Fundamentals Of Image Processing
Take Home Exam 2

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 frequency domain filtering techniques and JPEG image compression.**

I. Introduction

This Document has two main sections. First section is Frequency Domain Image Filtering with subsections: Edge Detection and Noise Reduction. Second section is JPEG Image Compression.

II. Frequency Domain Image Filtering

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the Fourier or frequency domain.

2-D discrete Fourier transform (DFT):

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-2j\pi(\frac{ux}{M}+\frac{vy}{N})}$$

u $\in \{0,1,2,...,M-1\}, v \in \{0,1,2,....,N-1\}$(1)

,where F(u,v) is the Fourier Transformed image (image spectrum) and f(x,y) is a digital image of size M × N.

Given the transform F(u,v), we can obtain f(x,y) by using the inverse discrete Fourier transform (IDFT):

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{2j\pi(\frac{ux}{M}+\frac{vy}{N})}$$

x $\in \{0,1,2,...,M-1\}, y \in \{0,1,2,....,N-1\}$(2)

To calculate the DFT of an image of M x N, it would need to perform MN (multiplications) x MN (additions) = $O(M^2N^2)$ operations. It is obvious that calculating the DFT of even a 2000x2000 pixel image directly takes to much time to complete.

Fortunately, the Fast Fourier Transform (FFT) algorithm for computing the DFT was developed. As the name implies, the FFT is an algorithm that determines DFT of an input significantly faster than computing it directly. Using recursive approach, the FFT reduces the time complexity of 2D-DFT from $O(N^2M^2)$ to $0(NMlog(NM))$ where NxM is the size of image.

To compare complexity of these two algorithm, let us suppose it took 1 nanosecond to perform one operation on a computer, then it would take the FFT algorithm approximately 30 seconds to compute the DFT for a problem of size $NM = 10^9$. In contrast, the direct algorithm would need several decades ($N^2M^2 = 10^{18}nanoseconds = 31.68years$).

It is obvious that The FFT algorithm is significantly faster than the direct implementation. However, it still lags behind the numpy implementation by quite a bit. One reason for this is the fact that the numpy implementation uses matrix operations to calculate the Fourier Transforms simultaneously.

Hence we will use numpy's implementation of FFT from now on.

A. Edge Detection

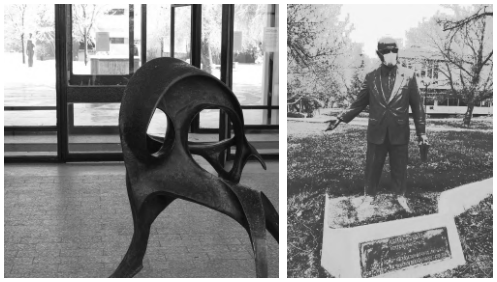

Fig. 1. Input Images

In this part we will try to implement Canny edge detection algorithm to above images in frequency domain.

For filtering in frequency domain we will be following these steps.

- Take the filters and the image into the frequency domain via DFT
- Concentrate high frequencies to the middle for better visualization and to facilitate filtering process. To achieve this we shift the zero-frequency component to the center of the spectrum
- Convolve input image with the filter in the frequency domain
- Reshift the image in frequency domain
- Return the frequency domain by using Inverse DFT

First we apply numpy's FFT to the input image via fft2 2-dimensional discrete Fourier transform function followed by fftshift function for shifting the zero-frequency component to the center of the spectrum.

```
img = cv2.imread(input_img_path, 0)
```

```
fourier = np.fft.fft2(img)
shifted = np.fft.fftshift(fourier)
```

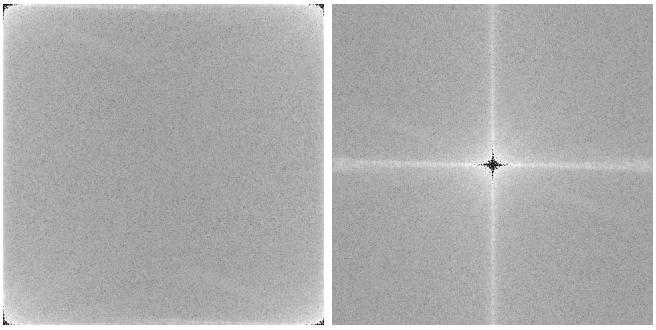

Fig. 2. Unshifted frequencies on the left and shifted frequencies on the right

In frequency domain we have tried some highPass filters. With ideal highpass filter(IHPF) ringing effect (artifacts that appear as spurious signals near sharp transitions in a signal) and edge distortion (thickened edges) observed. Although as we increase the radius (cutoff distance) the ringing effect is noticeably reduced, they were still prominent in near edge pixels.

$$H(u,v) = \begin{cases} 1 & D(u,v) > D_0 \\ 0 & D(u,v) \leq D_0 \end{cases} \quad (3)$$

,where D_0 is the cutoff frequency and $D(u,v) = \sqrt{(u - M/2)^2 + (v - N/2)^2}$

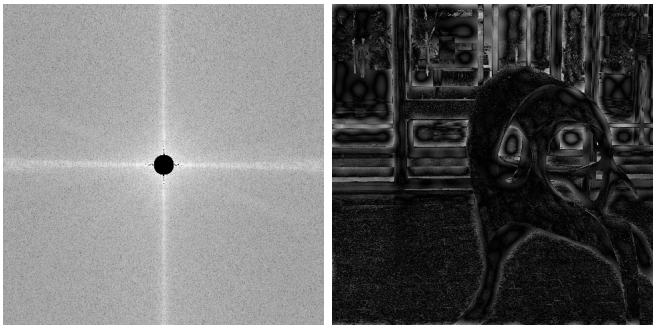

Fig. 3. IHPF with $D_0 = 15$ (480x480 image) on frequency and spatial domain

So we conclude that ideal filters are not suited for edge detection task since it prevents exactly what we want to achieve. Instead we use Gaussian highpass filter which causes no ringing artifacts and less edge distortions. To apply Gaussian filter first we created 5x5 Gaussian matrix in spatial domain. Then by padding with zeros we increased it size to the image size. Afterwards, by FFT we converted padded Gaussian to frequency domain and pointwise multiplying it with the image in frequency domain, we get the filtered image.

```
Gaussian = np.array([[2,  4,  5,  4, 2],
                     [4,  9, 12,  9, 4],
                     [5, 12, 15, 12, 5],
                     [4,  9, 12,  9, 4],
                     [2,  4,  5,  4, 2]])*(1/159)
padding = (number_of_rows - 5, number_of_columns -
    5)
Gaussian = np.pad(Gaussian, (((padding[0]+1)//2,
    padding[0]//2), ((padding[1]+1)//2, padding
    [1]//2)), 'constant', constant_values = 1)
Gaussian = np.fft.fft2(Gaussian)
Gaussian = np.fft.fftshift(Gaussian)
```

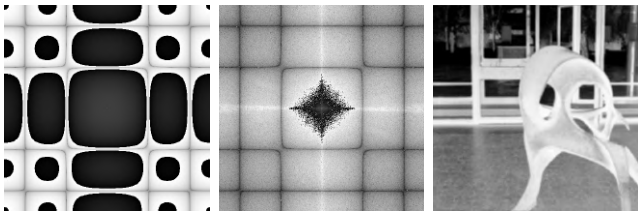

Fig. 4. Gaussian Highpass filter on frequency domain and filtered image in both frequency and spatial domain respectively

As expected high-pass filter alone is not appropriate for edge detection since it keeps all high-frequency features (e.g. sharp peaks and corners) which are usually not classified as edges. Next, we applied Sobel filter. Similar to Gaussian filter, first we convert 3x3 Sobel matrix in spatial domain to frequency domain by usig FFT followed by shifting. Then we multiplied with the image.

```
sobel_combined = np.array([[-1, -2, 1],
                           [-2, 0, 2],
                           [-1, 2, 1]])
padding = (number_of_rows - 3, number_of_columns -
    3)
sobel_combined = np.pad(sobel_combined, (((padding
    [0]+1)//2, padding[0]//2), ((padding[1]+1)//2,
    padding[1]//2)), 'constant')
```

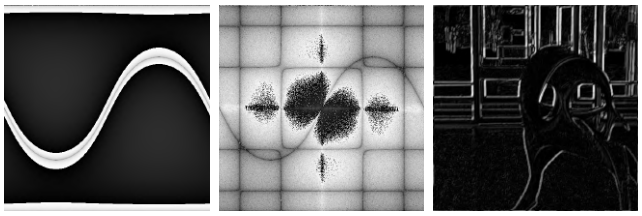

Fig. 5. Combined Sobel operator on frequency domain and Sobel filtered image in both frequency and spatial domain respectively

After filtering operations we have returned to spatial domain by reshifting followed by IFFT. Also we need to normalize the image since due to the multiplications we have done, we are way over the max value 255.

```
# Apply inverse shift
reshifted = np.fft.ifftshift(filtered)
# Return to spatial domain using Inverse Fourier
    Transformation
inversefourier = np.fft.ifft2(reshifted)
inversefourier = np.abs(inversefourier)
# Normalize the image
inversefourier -= inversefourier.min()
inversefourier = inversefourier*255 / inversefourier
    .max()
spatial = inversefourier.astype(np.uint8)
```

From now on we will work on spatial domain. With the Sobel filtering We sharpened the edges sufficiently. Now we have to get rid of the wrong and discontinued edges. We will do this by thresholding and edge tracking by hysteresis as we did in the THE1 previously.

```
# Thresholding
# If an edge pixels gradient value is higher than the high threshold value, it
      is marked as a strong edge pixel.
# If an edge pixels gradient value is smaller than the high threshold value and
      larger than the low threshold value, it is marked as a weak edge pixel
thresholded = np.zeros((number_of_rows, number_of_columns), np.uint8)
threshold_low = 30
threshold_high = 60

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

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

Fig. 6. After Thresholding

```
# Edge tracking by hysteresis
# To track the edge connection, blob analysis is applied by looking at a weak edge
      pixel and its 8-connected neighborhood pixels.
# As long as there is one strong edge pixel that is involved in the blob, that weak
      edge point can be identified as one that should be preserved.
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
```

End results:

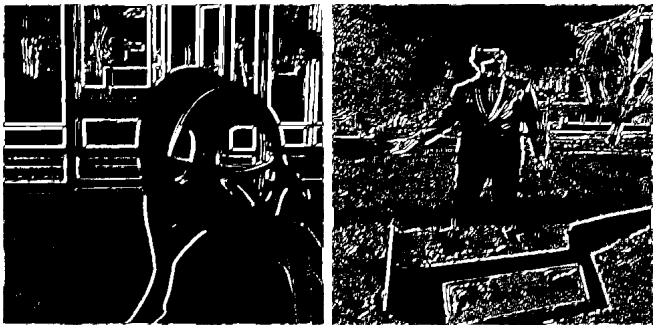

Fig. 7. Hysteresis applied

As a comparison with spatial domain, implementing edge detection in frequency domain turns out with sufficient results. The biggest advantage of frequency domain is the highpass filters. The high frequencies in the frequency domain displays a sharp change of image contrast from one pixel to another in spatial domain so they mostly corresponds to edges but not always. To actually compute edges, we need gradients, we need directional information which we do not have in the frequency domain. So edges are best described in spatial domain in my opinion. Another reason in favor of the spatial domain is the computational cost of the 2D FFT. In spatial domain we mostly convolute the image with smaller size filters whereas in the frequency domain we need to pad the filters and moreover we need to appy FTT to filters. Thus edge detection should be done more successfully in spatial domain in my thinking.

B. Noise Reduction

In this part we will work on two noisy specific images and will try to enhance these images in Fourier domain. To this end we will use various filters by trial and error methodology. Here the filters that we will be using: Ideal highpass and lowpass filters, bandreject and box filters.

Ideal band reject filter:

$$H(u,v) = \begin{cases} 0 & D_L \leq D(u,v) \leq D_H \\ 1 & otherwise \end{cases} \tag{4}$$

,where D_L and D_H are the lower and higher cutoff frequencies and $D(u,v) = \sqrt{(u - M/2)^2 + (v - N/2)^2}$

For the pass filters we create a matrix in the exact size of the image and fill it with full of 1's. Then starting at the center of this matrix we create a circle with given radius in which inside or outside is full of 0's according to type of the pass filter. If it is highpass, inside of the circle will be 0's as lower frequencies are gathered in the middle of the image matrix and converse is true for the lowpass filter. After we decide the entries of passfilter matrices, we simply multiply the given image with our filter to get the filtered image. Similarly for the bandreject filter. For the box filter we simply pinpoint a pixel and by multiplying with a filter we suppress the target pixel and its neighbors contained in the NxM matrix, where N and M parameters are given by us suited to number of pixels we wish to suppress.

```
def passFilter(image, filtertype, radius = 100):
    number_of_rows = image.shape[0]
    number_of_columns = image.shape[1]

    highPass_filter = np.ones((number_of_rows,
        number_of_columns), np.uint8)
    x_center, y_center = number_of_rows//2 ,
        number_of_columns//2
    x, y = np.ogrid[ :number_of_rows, :
        number_of_columns]
    if filtertype == "Highpass":
        circle_row, circle_column = np.where((x -
            x_center)**2 + (y - y_center )**2 <=
            radius**2 )
    elif filtertype == "Lowpass":
        circle_row, circle_column = np.where((x -
            x_center)**2 + (y - y_center )**2 >=
            radius**2 )
    highPass_filter[circle_row, circle_column] = 0

    return image * highPass_filter

def bandRejectFilter(image, band = (20,40)):

    number_of_rows = image.shape[0]
```

```
    number_of_columns = image.shape[1]
    inner,outer = band[:2]
    bandReject_filter = np.ones((number_of_rows,
        number_of_columns), np.uint8)

    x_center, y_center = number_of_rows//2 ,
        number_of_columns//2
    x, y = np.ogrid[ :number_of_rows, :
        number_of_columns]
    band_row, band_column = np.where(np.logical_and
        ((x - x_center)**2 + (y - y_center )**2 <=
        outer**2 ,(x - x_center)**2 + (y - y_center
        )**2 >= inner**2 ))
    bandReject_filter[band_row, band_column] = 0
    return image * bandReject_filter

def boxFilter(image, targetpixel, size):

    number_of_rows = image.shape[0]
    number_of_columns = image.shape[1]
    boxFilter = np.ones((number_of_rows,
        number_of_columns), np.uint8)
    boxFilter[targetpixel[0]-((size[0]-1)//2) :
        targetpixel[0]+((size[0]-1)//2),targetpixel
        [1]-((size[1]-1)//2) : targetpixel[1]+1+((
        size[1]-1)//2)] = 0

    return image * boxFilter
```

Now as always we start by reading the images in BGR format. Then we split the image to separate B,G,R channels and we work on all of them individually. We start by using FFT to convert images to frequency domain and then we shift them.

```
def enhance_3(path_to_3, output_path):
    img = cv2.imread(path_to_3)
    (B, G, R) = cv2.split(img)
    create_output_path(output_path)
    fourier = np.fft.fft2(img)
    shifted = np.fft.fftshift(fourier) # Shift the
        zero-frequency component to the center of
        the spectrum.
```

Input image:

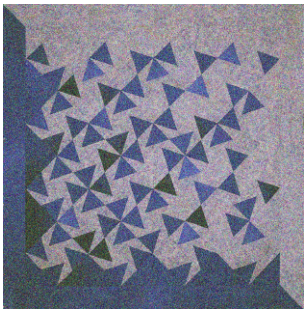

Fig. 8. Given image with BGR format in spatial domain

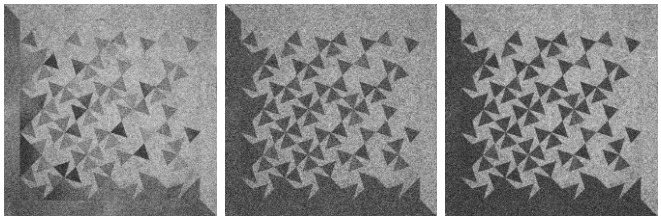

Fig. 9. All B,G,R channels in spatial domain respectively

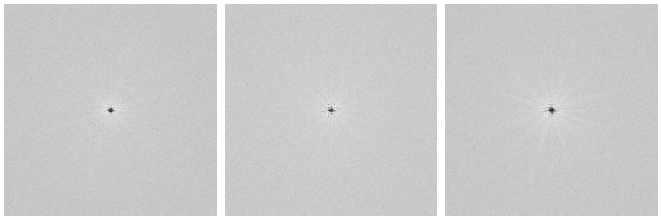

Fig. 10. All B,G,R channels in frequency domain respectively

By histogram analysis we observe that in all channels B,G and R number of pixels with intensity value 0 and 255 is significantly larger than the other intensity values. Since impulse noise (random noise or spike noise or salt and paper noise) introduces itself as casually occurring black and white pixels we suspect that it is impulse noise. So we applied the median filter as suggested in the literature but we were not satisfied with the result. Then we tried couple of linear filters such as mean and Gaussian but we still could not get a satisfying result. Afterwards we tried manual filters with manual inspection of the frequency domain. We observed high frequency components in some pattern and we tried to band-reject those components. Also with highpass filtering we get rid of the unnecessary parts of the image which holds no information rather than noise.

Example of high frequencies contributing none in channel R:

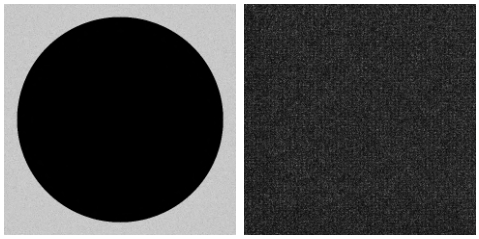

Fig. 11. Highpass filter on frequency domain and filtered image in both frequency domain respectively

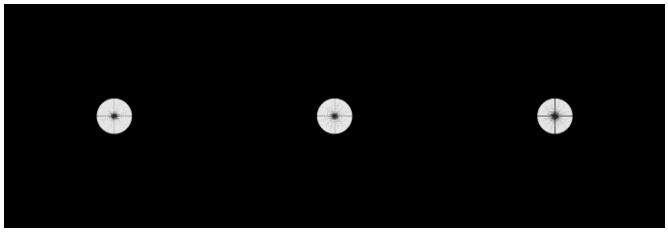

Fig. 12. Filtering in each B,G,R channel respectively

End Result:

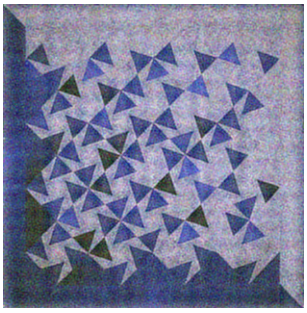

Fig. 13. Filtering in each B,G,R channel respectively

Second Input Image:

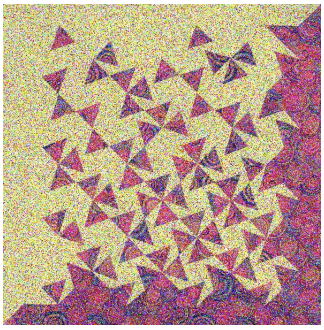

Fig. 14. Filtering in each B,G,R channel respectively

This time we know from THE1 that most of the information is on the G and R channels. Therefore while filtering we tried to be more precise in G and R channels to prevent loss of the details. To get rid of the redundant information and noise, we applied lowpass filters with various cutoff frequencies. However this caused blurring as a side effect. In the frequency domain, we searched by trial for components that may corresponding to noise but we could not find such components. We tried both bandpass and bandreject filters to various regions of image but again we were unsuccessful in eliminating the noise. After all these observations, we concluded that the noise is homogeneously distributed throughout the image. Thus we applied filters to remove details but preserve the overall shapes and patterns of the image.

We bandrejected some regions which seemed unnecessary to our eyes, but we observed that we lost a lot of information when we removed these regions.

For example here we can not see much but noise, but when we subtract this from the original image, we also lose to much information:

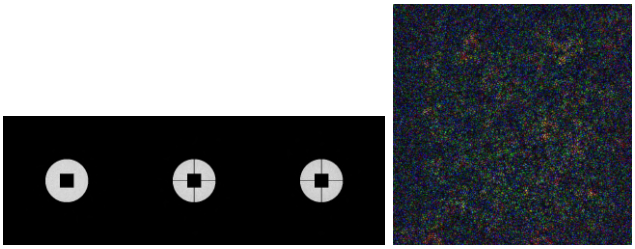

Fig. 15. Filtering in each B,G,R channel respectively and resulted image in spatial domain

After all the failures, we have applied the most pleasing filters, however dissatisfied we may be.

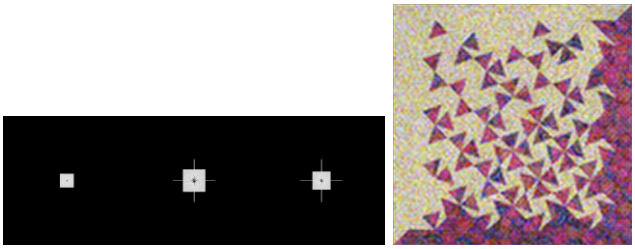

Fig. 16. Filtering in each B,G,R channel respectively and resulted image in spatial domain

In the THE1, where we aiming for noise reduction for the same images in spatial domain, we also not satisfied with the results. However the median and averaging filters we applied in spatial domain showed much better results and they were much less expensive operations due to much smaller kernel sizes. As a conclusion, we were thinking of doing much more successful noise reduction, since working in the frequency domain gives us the chance to suppress certain components and play with high and low frequencies. The reason why we could not achieve this may be that we work with trial and error method and we cannot find these components although they exist.

III. JPEG Image compression

In this section we will implement JPEG image compression.

We will be following these steps:

- Convert RGB to YCbCr color space
- Downsampling Chrominance Channels (Chroma subsampling)
- Discrete Cosine Transform (DCT)
- Quantization
- Encoding (Run-length and Huffman encoding)

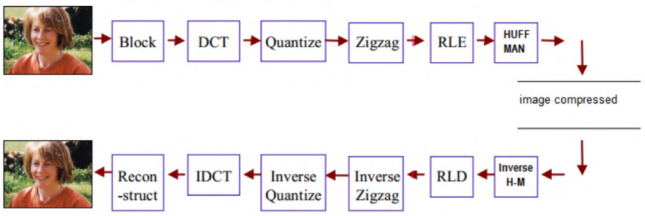

Fig. 17. JPEG Algorithm

As always we will start by reading the given image using python-opencv function cv2.imread. We know that the OpenCV library stores images in BGR format so in the next step we will convert BGR to JPEG's choice YC_rC_b.

```
def the2_write(input_img_path, output_path):
    # Read the input image
    img = cv2.imread(input_img_path)
```

A. Color Space Conversion

Color space conversion is the translation of the representation of a color from one basis to another. In this case with JPEG compression algorithm we will convert BGR to YC_rC_b. YC_rC_b, is a family of color spaces where Y is the luminance component and C_r and C_b are the red-difference and blue-difference chroma components. YC_rC_b values can be computed directly from RGB as follows:

$$Y = 0.299R + 0.587 * G + 0.114 * B$$

$$\begin{aligned} C_r &= 128 + 0.5 * R - 0.418688 * G - 0.081312 * B \\ C_b &= 128 - 0.168736 * R - 0.331264 * G + 0.5 * B \end{aligned} \quad (5)$$

,where Y, C_r and C_b with each input (B,G,R) have the full 8-bit range of [0,...,255]

```
def BGR2YCrCb(image):
    #image is BGR
    converted = np.zeros(image.shape, np.uint8)
    coefficients = [[0.299, 0.587, 0.114],
        [-0.168736, -0.331264, 0.5], [0.5,
        -0.418688, -0.081312]]

    converted[:,:,0] =  coefficients[0][0]*image
        [:,:,2] + coefficients[0][1]*image[:,:,1] +
        coefficients[0][2]*image[:,:,0]
    converted[:,:,1] =  coefficients[1][0]*image
        [:,:,2] + coefficients[1][1]*image[:,:,1] +
        coefficients[1][2]*image[:,:,0] + 128
    converted[:,:,2] =  coefficients[2][0]*image
        [:,:,2] + coefficients[2][1]*image[:,:,1] +
        coefficients[2][2]*image[:,:,0] + 128

    return converted
```

Using this function we convert the colorspace.

```
YCrCb = BGR2YCrCb(img)
(Y, Cr, Cb) = cv2.split(YCrCb)
```

B. Subsampling the Chrominance Channels C_r and C_b

The practice of encoding images with lower resolution for chroma information than for lumination information is known as chroma subsampling. Taking use of the lesser acuity of the human visual system for color differences than for luminance we simply discard significant amount chroma information but our eyes fail to notice it. Now, color in chroma channels will be reduced by factor of 2 in both directions vertical and horizontal, that is, Y is sampled at each pixel, where as C_r and C_b are sampled at every block of 2x2 pixels. Thus, for every 4 Y pixels, there will exist only 1 C_rC_b pixel.

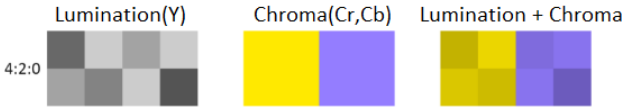

Fig. 18. 4:2:0 Chroma subsampling

In programming we applied 2x2 averaging filter to chroma channels. Then we simply skip every first row and column and take every second row and column of the image matrices and end up with an image both vertical and horizontal resolution is halved.

```
kernel = np.array([[1/4, 1/4], [1/4, 1/4]])
Cr = convolution2D(Cr, kernel)[::2,::2]
Cb = convolution2D(Cb, kernel)[::2,::2]
```

Basic 2d convolution:

```
def convolution2D(image, filter):
    m, n = filter.shape
    size = image.shape
    output = np.zeros(size, np.uint8)
    for i in range((m-1)//2, size[0] - ((m-1)//2) ):
        for j in range((m-1)//2, size[1] - (m-1)//2
            ): 4//2 = 2):
            output[i][j] =(1)*round(np.sum(image[i
                -((m-1)//2):i+1-((m-1)//2), j-((m-1)
                //2):j+1-((m-1)//2)]*filter))

    return output
```

C. Discrete Cosine Transform

A discrete cosine transform (DCT) is a method of representing a finite sequence of data points as a sum of cosine functions oscillating at different frequencies. The DCT has the property of preserving the majority of the visually essential information in an image with only a few coefficients containing all of the essential information about the image. So, the DCT is mostly utilized in image compression applications.

$$F(u,v)=\sqrt{\tfrac{2}{N}}\sqrt{\tfrac{2}{M}}c_i c_j \sum_{i=0}^{M-1}\sum_{j=0}^{N-1} f(i,j)cos[\tfrac{\pi u}{2N}(2i+1)]cos[\tfrac{\pi v}{2M}(2j+1)]$$

$$u\in\{0,1,2,...,M-1\}, v\in\{0,1,2,....,N-1\}$$

$$c_k=\begin{cases} \frac{1}{\sqrt{2}} & k=0 \\ 1 & otherwise \end{cases} \quad (6)$$

,where F(u,v) is the DCT applied image and f(i,j) is the image of size M × N.

We implemented the same DCT formula in programming with loops for summation. However due to time complexity of our implementation instead we will use OpenCV function cv2.dct for DCT.

```
def DCT(block):
    blocksize = block.shape[0]
    dct = np.zeros((blocksize,blocksize),np.float32
        )
    for u in range(0,blocksize):
        for v in range(0,blocksize):
            summation = 0
            for i in range(0,blocksize):
                for j in range(0,blocksize):
                    summation += block[i][j]*np.cos(
                        np.pi*u*(2*i+1)/(2*blocksize
                        ))*np.cos(np.pi*v*(2*j+1)
                        /(2*blocksize))
            if(u == 0):
                c1 = 1/np.sqrt(2)
            else:
                c1 = 1
            if(v == 0):
                c2 = 1/np.sqrt(2)
            else:
                c2 = 1
            summation *= c1*c2

            dct[u][v] = summation*np.sqrt(1/(2*
                blocksize))

    return dct
```

In JPEG, DCT will take an 8x8 image block and tell us how to reproduce it using an 8x8 matrix of cosine functions. Therefore first we convert the image into chunks of 8x8 blocks of pixels called minimum coding units (MCU), and scale the range of values of the pixels to [-128,128] so that they center on 0 . Afterwards we applied DCT to each block followed by quantization to compress the resulting block.

In programming, each channel is splited into 8x8 image blocks seperately followed by scaling. Example for C_r channel:

```
blocksize = 8

for r in range(0, Cr.shape[0]//8):
    for c in range(0, Cr.shape[1]//8):
        block = Cr[r*blocksize : (r+1)*blocksize, c*
            blocksize: (c+1)*blocksize]
        # Convert datatype to int16 so we can
            subtract 128 from each pixel
        block = block.astype(np.int16)-128
```

Original Sub-image

64	60	57	56	48	47	47	43
61	58	53	52	48	49	52	53
67	60	53	53	49	47	48	54
68	61	63	63	62	65	65	64
71	61	70	63	69	74	88	88
83	94	102	105	107	111	110	115
95	108	108	124	122	130	128	128
107	118	125	134	137	142	141	137

Shifted sub-image

-64	-68	-71	-72	-80	-81	-81	-85
-67	-70	-75	-76	-80	-79	-76	-75
-61	-68	-75	-75	-79	-81	-80	-74
-60	-67	-65	-65	-66	-63	-63	-64
-57	-67	-58	-65	-59	-54	-40	-40
-45	-36	-26	-23	-21	-17	-18	-13
-33	-20	-20	-4	-6	2	0	0
-21	-10	-3	6	9	14	13	9

Now we can use DCT to transform these 8x8 chunks to frequency domain followed by quantization. To quantize the frequencies obtained using DCT we use standard JPEG Quantization table. In the 8x8 matrix of coefficients we got through DCT, the top-left entries refer to low frequency part, and the bottom-right entries refers to high frequency part. So quantization table will have have very small values at the top-left part and very high values towards the bottom-right part to eliminate the high frequency part without much loss in the look of the image. We will use the standard JPEG quantization tables designed for luminance and chrominance channels.

```
QY = 1/np.array([[16,11,10,16,24,40,51,61],
                 [12,12,14,19,26,48,60,55],
                 [14,13,16,24,40,57,69,56],
                 [14,17,22,29,51,87,80,62],
                 [18,22,37,56,68,109,103,77],
                 [24,35,55,64,81,104,113,92],
                 [49,64,78,87,103,121,120,101],
                 [72,92,95,98,112,100,103,99]])

QC = 1/np.array([[17,18,24,47,99,99,99,99],
                 [18,21,26,66,99,99,99,99],
                 [24,26,56,99,99,99,99,99],
                 [47,66,99,99,99,99,99,99],
                 [99,99,99,99,99,99,99,99],
                 [99,99,99,99,99,99,99,99],
                 [99,99,99,99,99,99,99,99],
                 [99,99,99,99,99,99,99,99]])

for r in range(0, Cr.shape[0]//8):
        for c in range(0, Cr.shape[1]//8):
            block = Cr[r*blocksize : (r+1)*blocksize
                , c*blocksize: (c+1)*blocksize]
            block = block.astype(np.int16)-128
            block = block.astype(np.float32)
            block = cv2.dct(block)
            quantized = block * QC
            quantized = quantized.astype(np.int16)
```

After DCT

-376	-23	1	-2.5	-0.3	4	0.2	-2.6
-224	53	20	3.4	5	3	0.6	2.3
68	3.3	-14	-0.3	-2.8	-1.9	-4.7	-6.2
2.3	-8.9	-1.5	-3.8	-2.5	1.2	1.4	1.9
-8.4	1.2	1.9	3.3	-2.1	5	1.8	5.3
4.5	7.3	-7.4	1.9	1.3	-0.7	-1.5	-6
6.4	6.8	-3.2	-2.6	1.3	-2.1	1.7	1
-16	0.1	9	0.8	1.8	1.7	-1	1

After Quantization

-24	-23	0	0	0	0	0	0
-19	4	1	0	0	0	0	0
5	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig. 19.

We have now got the compressed output as a 2D array. We also know that a lot of them are zeroes. So, we will find a better way to store the sub-image than store its as a 2D array. In this step we start the encoding process.

D. Encoding

In JPEG compression algorithm, after quantization we use zig-zag encoding to convert the 2D array to 1D array. This encoding is preferred because most of the low frequency (most significant) information is stored at the beginning of the matrix and the zig-zag encoding stores all of that at the beginning of the 1D array.

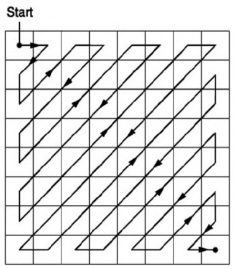

Fig. 20.

In programming we created an empty 2D list and filled this list as we traversing the quantized 2D array in zigzag order. Then by one more pass over this 2D list we got the 1D list. Now we are ready for the compression step.

```
zigzag_temp = [[] for x in range(blocksize*2 -1)]

for i in range(blocksize):
    for j in range(blocksize):
        temp = i+j
        if(temp%2 ==0):
            #add at the beginning of the list
            zigzag_temp[temp].insert(0,quantized[i][
                j])
        else:
            #add at the end of the list
            zigzag_temp[temp].append(quantized[i][j
                ])

zigzag = []
for sublist in zigzag_temp:
    for element in range(0,len(sublist)):
        zigzag.append(sublist[element])
```

Next we use Run-length encoding to compress repeated data. Run-length encoding (RLE) is a form of lossless data compression in which runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count.

After the zig-zag encoding, most of the zig-zag encoded 1D arrays had so many 0s at the end. Run-length encoding allows us to reclaim all that wasted space and use fewer bytes to represent all of those 0s.

For example Run-length encoding will convert list of 64 length [-7, 6, 3, 0, -5, 1, 0, -1, 2, 0, 0, 0, 0, 0, 0,...,0] into

[[-7 1][6 1][3 1][0 1][-5 1][1 1][0 1][-1 1][2 1][0 55]]

Here our implementation:

```
def runLengtEncoding(alist):
    i = 0
    n = len(alist)
    newlist = []
    while i < n:
        count = 1
        while (i < n-1 and alist[i] == alist[i+1]):
            count += 1
            i += 1
        i += 1
        newlist.append((alist[i-1],count))

    return newlist
```

We applied RLE to 1D zigzag list:

```
zigzag = np.asarray(runLengtEncoding(zigzag))
```

Now the next step is Huffman encoding. This can be done in two steps: Conversion of quantized DCT coefficients into an intermediate sequence of symbols and assignment of variable-length codes to the symbols. DCT coefficients are divided into ”DC coefficient” and ”AC coefficients”. DC coefficient is the coefficient with zero frequency in both dimensions(first enrty of the 8x8 block), and AC coefficients are remaining 63 coefficients with non-zero frequences.

In the intermediate symbol sequence, each nonzero AC coefficients is represented as combination with the “runlength” of zero-valued AC coefficients which precede it in the zig-zag. Each such runlength-nonzero AC coefficient combination is represented by a pair of symbols:

$$\textbf{symbol-1} : (RUNLEGTH, SIZE)$$

$$\textbf{symbol-2} : (AMPLITUDE)$$

Symbol-1 contains two pieces of information, RUNLENGTH and SIZE. Symbol-2 contains the single piece of information designated amplitude, which is simply the intensity value of the nonzero AC coefficient. SIZE is the number of bits used to encode AMPLITUDE. RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zig-zag sequence preceding the nonzero AC coefficient.

For example:

The number 7 can be represented with 3 bit: 111 $(1*2^2 + 1*2^1 + 1*2^0)$, so the size will be 3. Now suppose that before 7 there is no zero ,so runlength will be 0. Thus intermediate symbol-1 symbol-2 will be (0,3)(7).

For the negative integer values, we will use the representation called ones' complement. The ones' complement of a binary number is the value obtained by inverting all the bits in the binary representation of the number (swapping 0s and 1s).

```
def complement(bits):
    complement = ""
    for x in range(0,len(bits)):
        if bits[x] == '0':
            complement += "1"
        else:
            complement += "0"

    return complement
```

For example:

The number -5 can be represented with 3 bit: 010(5 can be represented with 101 so we invert it), so the size will be 3. Now suppose that before -5 there is only one zero, then runlength will be 1. Thus intermediate symbol-1 symbol-2 will be (1,3)(-5).

The special symbol-1 value (0, 0) means EOB (end-of-block) symbol, which terminates the 8×8 sample block.

Normally, since DC coefficients has no preceeding zero before them, we do not need to consider runlegth information. To compress the size which DC coefficient occupies, JPEG uses Delta encoding. Delta encoding is a method of storing or sending data as differences (deltas) between consecutive data. But in this case we will not discriminate between the DC and

AC coefficients except assigning Huffman codewords at the end.

Now, as a last step we will compute the Huffman codes for each of these nonzero coefficients. We will represent the symbol-1:(run/size) by bits, its Huffman code , and the corresponding symbol-2:(amplitude) by ones' complement amplitude representation.

Huffman coding assigns the shortest codelength to the most frequent symbols. As the relative frequency of the symbol decrease the corresponding code length of the codeword increases.

Huffman coding has two main passes: Forward pass and backward pass. The forward pass sorts the probability of symbols appearing throughout the image from the most frequent to the lowest.

Two of the lowest probabilities of the symbols are combined by adding them.

Step 2 is repeated until only two probabilities remain.

In the backward pass, the codewords are assigned to each reduced source, going backwards. We start by assigning 0 and 1 to the reduced probabilities, at the last step of source reduction. Then as we move backwards, we keep adding 0 and 1 to the unfolded probabilities.

In our implementation we applied this procedure using Tree data structure. But due to time complexity we later decided to use predefined standard codecs for Huffman codes.

Here our toy implementation of Huffman Coding procedure:

```
def huffmanEncoding(array):

    lookUpTableAC = [[(0,0),"1010"],[(0,1), "00"],[(0,2),"01"],[(0,3),"100"],[(0,4)
        ,"1011"],[(0,5),"11010"],[(0,6),"1111000"],[(0,7),"11111000"],[(0,8),"
        1111110110"],[(0,9),"1111111110000010"],[(0,10),"1111111110000011"],
        continues...]

    lookUpTableDC = [[(0,0),"00"],[(0,1),"010"],[(0,2),"011"],[(0,3),"100"],[(0,4),
        "101"],[(0,5),"110"],[(0,6),"1110"],[(0,7),"11110"],[(0,8),"111110"
        ],[(0,9),"1111110"],[(0,10),"11111110"],[(0,11),"111111110"]]
    output = []
    zerocount = 0
    for x in range(0,array.shape[0]):
        if (array[x][0] == 0):
            zerocount += array[x][1]
        else:
            if array[x][0] < 0 :
                output.append(((zerocount, len(bin(array[x][0])[3:])),array[x][0]))
            else:
                output.append(((zerocount, len(bin(array[x][0])[2:])),array[x][0]))

            zerocount = 0
        if (x == array.shape[0]-1) : # last element in the list
            output.append((0,0))
    huffman = []
    bitstream = ""
    for coeff in range (0,len(output)):
        if (coeff == len(output)-1): # end of the 8x8 block's is (0,0) coded as
            "1010"
            huffman.append(("1010"))
            bitstream += "1010"
        elif (coeff == 0):
            for codeword in lookUpTableDC:
                if codeword[0] == output[coeff][0]:

                    if output[coeff][1] < 0:
                        huffman.append((codeword[1],complement(bin(output[coeff
                            ][1])[3:])))
                        bitstream += codeword[1] + complement(bin(output[coeff][1])
                            [3:])
                    else:
                        huffman.append((codeword[1],bin(output[coeff][1])[2:]))
                        bitstream += codeword[1] + bin(output[coeff][1])[2:]

        else:
            for codeword in lookUpTableAC:
                if codeword[0] == output[coeff][0]:

                    if output[coeff][1] < 0:
                        huffman.append((codeword[1],complement(bin(output[coeff
                            ][1])[3:])))
                        bitstream += codeword[1] + complement(bin(output[coeff][1])
                            [3:])
                    else:
                        huffman.append((codeword[1],bin(output[coeff][1])[2:]))
                        bitstream += codeword[1] + bin(output[coeff][1])[2:]
    return bitstream
```

After Huffman encoding we have the bitstream representation of each 8x8 block. At last, we simply write these bitsteams to a .txt file format for each YC_rC_b channel.

Here the whole encoding process for Y channel:

```
blocksize = 8
    with open(output_path+"/compressed.txt", "w") as file:
        file.write("Y {} {} \n".format(number_of_rows,number_of_columns))
        for r in range(0, Y.shape[0]//8):
            for c in range(0, Y.shape[1]//8):
                block = Y[r*blocksize : (r+1)*blocksize, c*blocksize: (c+1)*
                    blocksize]
                block = block.astype(np.int16)-128
                block = block.astype(np.float32)
                block = cv2.dct(block)
                quantized = block * QY

                quantized = quantized.astype(np.int16)
                zigzag_temp = [[] for x in range(blocksize*2 -1)]
                # Zigzag traversal
                for i in range(blocksize):
                    for j in range(blocksize):
                        temp = i+j
                        if(temp%2 ==0):
                            zigzag_temp[temp].insert(0,quantized[i][j])
                        else:

                            zigzag_temp[temp].append(quantized[i][j])
                zigzag = []
                for sublist in zigzag_temp:

                    for element in range(0,len(sublist)):
                        zigzag.append(sublist[element])
                # Runlength encoding
                zigzag = np.asarray(runLengtEncoding(zigzag))
                # Apply Huffman Coding to 8x8 blocks and write the output bitstream
                    representing 8x8 block to a txt file.
                file.write(huffmanEncoding(zigzag)+"\n")
```

In this implementation by scaling, that is, multiplying quantization tables by constant we get different compression result. With smaller scaling factor we get more compression together with more loss of information. For scale 1, compressed image size turns out 3014.719KB where the original image was 6177.333KB so the compression ratio is 2.049. Therefore we lost half of the information. However since human eye is not capable of understanding the difference, we gained more than we lost as a storage space.

scale	compression ratio
1	2.049
0.50	5.356
0.10	7.804
0.05	10.283

E. Reading and Decoding the Compressed image

This time as shown in the Figure (17) previously we will apply the each step of encoding process in reverse order with inverse operations. Basically we will:

- Read the bitstreams from txt file and convert them to (symbol-1),(symbol-2) format
- Reconstruct 1D zigzag list and 8x8 quantized block
- Dequantize the 8x8 block
- Apply Inverse Discrete Cosine Transformation(IDCT) to block
- Rescale the 8x8 block
- Reverse chroma subsampling by resizing chroma channels
- Convert colorspace from YC_rC_b to BGR

In compressed.txt we have a line indicating which channel we are on(for the first line we also have the size of the image)

followed by lines corresponding bitstream representations of 8x8 blocks for each channel:

```
Y 1536 2048
11100110001010
11100101111010
.
Cr
10110101010
10110111010
.
Cb
011000001010
1010
.
```

Therefore we start reading this file line by line. For Y and chroma(C_rC_b) channels our decoding will have slight variations since we use different quantization and Huffman tables while encoding, and also because the chroma channels have size one quarter of the Y channel. So, by reading the file we will distinguish the channels by looking for lines including "Y" or "Cr" or "Cb". Then by using right lookup tables first we decode the Huffman codes followed by reconstructing quantized 8x8 matrix by reversing zigzag traversal. Afterwards, we simply multiply quantized 8x8 matrix by corresponding multiplicative inversed quantization table. After getting a dequantized 8x8 block, we apply IDCT followed by rescaling, that is, we simply add 128 to each entry of the 8x8 matrix. After all these, we will have all the 8x8 blocks. Now all that's left is constructing the image block by block. Remember, we previously subsampled the chroma channels. Now we need to resize chroma channels to the original size. After resizing chroma channels we simply merge all channels and hence we got the image back in YC_rC_b colorspace. At last we convert colorspace to BGR.

Here the whole process:

```
def the2_read(input_img_path):
    lookUpTableAC = [[(0,0),"1010"],[(0,1), "00"],[(0,2),"01"],[(0,3),"100"],[(0,4)
        ,"1011"],[(0,5),"11010"],[(0,6),"1111000"],[(0,7),"11111000"],[(0,8),"
        1111110110"],[(0,9),"1111111110000010"],[(0,10),"1111111110000011"],
        continues..]

    lookUpTableDC = [[(0,0),"00"],[(0,1),"010"],[(0,2),"011"],[(0,3),"100"],[(0,4),
        "101"],[(0,5),"110"],[(0,6),"1110"],[(0,7),"11110"],[(0,8),"111110"
        ],[(0,9),"1111110"],[(0,10),"11111110"],[(0,11),"111111110"]]

    QY = np.array([[16,11,10,16,24,40,51,61],
                   [12,12,14,19,26,48,60,55],
                   [14,13,16,24,40,57,69,56],
                   [14,17,22,29,51,87,80,62],
                   [18,22,37,56,68,109,103,77],
                   [24,35,55,64,81,104,113,92],
                   [49,64,78,87,103,121,120,101],
                   [72,92,95,98,112,100,103,99]])

    QC = np.array([[17,18,24,47,99,99,99,99],
                   [18,21,26,66,99,99,99,99],
                   [24,26,56,99,99,99,99,99],
                   [47,66,99,99,99,99,99,99],
                   [99,99,99,99,99,99,99,99],
                   [99,99,99,99,99,99,99,99],
                   [99,99,99,99,99,99,99,99],
                   [99,99,99,99,99,99,99,99]])
    scale = 1
    QY *= scale
    QC *= scale

    image = np.zeros((1536,2048,3),dtype=np.uint8)
    channel = 0
    blocksize = 8
    size = []
    with open(input_img_path, "r") as file:
        lines = file.readlines()
        for line in lines:
            if("Y" in line):
                channel = 0
                Q = QY
                r,c = 0,0
                size = line.split(" ")[1:3]
                cmax = int(size[1])//blocksize
            elif("Cr" in line):
                channel = 1
                Q = QC
                r,c = 0,0
                cmax = cmax//2
            elif("Cb" in line):
                channel = 2
                Q = QC
                r,c = 0,0
            else:
                decode = []
                cursor = 0
                count = 0
                for x in range(0,len(line)):
                    if count == 0 :
                        lookUpTable = lookUpTableDC
                    else:
                        lookUpTable = lookUpTableAC
                    for code in lookUpTable:
                        if(line[cursor:x] == code[1]):
                            count= 1
                            if(code[1]== "1010"):#end of the 8x8 block
                                decode.append((0,0))
                            else:
                                symbol1 = code[0]
                                value = line[x:x+symbol1[1]]
                                if (len(value)== 0 or value == "\n"):
                                    break
                                if value[0] == '0':
                                    value = int(complement(value),2)*(-1)
                                else:
                                    value = int(value,2)
                                decode.append((symbol1,value))
                                cursor = x+symbol1[1]

                # reconstruct zigzag
                length = 0
                zigzag = []
                for symbols in decode:
                    if(symbols == (0,0)):
                        for i in range(0,64-length):
                            zigzag.append(0)
                        break
                    else:
                        for i in range(0,(symbols[0])[0]):
                            zigzag.append(0)

                        zigzag.append(symbols[1])
                        length = len(zigzag)

                quantized = np.zeros((blocksize,blocksize),np.int16)
                if (len(zigzag) == 64):
                    tempx,tempy = 0,0
                    for i in range(0,63):
                        quantized[tempx][tempy] = zigzag[i]
                        if((tempx+tempy)%2 == 0): #traversal will start from bottom
                            to top
                            if tempx != 0:
                                dx = -1
                            else:
                                dx = 0
                            if tempy != 7:
                                dy = 1
                            else:
                                dy = 0
                        else: #traversal will start from top to bottom
                            if tempx != 7:
                                dx = 1
                            else:
                                dx = 0
                            if tempy != 0:
                                dy = -1
                            else:
                                dy = 0
                        tempx += dx
                        tempy += dy
                # Dequantize
                dequantized = quantized * Q
                dequantized = dequantized.astype(np.float32)
                # IDCT
                block = cv2.idct(dequantized)
                block = block.astype(np.int16)
                # Rescale
                block = block +128
                # Construct Image channels block by block
                image[r*blocksize : (r+1)*blocksize,c*blocksize: (c+1)*blocksize,
                      channel] = block
                if (c==cmax-1):
                    c = 0
                    r+= 1
                else:
                    c+= 1

    file.close()
    Y = image[:,:,0]
    Cr = image[:,:,1]
    Cb = image[:,:,2]
    # Reverse subsampling
    Cr4 = Cr[0:int(size[0])//2,0:int(size[1])//2]
    Cr = cv2.resize(Cr4, (2048, 1536), interpolation= cv2.INTER_LINEAR)
    Cb4 = Cb[0:int(size[0])//2,0:int(size[1])//2]
    Cb = cv2.resize(Cb4, (2048, 1536), interpolation= cv2.INTER_LINEAR)

    image[:,:,2] = Cr
    image[:,:,1] = Cb
    # Convert color space to BGR
    image = cv2.cvtColor(image, cv2.COLOR_YCrCb2BGR)
    # Show image
    cv2.imshow("reconstructed",image)
```

Since some values are suppressed to zero in the quantization step, we cannot fully return those values. This causes some pixels in reconstructed image to deviate from the original. But aside from the compression we get, this is pretty much

negligible. We can also increase or decrease the compression ratio by scaling the quantization table. However, this can result in the destruction of necessary information in the image.

Fig. 21. Input image

Reconstructed images:

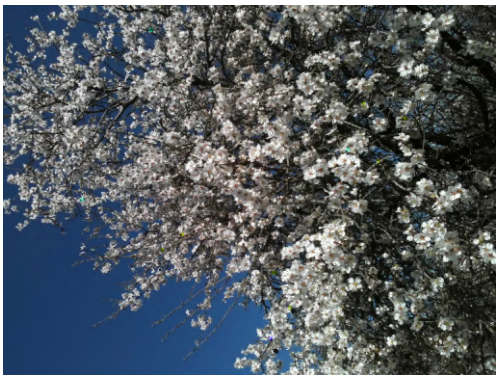

Fig. 22. Reconstructed image (Quantization scale = 1)

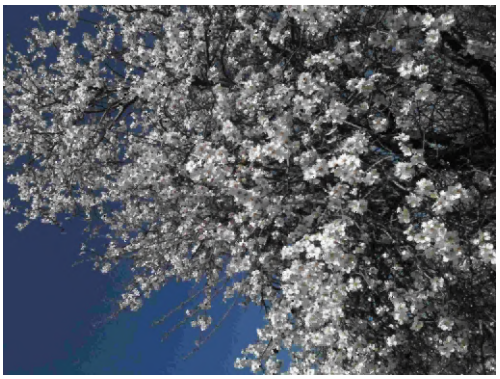

Fig. 23. Reconstructed image (Quantization scale = 0.50)

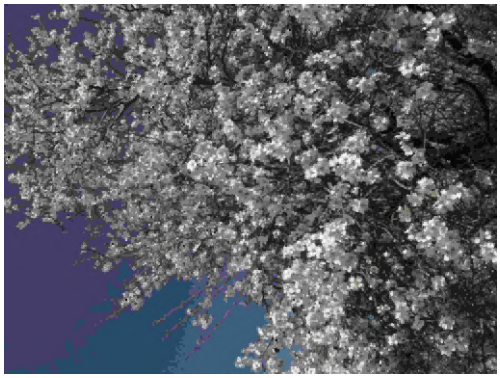

Fig. 24. Reconstructed image (Quantization scale = 0.10)

ACKNOWLEDGMENT

We implemented frequency domain image enhancement techniques using mainly trial-error methodology. We used various frequency domain filtering techniques to achieve desired goals for enhancement of the images and have seen advantages and disadvantages of working in frequency domain. Also we tried to implement JPEG compression algorithm which utilizes frequency domain by using DCT.

REFERENCES

[1] Muzhir Al-Ani and Fouad Awad. "THE JPEG IMAGE COMPRESSION ALGORITHM". In: *International Journal of Advances in Engineering Technology* 6 (May 2013), pp. 1055–1062.

[2] Arjunsreedharan. *JPEG 101 - how does JPEG work?* June 2016. URL: https://arjunsreedharan.org/post/146070390717/jpeg-101-how-does-jpeg-work.

[3] David R. Bull and Fan Zhang. "Lossless compression methods". In: *Intelligent Image and Video Compression Communicating Pictures*. Elsevier, 2021, pp. 225–270.

[4] *Chroma subsampling*. June 2021. URL: https://en.wikipedia.org/wiki/Chroma_subsampling.

[5] Rafael C. Gonzalez and Richard E. Woods. "Image Compression and Watermarking". In: *Digital Image Processing*. Pearson, 2018, pp. 539–594.

[6] *JPEG*. Jan. 2022. URL: https://en.wikipedia.org/wiki/JPEG.

[7] *JPEG: Colorspace transform, subsampling, DCT and quantisation¶*. URL: https://www.hdm-stuttgart.de/~maucher/Python/MMCodecs/html/jpegUpToQuant.html.

[8] Cory Maklin. *Fast fourier transform*. Dec. 2019. URL: https://towardsdatascience.com/fast-fourier-transform-937926e591cb.

[9] Cung Nguyen. "Fault Tolerant Huffman Coding for JPEG Image Coding System". In: ().

[10] Tongfeng Yang et al. "A new edge detection algorithm using FFT procedure". In: *Lecture Notes in Electrical Engineering* (2013), pp. 297–304. DOI: 10.1007/978-3-642-41407-7_29.

[11] *YCbCr*. Oct. 2021. URL: https://en.wikipedia.org/wiki/YCbCr.