

## 1. DCT image compression

```

1 def dct2(block):
2     n = 8
3     dct = np.zeros((n, n))
4     for u in range(n):
5         for v in range(n):
6             cu = 1 / np.sqrt(2) if u == 0 else 1
7             cv = 1 / np.sqrt(2) if v == 0 else 1
8             for i in range(n):
9                 for j in range(n):
10                    dct[u,v] += 0.25 * cu * cv * block[i,j] * np.cos((2*i+1)*u*np.pi/16) * np.cos((2*j+1)*v*np.pi/16)
11    return dct

1 def idct2(dct):
2     n = 8
3     block = np.zeros((n, n))
4     for i in range(n):
5         for u in range(n):
6             for v in range(n):
7                 cu = 1 / np.sqrt(2) if u == 0 else 1
8                 cv = 1 / np.sqrt(2) if v == 0 else 1
9                 block[i,j] += 0.25 * cu * cv * dct[u,v] * np.cos((2*i+1)*u*np.pi/16) * np.cos((2*j+1)*v*np.pi/16)
10    return block

1 def subsampling(ycbcr):
2     y = ycbcr[:, :, 0]
3     cb, cr = ycbcr[:, :, 1], ycbcr[:, :, 2]
4     return y, cb, cr

1 def compressed(image, n, w, table):
2     if len(image.shape) == 2:
3         h, w = image.shape
4         d = 1
5         image = np.expand_dims(image, axis=-1)
6     else:
7         h, w, d = image.shape
8         # divide the image into 8x8 pixels
9         num_blocks_h = h // 8
10        num_blocks_w = w // 8
11        max_value = np.array([1, -1, -1])
12        min_value = np.array([260, 260, 260])
13
14    dct_blocks = np.zeros((h, w, d))
15    quantized_blocks = np.zeros((h, w, d))
16    reconstructed_image = np.zeros((h, w, d))
17    for channel in range(d):
18        for i in range(num_blocks_h):
19            for j in range(num_blocks_w):
20                # subtract 128 from each pixel to fit dct domain
21                block = image[i*8:(i+1)*8, j*8:(j+1)*8, channel]
22                dct = dct2(block)
23                # lower-frequency coefficients
24                dct[:, :] = 0
25                dct[:, n:] = 0
26                dct_blocks[i*8:(i+1)*8, j*8:(j+1)*8, channel] = dct
27        for i in range(num_blocks_h):
28            for j in range(num_blocks_w):
29                dct = dct_blocks[i*8:(i+1)*8, j*8:(j+1)*8, channel]
30                q_block = np.round(dct / table)
31                if q_block[:, n:].max() > max_value[channel]:
32                    max_value[channel] = q_block[:, n:].max()
33                if q_block[:, n:].min() < min_value[channel]:
34                    min_value[channel] = q_block[:, n:].min()
35                quantized_blocks[i*8:(i+1)*8, j*8:(j+1)*8, channel] = q_block
36        step = (max_value[channel] - min_value[channel]) / (2**m - 1)
37        quantized_blocks = np.round(quantized_blocks / step) * step
38        for i in range(num_blocks_h):
39            for j in range(num_blocks_w):
40                block = quantized_blocks[i*8:(i+1)*8, j*8:(j+1)*8, channel]
41                block *= table
42                block = idct2(block)
43                block = block.astype(np.int16)
44                reconstructed_image[i*8:(i+1)*8, j*8:(j+1)*8, channel] = block
45
46    if d == 1:
47        reconstructed_image = np.squeeze(reconstructed_image, axis=-1)
48    return reconstructed_image

```

Firstly, I divided the image into 8x8 pixels as a block. For each block I do dct transform and save the lower n coefficients. Then I quantize each block with a quantization table, and calculate the min and max value after quantizing to compute step size. Next, the whole compressed image divide step\_size to save in the m bit. Finally do each step inverse and roll back to reconstruct the image. For part-b, it will convert rgb to ycbcr at first, then subsample cb and cr channel with half bits. Next, put y, cb, cr channel into compress function with paired quantization table and do the same thing as part-a. Finally upsample the channel and convert back to RGB image.

1. Show the reconstructed image (part-a) (n2m4, n2m8, n4m4, n4m8)



2. Compute compressed ratio and PSNR value

```

1 def psnr(original, compressed):
2     mse = np.mean((original - compressed)**2)
3     if mse == 0:
4         return 100
5     max_pixel = 255.0
6     return 20 * np.log10(max_pixel / np.sqrt(mse))

```

PSNR_cat.jpeg_n2_m4_a:	28.746293879028574
PSNR_cat.jpeg_n2_m8_a:	28.542108565620875
PSNR_cat.jpeg_n4_m4_a:	28.543641740409914
PSNR_cat.jpeg_n4_m8_a:	28.557537955824365
PSNR_Barbara.jpeg_n2_m4_a:	28.467302620286862
PSNR_Barbara.jpeg_n2_m8_a:	28.483469385968135
PSNR_Barbara.jpeg_n4_m4_a:	28.313417849010992
PSNR_Barbara.jpeg_n4_m8_a:	28.356640226091553

PSNR value is computed by above code.

Compressed ratio is related with n and m. The original image has 512\*512 pixel and 8bits/channel. The original block is divided to 8, and n is set as 2 or 4. If n = 2, it compress 16 times, else if n=4, it compress 4 times. Look at m, the original is 8bits, and set m as 4 or 8. If m = 8 then no compress here, else if m = 4, it compress 2 times. As the result, the image compressed ratio is: n2m4=32 times, n2m8=16 times, n4m4=8 times, n4m8=2 times.

3. Show the reconstructed RGB image (part-b)



4. Compute compressed ratio and PSNR value

```
PSNR_cat.jpeg_n2_m4_b: 28.64028999451119
PSNR_cat.jpeg_n2_m8_b: 28.43023065292798
PSNR_cat.jpeg_n4_m4_b: 28.69028959062699
PSNR_cat.jpeg_n4_m8_b: 28.724335150836144
PSNR_Barbara.jpeg_n2_m4_b: 27.84843579459112
PSNR_Barbara.jpeg_n2_m8_b: 27.854501481529322
PSNR_Barbara.jpeg_n4_m4_b: 27.790839738255745
PSNR_Barbara.jpeg_n4_m8_b: 27.847072988764296
```

The PSNR value is computed by the same method.

The compressed ratio is also similar with above one, however, it consider subsample so it only need half bits in cb and cr channel. It can compress more 1.5 time than part-a

5. Discussion

The result of part-a and part-b look really similar. But the brightness is different. The images which are produced in part-a are darker than produced in part-b. I consider that be affected by different quantized table. In part a, three channel use the same quantized table, however in part-b y channel apply to luminance table and cb cr channel apply to chrominance table. That is, in part-b it has less bit to focus on color but more channel to focus on bright. Moreover, I found in small n and small m condition, the images in part-b would miss a few color, some color would become white. I think it's affected by upsampling. Because of doing upsample, some pixel is miss and difficult to recover especially in small n and m.

## 2. Create your own FIR filters to filters audio signal


1. Discuss how you determine the filters:

Firstly, I analysis the spectrum of the input signal to identify the frequencies of the mixed song. Then, design 3 filters, which are low-pass, band-pass, and high-pass, with cut-off frequencies and window-size by Blackman window function.

2. How you implement the filter and convolutions to separate the mixed song and one.multiple fold echo?




```
1 def sin_fn(x):
2     return np.where(x == 0, 1.0, np.sin(x) / x)
```



```
1 def blackman_window(N):
2     n = np.arange(N)
3     return (0.42 - 0.5*np.cos(2*np.pi*n/(N-1)) + 0.08*np.cos(4*np.pi*n/(N-1)))
```

I designed the Blackman window function. For each filters, there is a Blackman window and applies the filter coefficients, which are calculated by sin function.



```
1 def low_pass_filter(fc, N, rate):
2     n = np.arange(N)
3     # Calculate the filter coefficients
4     h = 2 * fc/rate * sin_fn(2*np.pi*fc*(n-(N-1)/2)/rate)
5     h = h * blackman_window(N)
6     return h
7
8 def high_pass_filter(fc, N, rate):
9     # low-pass filter
10    lp_filter = low_pass_filter(fc, N, rate)
11    # Create high-pass filter
12    hp_filter = -lp_filter
13    hp_filter[N // 2] += 1
14    return hp_filter
15
16 def band_pass_filter(f1, f2, N, rate):
17    return low_pass_filter(f2, N, rate) - low_pass_filter(f1, N, rate)
```

Then, I implement 3 filters one by one. The first one is low-pass filter. I create a filter kernel with sin function, frequency and Blackman window. The second one is high-pass filter. It create a low-pass filter and subtracted it from a delta function to implement high-pass filter. The last one is band-pass filter, it is created by a low-pay and a high-pass filter.

I implement the convolution to fit the input signal and filters. It create an array with the input



```
1 def conv1(signal, filter):  
2     conv_sig = np.zeros_like(signal)  
3     # avoid edge effects  
4     N = len(filter)  
5     padded_sig = np.pad(signal, (N // 2, N // 2), mode="constant")  
6     for i in range(len(signal)):  
7         conv_sig[i] = np.sum(padded_sig[i:i + N] * filter)  
8     return conv_sig
```

signal and padded 0 to avoid edge effect. Then, using loop to go through each sample and computing the result by dot product.

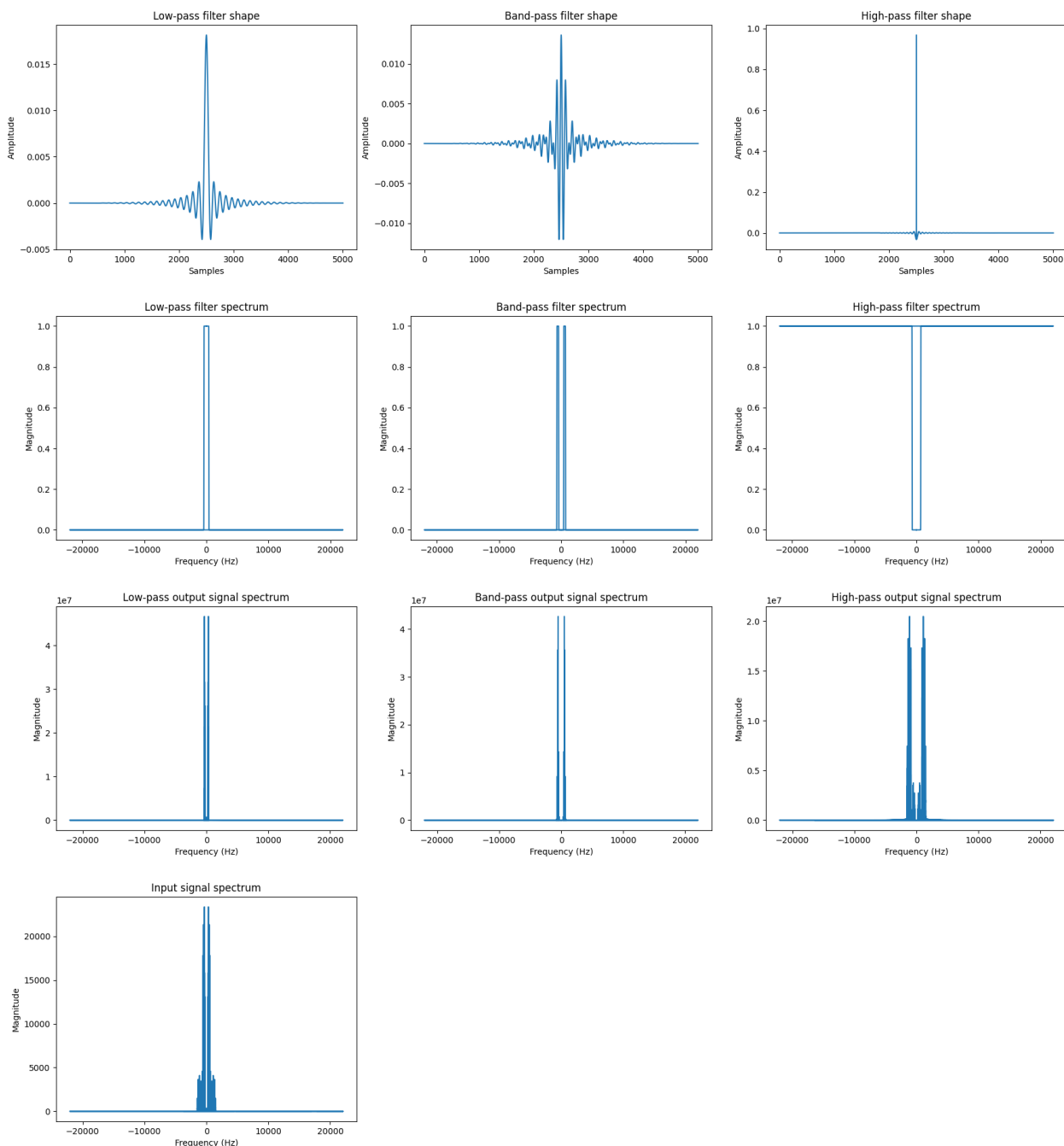
```
1 def apply_echo(signal, delay, alpha):  
2     echo_sig = np.zeros_like(signal)  
3     echo_sig[delay:] = signal[:-delay] * alpha  
4     return signal + echo_sig
```

To implement the one-fold echo, I create an array to write the delay signal. With multiplying signal value and attenuation factor(alpha), it can get the delay signal then plus with original signal.

```
1 echo_multiple = echo_one.copy()  
2 for i in range(2, 5):  
3     echo_multiple = apply_echo(echo_multiple, echo_delay * i, echo_alpha)
```

To implement multiple-fold echo, I use the same way described above as basis. Additionally, using loop to apply the process repeatedly and apply previous one echo signal.

## 3. Compare the spectrum and shape of the filters



The low-pass filter has a gradual shift in its passband to the stopband, decreasing the intensity of higher frequencies. The band-pass filter, meanwhile, allows a specific frequency range to pass through while also reducing the intensity of both lower and higher frequencies. The high-pass filter suppresses lower frequencies while permitting higher frequencies to pass through. Furthermore, the filters' shapes in the time domain reflect their impulse response, providing insight into their ringing and response traits.

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

Upon comparing the signals prior and after applying a reduction in sampling rates, I have noticed a discernible decrease in the frequency content of the downsampled signals. This decrease in frequency content results in a loss of high-frequency information, but it also leads to a smaller file

size, making it easier to store and transmit the data.

As per the Nyquist theorem, the downsampled signals exhibit a maximum frequency of 1000 Hz, while the original signals possess higher frequency components. This means that the downsampled signals can't represent high-frequency content as accurately as the original signals. However, in certain applications, such as voice calls or audio streaming, it may not be necessary to have high-frequency content, and therefore, downsampling may be a reasonable option.

Despite losing some high-frequency information, the main features of the songs remain preserved at the lower sampling rate, allowing us to appreciate the melody and rhythm of the music without compromising on the song's essential qualities. Additionally, downsampling can also help reduce computational complexity in signal processing, making it an important tool in modern audio processing.