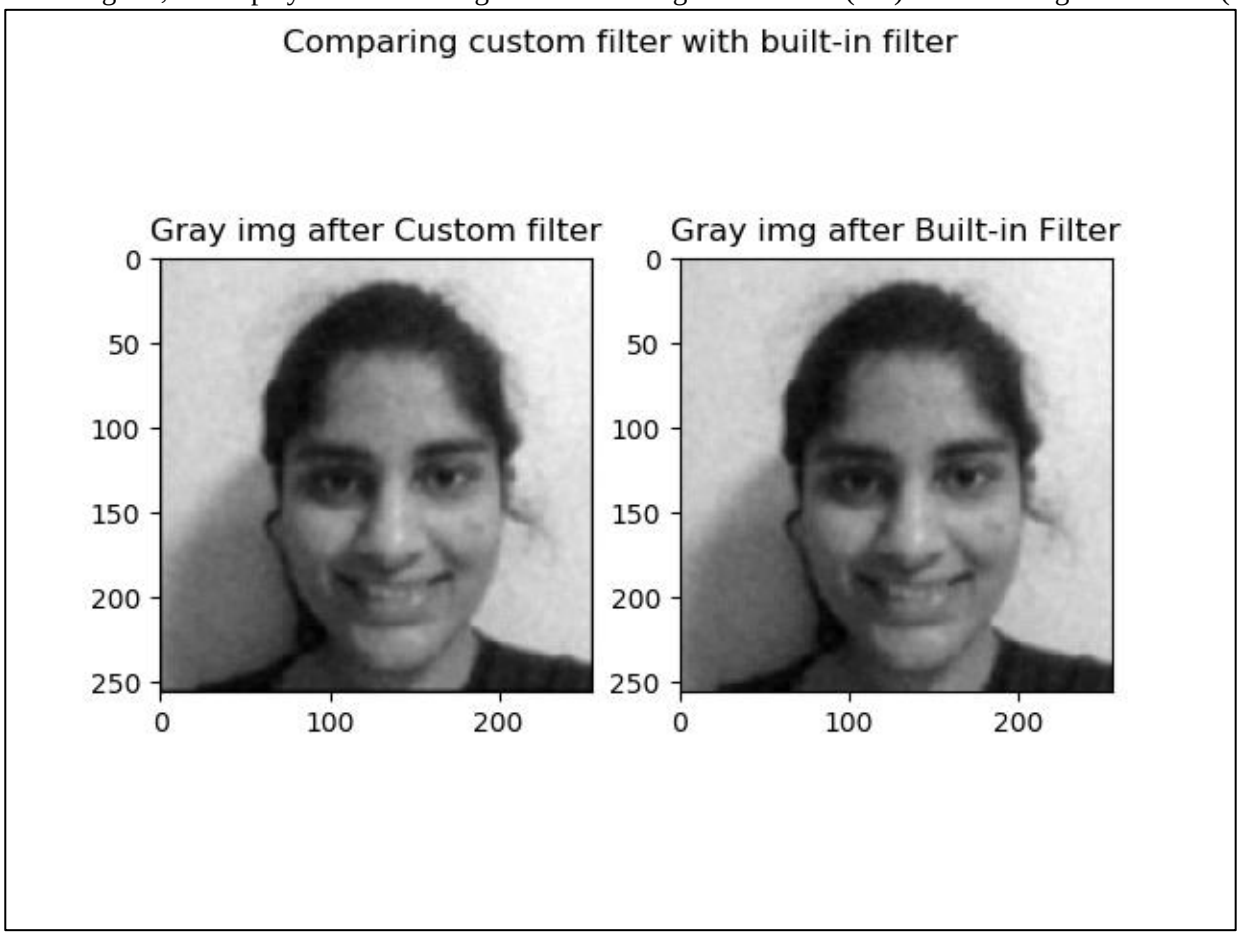


FIG 4.6 – IMAGE COMPARISON BETWEEN OUR CUSTOM GAUSSIAN FILTER AND BUILT-IN FUNCTION.

In the Fig 4.7, we display smoothed image histograms with custom gaussian filter (left) and built-in gaussian filter(right).

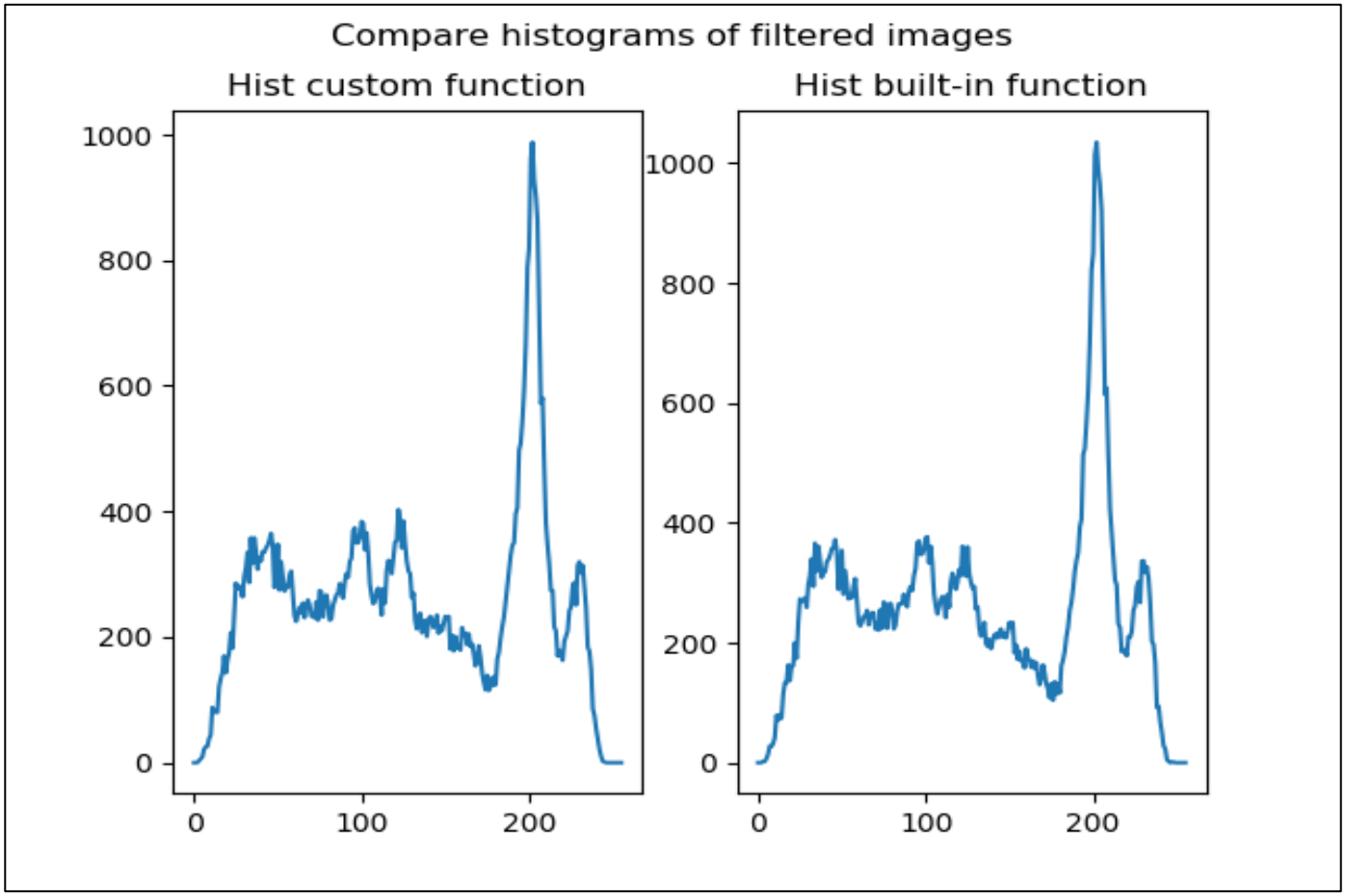


FIG 4.7 - HISTOGRAM COMPARISON BETWEEN OUR FUNCTION AND BUILT-IN SMOOTHING FUNCTION

Results

From Fig 4.6, we can see that visually, the custom filter performs very identically to the built-in function. The reason is that we perform the same operation as a gaussian blur would perform. Also, it can be seen from Fig 4.7 that the resulting images have very similar histograms which strengthens our claim that the filter written by us does as good a job as the built-in function.

TASK 5 – SOBEL FILTER

Documentation

To create a sobel filter, we use this matrix np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
We have already written the convolution function in Task 4. We use that function to convolve our image with the Sobel filter and get the final image. Please refer to Task4and5.py for the code.

Observations and Results

In the Fig 5.1, we display original gray image(left), image after applying vertical sobel filter (middle) and image after applying horizontal sobel filter (right). This is the result of our own custom sobel filter.

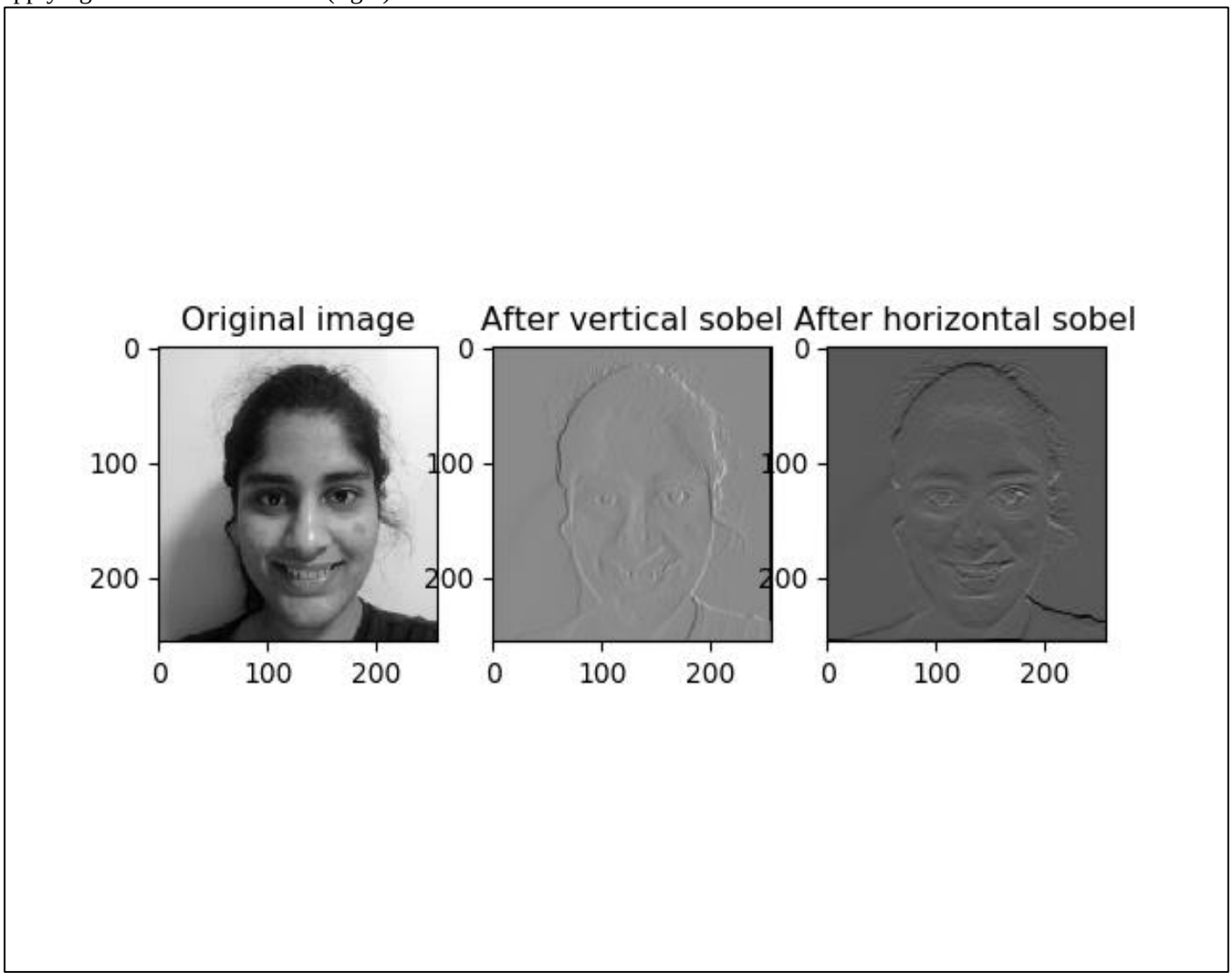


FIG 5.1 – VERTICAL AND HORIZNTAL EDGES AFTER APPLYING OUR SOBEL FILTER. NOTICE HOW THE VERTICAL FILTER HIGHLIGHTS VERTICAL EDGES ONLY AND SAME FOR HORIZONTAL.

In the Fig 5.2, we display a darker image, along with its vertical and horizontal edges using our Sobel filter.

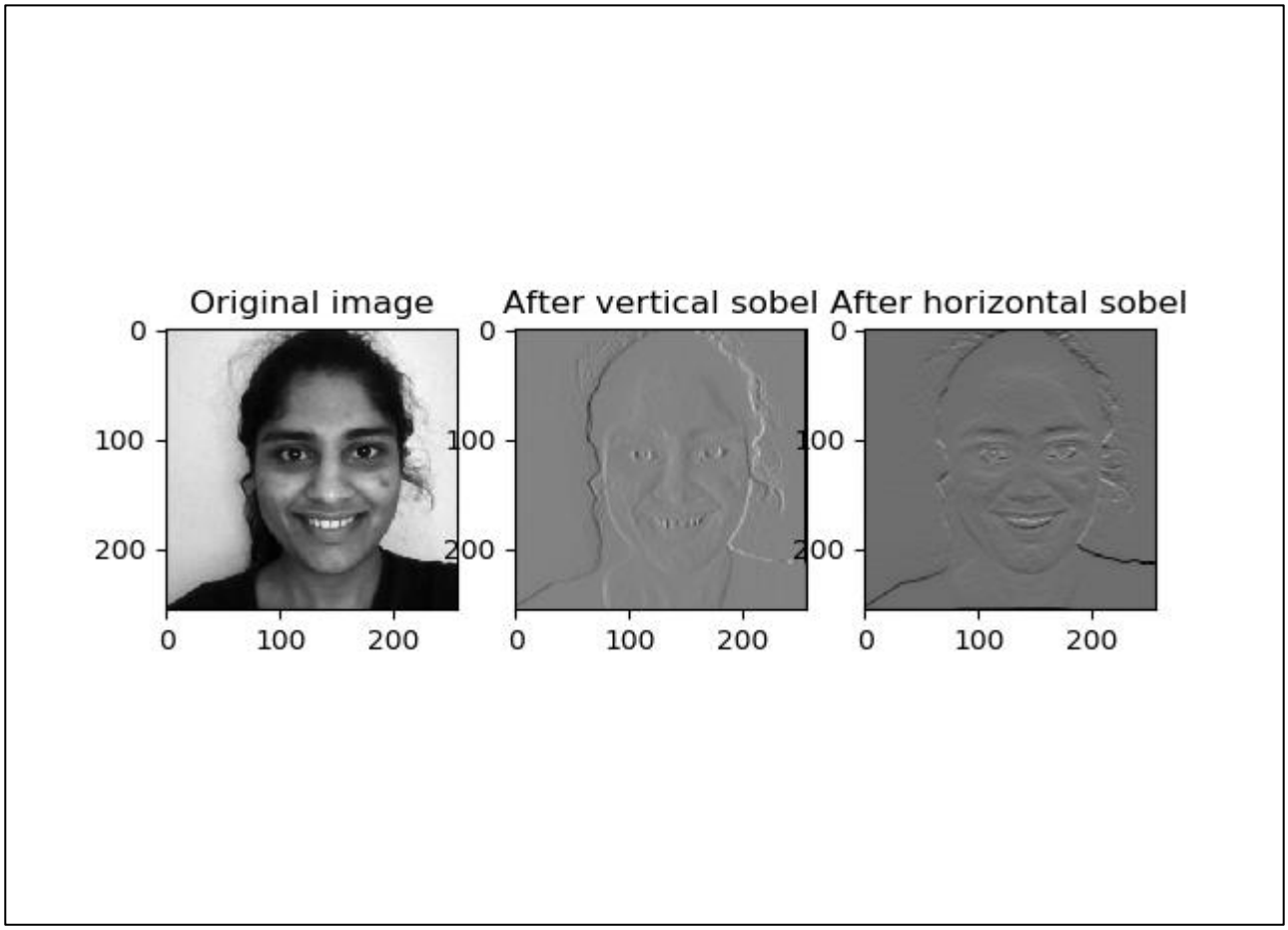


FIG 5.2 – VERTICAL AND HORIZNTAL EDGES AFTER APPLYING OUR SOBEL FILTER. NOTICE HOW THE VERTICAL FILTER HIGHLIGHTS VERTICAL EDGES ONLY AND SAME FOR HORIZONTAL.

In the Fig 5.3, we display original gray image(left), image after applying vertical sobel filter (middle) and image after applying horizontal sobel filter (right). This is the result of the built-in sobel filter – cv2.Sobel()

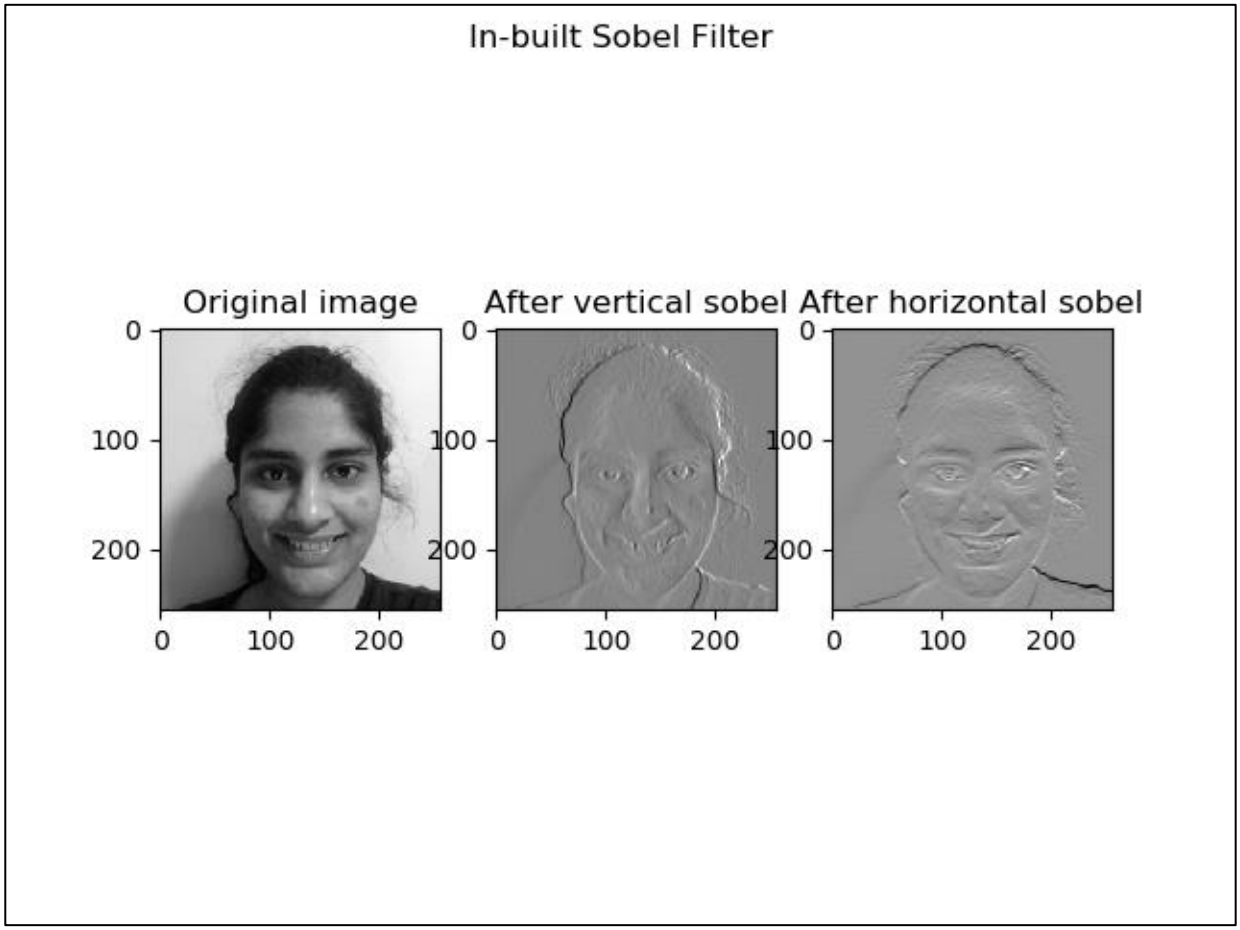


FIG 5.3 – VERTICAL AND HORIZNTAL EDGES AFTER APPLYING BUILT-IN SOBEL FILTER.

In the Fig 5.4, we display a darker image, along with its vertical and horizontal edges using cv2.Sobel() filter.

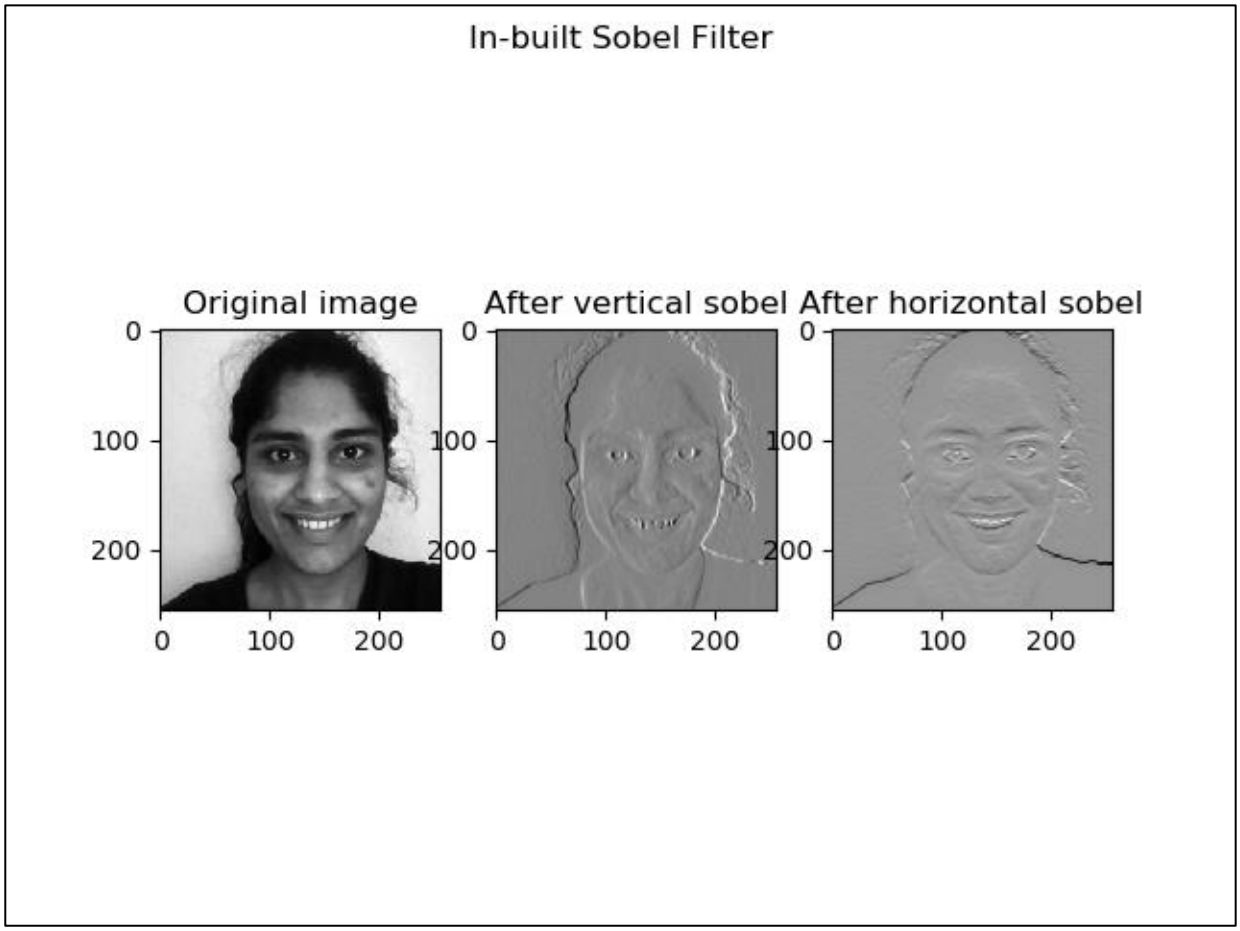


FIG 5.4 – VERTICAL AND HORIZNTAL EDGES AFTER APPLYING BUILT-IN SOBEL FILTER.

Analysis

A sobel filter is created such that it has a stark gradient -> [-1 0 1] which lends the property of identifying edges. If there is a vertical edge, the values to the left of it get cancelled by the values on the right and we get a dark pixel there. That is how we are able to detect edges. Same for horizontal [-1 0 1].T

From figure 5.1 and 5.2, we observe that the edges are darkened in the respective directions which we expect.

Also, from Fig 5.3 and 5.4, we can safely say that the results of our filter is the same as in-built sobel filter.

6.1 Rotation

Documentation

To rotate the image, we write a function **rotate** which gives rotated x and y values given an input point, a point around which to rotate and a theta to rotate by. Then we write a function **rotate_forward_map** which takes in an image and an angle of rotation and returns the rotated image. This function just takes the int values of rotated_x and rotated_y that we got from **rotate** function, hence we see holes in our rotated images. We don't use any interpolation technique in this function. Please refer to Task6.py for the code.

In the Fig 6.1, we display rotated images that we got from our function **rotate_forward_map**.

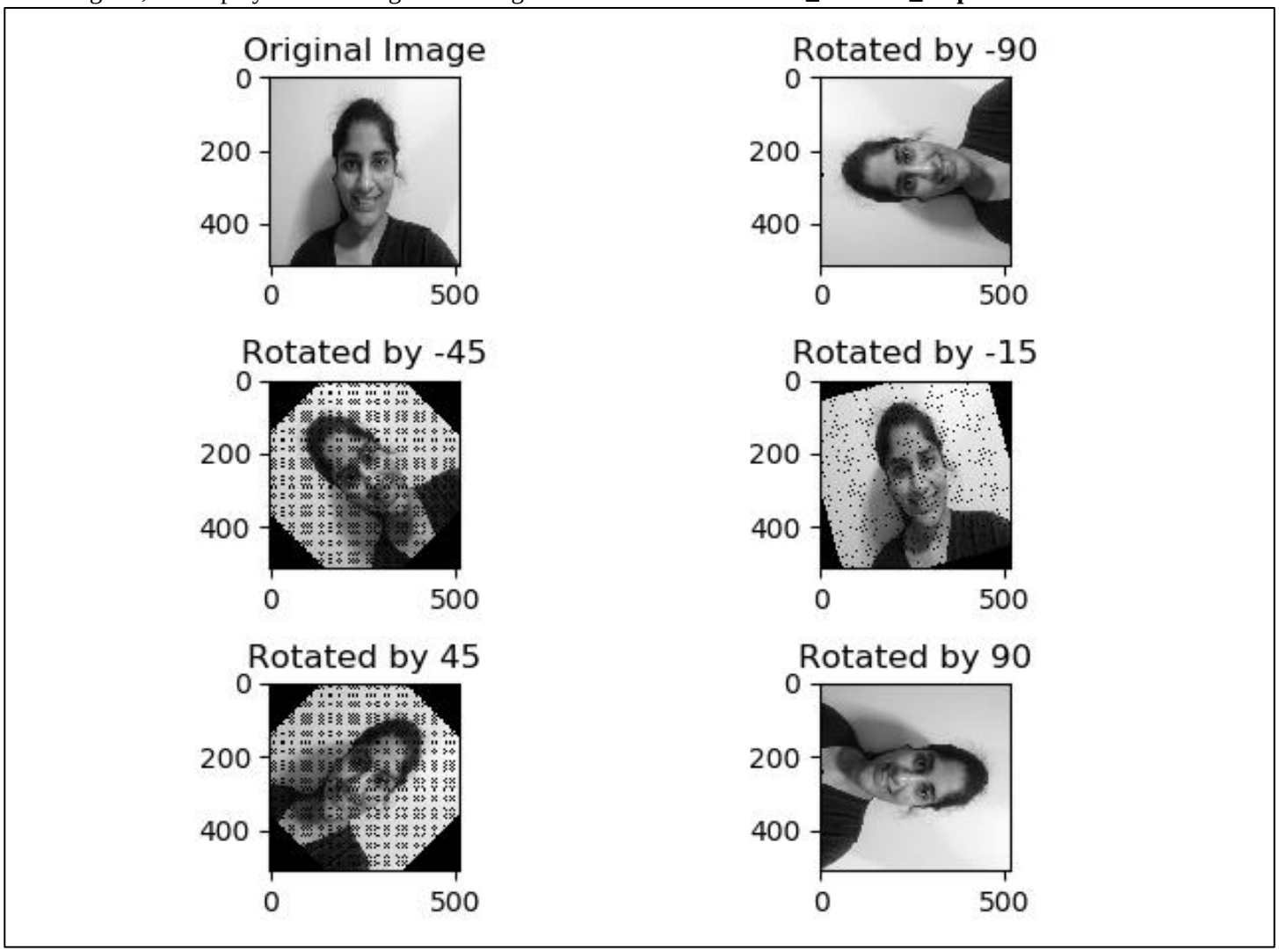


FIG 6.1 – ORIGINAL IMAGE AND ROTATED IMAGES BY DEGREES SPECIFIED IN THE FIGURE

Observation and Analysis

From Fig 6.1, we observe that the images have been rotated since we multiplied our x and y with a rotation matrix and got new x and y. But since the new x and y will not always be integers, we get a lot of holes in out image. Also, since we are using forward mapping here, the hole issue is not resolved here. We will see it in section 6.2 on how to resolve it.

6.2 Compare forward and backward mapping

Documentation

We already saw in section 6.1 how rotation can lead to holes. If we carefully look at our code in Task6.py – **rotate_forward_map**, we can see that not all points in the destination matrix are given a value and that is why we observe holes.

To counteract that, we implement a function called **backward_map** which iterates over the *destination* points and fills them with the pixel value at a point that we get from rotating by a negative theta (inverse mapping). Please refer Task6.py for the complete code.

Observations and Results

In the Fig 6.2, we compare and contrast images rotated using forward (left column) and backward (right column) mapping by degrees mentioned in the figure.

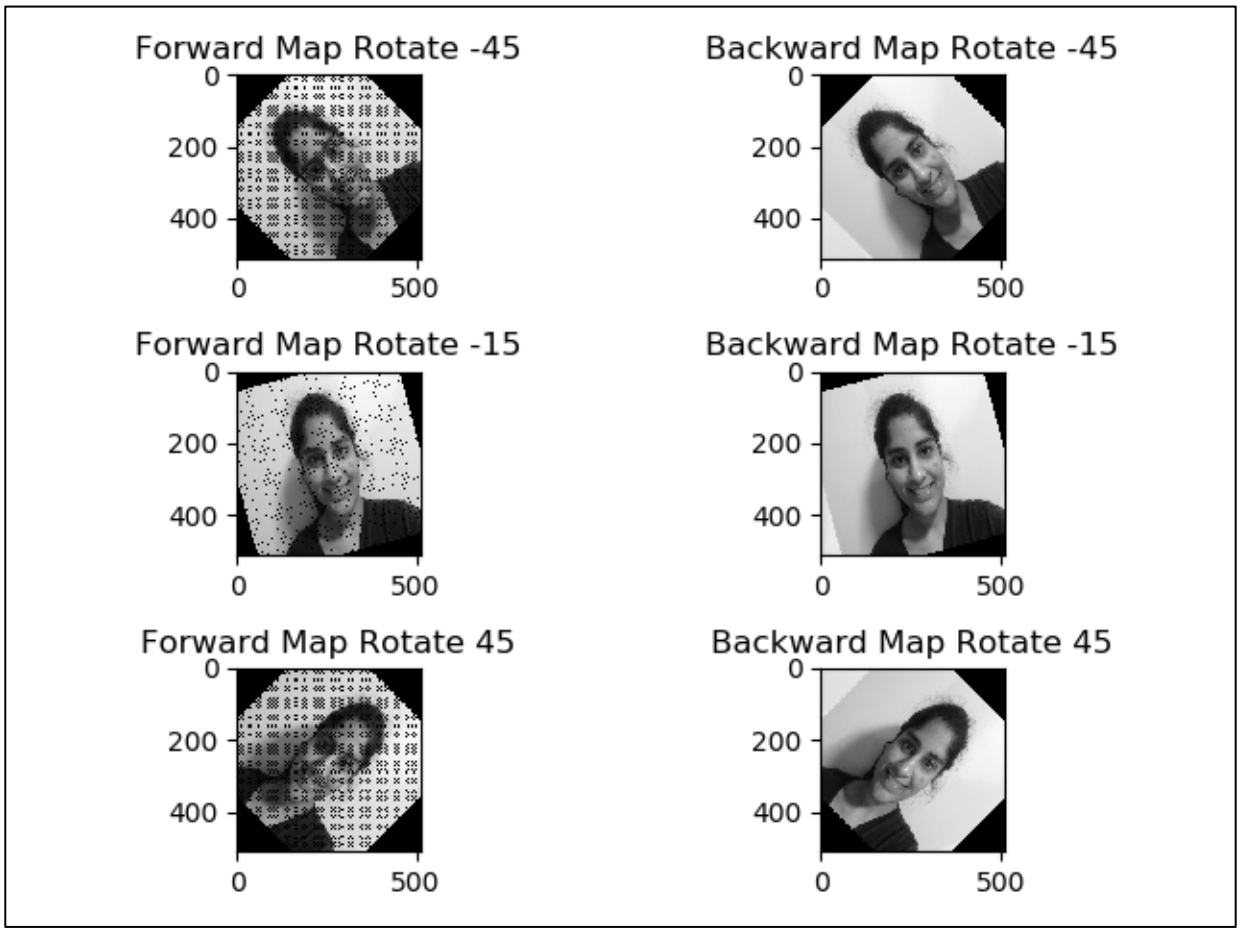


FIG 6.2 – VISUAL COMPARISON OF FORWARD AND BACKWARD MAPPING TECHNIQUES. BACKWARD MAPPING TAKES CARE OF HOLES AND IS THEREFORE A PREFERRED TECHNIQUE

Analysis

As discussed in the Documentation section 6.2, iterating over destination points leads to backward mapped rotated images not having any holes and therefore being visibly better as seen in Fig 6.2. The time complexity of both is same, therefore we almost always prefer backward/inverse mapping when we have to rotate images.

6.3 Comparison of Interpolation Techniques

Documentation

One of the techniques is to just take the integer values of our rotated coordinate values. This leaves holes in images. Can we use something else?

Another technique we use is called the nearest neighbour technique. In this, we find distance of our point to all its neighbours and get the nearest point. We assign to our rotated coordinates these new nearest neighbour coordinates and assign it the value at the original image (x,y before rotation). For details, please refer to Task6.py

We also implement bilinear interpolation taking help from the Wikipedia article on the same. The basic idea is to get linear combination of x and y to get an approximation of the value at rotated image.

Observations and Results

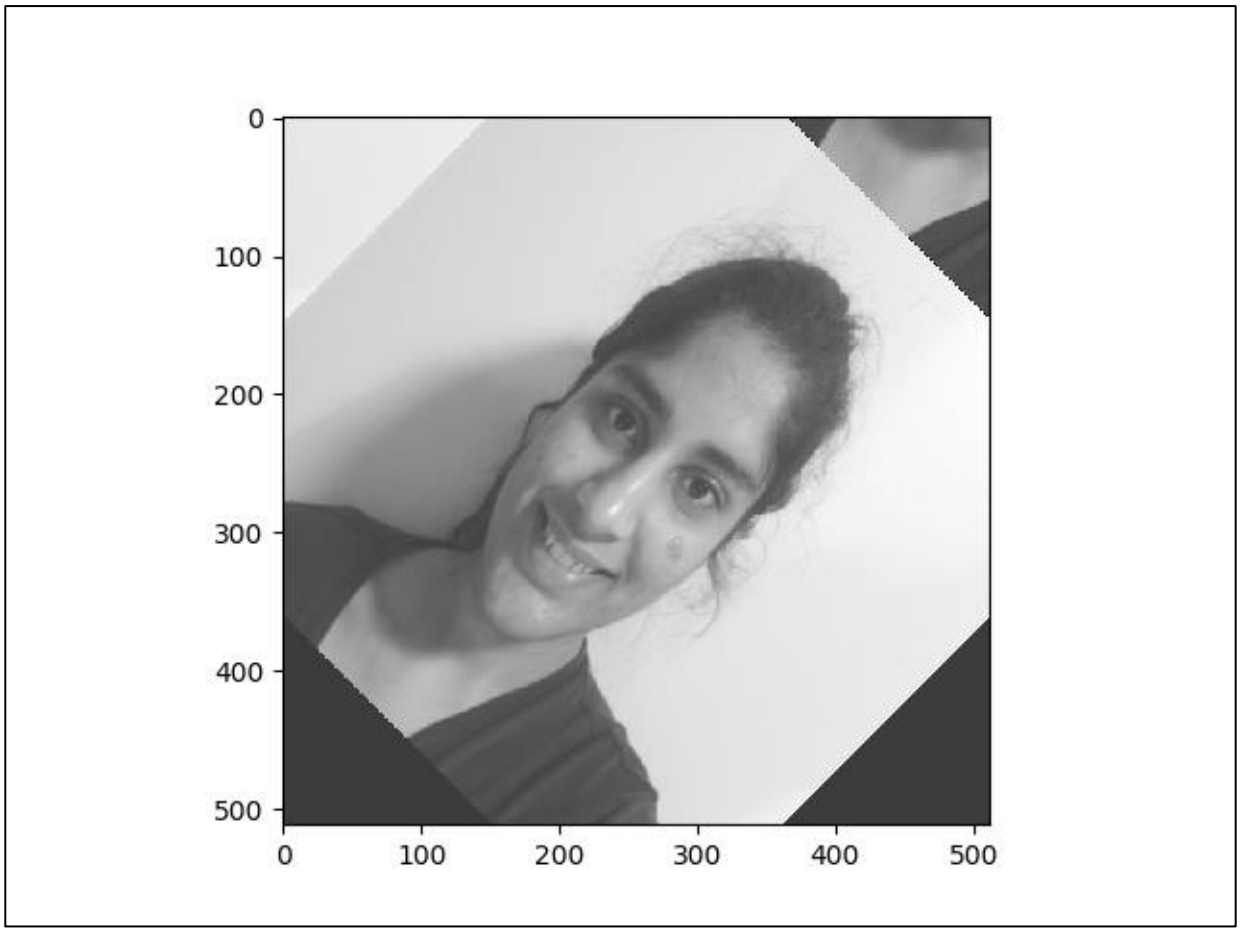


FIGURE 6.3 – RESULT OF BILINEAR INTERPOLATION USED WHILE ROTATION BY 45 DEGREES

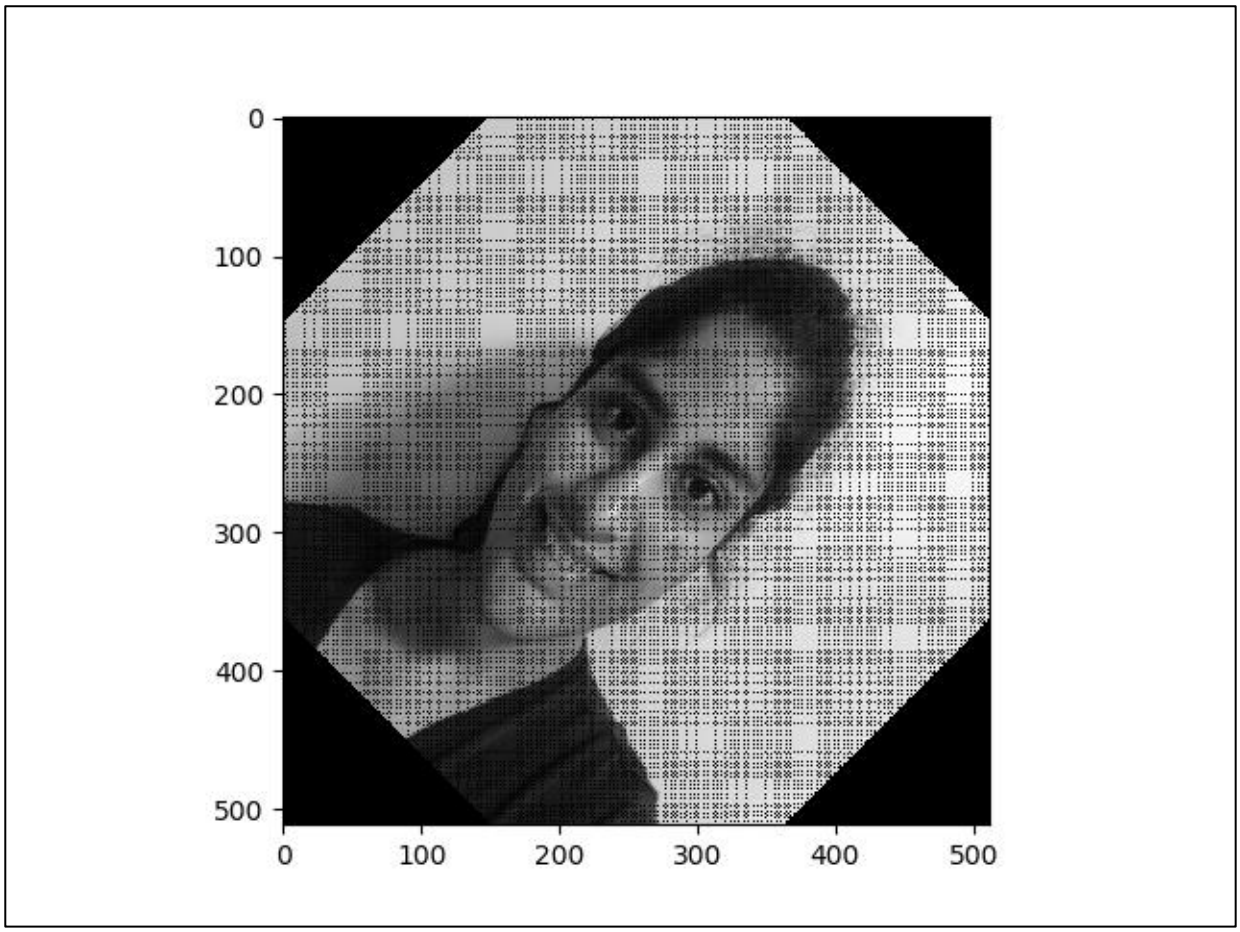


FIG 6.4 – RESULT OF NEAREST NEIGHBOUR INTERPOLATION USED TO ROTATE BY 45 DEGREES

Analysis

From Figure 6.4, we can see that bilinear mapping is visibly much better that nearest neighbour mapping. Also, the time complexity of nearest neighbour is lower than that of bilinear. Nearest neighbour's results can be blocky, whereas since bilinear uses a weighted average of the four nearest pixels, it gives a blurring effect but it's not blocky.

Thank you for your time and patience.