Clab-1 Report
ENGN4528

Kirat Alreja
u7119530

04/04/2022

Task 1

Python

(1) a = np.array([[1, 2, 3],[5, 2, 20]])
Creates an array of shape (2,3) with the given elements

(2) b = a[1 , :]
Slice the array and keep only the 2nd row with elements [5,2,20]

(3) f = np.random.randn(200,1)
Make a random array of 200 numbers in one row, with normal distribution

(4) g = f [f > 0]
Filters out the array, and keeps only positive numbers

(5) x = np.zeros (50) + 0.5
Make an array of 50 zeros, and add 0.5 to each elements. Effectively making an array of 50 numbers, all with a value of 0.5

(6) y = 0.5 * np.ones([1, len(x)])
Exactly the same operation as (5). It creates an an array of 50 ones, using the length of the previous 50 number array. Then, multiply 0.5 with all numbers in the array, effectively generating an array of 50 numbers, all with a value of 0.5

(7) z = x + y
Add the two arrays, adding the respective elements of each array. This creates an array of 50 numbers, each with a value of 1.0 (0.5 + 0.5).

(8) a = np.linspace(1,200)
Creates an array, with numbers equally spaced between the specified start & end intervals - in this case, 1 & 200.

(9) b = a[: :-1]
Reverses all the numbers of array

(10) b[b <35]=0
Replaces all the values greater than 35 in the array with the value 0

Task 2

This task is about resizing ‘image1.jpg’, and working with the individual RGB channels & their histograms. The default color space of openCV library is BGR, so I converted the same to RGB before performing any operations on the image. Then, for task 2.b, I separated the image into the three color channels by slicing the image array in the third dimension (Fig1).

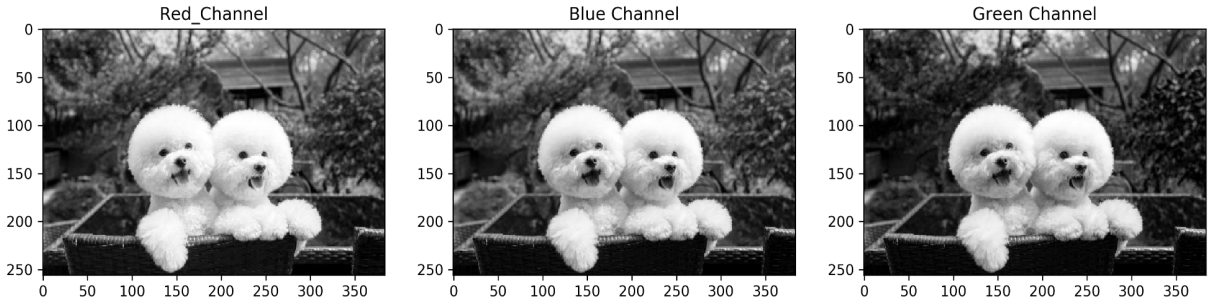

Fig1 : RGB Channels for ‘image1.jpg’

Observing the channels in grayscale side-by-side shows that there is no major visual difference, except for minor brightness & contrast changes. Next, I plotted the respective histograms for these images (Fig2).

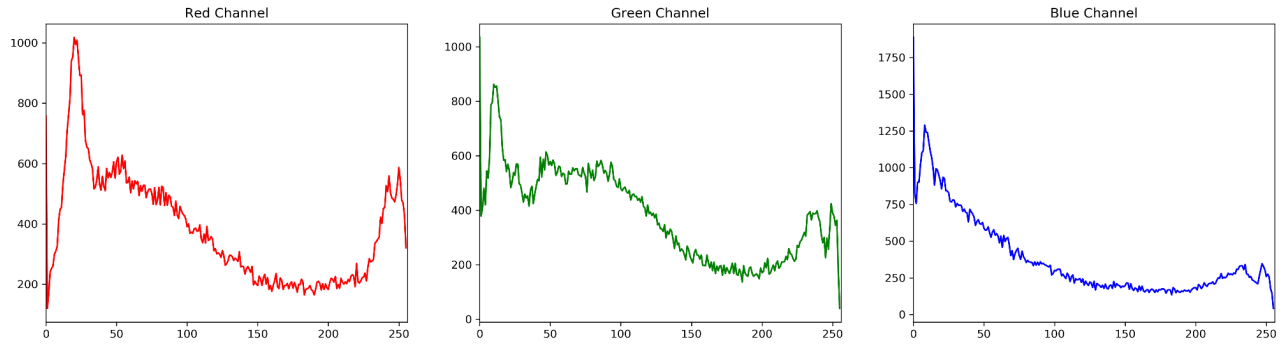

Fig2: Histograms for each channel in ‘image1.jpg’

We can observe that the gray levels are on the higher side for shadows, and the image is fairly well exposed otherwise. Then, I used histogram equalization on the original image & the three separate grayscale channel images (Fig3).

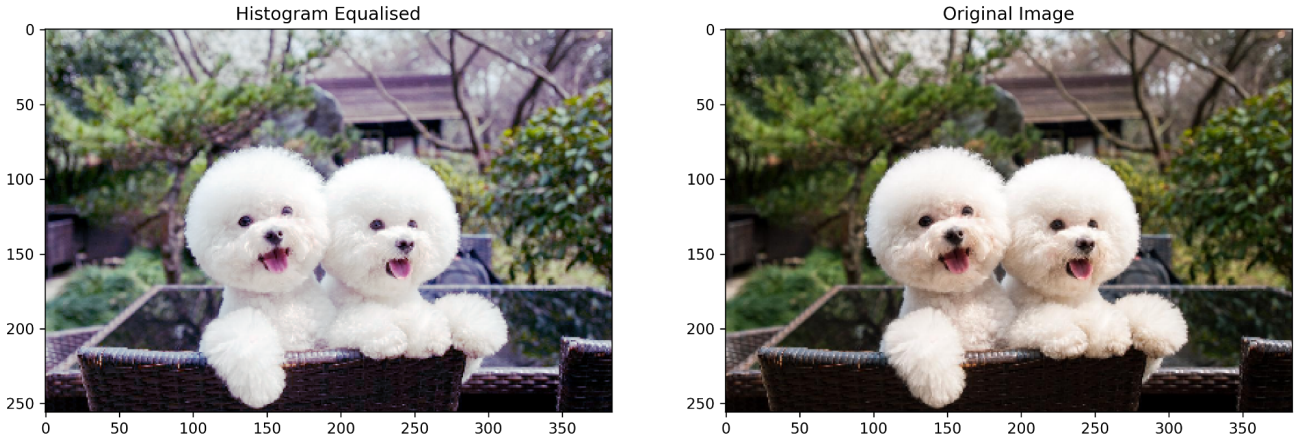

Fig3: Histogram Equalised vs Original Image 'image1.jpg'

Upon observing the two images side-by-side, we can see that the equalised image brings back the information lost by the high contrast in the shadows. The face of the puppies is much more evenly exposed. To understand the effect of histogram equalisation thoroughly, I equalised the three channels separately & then plotted them again (Fig4).

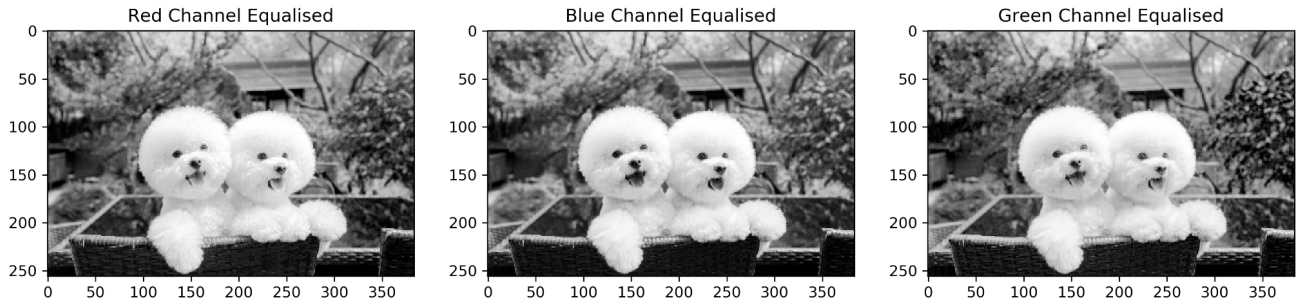

Fig4: Three channels equalised separately

Comparing the images in Fig4 to Fig1, the effect is histogram equalisation is clearly visible. In Fig1, we have brightness & contrast changes across channels. However, after equalisation, all three channels are similarly exposed - which results in an overall well-exposed RGB color image. Analysing the histograms for four of the equalised images, we can observe the same effect (Fig 5 & Fig 6).

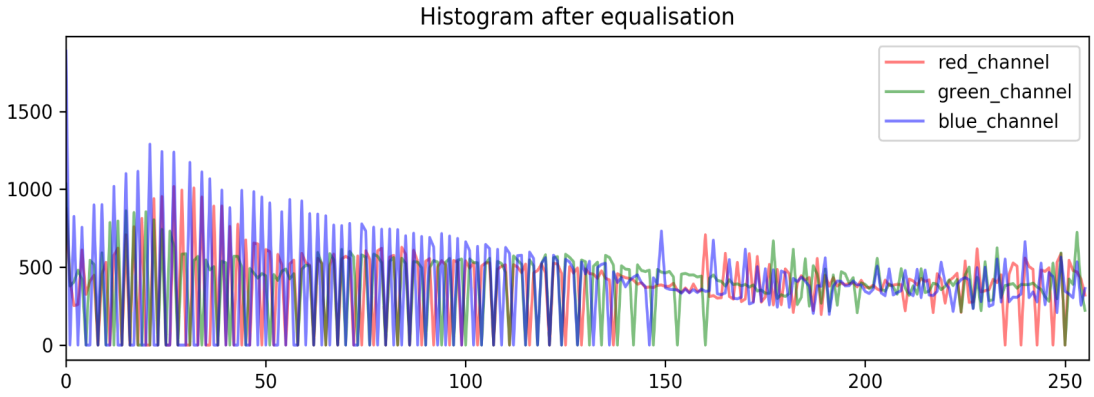

Fig5: Histogram for RGB after equalisation

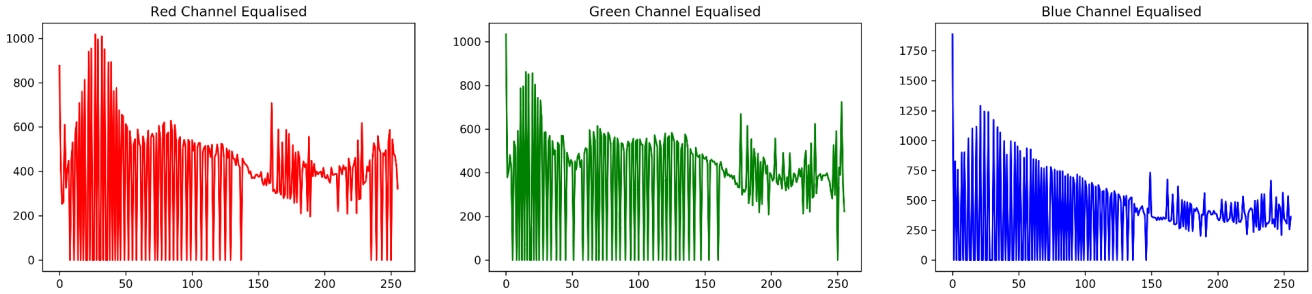

Fig6: Histogram for RGB channels equalised individually

Task 3

This task is about RGB-to-HSV conversion & visualising the HSV channels separately. To write the RGB-to-HSV conversion, I used the standard mathematical formula and coded a function that converts a single RGB pixel to its corresponding HSV value. Then I used a helper function 'imageToHSV' to iterate the conversion over every single pixel of an image. Applying the same over 'Figure2-a.png' yields the following result (Fig7).

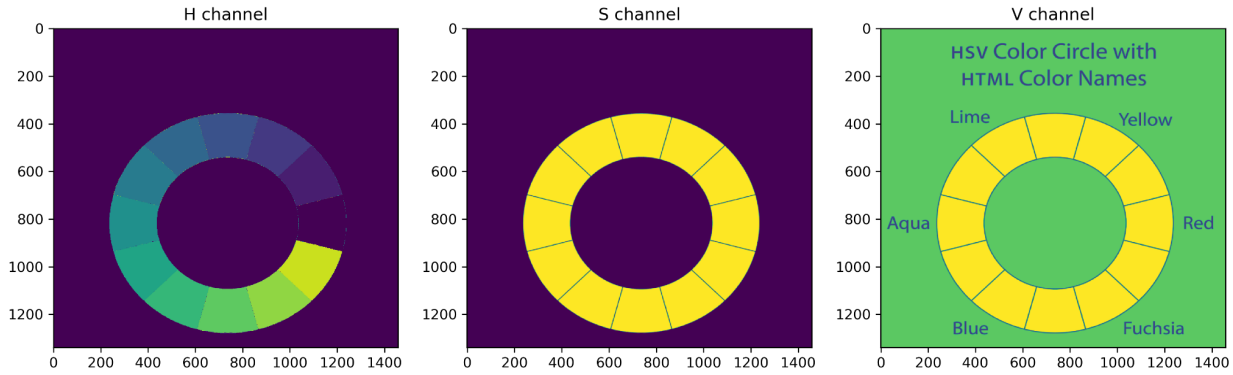

Figure 7: HSV channels for 'Figure2-a.png'

Now, to distinguish the hue channels in 'Figure2-b.png', I started by displaying the image in the hue channel. This helped me visualise the five solid colors properly (Fig8).

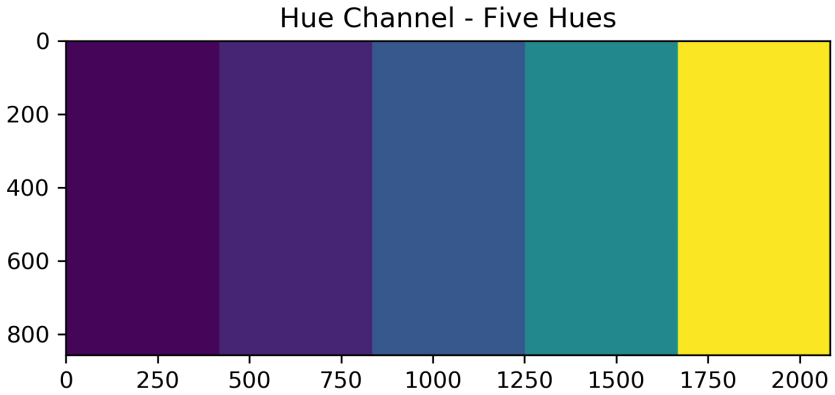

Figure 8: Hue channel of 'Figure2-b.png'

Since the colors are equidistant, I sliced the image array into five equal parts & then calculated the average hue value for each of them. I obtained the following results :

First hue 3.55
Second hue 20.53
Third hue 54.37
Fourth hue 92.76
Fifth hue 196.97

Task 4

This task is about adding gaussian noise to an image and then trying to eliminate it using a 7x7 gaussian filter operation. I started by cropping the image to focus on the facial area, and converting the same to grayscale (Fig9). Grayscale conversion makes the convolution process much faster, since there is only a single channel to be processed instead of three.

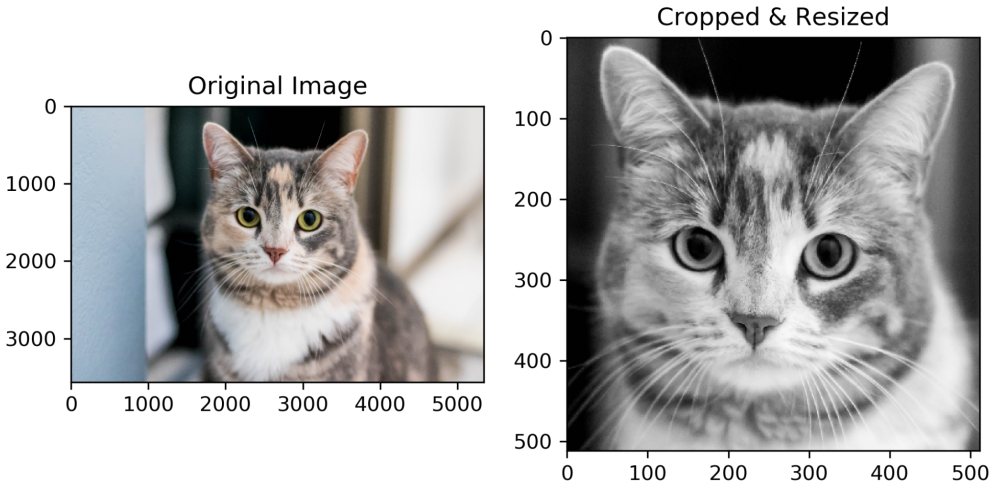

Figure 9: Original Image vs Cropped & Resized to 512x512

Now, adding gaussian noise to the image, we get the following result (Fig10). The image looks really noisy and loses quite a lot of detail & information around the face.

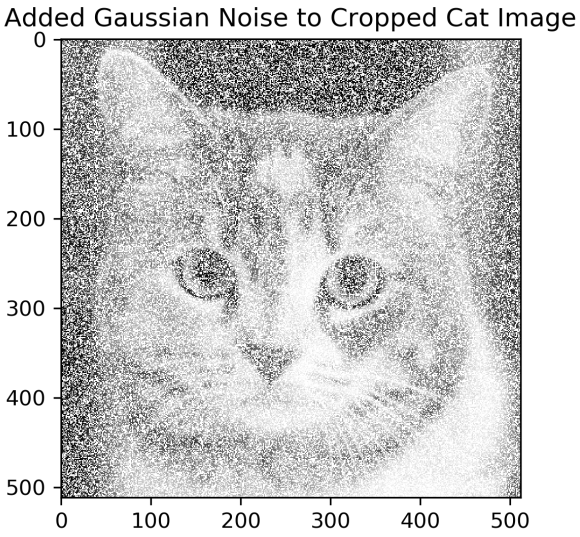

Figure 10: Gaussian noise on the cropped image

Visualising the histograms of the image before & after adding gaussian noise, we observe that the graphs are day & night apart - almost opposites of each other (Fig11).

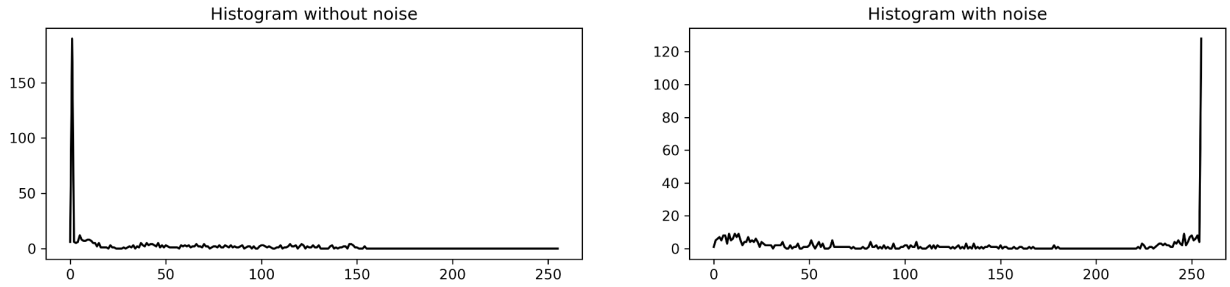

Figure 11: Histogram without noise vs Histogram with noise

Then, I implemented the gaussian filter operation by using the principle of convolution. My function 'my_Gauss_filter' works by creating an empty image with the size of the expected convolution result image. It applies the kernel all over the source image and keeps populating this empty image, producing the required result. 'getGaussianKernel' is a helper function which produces a gaussian kernel of required size & standard deviation. Applying a 7x7 gaussian filter over the noisy cat image, we observe that it can eliminate quite a lot of noise (Fig 12).

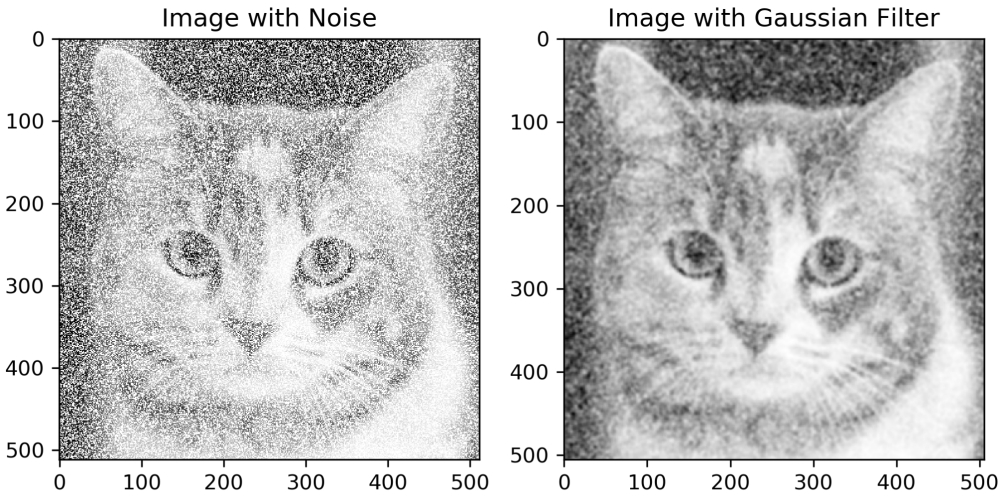

Figure 12: Image with Noise vs Image with Gaussian Filter

I did an extensive comparison of the gaussian filter's effect for different kernel & standard deviation values. First, I kept the kernel size constant at 7 and varied the standard deviation values (Fig 13). Then, I kept the standard deviation constant and varied the kernel sizes (Fig 14). From the results, it looks like the standard deviation does not make much difference after a certain point. In this case, any value more than five has no noticeable improvements or changes. On the other hand, the kernel size seems to have a greater effect with a steady blur as it increases.

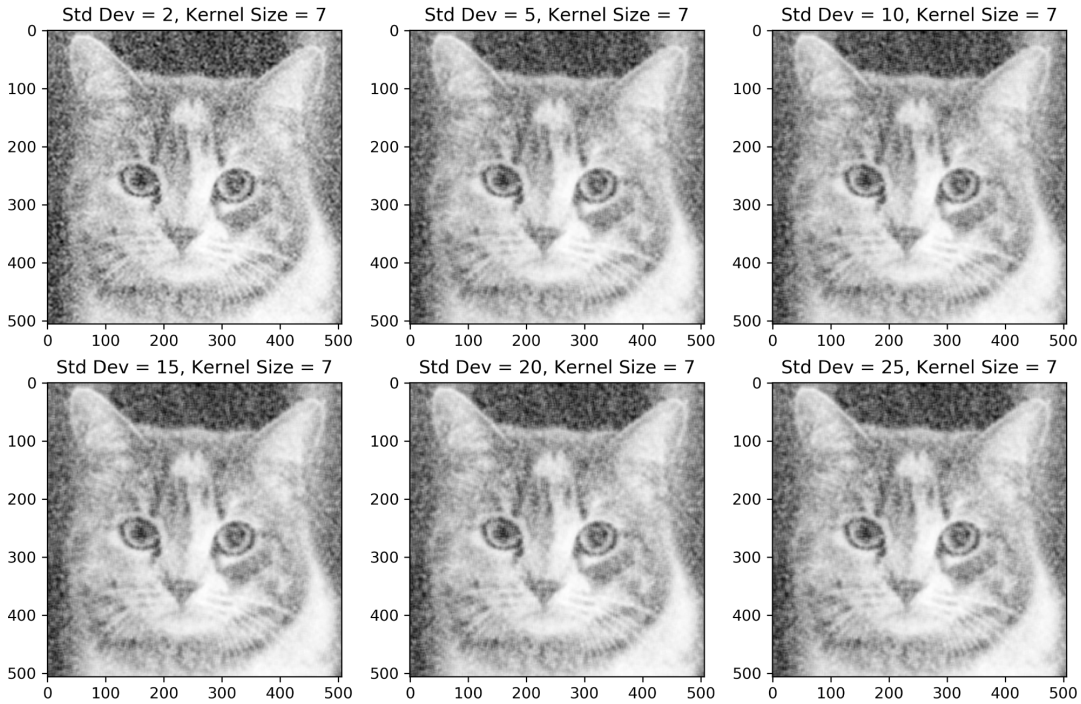

Figure 13 : Comparing different kernel sizes with constant standard deviation

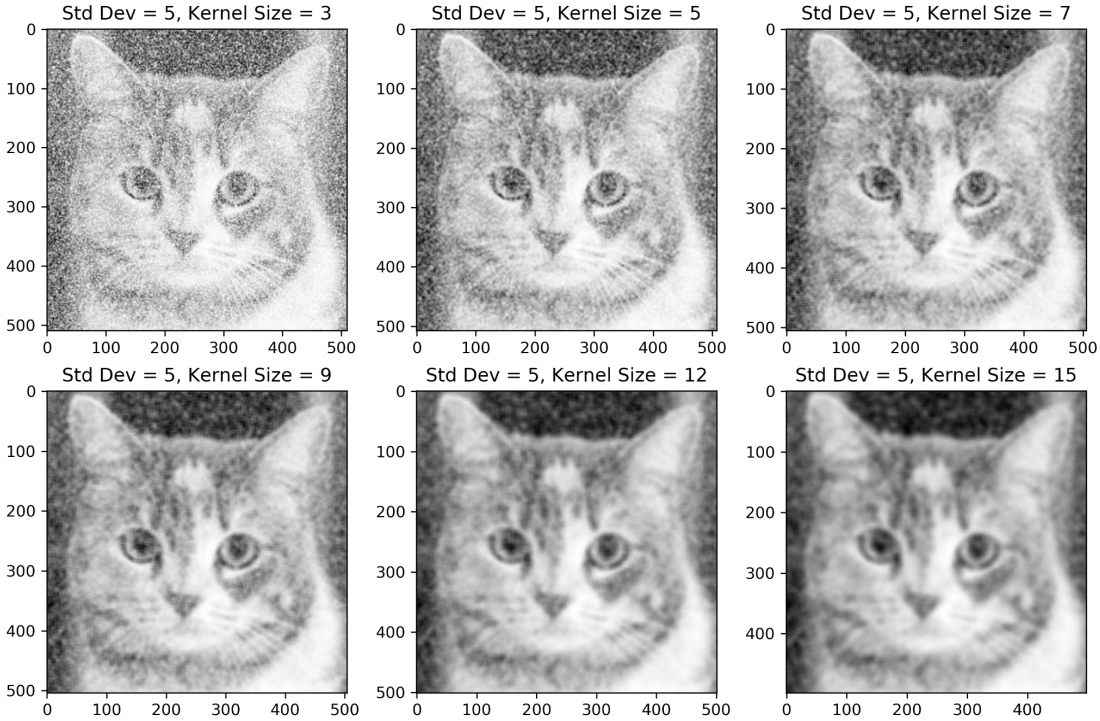

Figure 14 : Comparing different kernel sizes with constant standard deviation

Lastly, comparing my gaussian filter with the openCV gaussian filter, the result appears to be exactly the same. (Fig15)

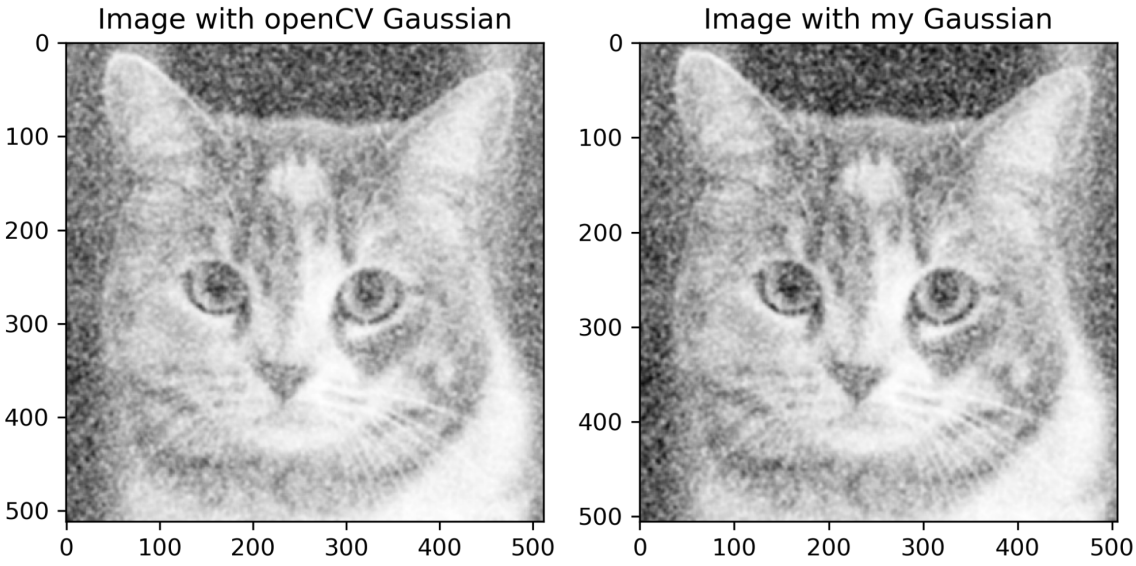

Figure 15: Comparing openCV gaussian with my gaussian

Task 5

This task is about using the 3x3 Sobel filter for edge detection. Since the convolution operation is exactly the same as Gaussian filter, I used the same ‘my_Gaussian_filter’ function and swapped out the gaussian kernel for the Sobel kernel. Testing the operation on ‘image2’ & ‘image4’, we get the following result (Fig16)

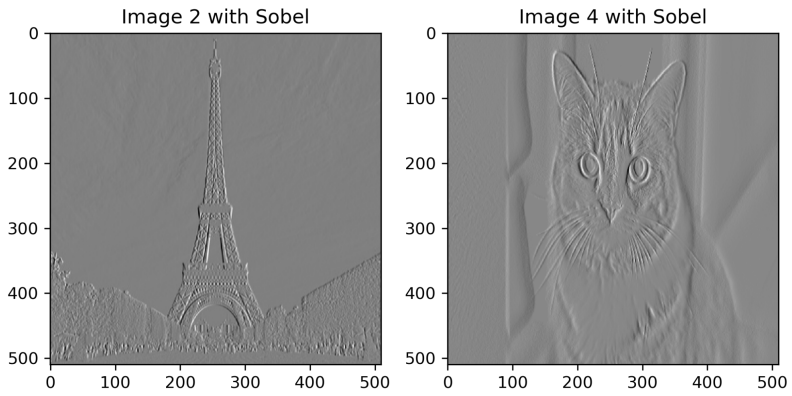

Figure 16: Image 2 & Image 4 with Sobel

As we can observe, the Sobel filter has successfully performed edge detection and we can clearly visualise the edges in the images. I also compared my filter with the built-in openCV version, and saw that the results are very similar. (Fig 17)

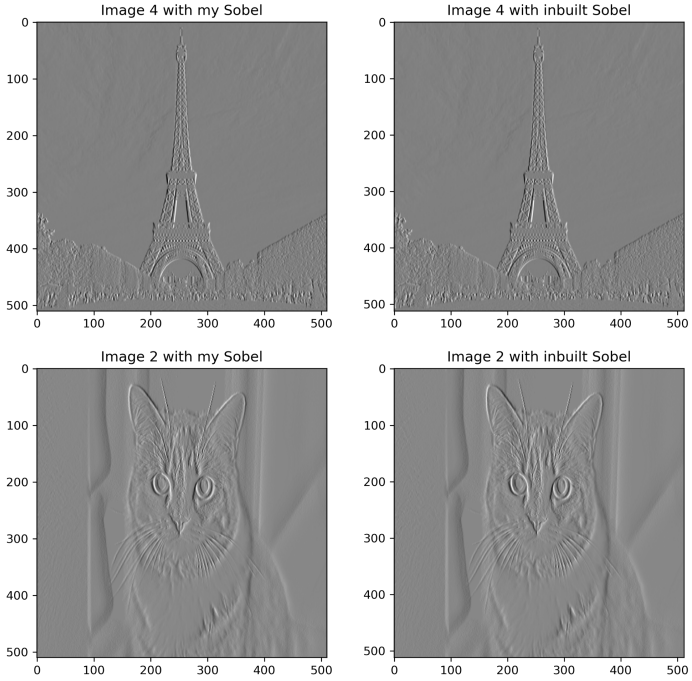

Figure 17: Comparing my Sobel with openCV Sobel

Edges in an image represent a sudden change in the intensity & value of pixels. They are regions of interest and can help analyze the structure of the image or object of attention. Sobel filter works by detecting this sudden change in intensity for each pixel of the image. It uses the first gradient and can be applied for either the x-axis or the y-axis. If the pixel has constant intensity, Sobel operation results in a zero vector. On the other hand, if the pixel has change in the intensity, the result is a vector that points in the direction of change. Thus, Sobel filter can help us observe the orientation of edges as well.

Sobel filter can be applied by using convolution operation on the image, where the kernel is

[-1, 0, +1]
[-2, 0, +2]
[-1, 0, +1]

Now, visualising the histogram of gradient orientation, we get (Fig 18)

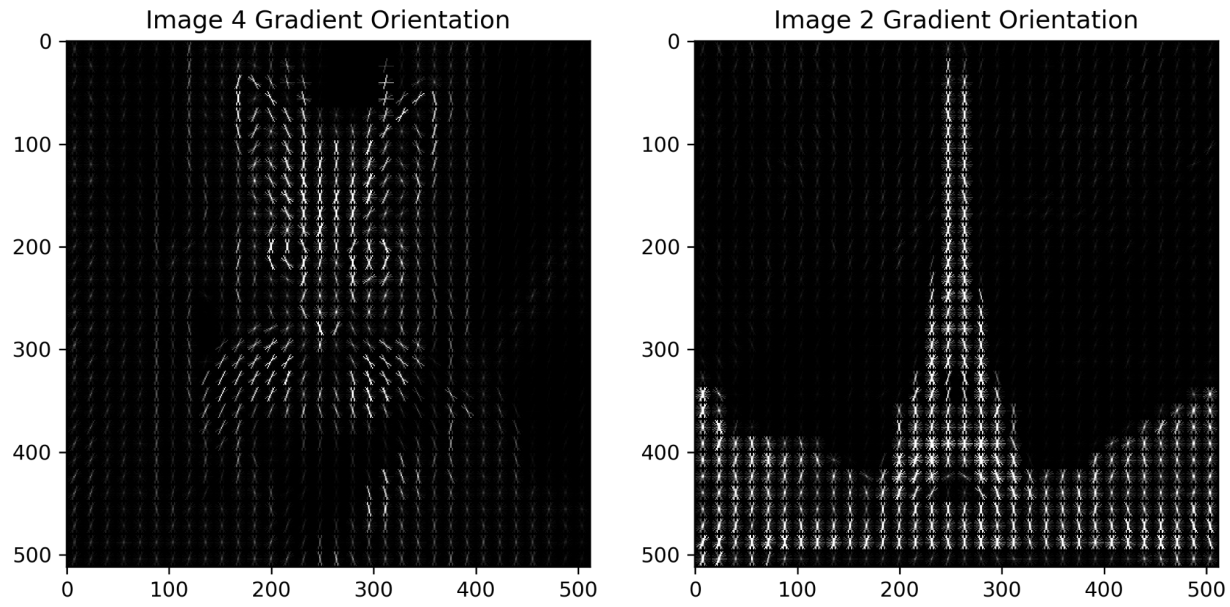

Figure 18. Histogram of Gradient Orientation

We can observe that the vectors do represent the overall direction of the edges, with good accuracy. The inaccuracy of edge orientation estimation can be caused by noise. If there is noise in the image, it will affect the ability of Sobel to detect the change in intensity. That happens because it works purely on calculating the gradient in the direction of change, and noise will abrupt this process. The workaround for using Sobel on a noisy image is to use a gaussian filter first to eliminate any possible noise. However, in this case, as our images do not have any noise - the Sobel filter works accurately & the direction of gradients is true to the actual edges.