Fundamentals Of Image Processing
Take Home Exam 1

1st Can Karagedik
Middle East Technical University
Ankara, Turkey
can.karagedik@metu.edu.tr

***Abstract*—This document is a report that explains the methodology and includes the analysis of the results about the assignment which we implementing fundamental spatial domain image enhancement techniques. We implemented these techniques using mainly trial-error methodology. We have seen how we can use image histograms for analysis to enhance images and various spatial filtering techniques to achieve desired goals for enhancement of the images.**

I. Introduction

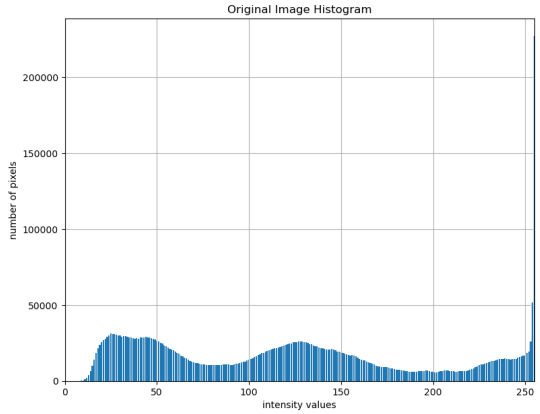
This Document has two main sections, namely Histogram Processing and Spatial Domain Image Filtering. The second section has subsections Convolution, Edge Detection and Noise Reduction.

II. Histogram Processing

A. Extracting the histogram

Given image is read using imread function of python-opencv library. Since the image provided is in gray-scale, flags parameter of imread function given as cv2.IMREAD_GRAYSCALE so that the image matrix (img) has only one channel. Then we simply loop over the matrix entries and count the number of pixel intensity values from 0 to 255 and we store them all in a list called histogram. Each value indicates the total number of pixels at that intensity (brightness) value (0,..,255).

```
img = cv2.imread(input_img_path, 0)
#0(cv2.IMREAD_GRAYSCALE) stands for GRAYSCALE

histogram  = [0]*256
for i in range(0, img.shape[0]):
        for j in range(0, img.shape[1]):
            #print("intensity value = ", img[i][j])
            histogram[img[i][j]] += 1
```



B. Generating the histogram using mixture of gaussians

In this part first we generate the probability distributions of the pixel intensity values by the given means and standard deviations using equation (1). Then by equation (2) we have the mixture of gaussians.

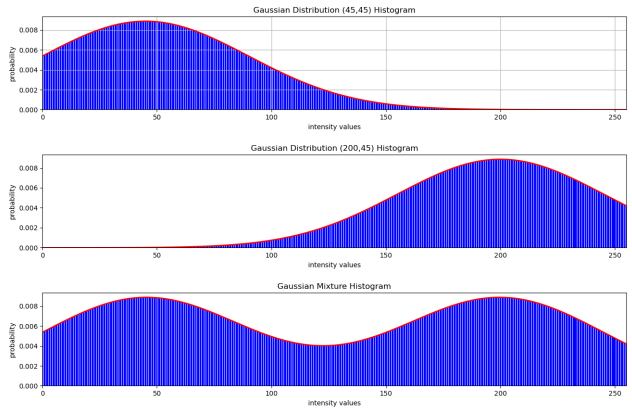
$$N(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{1}$$

$$P(x) \backsim N(\mu_1, \sigma_1^2) + N(\mu_2, \sigma_2^2) \tag{2}$$

,where x is an intensity value of the pixel, μ and σ are mean and standard deviation of the Gaussian distribution, respectively.

```
def GaussianDistribution(x, mean, standart_deviation
    ):
    return (1/(standart_deviation*math.sqrt(2*math.
        pi)))*math.exp((-1/2)*pow((x - mean)/
        standart_deviation,2))

bins = list(range(256))
gaussian_histogram_1  = [0]*256
gaussian_histogram_2  = [0]*256
gaussian_mixture = [0]*256
for x in bins:
    gaussian_histogram_1[x] = GaussianDistribution(x
        , m[0], s[0])*img_size
    gaussian_histogram_2[x] = GaussianDistribution(x
        , m[1], s[1])*img_size
    gaussian_mixture[x] = (gaussian_histogram_1[x] +
         gaussian_histogram_2[x])
```

Example:



Now we need to generate the histogram of the mixture of the gaussians. To this end we simply multiply each probability value by the number of pixels of the entire image. Since probability density function given as

$$p(r_j) = \frac{n(r_j)}{N} \tag{3}$$

,where $n(r_j)$ denotes number of pixels with the intensity value r_j and $N = \sum_{r_j \in [0,255]} n(r_j)$ denotes total number of pixels of the entire image. Thus, we get the gaussian mixture histogram.

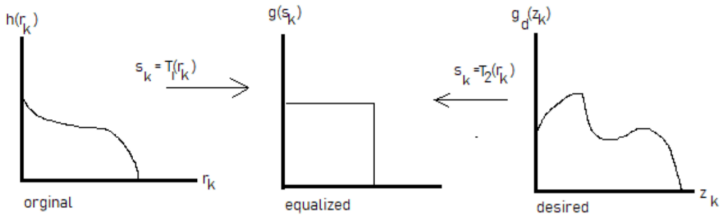
```
img_size = img.shape[0]*img.shape[1] # total number
    of pixels in the image

for x in bins:
        gaussian_histogram_1[x] =
            GaussianDistribution(x, m[0], s[0])*
            img_size
        gaussian_histogram_2[x] =
            GaussianDistribution(x, m[1], s[1])*
            img_size
        gaussian_mixture[x] = (gaussian_histogram_1[
            x] + gaussian_histogram_2[x])
```

Example:

C. Histogram matching(specification or stretching)

We want to stretch the image intensity histogram by a certain transformation so that the resulting histogram of the new image resembles the desired distribution. In this case the desired distribution is the gaussian mixture.

We will first equalize both original and specified histogram using the Histogram Equalization method as shown below.



Transformation function is corresponding cumulative histogram multiplied by some scalars as shown in equation (4). As cumulative histogram is a monotonically increasing function (guarantees one to one correspondence), we know that the transformation function is one to one, so it is invertible. Therefore by composition of first transformation with inverse of the second transformation we can get the mapping from original to specified histogram.

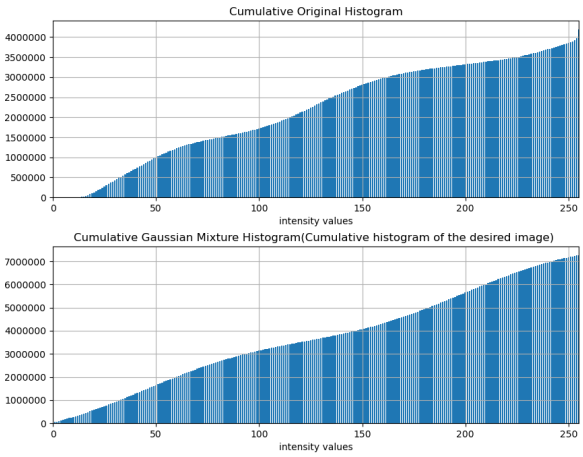
$$s_k = h_c(r_j) \times \frac{L-1}{M \times N} \tag{4}$$

,where s_k is the equalized image's color palette, $M \times N$ is the total number of pixels, L is the maximum number of colors that can be displayed at any time(since each pixel is represented by 8-bits $L = 2^8 = 256$), and $h_c(r_j)$ is the corresponding cumulative histogram according to formula given in equation (5).

$$h_c(r_j) = \sum_{j=0}^{255} h(r_j) \tag{5}$$

```
def CumulativeHistogram(histogram):
    cumulative_histogram  = [0]*256
    cumulative = 0
    i = 0

    for pixelvalue in histogram:
        cumulative += pixelvalue
        cumulative_histogram[i] = cumulative
        i += 1
    return cumulative_histogram

def HistogramEqualization(cumulative_histogram,
    image_size):
    equalized_histogram  = [0]*256
    i = 0
    for pixelvalue in cumulative_histogram:
       equalized_histogram[i] = round(((256-1)/
           image_size)*pixelvalue)
       i += 1
    return equalized_histogram
```

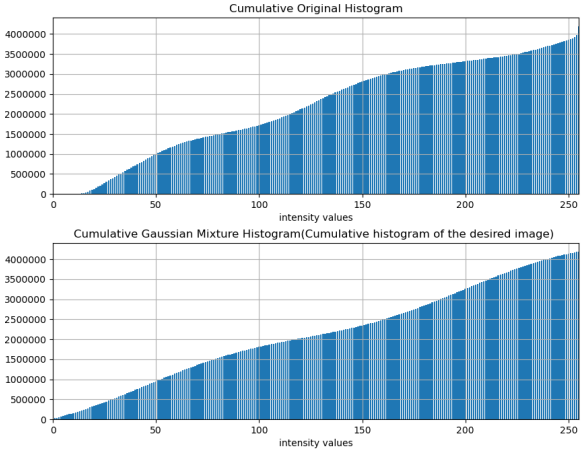
Example of cumulative histograms:



Observe that the sums of cumulative histograms are way different. We need to make them equal except the distribution of pixel values. So we multiply desired cumulative histogram with the ratio of sums.

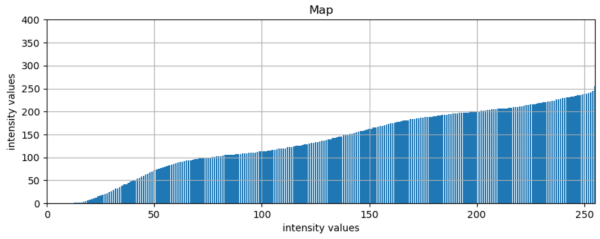
```
histogram_cumulative = CumulativeHistogram(histogram
    )
histogram_equalized = HistogramEqualization(
    histogram_cumulative,img_size)

histogram_gaussian_cumulative = CumulativeHistogram(
    gaussian_mixture)

ratio = histogram_cumulative[255] /
    histogram_gaussian_cumulative[255]

for i in range(0,256):
    histogram_gaussian_cumulative[i] =
        histogram_gaussian_cumulative[i] * ratio

histogram_gaussian_equalized = HistogramEqualization
    (histogram_gaussian_cumulative, img_size)
```

Except for the distribution of pixel values, the cumulative histograms are now identical:



Now we will create the look-up table that will represent our transformation function, which will convert the original image pixel values to the desired pixel values. In programming, look-up table is a list with length 256. Indexes from 0 to 255, each value represents mapping of intensity values.

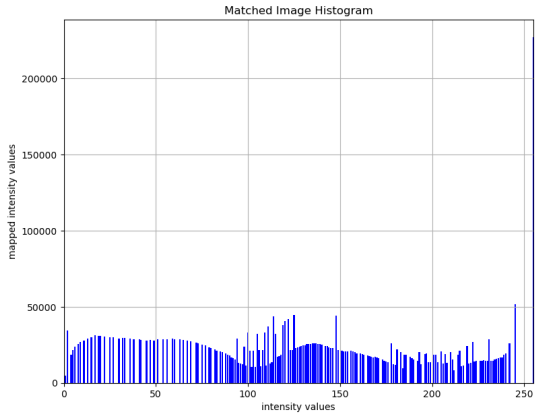
```
mapping = [0]*256

    for i in range (256):
        mapping[i]= histogram_gaussian_equalized[
            histogram_equalized[i]]
```



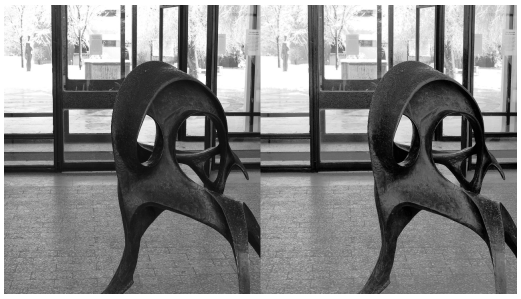
At last using this mapping we create the desired image and its histogram.

```
processed_image = np.zeros((img.shape[0],img.shape
    [1]), dtype=np.uint8)
matched_image_histogram = [0]*256

for i in range(0, img.shape[0]):

    for j in range(0, img.shape[1]):
    #print("intensity = ", img[i][j], "changed to "
        mapping[img[i][j]])
    processed_image[i][j] = mapping[img[i][j]]
    matched_image_histogram[mapping[img[i][j]]] += 1
```

Example of a matched image histogram:



End result:



III. Spatial Domain Image Filtering

A. Convolution

In this part we will implement convolution function. First we read the image in RGB format by using python-opencv library. Since the format is RGB we need to convolute each channel separately. We simply loop over R,G,B channels. Due to the nature of the convolution operation, we determine the range of starting and ending pixels to convolute(assuming the given filter is a square). Then we execute the convolution operation over the image.

```
def the1_convolution(input_img_path, filter):
    img = cv2.imread(input_img_path)
    m, n = filter.shape
    if (m == n): # check if the given filter(kernel)
         is square or not
        number_of_rows = img.shape[0]
        number_of_columns = img.shape[1]

        processed_img = np.zeros((number_of_rows,
            number_of_columns,3), np.uint8)

        for channel in range(img.shape[2]):
            for i in range((m-1)//2, number_of_rows
                - ((m-1)//2) ):
                for j in range((m-1)//2,
                    number_of_columns - (m-1)//2 ):
                processed_img[i][j][channel] =(1/9)*
                    round(np.sum(img[i-((m-1)//2):i
                    +1-((m-1)//2), j-((m-1)//2):j
                    +1-((m-1)//2), channel]*filter))
    return processed_img
```

B. Edge Detection

In this part we will implement Canny edge detection algorithm. The process of Canny edge detection algorithm can be broken down to five different steps:

- Apply Gaussian filter to smooth the image in order to remove the noise
- Find the intensity gradients of the image
- To eliminate erroneous edge detection responses, use gradient magnitude thresholding or lower bound cut-off suppression.
- To find potential edges, use a double-thresholding method.
- Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Image is read in GRAYSCALE format by using python-OpenCV library. We start with smoothing. In order to get rid of unnecessary floor pattern and shadows we applied Gaussian blur. To prevent lose of potential edges we choose Gaussian blur with size 5×5 . Smaller size filter resulted with unnecessary information and bigger filter with loss of edges. However, Gaussian blur applied by python-OpenCV function showed us much better results. So rather than convolution manually we used cv2.GaussianBlur function. But here our implementation in any case.

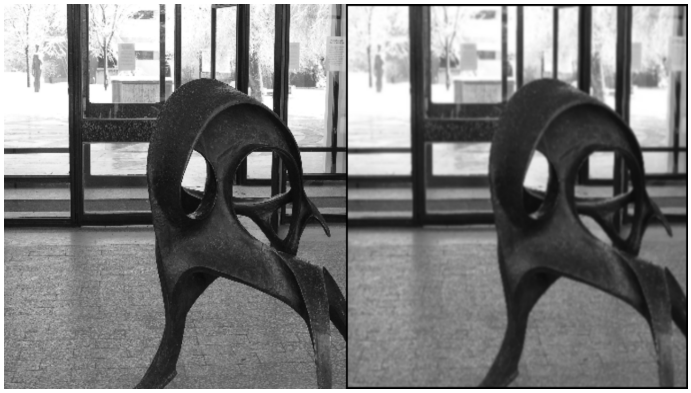
Gaussian distribution with mean $\mu = 0$ and $\sigma = 1$:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

We apply convolution with Gaussian Filter

```
def part2(input_img_path, output_path):
    img = cv2.imread(input_img_path, 0)
    number_of_rows = img.shape[0]
    number_of_columns = img.shape[1]

    GaussianBlur = np.array([[1,  4,  7,  4, 1],
                             [4, 16, 26, 16, 4],
                             [7, 26, 41, 26, 7],
                             [4, 16, 26, 16, 4],
                             [1,  4,  7,  4, 1]])
    filter =  GaussianBlur
    m, n = filter.shape

    for i in range((m-1)//2, number_of_rows - ((m-1)
        //2) ):
            for j in range((m-1)//2,
                number_of_columns - (m-1)//2 ):
                img[i][j] = (1/273)*round(np.sum
                    (img[i-2:i+3, j-2:j+3]*
                    filter))

                    OR

    img = cv2.GaussianBlur(img,(5,5),0)
```

Hence we have the smoothed image:



Now, we need to compute gradients using Sobel Operator. Sobel Operator is a specific type of 2D derivative mask which is efficient in detecting the edges in an image. We will use the following two masks:

X – Direction Kernel

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

Y – Direction Kernel

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

We again convolute the image using the both vertical and horizontal mask. Then using the equation (6) we get the gradient magnitudes. For the edge directions, we use the equation (7).

$$G = \sqrt{G_x^2 + G_y^2} \tag{6}$$

$$\theta = \arctan 2(G_y, G_x) \tag{7}$$

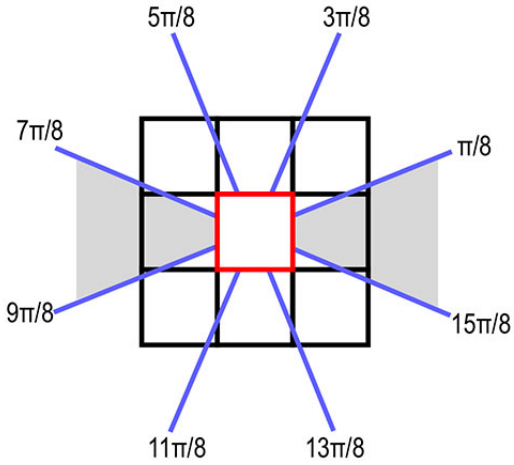
,where G_x and G_y denotes $Sobel_x \circledast Image$ and $Sobel_y \circledast Image$ respectively($\circledast$ is convolution operation), $\arctan 2$ is four-quadrant inverse tangent resulting θ in radians. After calculations we convert radians to degrees.

```
sobel_horizontal = np.array([[-1, 0, 1],
                             [-2, 0, 2],
                             [-1, 0, 1]])

sobel_vertical = np.array([[-1, -2, -1],
                           [ 0 , 0 , 0],
                           [ 1 , 2 , 1]])

gradient_magnitudes = np.zeros((number_of_rows,
    number_of_columns), np.uint8)
gradient_horizontal = np.zeros((number_of_rows,
    number_of_columns), np.uint8)
gradient_vertical = np.zeros((number_of_rows,
    number_of_columns), np.uint8)

for i in range(1, number_of_rows - 1):
    for j in range(1, number_of_columns - 1):
        gradient_vertical[i][j] = round(np.sum(img[i
            -1:i+2, j-1:j+2]*sobel_vertical))
        gradient_magnitudes[i][j] = math.sqrt(pow(
            round(np.sum(img[i-1:i+2, j-1:j+2]*
            sobel_horizontal)),2) + pow(round(np.sum
            (img[i-1:i+2, j-1:j+2]*sobel_vertical))
            ,2))

edge_directions = np.arctan2(gradient_horizontal,
    gradient_vertical)
edge_directions  = np.rad2deg(edge_directions )
```

Sobel applied to previous image:

Now, we have everything we need for Non-Maximum Suppression (edge thinning technique). Our goal is to get rid of the redundant/duplicate edges found by Sobel Edge Detection. Rather than having numerous lines for the same edge, we want simply one line to show it. The Non-Max Suppression Algorithm can help with this.

For each pixel we wii consider 8-neighborhood system, the neighboring pixels are located in horizontal, vertical, and diagonal directions (0°, 45°, 90°, and 135°). We will compare every pixel value against the two neighboring pixels in the gradient direction. If that pixel is a local maximum, it is retained as an edge pixel otherwise suppressed. This way only the largest responses will be left.




```
non_max = np.zeros((number_of_rows,
    number_of_columns), np.uint8)
for i in range(1, number_of_rows - 1):
    for j in range(1, number_of_columns - 1):
        edge_direction = edge_directions[i, j]

        #East-West(right-left) direction
        if((0 <= abs(edge_direction) < 180/8) or
            (7*180/8 < abs(edge_direction) <= 180)):
            neighbor_1 = gradient_magnitudes[i, j +
                1]
            neighbor_2 = gradient_magnitudes[i, j -
                1]
```

```
#Southeast-Northwest (bottom right-top left)
      direction
elif((180/8 <= edge_direction < 3*180/8) or
     (-7*180/8 <= edge_direction < -5*180/8))
     :
     neighbor_1 = gradient_magnitudes[i + 1,
          j - 1]
     neighbor_2 = gradient_magnitudes[i - 1,
          j + 1]

#North-South(up-down) direction
elif(3*180/8 <= abs(edge_direction) <=
     5*180/8):
     neighbor_1 = gradient_magnitudes[i + 1,
          j ]
     neighbor_2 = gradient_magnitudes[i - 1,
          j ]

#Northeast-Southwest (top right-bottom left)
      direction
elif((5*180/8 < edge_direction <= 7*180/8)
     or (-3*180/8 <= edge_direction < 180/8))
     :
     neighbor_1 = gradient_magnitudes[i + 1,
          j + 1]
     neighbor_2 = gradient_magnitudes[i - 1,
          j - 1]

if (gradient_magnitudes[i,j] >= neighbor_1 )
      and (gradient_magnitudes[i,j] >=
     neighbor_2):
     non_max[i,j] = gradient_magnitudes[i,j]
else:
     non_max[i,j] = 0
```

After Non-Max Suppression:

In an image, non-max suppression produces a more accurate representation of genuine edges. However, one can see that some of the edges are brighter than others. The brighter ones can be considered strong edges, whereas the lighter ones could be edges or noise. To address the issue of which edges are truly edges and which aren't, we apply double threshold followed by hysteresis.

Basically we will decide one low one high threshold values so that if an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. Ones gradient value lower than the low threshold will be suppressed to zero.

Since we work on a specific image, trial and error method is used to find suitable threshold values.

It is observed that picking low threshold too small(¡25) resulted with presence of non-edge pixels. Oppositely, picking low threshold too large(¿50) caused presence of false edges. Similarly, picking high threshold to small(¡125) caused false edges and picking high threshold to large(¿150) resulted with loss in real edges. Thus, by trial following values chosen: low threshold = 25, high threshold = 125.

Here are some trials:

Low threshold = 25, high threshold = 100:

Low threshold = 25, high threshold = 125

Low threshold = 25, high threshold = 200

Low threshold = 50, high threshold = 100

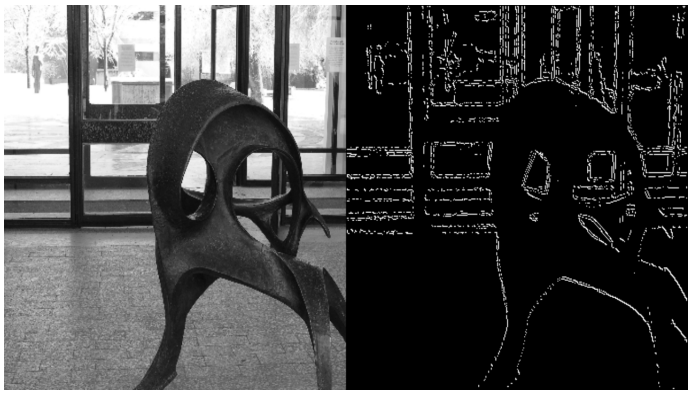
```
thresholded = np.zeros((number_of_rows,
    number_of_columns), np.uint8)
    threshold_low = 25
    threshold_high = 125

    strong_row, strong_column = np.where(non_max >=
        threshold_high )
    weak_row, weak_column = np.where( (non_max <=
        threshold_high ) & (non_max >= threshold_low
        ))
    zero_row, zero_column = np.where(non_max <
        threshold_low)

    thresholded[zero_row, zero_column] = 0
    thresholded[weak_row, weak_column] = 100
    thresholded[strong_row, strong_column] = 255
```

As planned, the strong edge pixels should be included in the final edge image because they are taken from the image's real edges. However, there will be some disagreement over the weak edge pixels, which can be retrieved from either the real edge or the noise/color variations. The weak edges should be eliminated to achieve an accurate outcome. A weak edge pixel resulting from real edges is usually related to a strong edge pixel, whereas noise responses are not. Thus blob analysis is used to track the edge connection by looking at a weak edge pixel and its eight linked neighbors. As long as the blob has at least one strong edge pixel, that weak edge point can be identified as one that should be maintained.

Hence we track edges by hysteresis.

```
for i in range(1, number_of_rows - 1):
        for j in range(1, number_of_columns - 1):
            if (thresholded[i,j] == 100):
                if (255 in [thresholded[i-1:i+2, j
                    -1:j+2].tolist()]):
                    thresholded[i, j] = 255
                else:
                    thresholded[i, j] = 0
```

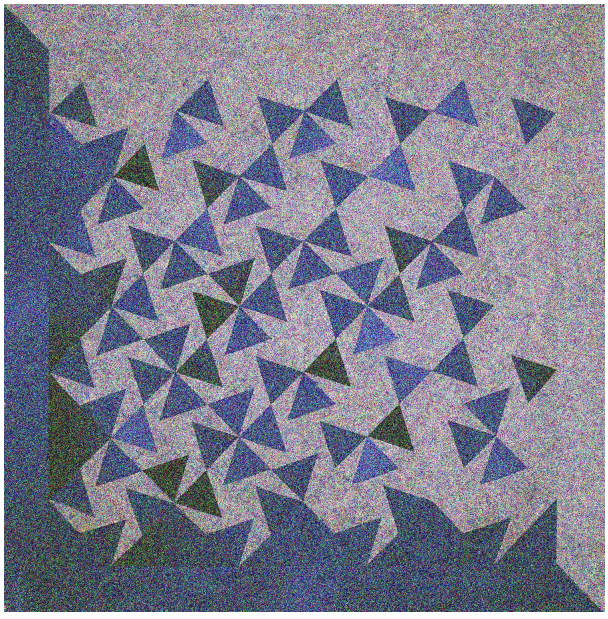
Finally we have the result:



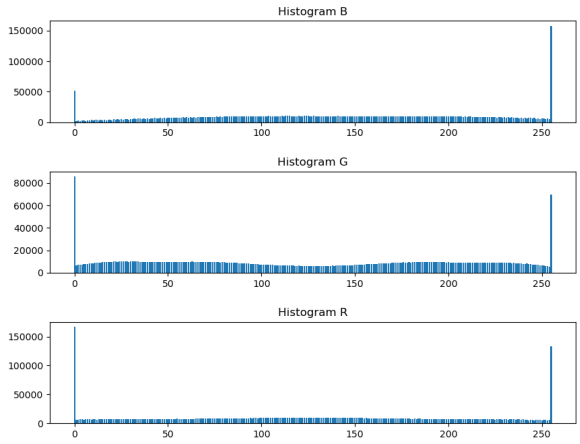
C. Noise Reduction

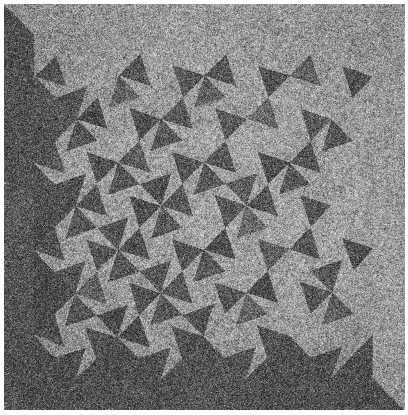
Here we are given two noisy images and we will try to enhance these images. We mainly try to improve the quality and information content of the image by trying commonly used practices such as contrast stretching, blurring, averaging and thresholding.

Given image:



First we analyze the histogram of the image

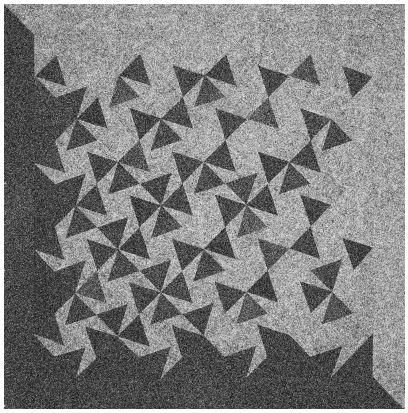




Red channel:

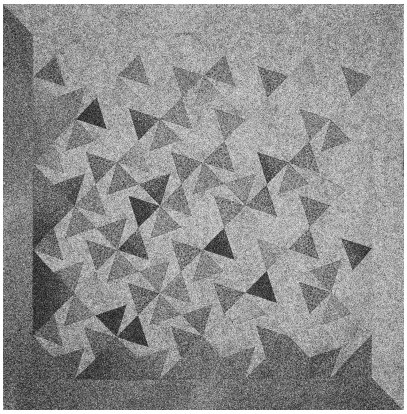
We immediately observe that in every channel most intensity values are piled up at the intensity values 0 and 255. Moreover, at 255 value Blue channel has largest number of pixels, and confirms the obvious that the most dominant channel is the Blue channel.

By observing each channel separately, we see that both green and red channel have much more contrast compared with blue channel. Since blue channel is much more dominant in the original image, we will binarize the both red and green channel but will preserve the intensity distribution of the blue channel. For better binary images we first apply averaging filter followed by median blur for smoothing. In addition, we equalize the blue channel for smooth transition between lighter and darker pixels.



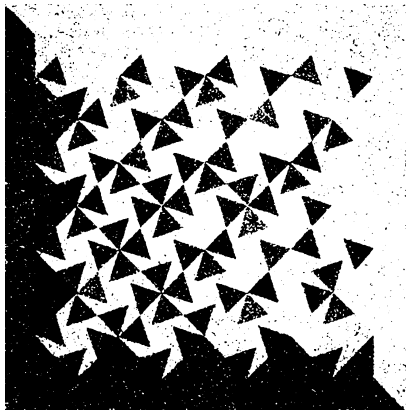
Blue channel:

```
def enhance_3(path_to_3 , output_path):
    img = cv2.imread(path_to_3)
    (B, G, R) = cv2.split(img)

    average5 = np.ones((5,5),np.float32)/25
    B = cv2.filter2D(B,-1,average5)
    B = cv2.medianBlur(B,5)
    B = cv2.equalizeHist(B)

    G = cv2.filter2D(G,-1,average5)
    G = cv2.medianBlur(G,5)
    ret,G = cv2.threshold(G,115,255,cv2.
        THRESH_BINARY)

    R = cv2.filter2D(R,-1,kernel5)
    R = cv2.medianBlur(R,5)
    ret,R = cv2.threshold(R,112,255,cv2.
        THRESH_BINARY)

    enhanced = cv2.merge((B,R,G))
    return enhanced
```

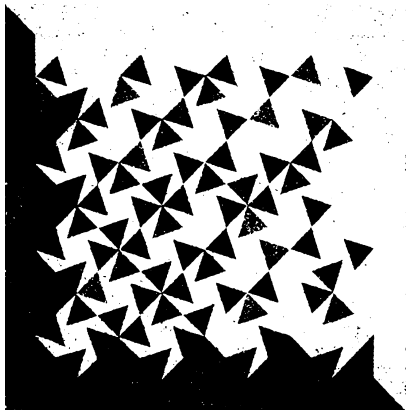


Green channel:

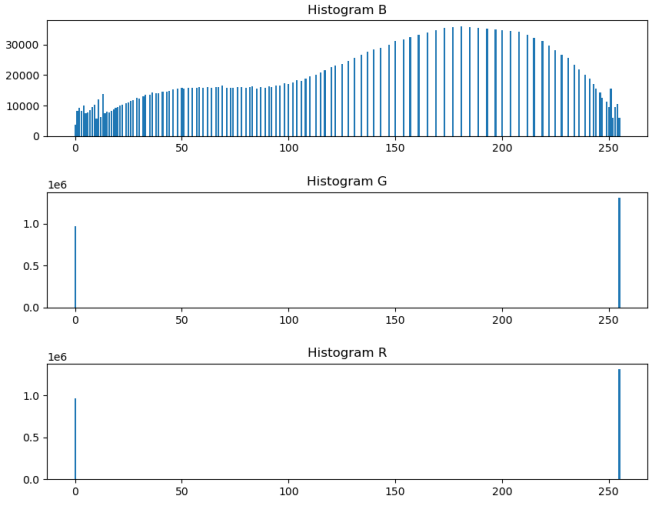
Binarized green channel:



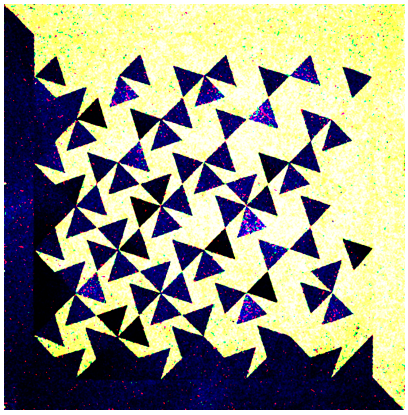
Binarized red channel:



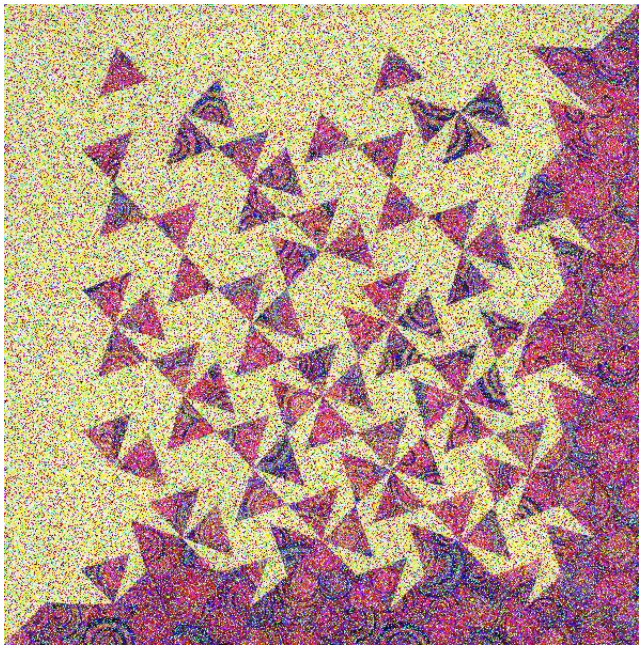
Enhanced image histogram:



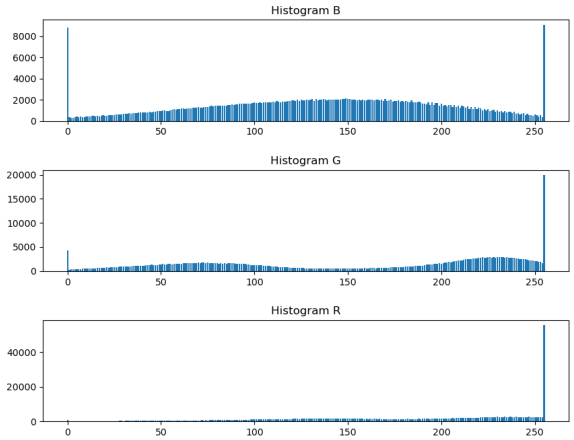
Enhanced image :



Second image:

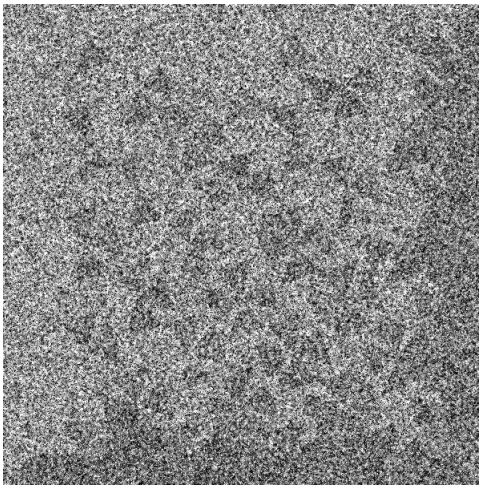


Again we analyze the histogram of the image

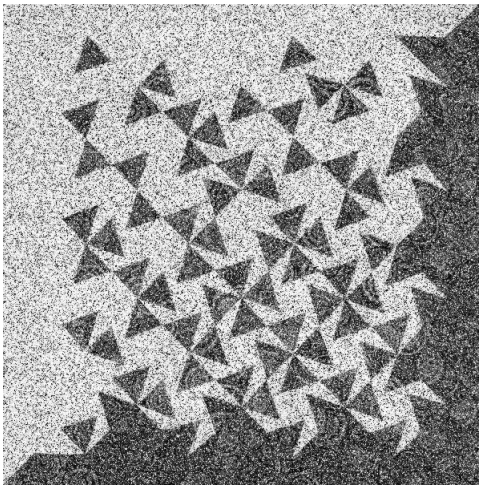


This time the dominant channel is red channel and green channel has the most contrast. By sharpening followed by

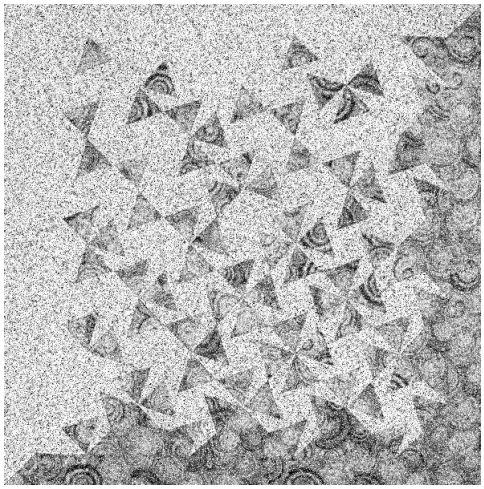
median blurring we get rid of the noise in the green channel. For the red channel, we sharpen the image to make circular pattern more obvious then apply averaging filter followed by median blurring for noise cancelling. However for the blue channel we do not much data to gather. Thus we sharpen the blue channel to get as much pattern data as possible then applied averaging filter for smoothing. Hence the enhanced image turns out with much less noise but also decrease in details on circular patterns observed in the original image. For better enhancement further analysis and methods is needed. Blue channel:



Green channel:

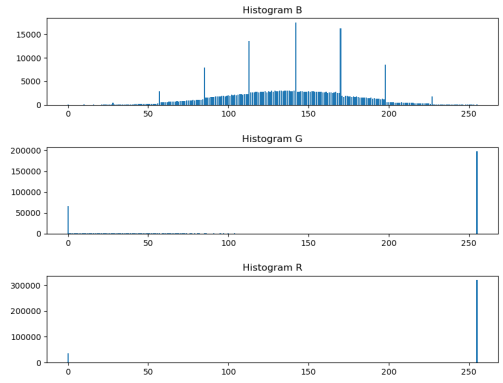


Red channel :

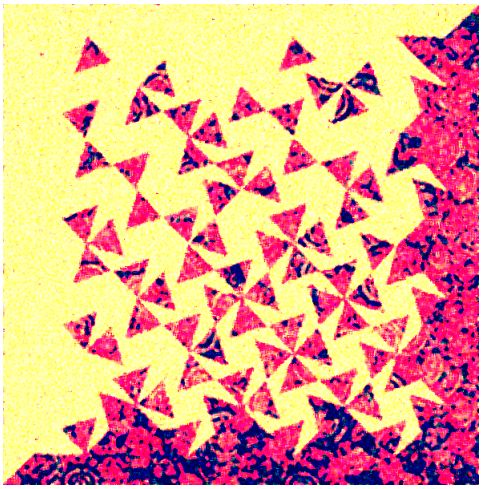


```
def enhance_4(path_to_4 , output_path):
    img = cv2.imread(path_to_4)

    (B, G, R) = cv2.split(img)
    cv2.imwrite(output_path + "B.png",B)
    cv2.imwrite(output_path + "G.png",G)
    cv2.imwrite(output_path + "R.png",R)

    average5 = np.ones((5,5),np.float32)/25
    average3 = np.ones((3,3),np.float32)/9
    sharpen = np.array([[-1,-1,-1], [-1,9,-1],
        [-1,-1,-1]])

    R = cv2.filter2D(R, -1, sharpen)
    R = cv2.filter2D(R,-1, average5)
    R = cv2.medianBlur(R,5)
    ret,R = cv2.threshold(R,115,255,cv2.
        THRESH_BINARY)

    B = cv2.filter2D(B, -1, sharpen)
    B = cv2.filter2D(B,-1, kaverage3)

    G = cv2.filter2D(G, -1, sharpen)
    G = cv2.medianBlur(G,5)

    enhanced = cv2.merge((B,G,R))
    return enhanced
```

Histogram of enhanced image:



Enhanced image :



REFERENCES

[1] Gonzalez, Rafael C.; Woods, Richard E. (2008). Digital Image Processing (3rd ed.). Prentice Hall. pp. 133–153. ISBN 9780131687288.

[2] Green, B. (2002, January 1). Canny Edge Detection Tutorial. Retrieved December 3, 2014