CENG 466 - Fundamentals of Image Processing Take Home Exam 1 Report

1st Batuhan Çalışkan
Computer Engineering
Middle East Technical University
Ankara, Turkey
batuhan.caliskan@metu.edu.tr

2nd Alişan Yıldırım
Computer Engineering
Middle East Technical University
Ankara, Turkey
alisan.yildirim@metu.edu.tr

***Abstract*—Histogram processing and spatial domain image filtering (including convolution, edge detection and noise reduction) are some basic methods in image processing. This report presents practice of these methods.**

I. Introduction

This report presents for the THE1 (Take Home Exam 1) of the course Fundamentals of Image Processing. Histogram processing part have histogram equalization and histogram specification and Spatial domain image filtering part have edge detection and noise reduction techniques.

Also, there are some code snippets that explains procedure which perform the image processing techniques.

II. Histogram Processing

Histogram matching is used for generating a mapping function from pixel values to pixel values. It takes two arguments such as original image histogram which is the target histogram and other image histogram which is going to change. Purpose of this technique is finding transition matrix from reference image to target image. The procedure for this assignment is explained in detail in this section.

Firstly, started with looking out if output path is valid. If it is not, then create directory for the output; as you can see, below code snippet is doing this assignment.

```
def MakeDir(output_path):
    path_list =output_path.split("/")
    writen_path=""
    for path in path_list:
        if path=="":
            continue
        if path=="." or path=="~":
            writen_path+=path+"/"
            continue
        writen_path+=path
        if not os.path.exists(writen_path):
            os.mkdir(writen_path)
        writen_path+="/"

def part1(input_img_path, output_path, m,s):
    if output_path[-1]!="/":
        output_path+="/"
    if not os.path.exists(output_path):
        MakeDir(output_path)
    ...
```

There are some examples images below. They will be used for testing.

Fig. 1. First original image

After creating directory, the histogram of original image was calculated. The histogram is used for representing brightness array of the image. For finding it, increments the corresponding value of pixel while program is traversing each pixels.

Fig. 3 is the original histogram of first image and Fig. 4 is the original histogram of second image.

To find original histogram of the image, firstly should create an array h with L gray values and initialize with 0 values. Then, find the histogram such as

$$h(r_k) = h(r_k) + 1$$

.

After the calculation of original image histogram, we calculated the cumilative histogram of the original image

Fig. 2. Second original image

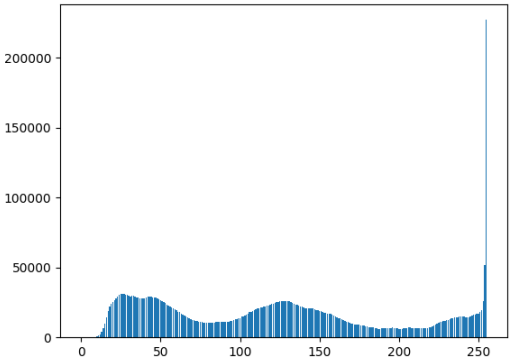

Fig. 3. Original histogram of first image

with the given formula;
$h_c(r_k) = h_c(r_k - 1) + h(r_k)$

Next, for generating the histogram, Mixture of Gaussians is used with the given means and deviations. The formula is;

$$P(X) \sim N(\mu_1, \sigma_1^2) + N(\mu_2, \sigma_2^2)$$

.

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

For applying this formula, numpy module can be used such as;

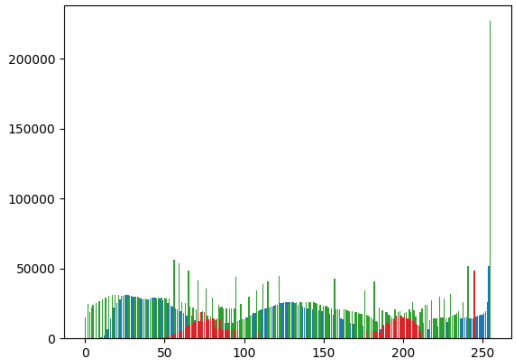

Fig. 4. Original histogram of second image

```
...
Histogram_G = np.zeros([256], np.int32)
for x in range(0, len(m)):
    Histogram_G = np.add(Transform_1, np.random.
        normal(m[x], s[x], 256))
...
```

where the m is μ and s is σ and 256 is output shape. np.random.normal draws random samples from a normal (Gaussian) distribution. Also, the probability density for the Gaussian distribution is same as the formula.

Fig. 5 is the gaussian histogram of first image and Fig. 6 is the gaussian histogram of second image.

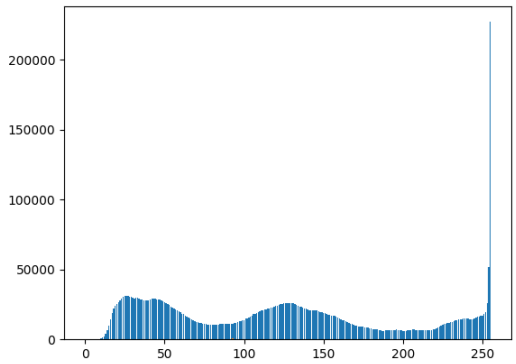

Fig. 5. Gaussian histogram of first image

Now, mapping function and cumulative histogram are needed for matching the image. To find the mapping function, values in the cumulative histogram should be multiplied by a constant.

$$c = (L-1)/(width * height)$$

$L-1$ is the number of brightness level and with this number, constant can be calculated. This calculation is for original images transform function. For the desired image' function

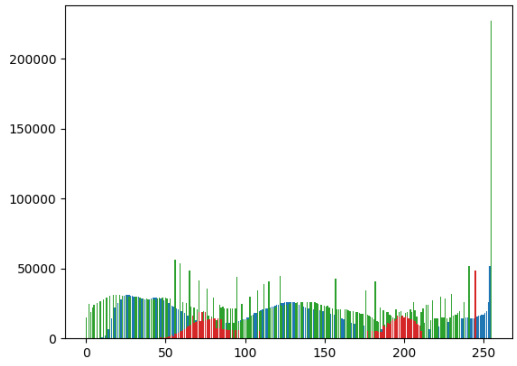

Fig. 6. Gaussian histogram of second image

we need to multiply the cumilative histogram of gaussian histogram with following value

$$c = (L-1)/(max(Histogram\_Gaussian\_Cumilative))$$

```
...
Histogram_Cumilative = np.zeros ([256], np.int32)
Histogram_Cumilative[0] = Histogram[0]
for x in range(1, 256):
        Histogram_Cumilative[x] +=
            Histogram_Cumilative[x-1] + Histogram[x
            ]

Transform_1 = np.zeros ([256], np.int32)
for x in range(0, 256):
    Transform_1[x] = round( (255 / (Image_Height *
        Image_Width) ) * Histogram_Cumilative[x])
Histogram_G_Cumilative = np.zeros([256], np.int32)
Histogram_G_Cumilative[0] = Histogram_G[0]
for x in range(1, 256):
    Histogram_G_Cumilative[x] +=
        Histogram_G_Cumilative[x - 1] +
        Histogram_G[x]

Transform_2 = np.zeros ([256], np.int32)
for x in range(0, 256):
    Transform_2[x] = round((255 / np.max(
        Histogram_G_Cumilative)) *
        Histogram_G_Cumilative[x])
...
```

Then, in order to create a matching function we have to following formula:

$$z_k = T_2^{-1}(T_1(r_k))$$

To find T_2^{-1} we need to change domains of this function, since the inverse of matrix is not applicable. To change domain we need an index finder function and the following lines provides that indexing function.

```
def FindIndex(Arr, val):
    for x in range(0, len(Arr)):
        if Arr[x] == val:
            return x
    return val

    ...
    for x in range(0, Image_Height):
        for y in range(0, Image_Width):
            Input_Image[x][y] = FindIndex(Transform_2,
                Transform_1[Input_Image[x][y ][0]])
    ...
```

As outputs, Fig. 7 and Fig. 8 are matched images for first and second images.

Fig. 7. Matched image of first image

To calculate matched image histogram, same thing which has been done at the beginning should be done. Firstly should create an array h with L gray values and initialize with 0 values. Then, increment by one corresponding pixel while going through of them:

```
...
Histogram = np.zeros ([256], np.int32)
for x in range(0, Image_Height):
    for y in range(0, Image_Width):
        Histogram[Input_Image[x][y ][0]] += 1
...
```

Finally, Fig. 9 and Fig. 10 are outputs of matched image histograms of first and second images.

To conclude, it is crystal clear histogram equalization increases the contrast of the image. While it affects some images hugely, also it can affect some images in a little way.

Fig. 8. Matched image of second image

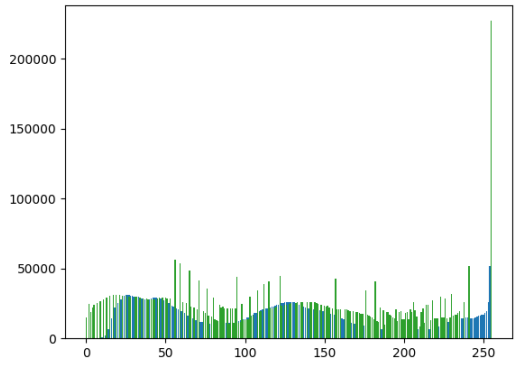

Fig. 9. Matched image histogram of first image

As can be seen, Fig. 7 was affected just little while Fig. 8 changed completely. Because of using reference image, Fig. 8 was produced better.

III. Spatial Domain Image Filtering

Spatial filtering is a technique which changes the brightness value of a pixel, depending on its position.

A. Convolution

Our convolution function first reads the input image then calls the our separate convolution function that takes image and filter. Function traverses the image and then apply filter

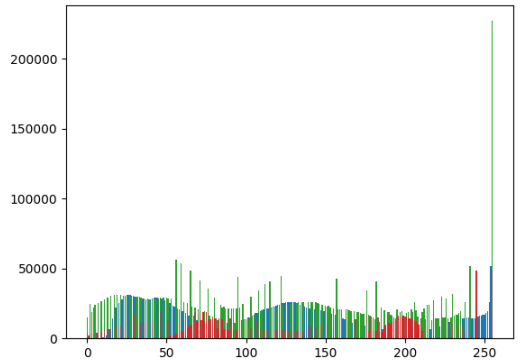

Fig. 10. Matched image histogram of second image

to the current pixel. Important point is that we take the pixels that are not in the image array as 0 value. Our filter variable should be 2n+1 x 2n+1 in order to calculate the filter correctly.

B. Edge Detection

To create a proper edge filter algorithm, first we need to blur image a little bit. This process will help to smooth edge transitions and will help the reduce noise in the image. To blur image we have used the numpy. Following line is the Blur function that we have used;

```
Smoothed_Image = cv.blur(Input_Image, (15, 15))
```

Sobel filters combine the edge maps, obtained at the output of gradient filters in vertical and horizontal directions to define the magnitude and directions of edge pixels. Because of the definition of it, it is suitable for this purpose, edge detection.

At first we have tried only 2 filters which are horizontal and vertical. In result we have get edges at only 2 directions. Thus we have used the 4 sobel filter in order to get edges at the 4 direction.

$$S_{east} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad S_{north} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$S_{west} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S_{south} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Code snippet can be written as:

```
...
Edge_Image_x_1 = Convolution(Smoothed_Image, [[−1,
    0, 1], [−2, 0, 2], [−1, 0, 1]], 0)
Edge_Image_y_1 = Convolution(Smoothed_Image, [[ 1,
    2, 1], [ 0, 0, 0], [−1, −2, −1]], 0)
Edge_Image_x_2 = Convolution(Smoothed_Image, [[1,
    0, −1], [2, 0, −2], [1, 0, −1]], 0)
Edge_Image_y_2 = Convolution(Smoothed_Image, [[
    −1, −2, −1], [ 0, 0, 0], [1, 2, 1]], 0)
...
```

After the filtering process, we have add the above results to create output image. Then we have clamped the output image' intensity values with the following function.

```
def EdgeClamper(Input_Image, val):
Image_Height = Input_Image.shape[0]
Image_Width = Input_Image.shape[1]

for x in range(0, Image_Width):
    for y in range(0, Image_Height):
        if Input_Image[y][x] < val:
            Input_Image[y][x] = 0
        else:
            Input_Image[y][x] = 255
return Input_Image
```

This function takes the input image and for each pixel, if piksel is greater than value makes it 255 and less than value makes it 0. This function is necessary to see the convinient result.

After the clamp we have get the images at Figure 11 and Figure 12 which are the edges of the images at Figure 1 and Figure 2 repectively.

Fig. 11. Edge detection of first image

To conclude Getting all edges is difficult proccess and none of the algorithms in sector provides general solution for edge detection. We have used the easy yet the powerfull method sobel filtering to get edge filtered image.

C. Noise Reduction

In both functions we have used the past mkdir function to create output path if it doesn't exist.

1) Enhance3: In order to create a proper noise reduction filter we first used the avarage filtering with equal weights. The image still had readable noise. Thus, we put some weight to the center piksels and make the avaraging filter 5x5. Our filter is

$$1/25 * \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 3 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Fig. 12. Edge detection of second image

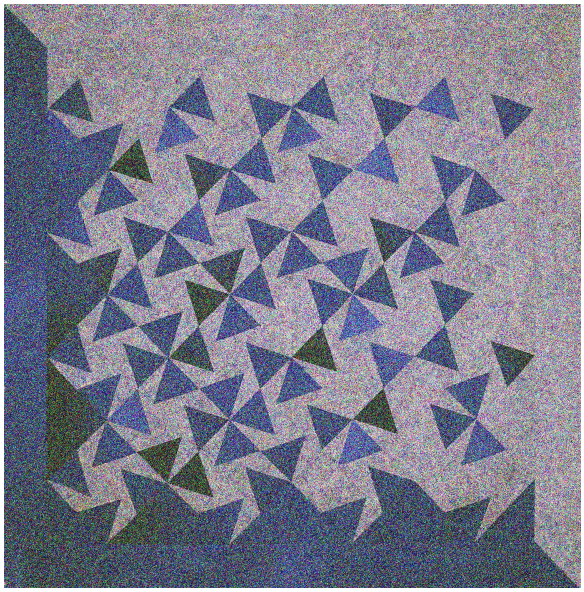

Fig. 13. Third original image

We have applied this filter with our convolution function for each R, G and B channel, then we have created the output image with mixing these channels. Then we have get the image at the Figure 14.

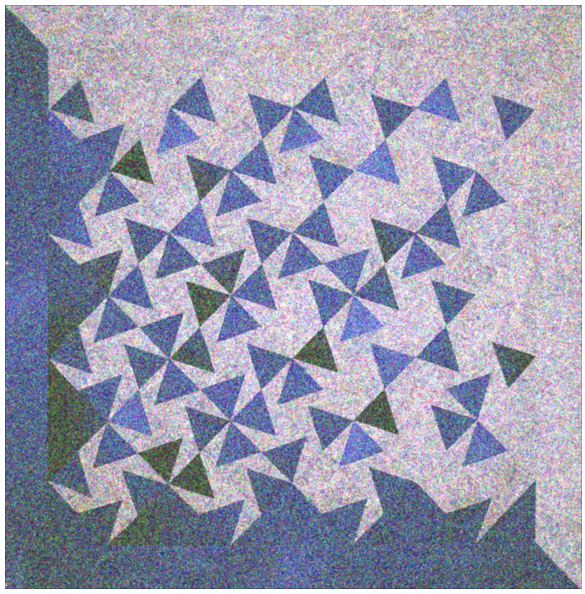

Fig. 14. Enhance with convolution of third image

To conclude Avaraging is not the best solution but it reduces the noise a lot. Because of the avaraging filter we get the piksel values at the neighbour pixels. This results in brighter image than usual.

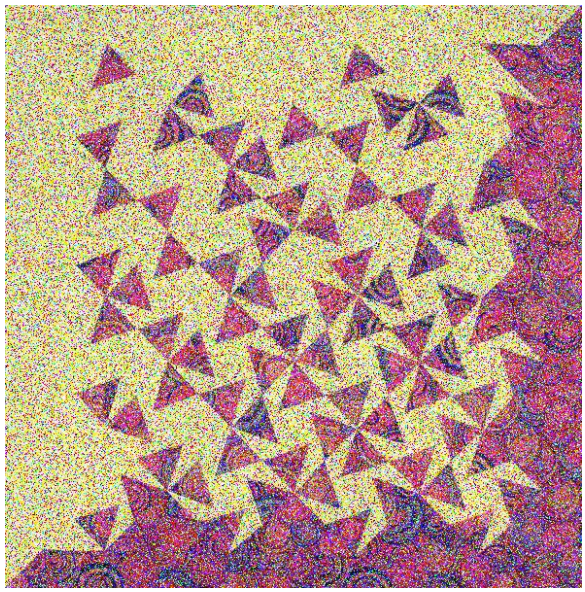

Fig. 15. Fourth original image

2) Enhance4: First we have tried the first filter that we have used in enchance3 function. The following result is the mean(avaraging) filter result of the 4th image(Fig. 15).

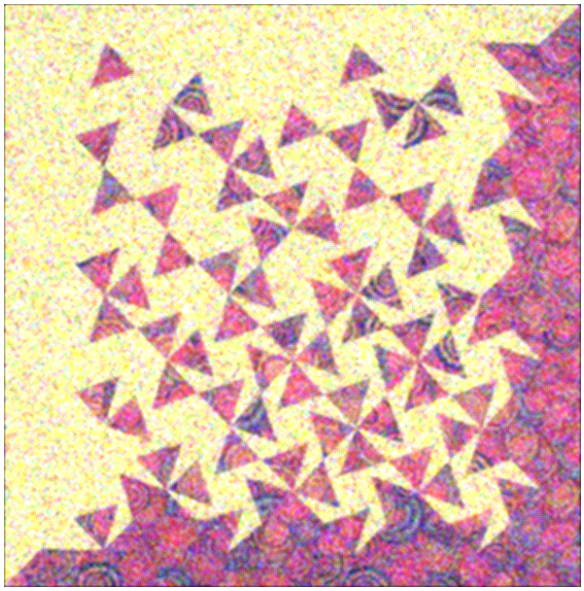

Fig. 16. Enhance with avaraging filter of fourth image

This image(Fig. 16) is still noisy and a little bright than original. Thus we have tried another filtering method which is median filtering. Median filter creates a data array which is calculated by taking the neighbour pixels of center pixel. Then we sorted this array and used np.median() function to calculate median. These lines are corresponds the median filtering function.

```
def MedianFilter(Input_Image, size, color):
    Image_Height = Input_Image.shape[0]
    Image_Width = Input_Image.shape[1]

    New_Image = np.zeros((Image_Height, Image_Width),
        np.int32)

    Arr = np.zeros([size*size], np.int32)
    for y in range(0, Image_Height):
        for x in range(0, Image_Width):
            for i in range(0, size):
                for j in range(0, size):
                    if not ((x + (i - size//2)) < 0 or (
                        x + (i - size//2)) > (
                        Image_Width - 1) or (y + (j -
                        size//2)) < 0 or (y + (j - size
                        //2)) > (Image_Height -1)):
                        Arr[j*size + i] = (Input_Image[
                            y + (j - size//2)][x + (i -
                            size//2)][color])
            Arr.sort()
            New_Image[y][x] = np.median(Arr)
    return New_Image
```

The Figure 17 is the result of the median filtered image4. As we have seen the median filter have performed a better result in this image.

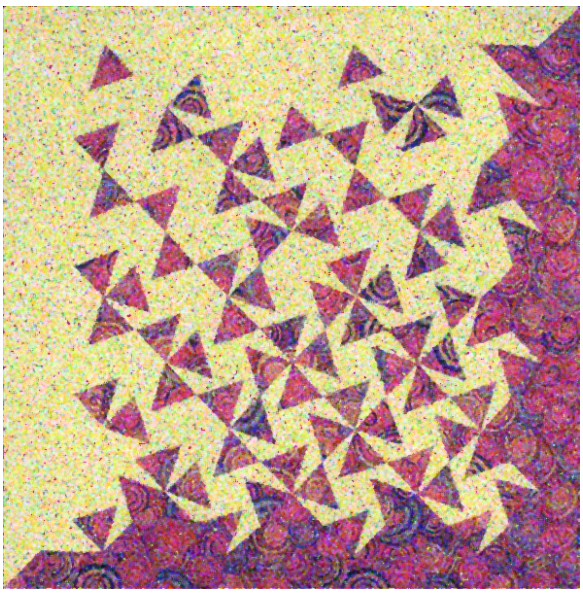

Fig. 17. Enhance with median filter of fourth image

To conclude Avaraging filter works noisy and brighter because of the contrast of the image. So we have tried the median filter. Median filter handles the contrast by considering only one pixel value for a pixel. Thus the result image is less noisy and less brighter than original.