HW2 Report Gideon Levi 109006114

1. DCT Image Compression

```python
Q1 > 1.py > ...
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import cv2
import math
import scipy

def imshow(image):
    if len(image.shape) == 3:
        # Height, width, channels
        pass
    else:
        # Height, width - must be grayscale
        # convert to RGB, since matplotlib will plot in a weird colormap (instead of black = 0, white = 1)
        image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)
    # Draw the image
    plt.imshow(image)
    # Disable drawing the axes and tick marks in the plot, since it's actually an image
    plt.axis('off')
    plt.show()

n = int(input('Enter n:'))
m = int(input('Enter m:'))
select_img = int(input('Enter 1 for Barbara, 2 for Cat:'))
if select_img == 1:
    input_img = cv2.imread('./Barbara.jpg')
    img_name = 'bar'
elif select_img == 2:
    input_img = cv2.imread('./cat.jpg')
    img_name = 'cat'
input_img_rgb = cv2.cvtColor(input_img, cv2.COLOR_BGR2RGB)
# imshow(input_img_rgb)
```

- For this problem, first I import needed modules such as numpy, matplotlib, cv2, math, and scipy. I also initialize an imshow() function to show an image. Then I ask input for value of “n”, “m”, and to select the picture to be compressed. Then I read the selected image with imread() function, and convert it from BGR to RGB with cvtColor() function.

```python
def keep_n_per_block(img, n):
    x_lim = img.shape[0]
    y_lim = img.shape[1]
    result_img = np.zeros((x_lim, y_lim))

    for j in range(0, y_lim, 8):
        for i in range(0, x_lim, 8):
            result_img[i:(i+n), j:(j+n)] = img[i:(i+n), j:(j+n)]
    return result_img
```

- I also initialize a keep_n_per_block() function to keep only the lower-frequency (i.e. upper-left-n-by-n) coefficients in the 2D DCT domain 8x8 block, and set the rest of the coefficients in the block to 0.

(a) Implement the simplified DCT compression process for n = 2, 4 and m = 4, 8.

```python
#COMPRESS
rgb_quantization_table = np.array([[ 8, 6, 6, 7, 6, 5, 8, 7],
                                   [ 7, 7, 9, 9, 8,10,12,20],
                                   [13,12,11,11,12,25,18,19],
                                   [15,20,29,26,31,30,29,26],
                                   [28,28,32,36,46,39,32,34],
                                   [44,35,28,28,40,55,41,44],
                                   [48,49,52,52,52,31,39,57],
                                   [61,56,50,60,46,51,52,50]])


input_img_rgb_float = input_img_rgb.astype(np.float32)
x_lim = input_img_rgb.shape[0]
y_lim = input_img_rgb.shape[1]

#divide the img to 3 channels
input_img_r_float = np.zeros((x_lim, y_lim))
input_img_g_float = np.zeros((x_lim, y_lim))
input_img_b_float = np.zeros((x_lim, y_lim))
for j in range(0, y_lim):
    for i in range(0, x_lim):
        input_img_r_float[i, j] = input_img_rgb_float[i, j][0]
        input_img_g_float[i, j] = input_img_rgb_float[i, j][1]
        input_img_b_float[i, j] = input_img_rgb_float[i, j][2]
```

(b) For part (a), first I create the rgb quantization table as 8x8 numpy array. Then I change the datatype of the input image to float, get the size of the image, and divide the image into 3 numpy arrays which are arrays for r, g, and b channels. I store the r channel to “input_img_r_rgb”, g channel to “input_img_r_rgb”, and b channel to “input_img_b_rgb”.

```python
dct_img_r = np.zeros((x_lim, y_lim))
dct_img_g = np.zeros((x_lim, y_lim))
dct_img_b = np.zeros((x_lim, y_lim))

for j in range(0, y_lim, 8):
    for i in range(0, x_lim, 8):
        dct_img_r[i:(i+8), j:(j+8)] = dct2d(input_img_r_float[i:(i+8), j:(j+8)])
        dct_img_g[i:(i+8), j:(j+8)] = dct2d(input_img_g_float[i:(i+8), j:(j+8)])
        dct_img_b[i:(i+8), j:(j+8)] = dct2d(input_img_b_float[i:(i+8), j:(j+8)])

dct_img_r = keep_n_per_block(dct_img_r, n)
dct_img_g = keep_n_per_block(dct_img_g, n)
dct_img_b = keep_n_per_block(dct_img_b, n)
```

- I create a function to apply DCT and IDCT on 2D image called dct2d() and idct2d() function based on the formula on PPT. Then for each channel’s image, I divide the image into blocks of 8x8 pixels and apply 2D DCT for each block using dct2d() function I define earlier, and store each channel’s DCT coefficient to an numpy array.
- After it I keep only upper left nxn block coefficients in the 2D DCT domain for each 8x8 block by invoking keep_n_per_block() function, passing the result of DCT and n as arguments.

```python
#Quantize using quantization table
for j in range(0, y_lim, 8):
    for i in range(0, x_lim, 8):
        for y in range(0, 8):
            for x in range(0, 8):
                dct_img_r[(i+x), (j+y)] = round(dct_img_r[(i+x), (j+y)]/rgb_quantization_table[x, y])
                dct_img_g[(i+x), (j+y)] = round(dct_img_g[(i+x), (j+y)]/rgb_quantization_table[x, y])
                dct_img_b[(i+x), (j+y)] = round(dct_img_b[(i+x), (j+y)]/rgb_quantization_table[x, y])
```

- Then I quantize using rgb quantization table which I initialized earlier. I implement it using nested for loops, and divide each 8x8 block of the DCT coefficients with the 8x8 rgb quantization table.

```python
#uniform quantization in m bits
max_value = max(np.max(dct_img_r), np.max(dct_img_g), np.max(dct_img_b))
min_value = min(np.min(dct_img_r), np.min(dct_img_g), np.min(dct_img_b))
step_size = (max_value - min_value) / 2**m
offset = round(min_value / step_size)
# print(max_value, min_value, step_size)

for j in range(0, y_lim):
    for i in range(0, x_lim):
        dct_img_r[i, j] = round(dct_img_r[i, j]/ step_size) - offset
        dct_img_g[i, j] = round(dct_img_g[i, j]/ step_size) - offset
        dct_img_b[i, j] = round(dct_img_b[i, j]/ step_size) - offset
```

- After quantization with quantization table, I do another quantization which is uniform quantization in m bits. First, I calculate the step size which is max value – min value of coefficients in each channel, divided by 2^m. I also calculate the offset value which are going to be added to the final value of the quantized coefficients to make all of the coefficients to be non-negative values and easier to save as an image. Note that this offset will not increase the number of bits used, it will just shift the quantized coefficients to the positive side. I implement this uniform quantization by nested for loops, and I divide each DCT coefficients by the step size, and subtract it by the offset value.

```python
#combine 3 channels to 1 img
dct_img_rgb = np.zeros((x_lim, y_lim, 3))
# dct_img_rgb = input_img_rgb.copy() #just to get array with same size
for j in range(0, y_lim):
    for i in range(0, x_lim):
        dct_img_rgb[i, j][0] = dct_img_r[i, j]
        dct_img_rgb[i, j][1] = dct_img_g[i, j]
        dct_img_rgb[i, j][2] = dct_img_b[i, j]
```

- Finally, I combine all the 3 channel’s DCT coefficients to an image by using nested for loops. First I initialize an numpy array with the same size as input image, and for each pixel, I assign each channel’s DCT coefficients to be the value of this image. Then we are done for the DCT compression.

```python
#DECOMPRESS

#divide the img to 3 channels
idct_img_r = np.zeros((x_lim, y_lim))
idct_img_g = np.zeros((x_lim, y_lim))
idct_img_b = np.zeros((x_lim, y_lim))
for j in range(0, y_lim):
    for i in range(0, x_lim):
        idct_img_r[i, j] = dct_img_rgb[i, j][0]
        idct_img_g[i, j] = dct_img_rgb[i, j][1]
        idct_img_b[i, j] = dct_img_rgb[i, j][2]
```

- Before I start to decompress the compressed image, I first divide the compressed image to 3 channels using nested for loops, the separated results are “idct_img_r” for red channel, “idct_img_g” for green channel, and “idct_img_b” for blue channel. These values are still the quantized dct coefficients, I call this idct only to differentiate the compression and decompression process, and because these values are going to be applied idct later.

```python
#Uniform Unquantization
for j in range(0, y_lim):
    for i in range(0, x_lim):
        idct_img_r[i, j] = (idct_img_r[i, j] + offset)*step_size
        idct_img_g[i, j] = (idct_img_g[i, j] + offset)*step_size
        idct_img_b[i, j] = (idct_img_b[i, j] + offset)*step_size
```

- Then I unquantize the dct coefficients with uniform unquantization. I implement this by using nested for loops, and for each pixel, I add the values by offset values, and multiply it by step size.

```python
#Unquantize using quantization table
for j in range(0, y_lim, 8):
    for i in range(0, x_lim, 8):
        for y in range(0, 8):
            for x in range(0, 8):
                idct_img_r[(i+x), (j+y)] = idct_img_r[(i+x), (j+y)]*rgb_quantization_table[x, y]
                idct_img_g[(i+x), (j+y)] = idct_img_g[(i+x), (j+y)]*rgb_quantization_table[x, y]
                idct_img_b[(i+x), (j+y)] = idct_img_b[(i+x), (j+y)]*rgb_quantization_table[x, y]
```

- Then I unquantize the coefficients again with quantization table. I implement it using nested for loops, and multiply each 8x8 block of these coefficients with the 8x8 rgb quantization table.

```python
#IDCT for each channel
result_img_r = np.zeros((x_lim, y_lim))
result_img_g = np.zeros((x_lim, y_lim))
result_img_b = np.zeros((x_lim, y_lim))

for j in range(0, y_lim, 8):
    for i in range(0, x_lim, 8):
        result_img_r[i:(i+8), j:(j+8)] = idct2d(idct_img_r[i:(i+8), j:(j+8)])
        result_img_g[i:(i+8), j:(j+8)] = idct2d(idct_img_g[i:(i+8), j:(j+8)])
        result_img_b[i:(i+8), j:(j+8)] = idct2d(idct_img_b[i:(i+8), j:(j+8)])
```

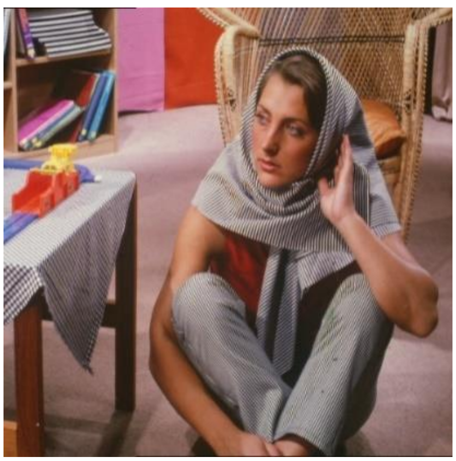
- Then I apply IDCT to these DCT coefficients to get the decompressed value of the image. For each channel’s DCT coefficients, I divide the coefficients into blocks of 8x8 pixels and apply 2D IDCT for each block using idct2d() function, and store each channel’s resulting value to a numpy array.

```python
#combine 3 channels to 1 img
result_img_rgb = input_img_rgb.copy() #just to make array with same size
for j in range(0, y_lim):
    for i in range(0, x_lim):
        R = result_img_r[i, j]
        G = result_img_g[i, j]
        B = result_img_b[i, j]
        #prevent overflow or underflow
        result_img_rgb[i, j][0] = R if R <= 255 and R >= 0 else 255 if R > 255 else 0 #R part
        result_img_rgb[i, j][1] = G if G <= 255 and G >= 0 else 255 if G > 255 else 0 #G part
        result_img_rgb[i, j][2] = B if B <= 255 and B >= 0 else 255 if B > 255 else 0 #B part
```

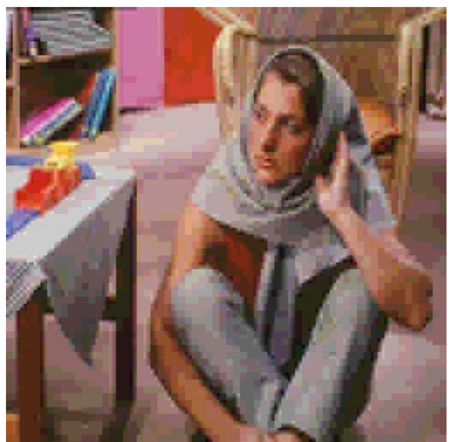
- Then I combine the 3 arrays that store the resulting value of idct to 1 image. I implement this by using nested for loops, and I set 255 as the upper limit and 0 as the under limit of the values to prevent overflow and underflow.

```
imshow(result_img_rgb)
cv2.imwrite('./output/'+img_name+'_n'+str(n)+'m'+str(m)+'_a.jpg', cv2.cvtColor(result_img_rgb, cv2.COLOR_RGB2BGR))

#compute compression ratios
compress_ratio = (x_lim*y_lim*8*3)/(x_lim*y_lim*m)
print('Compression ratio:', compress_ratio)

#compute PSNR values
mse = np.mean((input_img_rgb.astype(np.float32)-result_img_rgb.astype(np.float32))**2)
max_pixel = 255.0
psnr = 20*math.log10(max_pixel / math.sqrt(mse))
print('PSNR value:', psnr)
```

- Then I show the decompressed image, and convert it from RGB to BGR color channel before writing the image to a jpg file.
- And finally, I compute the compression ratios and PSNR values of the compressed image. Since in the original image each pixel uses 24 bits (8 bits x 3 channels), and in the compressed image each pixel only used m bits, so the compression ratio is 24/m. For PSNR values, I calculate it by first calculate the mean squared error value, and since the peak value of a pixel is 255, the PSNR is 20 multiplied by log 10 of (255/mse).



Original



n=2, m=4

Compression ratio: 6.0
PSNR value: 24.183193610466148

n=2, m=8

Compression ratio: 3.0
PSNR value: 25.612204866470023

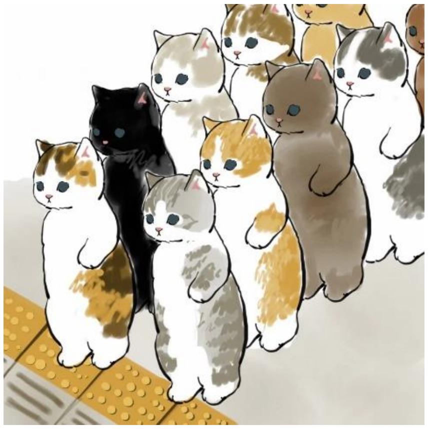
d

n=4, m=4

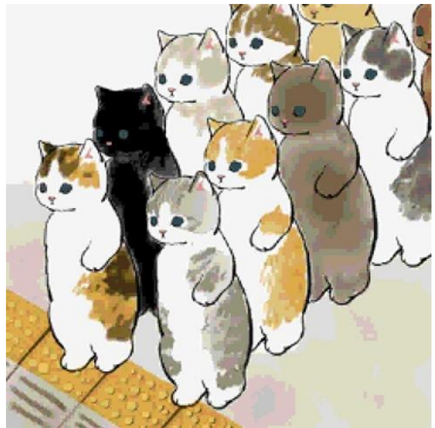
Compression ratio: 6.0
PSNR value: 24.802319105392108

n=4, m=8

Compression ratio: 3.0
PSNR value: 28.996261616913316



Original

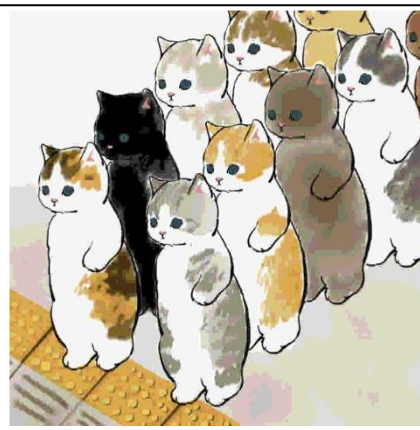


n=2, m=4

Compression ratio: 6.0
PSNR value: 22.869370524163557

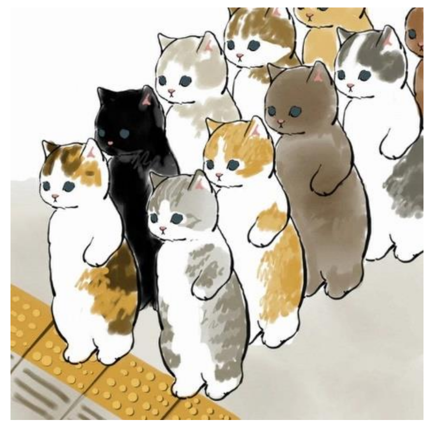
n=2, m=8

Compression ratio: 3.0
PSNR value: 24.110154650606578



n=4, m=4

Compression ratio: 6.0
PSNR value: 25.60261390794929



n=4, m=8

Compression ratio: 3.0
PSNR value: 33.16800271987338

Discussion for (a)

- When m=4, the compression ratio is 6.0, which causes the decompressed image to look really grainy. But for m=8, although the compression ratio is less with 3.0, but the decompressed image look more clearer and the difference with the original image is not that obvious.
- For m=4, the difference from changing n=2 to n=4 is not much for Barbara image, but a lot for cat image. For m=8, the difference from changing n=2 to n=4 is a lot for both Barbara image and cat image.
- For n=2, PSNR value of Barbara image is larger than PSNR value of cat image, meaning the distortion rate of the compressed Barbara image is less than the distortion rate of the compressed cat image. But for n=4, PSNR value of Barbara image is smaller than the cat image. Meaning the compression with n=2 works better for Barbara image and compression with n=4 works better for cat image.

(c) Use the same process in (a) with image transformed to YCbCr color space with 4:2:0 chrominance subsampling.

```python
#(b)

input_img_ycc = input_img_rgb.copy()
x_lim = input_img_ycc.shape[0]
y_lim = input_img_ycc.shape[1]

#convert input img from RGB color space to YCbCr color space
for j in range(0, y_lim):
    for i in range(0, x_lim):
        input_img_ycc[i, j][0] = 16+ 0.257*input_img_rgb[i, j][0] + 0.564*input_img_rgb[i, j][1] + 0.098*input_img_rgb[i, j][2] #Y part
        if j%2 == 0 and i%2 == 0: #sample chrominance
            Cb = 128 - 0.148*input_img_rgb[i, j][0] - 0.291*input_img_rgb[i, j][1] + 0.439*input_img_rgb[i, j][2]
            Cr = 128 + 0.439*input_img_rgb[i, j][0] - 0.368*input_img_rgb[i, j][1] - 0.071*input_img_rgb[i, j][2]
            #4:2:0 chrominance subsampling
            for y in range(j, j+2):
                for x in range(i, i+2):
                    if y < y_lim and x < x_lim:
                        input_img_ycc[x, y][1] = Cb
                        input_img_ycc[x, y][2] = Cr
```

- For part (b), I first convert the image from RGB color space to YCbCr color space with 4:2:0 chrominance subsampling. I implement this by using nested for loops, and for every pixel I calculate the Y value of the pixel with the formula provided by the TA in discussion forum. And if the indexes of the pixel in x-axis and y-axis are both multiply of 2, I calculate the chrominance value by the formula, and use this chrominance value for the pixel, the pixel's right pixel, the pixel under, and the pixel on right under side of this pixel.

```python
#COMPRESS
lumin_quantization_table = np.array([[16,11,10,16,24,40,51,61],
                                     [12,12,14,19,26,58,60,55],
                                     [14,13,16,24,40,57,69,56],
                                     [14,17,22,29,51,87,80,62],
                                     [18,22,37,56,68,109,103,77],
                                     [24,36,55,64,81,104,113,92],
                                     [49,64,78,87,103,121,120,101],
                                     [72,92,95,98,112,100,103,99]])

chrom_quantization_table = 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]])
```

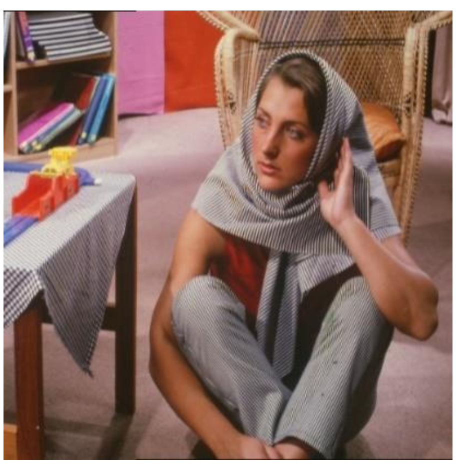
- Then I initialize the luminance quantization table and chrominance quantization table.
- After it I compress the image in YCbCr color space the similarly as in (a), the only differences are when I quantize a value using quantization table, instead of using rgb quantization table, I use these luminance and chrominance quantization table. And another difference is for the uniform quantization, because the value of the luminance part and chrominance part differs quite a lot, I separate the uniform quantization to 2 part, one for the luminance part and one for the chrominance part. Note that separating the uniform quantization won't increase the number of bits required for each pixel, as both part are using the same range of numbers.

```python
#IDCT for each channel
result_img_y = np.zeros((x_lim, y_lim))
result_img_cb = np.zeros((x_lim, y_lim))
result_img_cr = np.zeros((x_lim, y_lim))
for j in range(0, y_lim, 8):
    for i in range(0, x_lim, 8):
        result_img_y[i:(i+8), j:(j+8)] = cv2.idct(idct_img_y[i:(i+8), j:(j+8)])
        result_img_cb[i:(i+8), j:(j+8)] = cv2.idct(idct_img_cb[i:(i+8), j:(j+8)])
        result_img_cr[i:(i+8), j:(j+8)] = cv2.idct(idct_img_cr[i:(i+8), j:(j+8)])



#convert img from YCbCr color space to RGB color space
converted_img = input_img_rgb.copy() #just to make array with same shape

for j in range(0, y_lim):
    for i in range(0, x_lim):
        R = 1.164*(result_img_y[i, j]-16) + 0*(result_img_cb[i, j]-128) + 1.596*(result_img_cr[i, j]-128)
        G = 1.164*(result_img_y[i, j]-16) - 0.382*(result_img_cb[i, j]-128) - 0.813*(result_img_cr[i, j]-128)
        B = 1.164*(result_img_y[i, j]-16) + 2.017*(result_img_cb[i, j]-128) + 0*(result_img_cr[i, j]-128)
        #prevent overflow or underflow
        converted_img[i, j][0] = R if R <= 255 and R >= 0 else 255 if R > 255 else 0 #R part
        converted_img[i, j][1] = G if G <= 255 and G >= 0 else 255 if G > 255 else 0 #G part
        converted_img[i, j][2] = B if B <= 255 and B >= 0 else 255 if B > 255 else 0 #B part
```

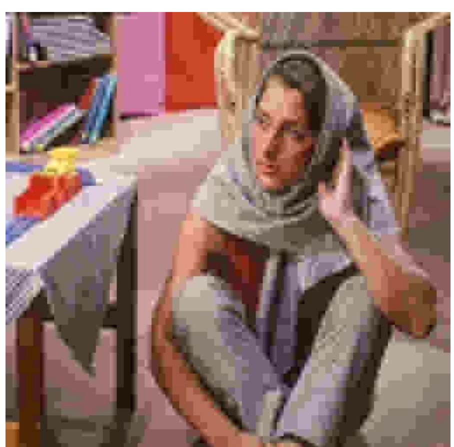
- After applying IDCT to each channel, I convert the image from YCbCr color space back to RGB color space. I implement this using nested for loops. For each pixel I calculate the R, G, and B values from Y, Cb, Cr value using the formula, and with similar ways as in part (a) I prevent overflow and underflow when assigning the values of R, G, and B to the converted image.

```python
imshow(converted_img)
cv2.imwrite('./output/'+img_name+'_n'+str(n)+'m'+str(m)+'_b.jpg', cv2.cvtColor(converted_img, cv2.COLOR_RGB2BGR))

#compute compression ratios
compress_ratio = (x_lim*y_lim*8*3)/(x_lim*y_lim*m*2)
print('Compression ratio:', compress_ratio)

#compute PSNR values
mse = np.mean((input_img_rgb.astype("float")-converted_img.astype("float"))**2)
max_pixel = 255.0
psnr = 20*math.log10(max_pixel / math.sqrt(mse))
print('PSNR value:', psnr)
```

- Finally, I show the image and convert it to BGR color space before writing it as jpg file, and I calculate the compression ratio and PSNR values of the compressed image.



Original

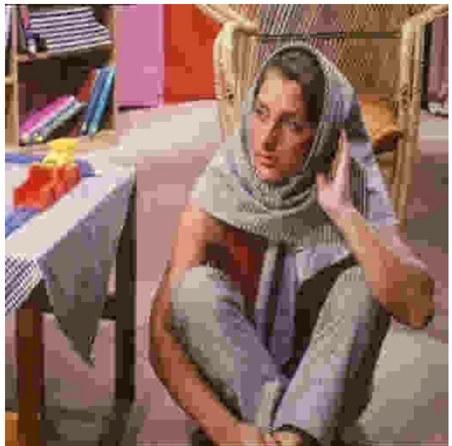


n=2, m=4

Compression ratio: 6.0
PSNR value: 22.867156239363812

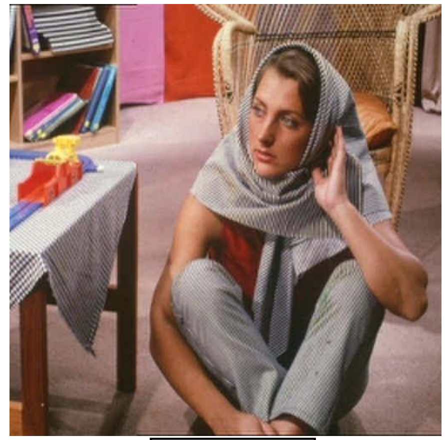
n=2, m=8

Compression ratio: 3.0
PSNR value: 24.59709866216589



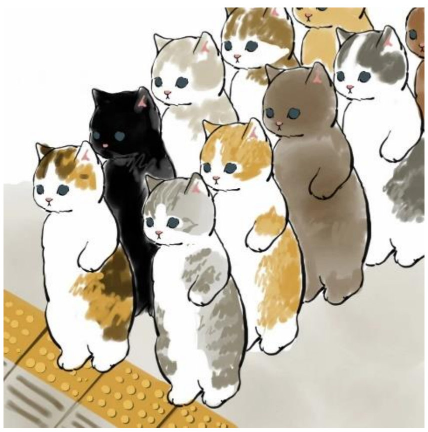
n=4, m=4

Compression ratio: 6.0
PSNR value: 23.468393955946013



n=4, m=8

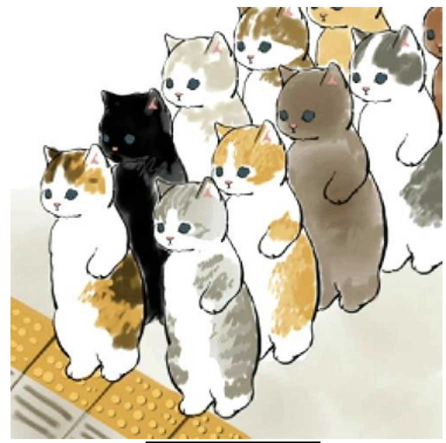
Compression ratio: 3.0
PSNR value: 26.906162649719263



Original

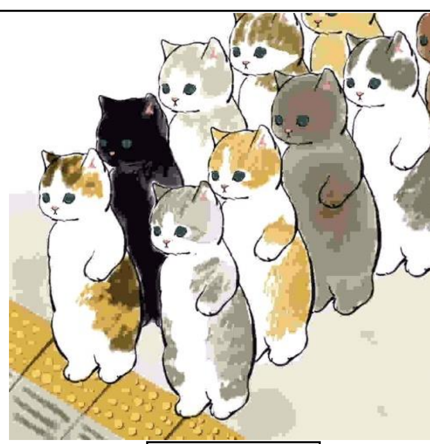
n=2, m=4

Compression ratio: 6.0
PSNR value: 21.980021548395428



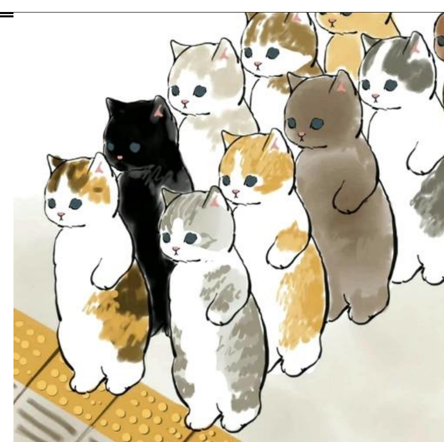
n=2, m=8

Compression ratio: 3.0
PSNR value: 23.133265577842977



n=4, m=4

Compression ratio: 6.0
PSNR value: 24.29434131681066



n=4, m=8

Compression ratio: 3.0
PSNR value: 28.205142267360635

Discussion for (b)

- When m=4, the compression ratio is 6.0, which causes the decompressed image to look really grainy. But for m=8, although the compression ratio is less with 3, but the decompressed image looks more clearer and the difference with the original image is not that obvious.
- For m=4, the difference from changing n=2 to n=4 is not much for Barbara image, but a lot for cat image. For m=8, the difference from changing n=2 to n=4 is a lot for both Barbara image and cat image.
- For n=2, PSNR value of Barbara image is larger than PSNR value of cat image, meaning the distortion rate of the compressed Barbara image is less than the distortion rate of the compressed cat image. But for n=4, PSNR value of Barbara image is smaller than the cat image. Meaning the compression with n=2 works better for Barbara image and compression with n=4 works better for cat image.

Discussion

- The decompressed image with compression on YCbCr color spaces are more blocky when m=4, and a little bit more yellowish than the decompressed image with compression on RGB color spaces.
- I think the compression quality with RGB color space is better than compression on YCbCr color space as the PSNR values of images compressed in RGB colos space are larger than PSNR values of images compressed in YCbCr color space.

2. Create your own FIR filters to filter audio signal

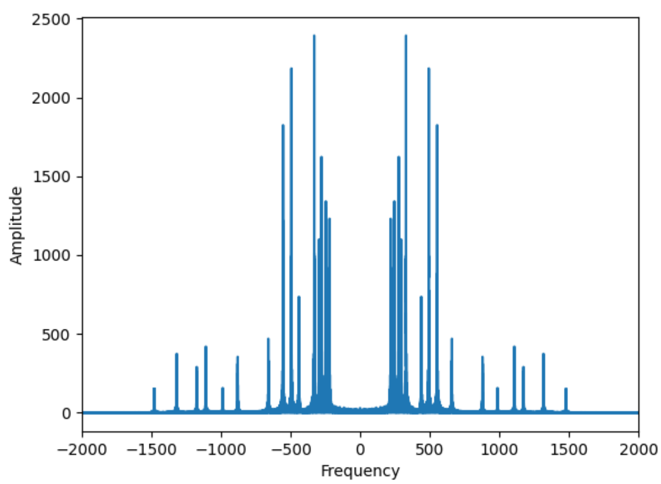
```
Q2 > 2.py > ...
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import cv2
import math
import scipy.io.wavfile

audio = scipy.io.wavfile.read("./HW2_Mix_2.wav")
sampling_rate = audio[0]
print('Audio sampling rate:', sampling_rate)
audio = np.array(audio[1], dtype=float)

#TIME DOMAIN SIGNAL
plt.plot(audio[:])
plt.ylabel("Amplitude")
plt.xlabel("Time")
plt.show()
```

- Before starting, first I import needed modules such as numpy, matplotlib, math, and scipy.io.wavfile. Then I read the audio to be processed with scipy.io.wavfile.read() function, the return value of this function is an array with 2 elements, the first element is the sampling rate and the second element is the signal data of the audio.

- Then I using matplotlib module, I plot the signal data of the audio and show it.

```
#FREQUENCY DOMAIN SIGNAL
# Calculate n/2 to normalize the FFT output
n = audio.size
normalize = n/2
fft_signal = np.fft.fft(audio)
fft_freq = np.fft.fftfreq(n, d=1.0/sampling_rate)

plt.plot(fft_freq, np.abs(fft_signal/normalize))
plt.ylabel("Amplitude")
plt.xlabel("Frequency")
plt.xlim(-2000, 2000)
plt.savefig('./output/input.png')
plt.show()
```

- I transform the time domain signal to Frequency domain signal using Fast Fourier Transform. I implement fast fourier transform by using np.fft.fft() and np.fft.fftfreq() functions to get the values on x-axis and y-axis, and I then plot them and save the plot as "input.png". Here before saving I set the x-axis min limit to -2000 and max limit to 2000 to make the signal larger and clearer to see.



- Based on the frequency domain graph, I notice that the frequencies is divided into 3 big parts, around 0Hz-380Hz, 380Hz-780Hz, and 780Hz-1550Hz. So, based on this observation I decided to use lowpass filter as the first filter to filter frequencies ranging in 0Hz-380Hz, bandpass filter as the second filter for frequencies ranging around 380Hz-780Hz, and highpass filter as the third filter for frequencies at and larger than 780Hz.

```python
# FILTER 1 (LOWPASS FILTER)
cutoff_freq = 380
cutoff_freq = cutoff_freq/sampling_rate
w_c = 2*math.pi*cutoff_freq
filter_order = 32000
middle = filter_order//2
low_filter = np.zeros((filter_order), dtype=float)

for i in range(-middle, middle):
    if i == 0:
        low_filter[middle] = 1
    else:
        low_filter[i+middle] = math.sin(2*math.pi*cutoff_freq*i)/(math.pi*i)
low_filter[middle] = 2*cutoff_freq

#multiply the elements of filter by a windowing function
#blackmann window function
for i in range(0, filter_order):
    low_filter[i] = low_filter[i] * (0.42 + 0.5*math.cos((2*math.pi*i)/(n-1)) + 0.08*math.cos((4*math.pi*i)/(n-1)))
```

- For the first filter (lowpass filter), I implement it by first designing the filter. I set the cutoff frequency and the filter order of the filter, and initialize an numpy array to store the coefficients of the filter. Then I use for loops to set the filter coefficient values using the formula sin(2*pi*cutoff_freq*i)/(pi*i).
- After the for loops is done, I again use another for loop to multiply the element of filter by a blackmann windowing function. For each element in the filter, I multiply it with the blackmann windowing formula.

```python
#1D CONVOLUTION
filtered_signal_1 = audio.copy() #just to get array with same size
# Pad the input signal with zeros to compute the convolution in the "same" mode
pad_length = len(low_filter) // 2
audio_padded = np.pad(audio, (pad_length, pad_length), mode='constant')
# Compute the convolution
for i in tqdm.trange(len(audio)):
    filtered_signal_1[i] = np.dot(audio_padded[i:i+len(low_filter)], low_filter)
```

- After the filter is completely initialized, I start the 1D convolution. The idea to implement it is to first pad the input signal with zeros so that the output signal has the same length as the input signal, then I perform the convolution. In the code, I first compute the padding length as half the length of the filter, then I pad the input signal with zeros using the np.pad(), and I computue the convolution by iterating over the indices of the output signal and using the np.dot() function to compute the dot product of the corresponding slice of the padded input signal and the filter.

```python
#output audio
scipy.io.wavfile.write('./output/Filter1Lowpass_380.wav', sampling_rate, np.int16(filtered_signal_1))

#output spectrums
# Calculate n/2 to normalize the FFT output
n = filtered_signal_1.size
normalize = n/2
fft_signal_1 = np.fft.fft(filtered_signal_1)
fft_freq_1 = np.fft.fftfreq(n, d=1.0/sampling_rate) #(?)

plt.plot(fft_freq_1, np.abs(fft_signal_1/normalize))
plt.ylabel("Amplitude")
plt.xlabel("Frequency")
plt.title('Output spectrum by filter 1 (Lowpass)')
plt.xlim(-2000, 2000)
plt.savefig('./output/output_by_Filter1Lowpass.png')
plt.show()
```

- After having the output signal saved in “filtered_signal_1” array, I write the output signal using scipy.io.wavfile.write() function.

- I also transform the output signal to its frequency domain signal with similar ways as when I transform the input signals, and I save the figure before showing it.

```python
#filter shape
plt.plot(low_filter)
plt.title('Filter 1 (Lowpass) shape')
plt.xlim(11000, 21000)
plt.savefig('./output/Filter1Lowpass_shape.png')
plt.show()

#filter spectrums
n = filter_order
normalize = n/2
fft_signal_1 = np.fft.fft(low_filter)
fft_freq_1 = np.fft.fftfreq(n, d=1.0/sampling_rate) #(?)
plt.plot(fft_freq_1, np.abs(fft_signal_1/normalize))
plt.title('Filter 1 (Lowpass) filter spectrum')
plt.ylabel("Amplitude")
plt.xlabel("Frequency")
plt.xlim(-2000, 2000)
plt.savefig('./output/Filter1Lowpass_spectrum.png')
plt.show()
```

- I also plot the shape of the filter in time domain, and set the x-axis limit to 11000 until 21000 to make it clearer, and save the figure before showing it.
- Then I transform the filter to its frequency domain with fast fourier transform, set the x-axis limit to -2000 until 2000, and save the figure before showing it.

```python
#DOWN SAMPLE 2KHZ
original_sampling_rate = sampling_rate
new_sampling_rate = 2000
downsample_factor = original_sampling_rate / new_sampling_rate
num_samples_downsampled = int(len(filtered_signal_1)/downsample_factor)
downsampled_signal = np.zeros((num_samples_downsampled), dtype=float)
#downsample the signal
for i in range(num_samples_downsampled):
    downsampled_signal[i] = filtered_signal_1[int(i * downsample_factor)]
#output audio
scipy.io.wavfile.write('./output/Filter1Lowpass_380_2khz.wav', new_sampling_rate, np.int16(downsampled_signal))
```

- To down sample the filtered signal to 2000Hz, I first set the audio sampling rate as the original sampling rate and 2000 as the new sampling rate. Then I calculate the down sample factor by dividing original sampling rate by new sampling rate.
- After it I calculate the number of samples there will be after the down sample, I calculate this by dividing the length of the filtered audio by the down sample factor. With this number I create a new numpy array to later store the down sampled signal.
- Then using for loop, I set the i-th index value of down sample signal to be the value of the (i*downsample factor)-th index value of the filtered signal.
- Then I write the down sampled signal as a wav file.

```python
#ONE-FOLD ECHO
one_echo_filter = np.zeros((3201), dtype=float)
one_echo_filter[0] = 1
one_echo_filter[3200] = 0.8

one_echo_filtered_signal = filtered_signal_1.copy() #just to get array with same size
# Pad the input signal with zeros
pad_length = len(one_echo_filter) // 2
audio_padded = np.pad(filtered_signal_1, (pad_length, pad_length), mode='constant')
# Compute the convolution
for i in tqdm.trange(len(audio)):
    one_echo_filtered_signal[i] = np.dot(audio_padded[i:i+len(one_echo_filter)], one_echo_filter)
#output audio
scipy.io.wavfile.write('./output/Echo_one.wav', sampling_rate, np.int16(one_echo_filtered_signal))
```

- I also apply one-fold echo and multiple-fold echo to the filtered signal before downsampling. I implement one-fold echo by first initializing the one-echo filter, I do this by creating a numpy array of zeros with length 3201, and float as the data type. Then I set the first element of this filter to be 1 and last element of this filter to be 0.8.

- Then using the same way as the lowpass filter convolution earlier, I compute the one-echo filtered signal. The only difference is the filter I use for the convolution is one-echo filter instead of lowpass filter, and the filtered signal is the low pass filtered signal instead of the original audio.

- Then I write the one-echo filtered signal as wavfile.

```
#MULTIPLE-FOLD ECHO
multi_echo_filter = np.zeros((3201), dtype=float)
multi_echo_filter[0] = 1
multi_echo_filter[3200] = -0.8

multi_echo_filtered_signal = filtered_signal_1.copy() #just to get array with same size
# Pad the input signal with zeros
pad_length = len(multi_echo_filter) // 2
audio_padded = np.pad(filtered_signal_1, (pad_length, pad_length), mode='constant')
# Compute the convolution
for i in tqdm.trange(len(audio)):
    multi_echo_filtered_signal[i] = np.dot(audio_padded[i:i+len(multi_echo_filter)], multi_echo_filter)
#output audio
scipy.io.wavfile.write('./output/Echo_multiple.wav', sampling_rate, np.int16(multi_echo_filtered_signal))
```

- To apply the multiple-fold echo to the filtered signal before down sampling, I implement it the as how I implement the one-fold echo, the only difference is the filter. For the multi-echo filter, the last element value is -0.8 instead of 0.8 in one-fold echo.

```
# FILTER 2 (BANDPASS FILTER)
freq_1 = 380
freq_2 = 780
freq_1 = freq_1/sampling_rate
freq_2 = freq_2/sampling_rate
filter_order = 32000
middle = filter_order//2
bandpass_filter_2 = np.zeros((filter_order), dtype=float)

for i in range(-middle, middle):
    if i == 0:
        bandpass_filter_2[middle] = 1
    else:
        bandpass_filter_2[i+middle] = math.sin(2*math.pi*freq_2*i)/(math.pi*i) - math.sin(2*math.pi*freq_1*i)/(math.pi*i)
bandpass_filter_2[middle] = 2*(freq_2-freq_1)

#blackmann window function
for i in range(0, filter_order):
    bandpass_filter_2[i] = bandpass_filter_2[i] * (0.42 + 0.5*math.cos((2*math.pi*i)/(n-1)) + 0.08*math.cos((4*math.pi*i)/(n-1)))
```

- For the second filter, which is bandpass filter for frequencies ranging in 380Hz-780Hz, I implement it by first initializing a numpy array with filter order as its length to store the coefficients of this bandpass filter. I also set the 380 as the frequency 1 and 780 as frequency 2.
- Then I use nested for loops to set the filter coefficient values using the formula sin(2*pi*frequency 1*i)/(pi*i) minus sin(2*pi*frequency 2*i)/(pi*i).
- After the for loops is done, I again use another for loop to multiply the element of filter by a blackmann windowing function. For each element in the filter, I multiply it with the blackmann windowing formula.

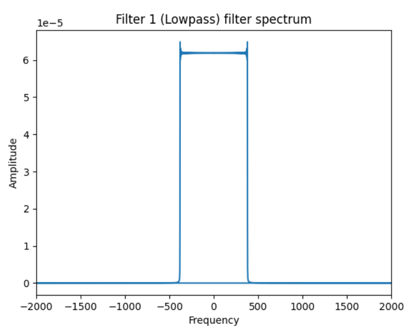
```
filtered_signal_2 = audio.copy() #just to get array with same size
# Pad the input signal with zeros
pad_length = len(bandpass_filter_2) // 2
audio_padded = np.pad(audio, (pad_length, pad_length), mode='constant')
# Compute the convolution
for i in tqdm.trange(len(audio)):
    filtered_signal_2[i] = np.dot(audio_padded[i:i+len(bandpass_filter_2)], bandpass_filter_2)
```

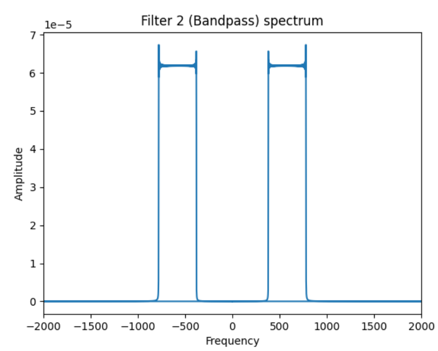
- Then I compute the convolution of the original audio and the filter, save it to “filtered_signal_2” array, and write it as wav file.
- Then I show the output spectrums, filter shape, and filter spectrums and save it as png file with the same way I do it in the first filter.
- I also down sample this bandpass filtered signal to 2000Hz, and write it as wav file. I implement this with the same way when I down sample the lowpass filtered signal to 2000Hz.

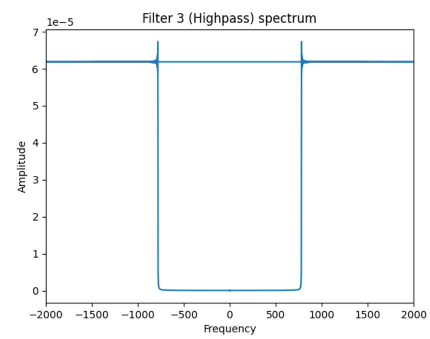
```
#FILTER 3 (HIGHPASS FILTER)
cutoff_freq = 780
cutoff_freq = cutoff_freq/sampling_rate
w_c = 2*math.pi*cutoff_freq
filter_order = 32000
middle = filter_order//2
highpass_filter = np.zeros((filter_order), dtype=float)

for i in range(-middle, middle):
    if i == 0:
        highpass_filter[middle] = 1
    else:
        highpass_filter[i+middle] = -math.sin(2*math.pi*cutoff_freq*i)/(math.pi*i)
highpass_filter[middle] = 1 - 2*cutoff_freq
```

- As for the 3rd filter which is a highpass filter, I implement it similarly as the way I implement the first filter. The only difference is the value of the cutoff frequency which is 780 here instead of 380, and the formula to calculate the filter coefficient is changed to -sin(2*pi*cutoff_freq*i)/(pi*i), and 1-2*cutoff freq for when i = 0.
- For the other part it is all the same as filter 1.
- I also down sample this bandpass filtered signal the same way as I down sample filtered signal in other filters.

Discussion

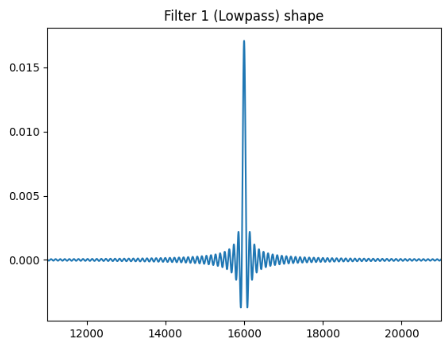


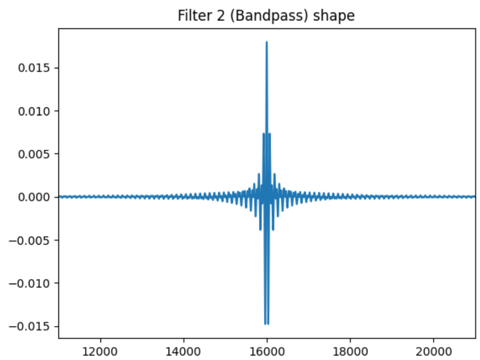


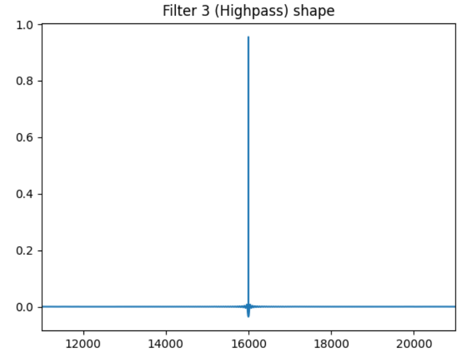


- The spectrum of the second filter is separated to two blocks like the 2 other filters as the second filter filters frequencies ranging in the middle (380Hz-780Hz).

- The total width of a block in spectrum of the first and second filter is narrower than the total width of a block in spectrum of the third filter. This is because the range of frequencies that the third filter filters on (780Hz-22500Hz) is larger than the other 2 filters.







- The higher the frequencies a filter is filtering on, the tighter the filter shape. I think it is because higher frequencies will have more waves density, which causes the shape of the filter to be tighter.

Briefly compare the difference between signals before and after reducing the sampling rates.

- For the signals filtered by first filter and second filter which are signals with frequencies ranging around 0Hz-380Hz and 380Hz-780Hz respectively, down sampling to 2000Hz have no significant difference to our ears, as according to Nyquist's sampling theorem, 2000Hz is more than twice the highest frequency component in the signals filtered by first filter and second filter.
- But for signals filtered by third filter, which are signals with frequencies ranging around 780Hz-1550Hz, down sampling to 2000Hz will results in aliasing as 2000Hz is less than twice the highest frequency component in the signals filtered, and there aren't enough sample points from which to accurately interpolate the sinusoidal form of the filtered wave.