CENG 466 - Fundamentals of Image Processing Take Home Exam 2 Report

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

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

***Abstract*—Frequency domain image filtering (including edge detection and noise reduction) and JPEG image compression are some basic methods in image processing. This report presents practice of these methods.**

I. Introduction

This report presents for the THE2 (Take Home Exam 2) of the course Fundamentals of Image Processing.Frequency domain image filtering part have edge detection and noise reduction techniques and JPEG image compression part has compression and decompression methods.

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

II. Frequency Domain Image Filtering

Frequency domain image filtering enables us to represent images in terms of a bunch of oscillations in different frequencies.

A. Edge Detection

From the lecture notes, high pass filters pass only the high frequency components of the image, outside a circle centered at the origin of $F(u, v)$. High frequency components represent the edges of an image . Thus, high pass filters work as edge detectors, so we need to use high pass filters. High pass and low pass filters are complement of each other. Following equation shows the relation between them;

$$H_{HP}(u, v) = 1 - H_{LP}(u, v)$$

There are original images as you can see at Fig1 and Fig2.

There are gaussian high pass edge detected images for first and second images you can see at Fig3 and Fig4.

As talked before, we used gaussian high pass filter and created it with this code snippet:

```
...
def gauss_2D(sigma, mu, height, width):
    x, y = np.meshgrid(np.linspace(-1, 1, height), np.
        linspace(-1, 1, width))
    d = np.sqrt(x * x + y * y)
    g = np.exp(-((d - mu) ** 2 / (2.0 * sigma ** 2)))
    return g
...
```

Fig. 1. First original image

```
    gaussian_array = gauss_2D(.5, 0, Image_Height,
        Image_Width)
    gaussian_array = np.subtract(np.full((Image_Width,
        Image_Height), 1), gaussian_array)
    gaussian_array *= np.full((Image_Width,
        Image_Height), 255)
    gaussian_array = np.abs(gaussian_array)
...
```

After we created high pass filter from low pass filter using the equation, took discrete fourier transformation of input image and shifted it. Then, we multiplied it with tranpose of gaussian array, therefore inverse discrete fourier tranformation and inverse shifting was applied. There are edge detected image for first and second images from THE1 as you can see at Fig5 and Fig6.

If we compare both spatial domain and frequency domain edge detection, we can clearly see the outputs of them different. Edges can sharply seen in the outputs of The1 better than The1 outputs but The2 outputs edges are more detailed than

Fig. 2. Second original image

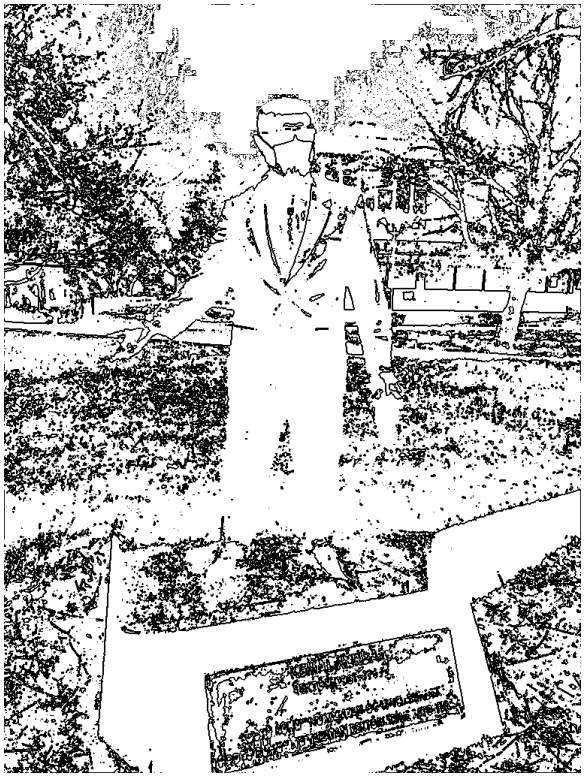

Fig. 4. Edge detection of second image with gaussian high pass filter

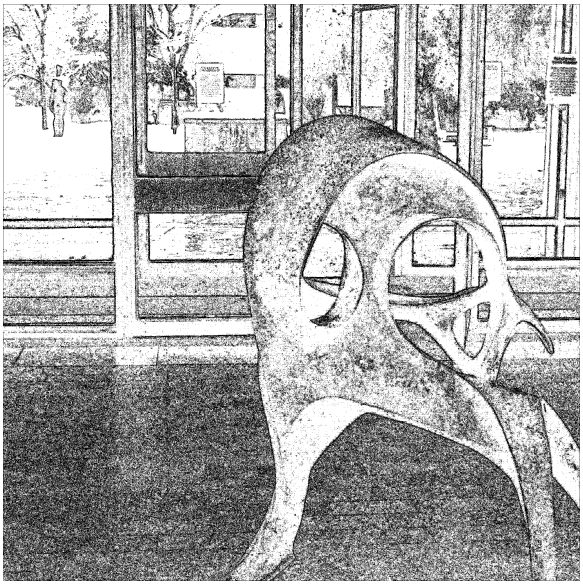

Fig. 3. Edge detection of first image with gaussian high pass filter

Fig. 5. Edge detection of first image in THE1

The1 outputs edges. The reason of this situation is that edge clamper is used in The1. With edge clamper, we set color pixel 0 and 255, if specific value is bigger or lesser than the image pixel. Therefore, with high pass filter, program compiles much faster than convolution method with sobel filters because with convolution method in The1 there are four for loops, so The1 program works slower than The2 program.

B. Noise Reduction

From the lecture notes, low pass filters pass only the low frequency components of the image, within a circle, around the origin of $F(u, v)$. Low frequency components represent the smooth regions. Thus, low pass filters work as smoothing filters, which suppresses the edges, yielding a blurred image at the output of the filter. For both Enhance3 and Enhance4

Fig. 6. Edge detection of second image in THE1

we used gaussian and ideal low pass filters. And, we would apply both of the images Fig7 and Fig8.

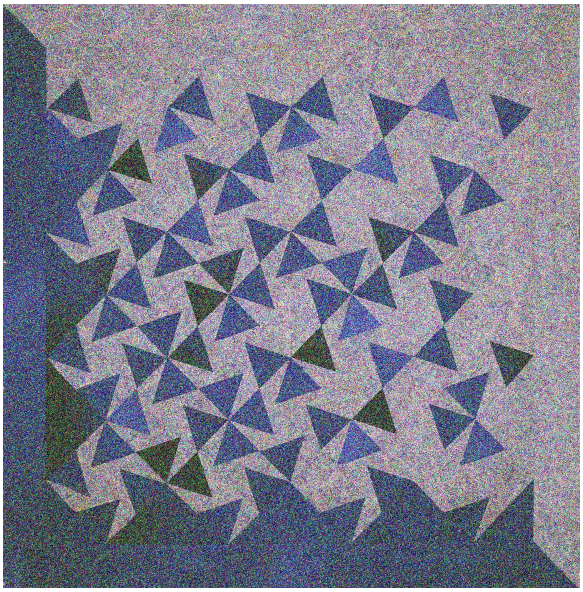

Fig. 7. Third original image

1) Enhance3: For Enhance3, ideal low pass filter is commented. Also like edge detection, same steps were applied for noise reduction parts. The only difference is;

```
...
# low_pass = ideal_lowpass(Image_Height,
      Image_Width, .25)
Red_Channel = Input_Image[:, :, 0]
Green_Channel = Input_Image[:, :, 1]
Blue_Channel = Input_Image[:, :, 2]
...
    spectrum = np.fft.fftshift(np.fft.fft2(Red_Channel)
        )
    spectrum *= gaussian_array_3d[:, :, 0]
    # spectrum *= low_pass
    img_back = np.fft.ifft2(np.fft.ifftshift(spectrum))
    img_r = np.real(img_back)
...
```

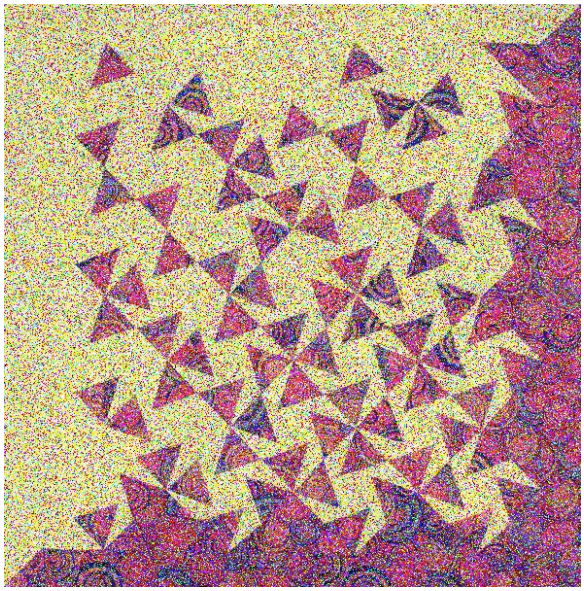

Fig. 8. Fourth original image

We examine all color channel and did not do discrete transform but only shift, filter and inverse shift. You can see the output for third image on Fig9 and for fourth image on Fig10.

2) Enhance4: For Enhance4, gaussian low pass filter is commented. Also like enhance3, same steps were applied for noise reduction parts. The only difference is;

```
...
def ideal_lowpass(height, width, r):
    arr = np.zeros((height, width))
    for h in range(height):
        for w in range(width):
            h_dif = abs(h - height / 2)
            w_dif = abs(w - width / 2)
            ratio = (h_dif ** 2 + w_dif ** 2) ** .5 /
                height
            if ratio < r:
                arr[h][w] = 1
    return arr
...
```

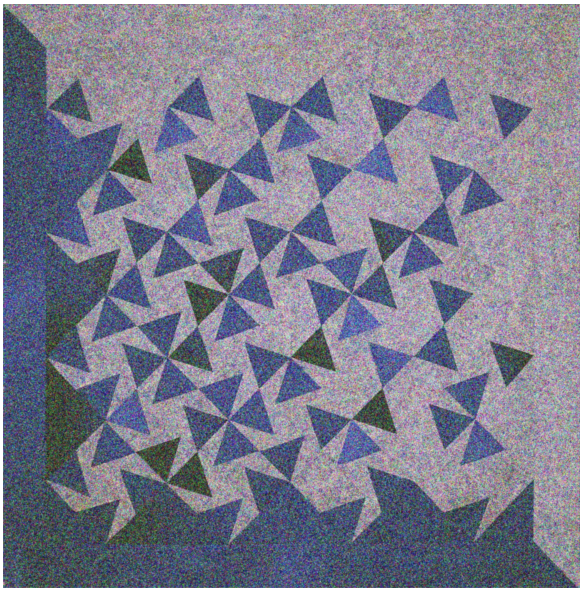

Fig. 9. Noise reduction of third image with gaussian low pass filter

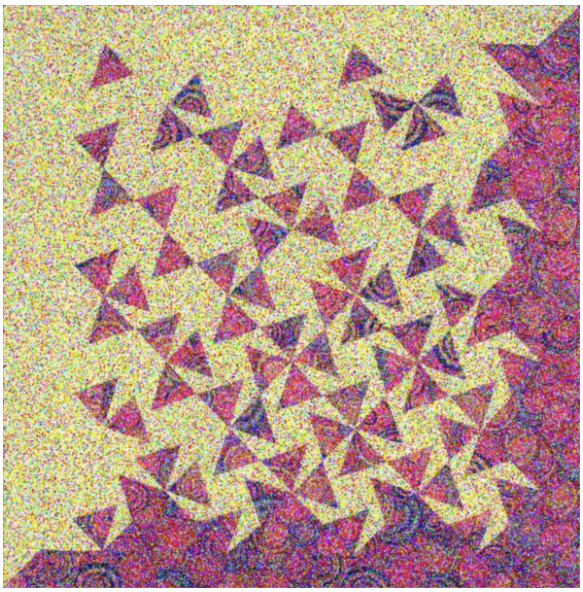

Fig. 10. Noise reduction of fourth image with gaussian low pass filter

```
low_pass = ideal_lowpass(Image_Height, Image_Width,
    .25)

Red_Channel = Input_Image[:, :, 0]
Green_Channel = Input_Image[:, :, 1]
Blue_Channel = Input_Image[:, :, 2]

spectrum = np.fft.fftshift(np.fft.fft2(Red_Channel)
    )
# spectrum *= gaussian_array_3d [:, :, 0]
spectrum *= low_pass
img_back = np.fft.ifft2(np.fft.ifftshift(spectrum))
img_r = np.real(img_back)
```

...

Function $ideal_lowpass()$ is used for creating ideal low pass filter. You can see the output for third image on Fig11 and for fourth image on Fig12.

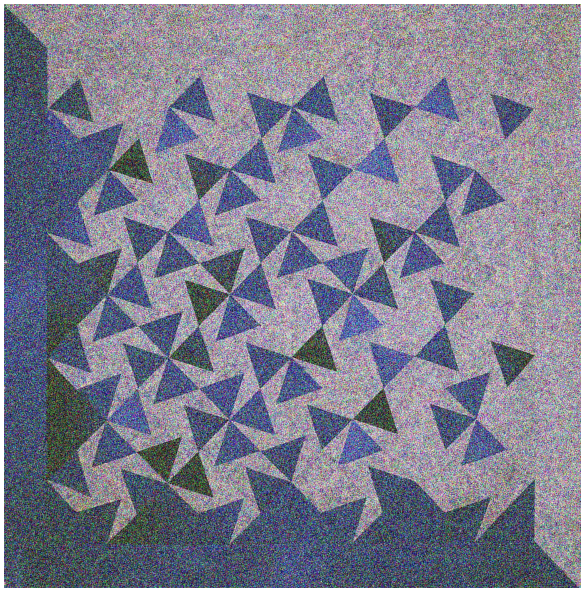

Fig. 11. Noise reduction of third image with ideal low pass filter

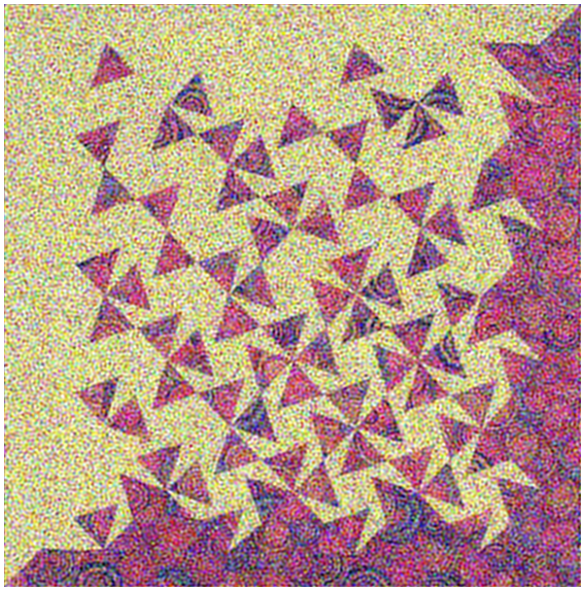

Fig. 12. Noise reduction of fourth image with ideal low pass filter

There are noise reduction images for first and second images from THE1 as you can see at Fig13, Fig14 and Fig15.

First of all for The2 outputs, as we expected, there are some non-smooth irregularities due to the effect of the discontinuity in the ideal low pass filter, whereas, Gaussian low pass filters have a smoother output. If we compare both noise reduction of The1 and The2 outputs, we can see there are some differences

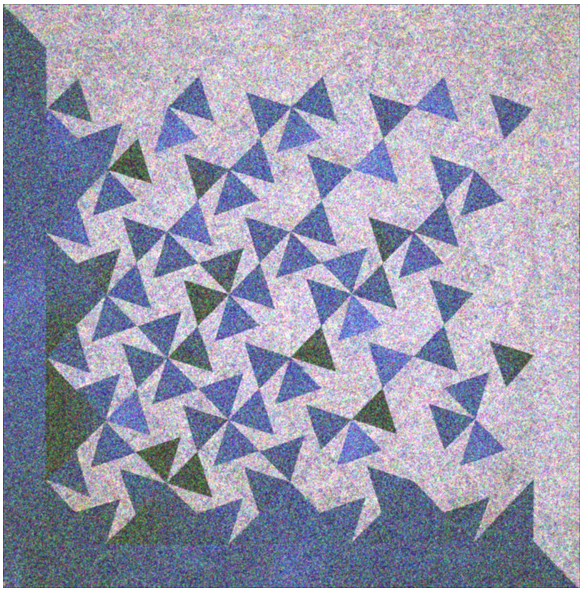

Fig. 13. Noise reduction with averaging filter of third image

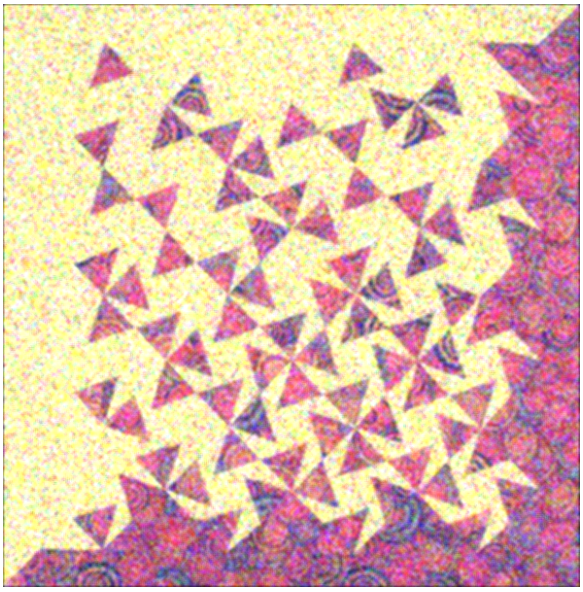

Fig. 14. Noise reduction with averaging filter of fourth image

between them. For third image, The1 output is noiser than The2 gaussian low filter output but less noiser than The2 ideal low filter output. Also, The1 output is the brightest one. For fourth image, The1 outputs are brighter than The2 outputs, but they are less noiser than The2 outputs. Because of the averaging filter, we get the pixel values at the neighbour pixels,so this results in brighter image than The2 outputs.

III. JPEG Image Compression

A. Write

JPEG is one of the lossy image compression methods. It uses several different techniques to compress images such as

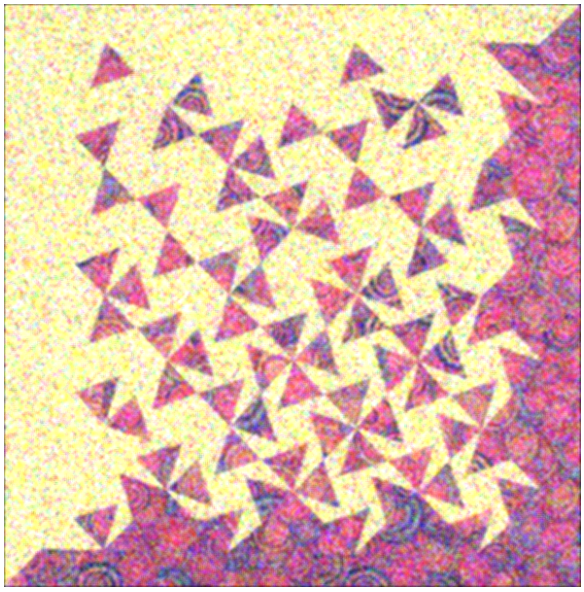

Fig. 15. Noise reduction with median filter of third image

downsampling, YCbCr color space, Discrete Cosine Transform, and Huffman Coding. To compress an image to jpeg, there are several steps.

First we change color space to YCbCr space.

```
img_ycbcr = np.array(Image.open(input_img_path).
    convert('YCbCr'))
img_y = img_ycbcr[:, :, 0]
img_Cb = img_ycbcr[:, :, 1]
img_Cr = img_ycbcr[:, :, 2]
```

YCbCr color space is used in order to separate intensity with colors. The first channel which is the Y channel is used to represent luminance, the second channel is Chromatic blue, and the third Channel is Chromatic red. Chromatic blue and red channels can be downsampled because there is no such difference at near colors in chromatic color space. To downsample an image following function is used.

```
def down_sampler(Image):
    kernel = np.ones((4, 4))
    convolved = convolve2d(kernel, Image, mode='
        valid')
    a_downsampled = convolved[::4, ::4] / 16
    return a_downsampled
```

This function uses convolution to downsample the image. Convolution is done with classic avaraging filter with all values of one.

Second, we need to transform the image's luminance channel to the frequency domain with Discrete Cosine Transform. But first, we need to divide the image into 8x8 blocks. Then for each block, Discrete Cosine Transform is applied to the block. So 64 different cosine waves are used for each block to obtain separately encoded blocks. But to apply DCT, we first need to shift all pixel values of the block by -128 in order to

center values around 0. This helper function does the shifting job.

```
def block_shifter(Arr, sign):
    for x in range(8):
        for y in range(8):
            Arr[x][y] += sign * 128.0
    return Arr
```

This function takes the block and the sign of the shift value then shifts all the pixels.

Then Discrete cosine transform is applied by cv.dct() function from OpenCV library. After the DCT we divide the block to the quantization table in order to get rid of not that important values of DCT. Then we zigzag the 2D array in order to get a correctly ordered One-Dimensional array. The following code part is the zigzag function.

```
def zigzag(input):
    # initializing the variables
    # ----------------------------------
    h = 0
    v = 0

    vmin = 0
    hmin = 0

    vmax = input.shape[0]
    hmax = input.shape[1]

    # print(vmax ,hmax )

    i = 0

    output = np.zeros((vmax * hmax))
    # ----------------------------------

    while ((v < vmax) and (h < hmax)):

        if ((h + v) % 2) == 0:  # going up

            if (v == vmin):
                # print(1)
                output[i] = input[v, h]  # if we
                    got to the first line

                if (h == hmax):
                    v = v + 1
                else:
                    h = h + 1

                i = i + 1

            elif ((h == hmax - 1) and (v < vmax)):
                # if we got to the last column
                # print(2)
                output[i] = input[v, h]
                v = v + 1
                i = i + 1

            elif ((v > vmin) and (h < hmax - 1)):
                # all other cases
                # print(3)
                output[i] = input[v, h]
                v = v - 1
                h = h + 1
                i = i + 1


        else:  # going down

            if ((v == vmax - 1) and (h <= hmax -
                1)): # if we got to the last line
                # print(4)
                output[i] = input[v, h]
                h = h + 1
                i = i + 1

            elif (h == hmin):  # if we got to the
                first column
                # print(5)
                output[i] = input[v, h]

                if (v == vmax - 1):
                    h = h + 1
                else:
                    v = v + 1

                i = i + 1

            elif ((v < vmax - 1) and (h > hmin)):
                # all other cases
                # print(6)
                output[i] = input[v, h]
                v = v + 1
                h = h - 1
                i = i + 1

            if ((v == vmax - 1) and (h == hmax - 1)):
                # bottom right element
                # print(7)
                output[i] = input[v, h]
                break

        # print('v:',v,',  h:',h,',  i:', i)
        return output
```

This function takes 2D array and returns 1D zigzagged array .

Third, we need to apply Huffman coding. Dahuffman library is used for this job. First, it takes the codec data which is all values in the block. Then we encode it with Huffman coding with respect to this codec. After creating the codec and the

encoded block, we save them in order to put them in the jpeg class.

Lastly jpeg object is created. Following class shows the jpeg class and its values.

```
class JPEG:
    def __init__(self, h, w, im_h, im_w):
        self.block_codecs = np.empty((h, w), dtype=
            object)
        self.encoded_blocks = np.empty((h, w), dtype
            =object)
        self.downSampled_Cr = np.zeros((im_h//4,
            im_w//4))
        self.downSampled_Cb = np.zeros((im_h//4,
            im_w//4))
```

After saving all the values such as codec list, encoded block list and Cr-Cb downsampled images we save this class to the disk with the pickle library. Txt file is used in order to store bytes. At last, the saved file name and path returned. The following lines show the whole code.

```
def the2_write(input_img_path, output_path):
    if not os.path.exists(output_path):
        os.mkdir(output_path)

    # Image in YCbCr space
    img_ycbcr = np.array(Image.open(input_img_path).
        convert('YCbCr'))
    Image_Height = img_ycbcr.shape[0]
    Image_Width = img_ycbcr.shape[1]

    img_y = img_ycbcr[:, :, 0]
    img_Cb = img_ycbcr[:, :, 1]
    img_Cr = img_ycbcr[:, :, 2]

    ds_img_Cb = down_sampler(img_Cb)
    ds_img_Cr = down_sampler(img_Cr)

    height_block_count = Image_Height // 8
    width_block_count = Image_Width // 8

    block_codecs = np.empty((height_block_count,
        width_block_count), dtype=object)
    encoded_blocks = np.empty((height_block_count,
        width_block_count), dtype=object)

    for h in range(height_block_count):
        h_start = h * 8
        h_end = h_start + 8
        for w in range(width_block_count):
            w_start = w * 8
            w_end = w_start + 8

            block_y = np.float32(img_y[h_start:
                h_end, w_start:w_end])

            block_y_shift = block_shifter(block_y,
                -1)

            dct_block_y = cv.dct(block_y_shift)

            quantized_y = dct_block_y //
                quantization_table

            ordered_dct_block_y = zigzag(quantized_y
                ).astype(int)

            codec = HuffmanCodec.from_data(
                ordered_dct_block_y.flatten())
            block_codecs[h][w] = codec

            encoded = codec.encode(
                ordered_dct_block_y)
            encoded_blocks[h][w] = encoded

    jpeg = JPEG(height_block_count,
        width_block_count, Image_Height,
        Image_Width)
    jpeg.block_codecs = block_codecs
    jpeg.encoded_blocks = encoded_blocks
    jpeg.downSampled_Cb = ds_img_Cb
    jpeg.downSampled_Cr = ds_img_Cr

    file = open(output_path + "jpeg.txt", 'wb')
    pickle.dump(jpeg, file)
    file.close()

    return output_path + "jpeg.txt"
```

Then to get the image back from the jpeg binary txt file we need to decompress it.

B. Read

First, read the file and fill the jpeg class variables and local function variables. Then create the output image and divide its Y channel into 8x8 blocks. Now doing all the steps in the compressing function reversed gives the original image. For each block get the block's decoded value from its encoded value with respect to its codec. After that inverse zigzag is uset to get 8x8 block from 1D array. Then multiplying block with the quantization table gets the dct value back. After Inverse DCT, values are shifted by 128 in order to get the non-0-centered pixel values. With all these steps we obtain the luminance (Y) channel of the image back.

Second, we upsampled the Chromatic red and blue channel with the helper upsampler function. After that to make the image in RGB space the following converter function is used.

```
def ycbcr2rgb(im):
    xform = np.array([[1, 0, 1.402], [1, -0.34414,
        -.71414], [1, 1.772, 0]])
    rgb = im.astype(float)
    rgb[:,:,[1,2]] -= 128
```

```
    rgb = rgb.dot(xform.T)
    np.putmask(rgb, rgb > 255, 255)
    np.putmask(rgb, rgb < 0, 0)
    return np.uint8(rgb)
```

Finnaly the RGB image is obtained. The following code shows the whole decompression function.

```
def the2_read(input_img_path):
    file = open(input_img_path, 'rb')
    data_pickle = file.read()
    file.close()
    jpeg = pickle.loads(data_pickle)

    block_codecs = jpeg.block_codecs
    encoded_blocks = jpeg.encoded_blocks

    height_block_count = block_codecs.shape[0]
    width_block_count = block_codecs.shape[1]

    Output_Image = np.zeros((height_block_count * 8,
        width_block_count * 8, 3))

    for h in range(height_block_count):
        h_start = h * 8
        h_end = h_start + 8
        for w in range(width_block_count):
            w_start = w * 8
            w_end = w_start + 8

            decoded = block_codecs[h][w].decode(
                encoded_blocks[h][w])

            quantized = inverse_zigzag(decoded, 8,
                8)

            dct_block_y = quantized *
                quantization_table

            block_y = np.array(cv.idct(dct_block_y))
                .astype(int)

            block_y_reshifted = block_shifter(
                block_y, 1)

            for x in range(8):
                for y in range(8):
                    Output_Image[h_start + x][
                        w_start + y][0] = min(
                        block_y_reshifted[x][y],
                        255)

    up_sampled_Cb = up_sampler(jpeg.
        downSampled_Cb, height_block_count * 2,
        width_block_count * 2)
    up_sampled_Cr = up_sampler(jpeg.
        downSampled_Cr, height_block_count * 2,
        width_block_count * 2)

    for x in range(height_block_count * 8):
        for y in range(width_block_count * 8):
            Output_Image[x][y][1] = up_sampled_Cb[
                x][y]
            Output_Image[x][y][2] = up_sampled_Cr[x
                ][y]

    Output_Image_RGB = ycbcr2rgb(Output_Image)

    original_size = height_block_count *
        width_block_count * 64 * 24

    compressed_size = os.stat(input_img_path).
        st_size

    print(original_size, compressed_size,
        original_size / compressed_size)

    plt.imshow(Output_Image_RGB)
    plt.show()
```

Fig. 16. 5.png

C. Summary

Our JPEG compression code is tested on 5.png(Fig. 16). Original image size is 75497472 bytes and the compressed size is 41311796 bytes. There is 1.82 compression ratio.

We have tried the default quantization table which is:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 10 & 51 & 61 \\ 12 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 13 & 16 & 24 & 40 & 57 & 59 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 103 & 92 \\ 49 & 64 & 78 & 77 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 109 \end{bmatrix}$$

But with this matrix loss ratio was too high and image was distrupted. Then we have created our quantizationt table

which is:

$$\begin{bmatrix} 1 & 1 & 1 & 2 & 4 & 6 & 10 & 15 \\ 1 & 1 & 1 & 2 & 4 & 6 & 10 & 15 \\ 1 & 1 & 1 & 2 & 4 & 6 & 10 & 15 \\ 2 & 2 & 2 & 2 & 4 & 6 & 10 & 15 \\ 4 & 4 & 4 & 4 & 4 & 6 & 10 & 15 \\ 6 & 6 & 6 & 6 & 6 & 6 & 10 & 15 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 15 \\ 15 & 15 & 15 & 15 & 15 & 15 & 15 & 15 \end{bmatrix}$$

By using our algorithm and this quantization table we obtain the image(Fig. 17).

Fig. 17. Result.png

As it can be seen on the output(Fig. 17.), the blue parts near the edges of the image are more denser. This is because of the down and upsampling of the Chromatic blue and red spaces. Moreover, there are some decrease in color resolution in the image. That is because of the Discrete Cosine Transform and truncations of the lowest frequency values.