multimedia-hw2

by 108062138 Po-Yu, Wu

Readme.md (http://Readme.md) contains only the *execution* part in `report.pdf`

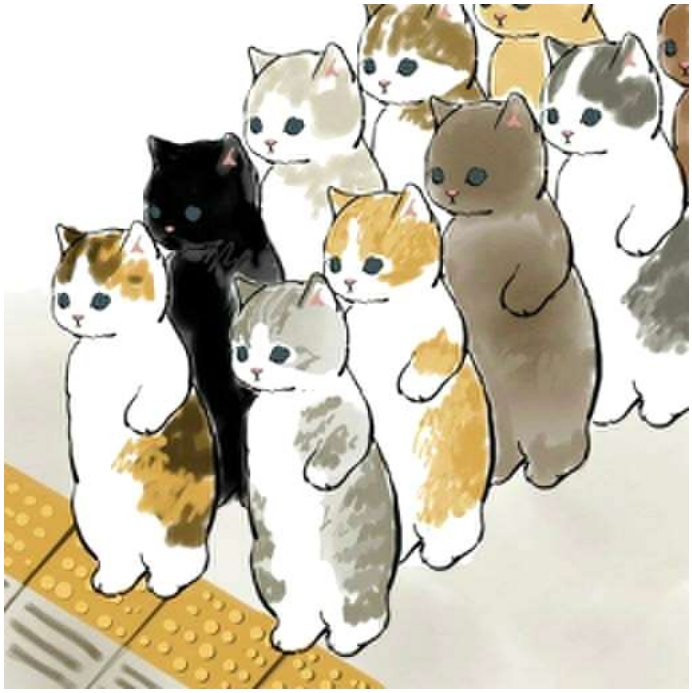
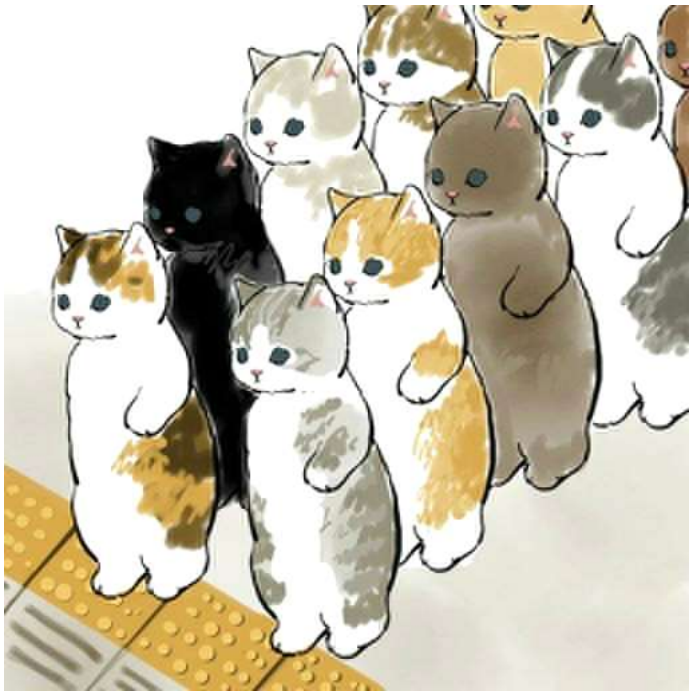
report link (https://hackmd.io/@sBeNJ4fqRNqa67PhyWWV4A/SkhrOtKGh)

1. DCT image compression

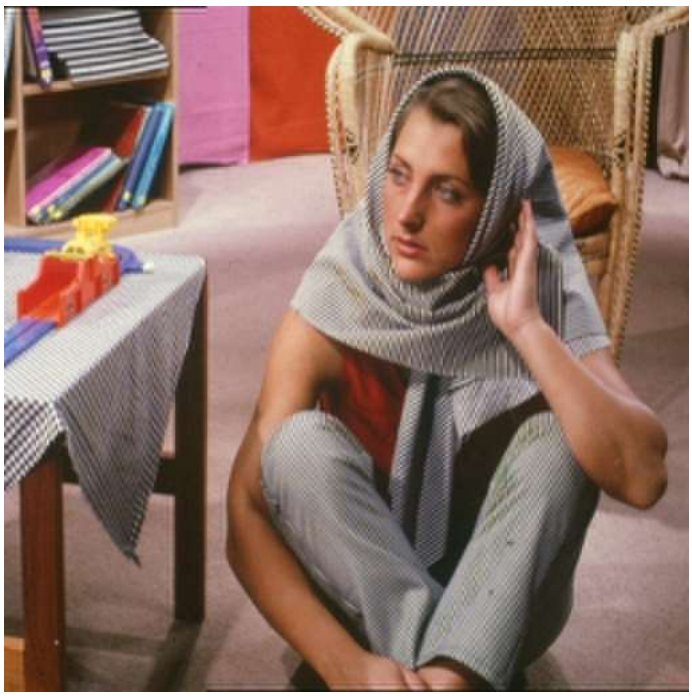
(a) Implement the simplified DCT compression process above for n = 2, 4 and m = 4,8 respectively, and then apply it to the attached image

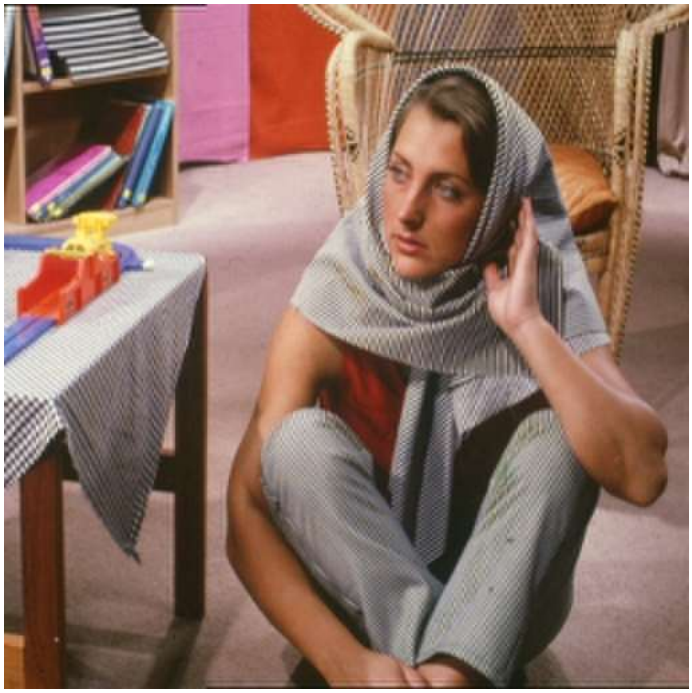
1. Show the reconstructed images for these four different cases. [2*4 images]

- 圖片擺放按照下列順序: `[{2,4}, {2,8}, {4,4}, {4,8}]` .
  - `cat.jpg`：
    - 

- 
- 
- 

- Barbara.jpg：
  - 
  - 





2. Compute the compression ratios and the PSNR values of these four reconstructed images and discuss the best rate-distortion choice.

- 直行是m而橫列是n。每一個格子是由一個tuple組成。
  - tuple: `{COMPRESSION RATE, PSNR}`, where \(PSNR=10log_{10}^{x_{peak}^2/{MSE}}\).
- 觀察表格，可以發現到m對於PSNR值的影響不大，所以我認為若是依照PSNR值來判斷高下的話，使用較小的m即可，因為這可以大大的節省資料傳輸量。此外，我觀察到，較大的n會給我帶來較大的PSNR值。是故，我會選擇n=4及m=4，以將PSNR值最大提升的同時，得到最好的壓縮率。

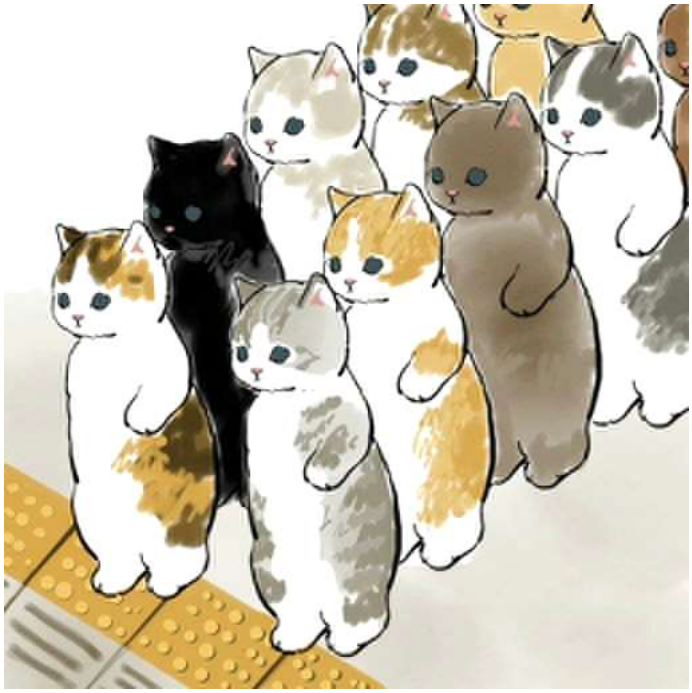
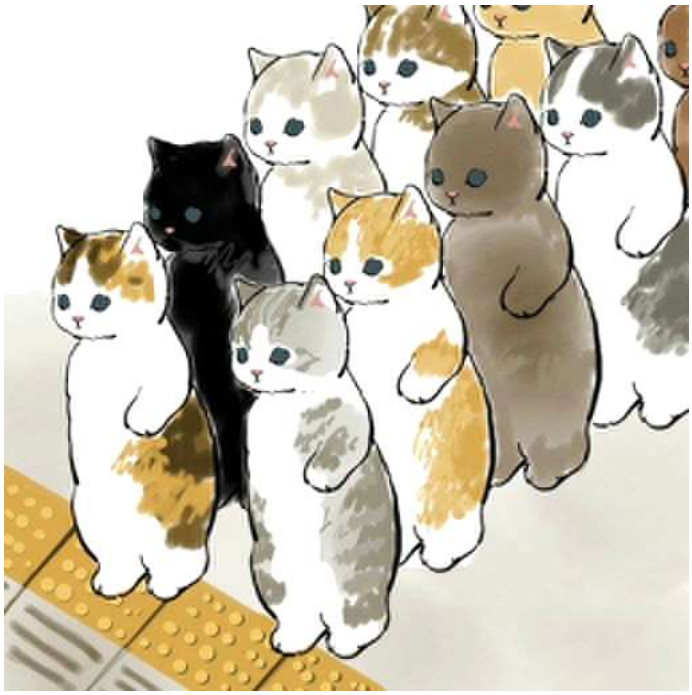
cat	m=4	m=8
n=2	{0.03, 26.23}	{0.06, 26.23}
n=4	{0.12, 32.45}	{0.25, 32.45}

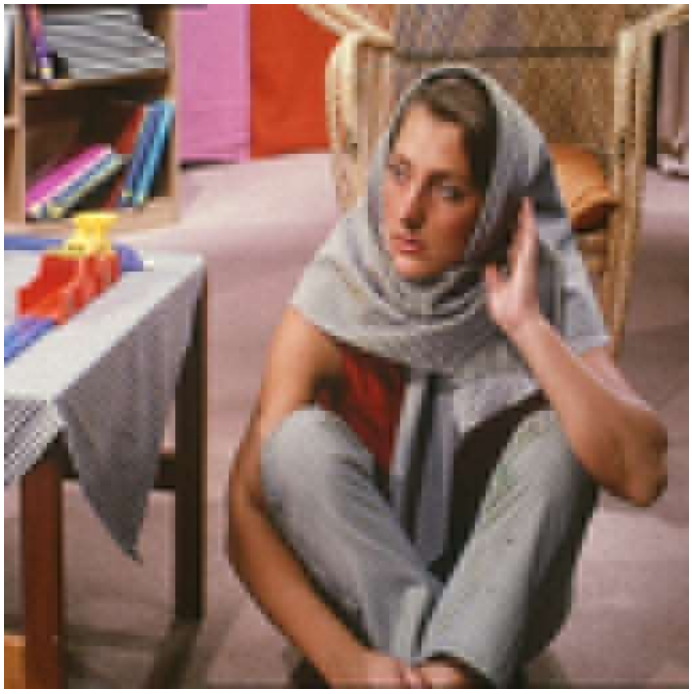
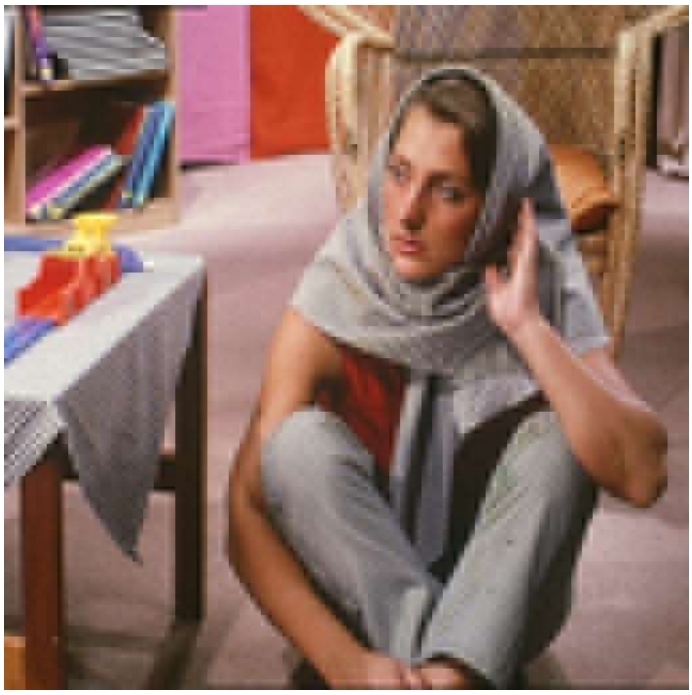
Bar	m=4	m=8
n=2	{0.03, 30.18}	{0.06, 30.18}
n=4	{0.12, 33.83}	{0.25, 33.83}

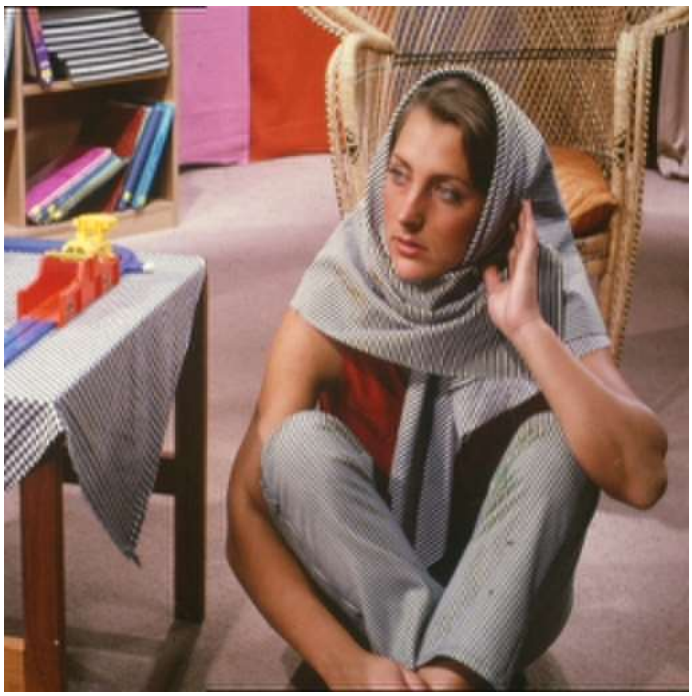
(b) Use the same process in (a) with image transformed to YCbCr color space with 4:2:0 chrominance subsampling.

1. Show the reconstructed images in RGB space. [2*4 images]

- 圖片擺放按照下列順序: `[{2,4}, {2,8}, {4,4}, {4,8}]`
  - `cat.jpg`
    - 

- 
- 
- 

- Barbara.jpg
  - 
  - 

- 
- 

2. Compute the compression ratios and the PSNR values of the four reconstructed images and discuss the best rate-distortion choice at the report

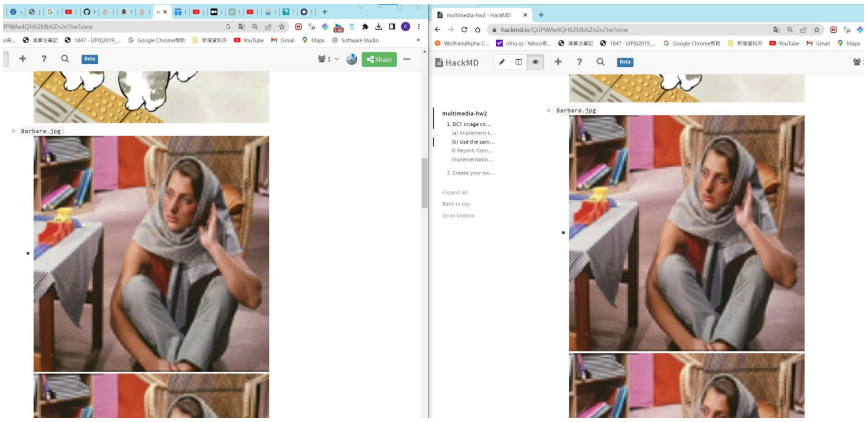
- 觀察表格，可以發現到m對於PSNR值的影響不大，所以我認為若是依照PSNR值來判斷高下的話，使用較小的m即可，因為這可以大大的節省資料傳輸量。此外，我觀察到，較大的n會給我帶來較大的PSNR值。是故，我會選擇n=4及m=4，以將PSNR值最大提升的同時，得到最好的壓縮率。

cat	m=4	m=8
n=2	{0.01, 26.24}	{0.03, 26.24}
n=4	{0.06, 32.44}	{0.12, 32.44}

Bar	m=4	m=8
n=2	{0.01, 30.19}	{0.03, 30.19}
n=4	{0.06, 33.81}	{0.06, 33.81}

© Report: Compare the differences between the results with DCT compression performed in two color spaces in (a) and (b) and the results of the two given images.

- 首先是對於同一個sampling方式下的分析:
  - 平行對比，我發現隨著n的放大，圖片的紋理會變得較為清晰，不會方塊畫。
  - 接著是垂直對比，隨著m的放大，圖片的顏色變得較為飽滿、豐富。在n以及m都較小的時候，n以及m的放大其效果比較顯著，但隨著n(或者是m)漸大，我的肉眼辨難以分辨。
- 在不同sampling下的分析:
  - 我對 `Barbara.jpg` 採樣{2,2}，因為在更大的n或是m下，我的眼睛完全無法比對。左圖是單純的YCbCr，而右圖有進行subsampling 4:2:0的處理。我發現在女生的臉頰處，其顏色若是有進行subsampling則會較深，且感覺深的部分涵蓋的範圍較大。



  - 我對 `cat.jpg` 採樣{2,2}，因為在更大的n或是m下，我的眼睛完全無法比對。左圖是單純的YCbCr，而右圖有進行subsampling 4:2:0的處理。我也有發現類似女生臉頰經過subsampling後變黑的狀況。且見第三排的土黃貓，經過處理後貓的手部在subsampling的部

分顏色變得較深。

- 綜合(a)-2以及(b)-2的表格，可以發現在n=2之時，進行subsampling竟然可以獲得較高的PSNR值(+0.01)，而在n=4之情境下，做subsampling其實也不會落差太大(-0.01)。在這樣的權衡之下，我會願意去使用4:2:0的subsampling，因為他的壓縮率基本上快要到普通YCbCr的兩倍以上，但是又可以維持不差的PSNR值。

(d) implementation discussion

> 實做細節不討論整體流程，只記錄一些我認為在實作上值得注意的細節

- 我計算DCT的方式不用老師給的sigma，而是直接使用numpy的矩陣相乘。在這種情況下可以大大的加速DCT的運算，而考量到iDCT是DCT的反運算，我也是直接使用numpy的矩陣乘法。唯一的overhead部分是計算DCT矩陣而這個是constant time，故這個overhead可以忽略。

```
def generate_DCT_matrix():
    res = np.zeros((8,8))
    for i in range(8):
        for j in range(8):
            if i == 0:
                res[i,j] = 1/np.sqrt(8)
            else:
                res[i,j] = np.sqrt(2/8) * np.cos(((2*j+1)*i*np.pi)/16)
    return res
def DCT(block, T):
    res = np.zeros((8,8))
    res = np.dot(np.dot(T, block), T.T)
    return res
def iDCT(block, T):
    res = np.zeros((8,8))
    res = np.dot(np.dot(T.T, block), T)
    return res
```

- 此外，考量到 `uniform_quantization` 有要除上step，但是因為step是源於 `(np.max(res) - np.min(res))/(2**m)` ，所以step有可能是0，進而產生255的效果。為了預防這個問題產生，我有對其進行數值保護以去除除以0的問題。

```
def uniform_quantization(block, n, m, channel):
    ...
        step = (np.max(res) - np.min(res))/(2**m)
        #print('step is ', step)
        if step <= 0.0000001:
            step = 64/(2**m)
        res = res/step
    return res
```

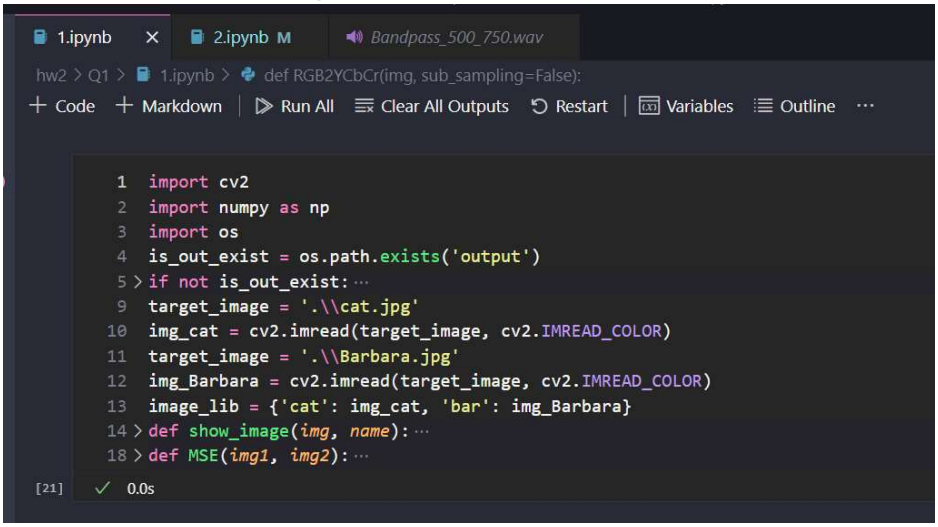
- 最後，在實作(b)的時候我對於subsampling是使用一個特殊的python語法， `u[1::2,1::2] = u[::2, ::2]` ，這個可以有效地將 `4:2:0` 的subsampling實做出來。

```
def RGB2YCbCr(img, sub_sampling=False):
    if sub_sampling:
        img_y_cb_cr = np.zeros(img.shape, dtype = int)
        y = (0.257 * img[:,:,0]) + (0.564 * img[:,:,1]) + (0.098 * img[:,:,2]) + 1
        u = -(0.148 * img[:,:,0]) - (0.291 * img[:,:,1]) + (0.439 * img[:,:,2]) +
        v = (0.439 * img[:,:,0]) - (0.368 * img[:,:,1]) - (0.071 * img[:,:,2]) + 1
        u[1::2,1::2] = u[::2, ::2]
        v[1::2,1::2] = v[::2, ::2]
        img_y_cb_cr = np.dstack((y, u, v))
    ...

    return img_y_cb_cr
```

(e) execution

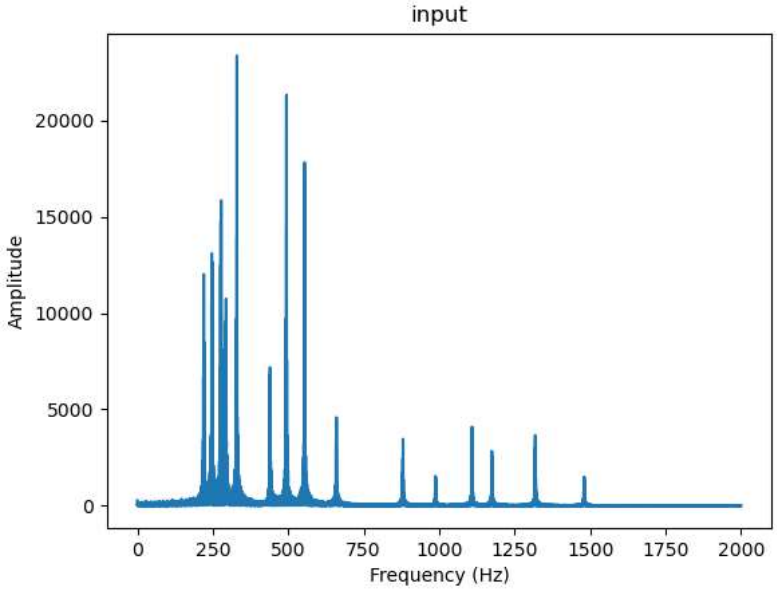
- 按下 `clear all output`
- 接著按下 `run all`
- 會將結果存放在output這個資料夾中



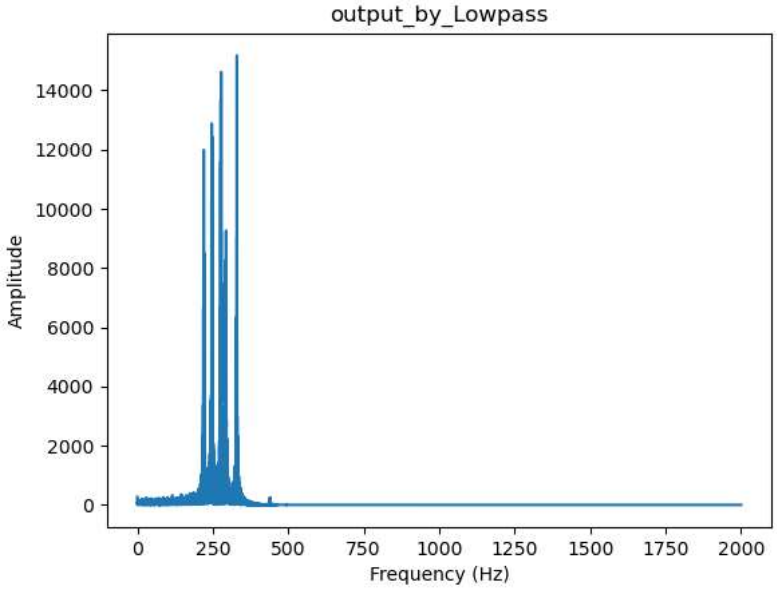
2. Create your own FIR filters to filter audio signal (40%)

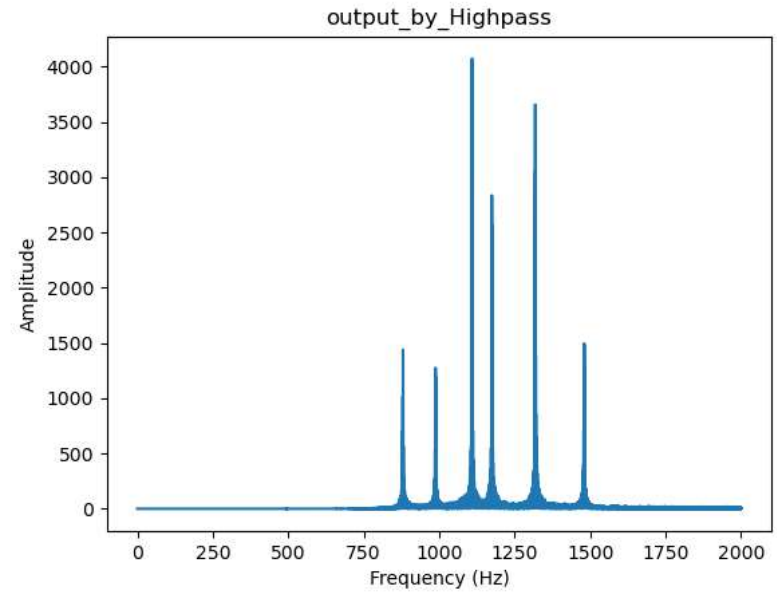
(a) Image results:

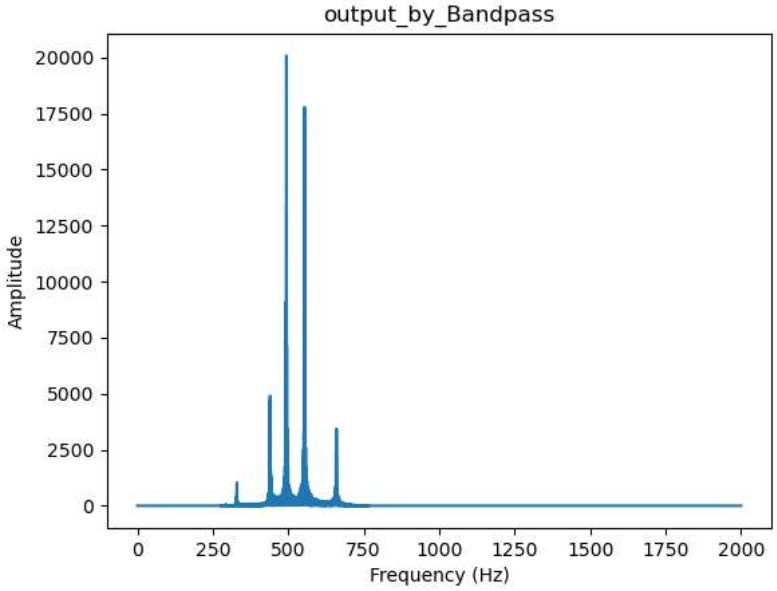
THE SPECTRUM OF THE INPUT SIGNAL

- 

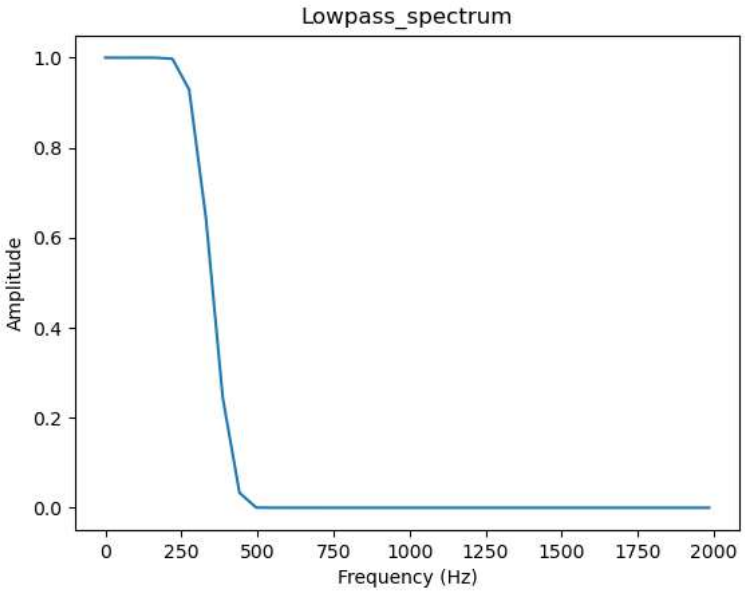
The spectrums of the output signals. (Before echo.) [3 images]

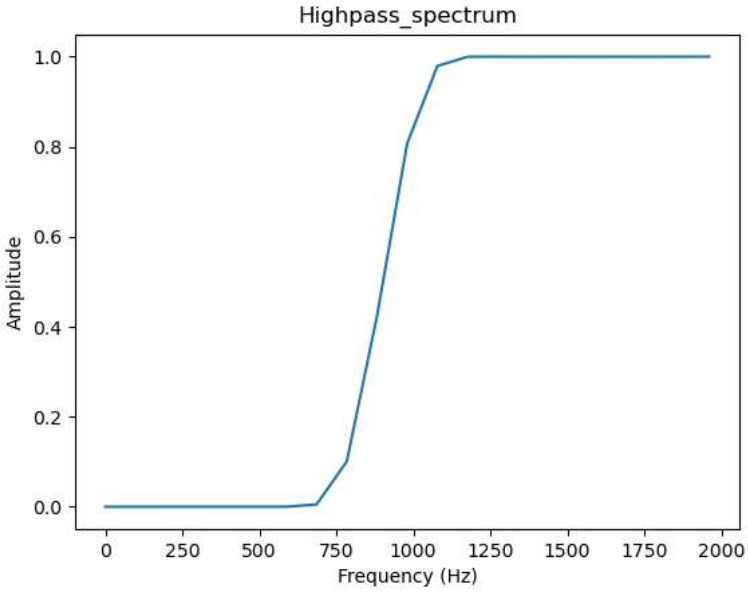
- 

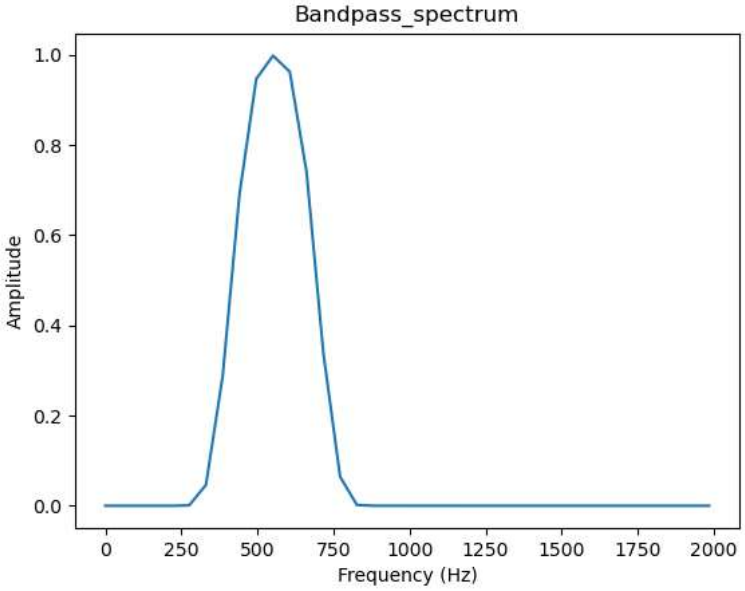
- 

- 

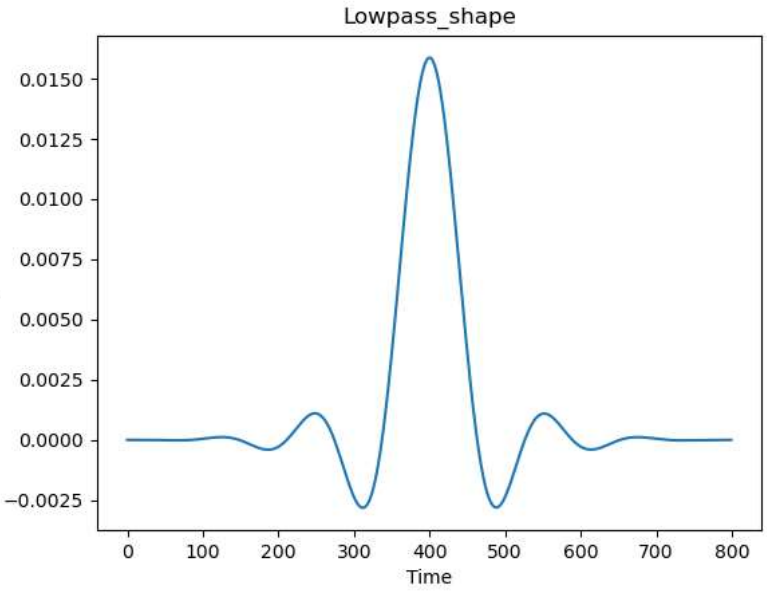
The spectrums of the filters. [3 images]

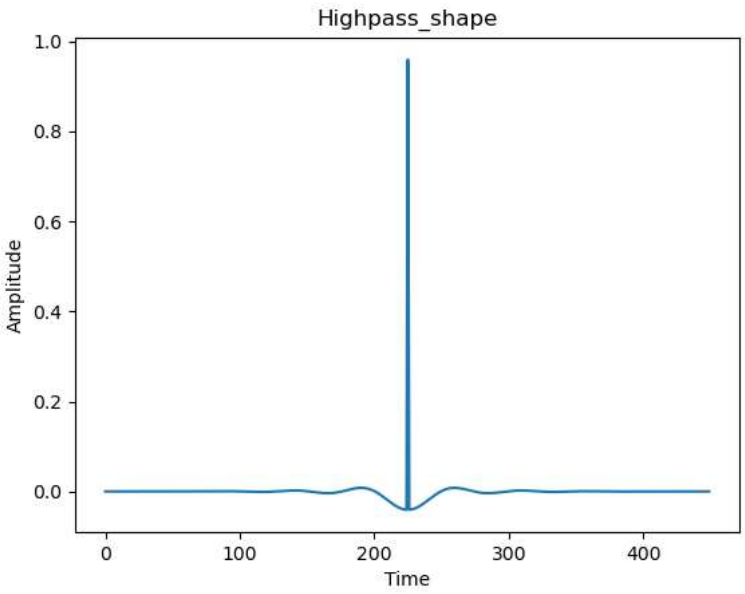
- 

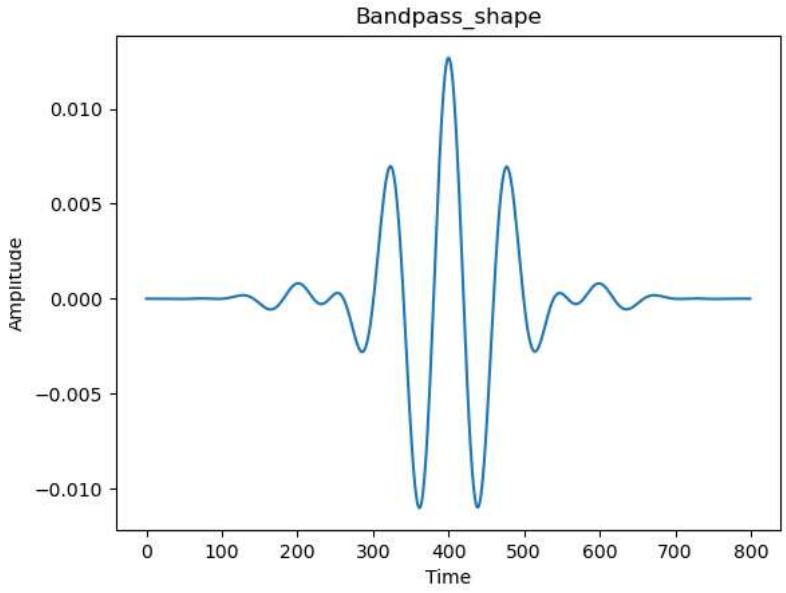
- 

- 

The shapes of the filters (time domain) [3 images]

- 

- 



(b) Audio results:

- 已經在code裡面存入

© Report:

1. Discuss how you determine the filters.

- Q2是由三首歌組成:小蜜蜂/兩隻老虎/瑪莉有隻小綿羊
  - 小蜜蜂:
    - 按照助教提供的hint，先設計出一個low pass filter。
    - 接著我將window設定為450。
    - 然後測試各種不同的窗戶 `['rectangular', 'hamming', 'hanning', 'blackman']` ，我最後選擇的是 `blackman` 。
    - 將 `f_cutoff` 依照1000、500、250的方式遞降，逐漸地發現在350左右可以獲得小蜜蜂。
  - 兩隻老虎:
    - 老師上課也只有講到四種filter: lowpass/highpass/bandpass/bandstop。我猜想下一種便是high pass filter。
    - 和小蜜蜂依樣，我將window設定為450。
    - 然後測試各種不同的窗戶 `['rectangular', 'hamming', 'hanning', 'blackman']` ，我最後選擇的是 `blackman` 。

        - 將 `f_cutoff` 依照250、500的方式遞升，逐漸地發現在900左右可以獲得兩隻老虎。
    - 瑪莉有隻小綿羊:
        - 由於我在進行FFT的時候就有對於過高頻的部分進行限縮，所以我思考[-,450], [900,+]均以經被使用的情況下，那應該是暗指我要對[450,900]進行處理。於是我猜用bandpass filter搭配 `f_cutoff_low, f_cutoff_high` 來處理。
        - 我將window設定為800
        - 然後測試各種不同的窗戶 `['rectangular', 'hamming', 'hanning', 'blackman']` ，我最後選擇的是 `blackman` 。
        - 將{ `f_cutoff_low` , `f_cutoff_high` }設定為{415、695}，便可以找到瑪莉有隻小綿羊

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

- 幾本上filter都長得大同小異，我拿 `low_pass_filter` 作為範例:每一格filter在line8及10不同，其他都依樣。
    - 就以 `low_pass_filter` 為例:
        - line8是 `my_filter[mid+i] = np.sin(2*np.pi*f_cutoff*i)/(np.pi*i)` ，
          在 `high_pass_filter` 是 `my_filter[mid+i] = -np.sin(2*np.pi*f_cutoff*i)/(np.pi*i)` ，
          在 `band_pass_filter` 則是 `my_filter[mid + i] = np.sin(2*np.pi*f_cutoff_high*i)/(np.pi*i) - np.sin(2*np.pi*f_cutoff_low*i)/(np.pi*i)`
        - 而line10是 `2 * f_cutoff` 而
          在 `high_pass_filter` 則是 `1 - 2 * f_cutoff` ，
          在 `band_pass_filter` 則是 `my_filter[mid] = 2 * (f_cutoff_high - f_cutoff_low)`

```
def low_pass_filter(f_cutoff, f_sampling_rate, N, window_type):
    #normalize cutoff frequency
    my_filter = np.zeros((N,))
    f_cutoff = f_cutoff / f_sampling_rate
    mid = int(N/2)
    for i in range(-mid,mid):
        if i!=0:
            my_filter[mid+i] = np.sin(2*np.pi*f_cutoff*i)/(np.pi*i)
        else:
            my_filter[mid] = 2 * f_cutoff
    #convert ideal filter to low pass filter
    my_ideal_filter = window(my_filter, N, window_type)
    return my_ideal_filter
```

- 而每一個filter對於window的選擇都會建基於 `window_type` 這個argument上，window的實作如下:基本上只是一個switch case得東西，其實做內容於silde的75可以找到1對1的呼應。

```
def window(ideal_filter, N, window_type):
    realistic_filter = ideal_filter.copy()
    for i in range(N):
        if   window_type == 'rectangular':
            realistic_filter[i] = ideal_filter[i]
        elif window_type == 'hamming':
            realistic_filter[i] = ideal_filter[i] * (0.54+0.46*np.cos(2*np.pi*i/N)
        elif window_type == 'hanning':
            realistic_filter[i] = ideal_filter[i] * (0.5+0.5*np.cos(2*np.pi*i/N))
        elif window_type == 'blackman':
            realistic_filter[i] = ideal_filter[i] * (0.42 - 0.5*np.cos(2*np.pi*i/(
    return realistic_filter
```

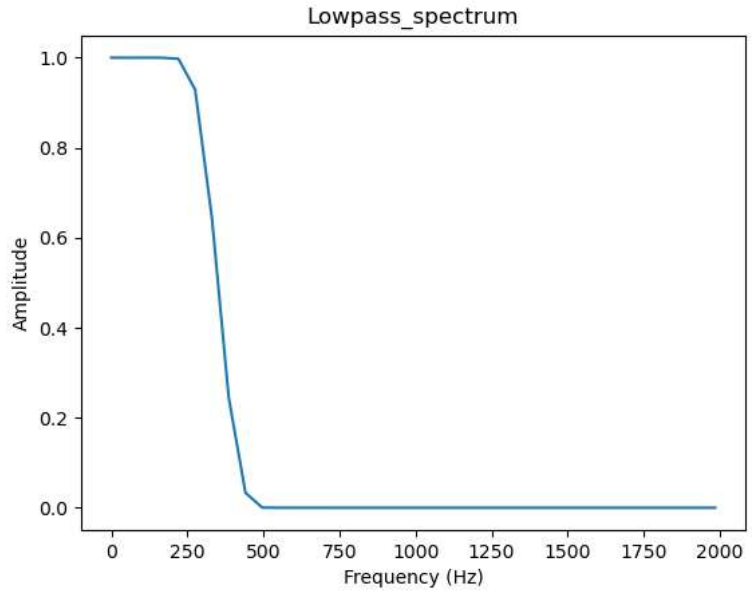
- 我對於convolution的實作如下所示。其中有一個小技巧，節省了我很多時間。且見 `#return np.convolve(input_audio, my_filter, 'same')#speed up line` ，這段本來不應該出現的，但是我發現每一次重跑Q2都要花十幾分鐘，我是個急性子的人，實在等不下去，然後我就想說先用一下numpy提供得convolution，後來發現驚為天人，步道一秒就跑完了。於是我便採取以下策略:
  - 測試時，使用numpy.convolve
  - 測試完成，改回我自己手寫的convolution
- 使用此策略節省我很多時間OuO

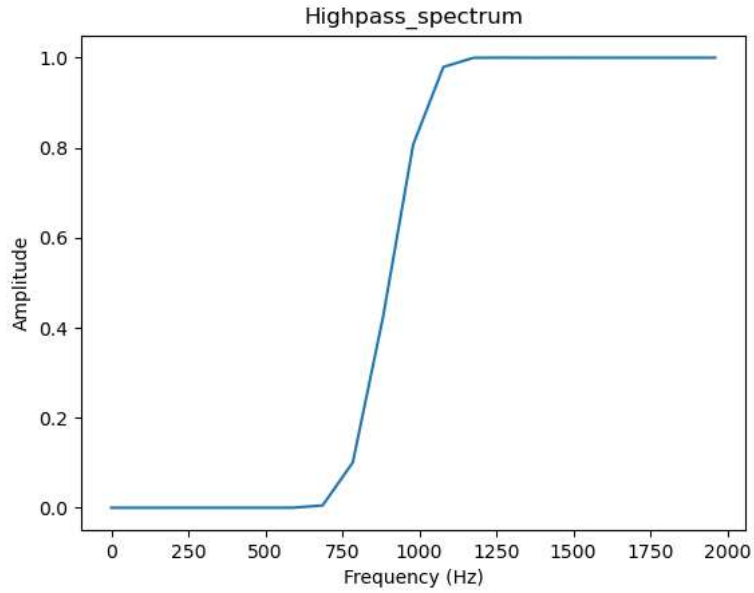
```
def masking(input_audio, my_filter, N):
    res = np.zeros(input_audio.shape)
    #return np.convolve(input_audio, my_filter, 'same')#speed up line
    for i in range(input_audio.shape[0]):
        for j in range(N):
            if i-j >= 0:
                res[i] = res[i] + input_audio[i-j] * my_filter[j]
    return res
```

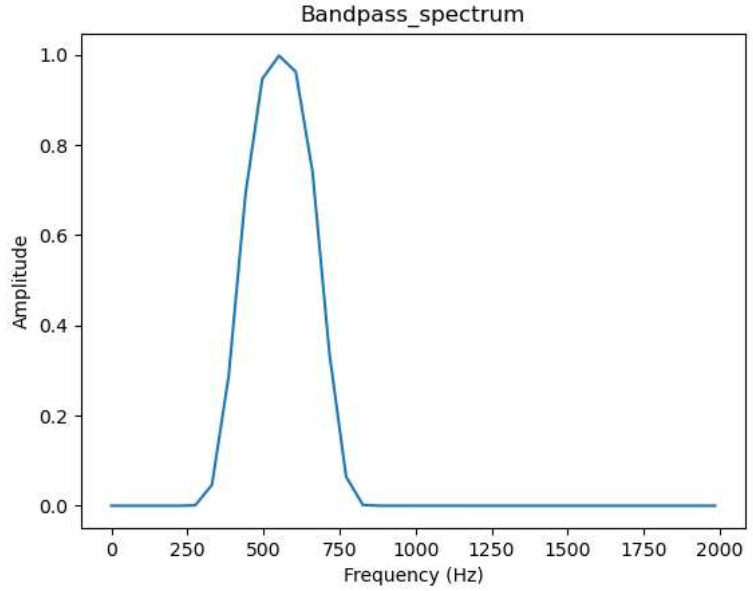
- 對於multiple fold的處理，我是照抄slide提供的multiple fold做法:

```
def multiple_fold_echo(input_audio, delay, gain):
    res = np.zeros(input_audio.shape)
    for i in range(input_audio.shape[0]):
        if i-delay >= 0:
            res[i] = input_audio[i] + gain * res[i-delay]
        else:
            res[i] = input_audio[i]
    return res
```

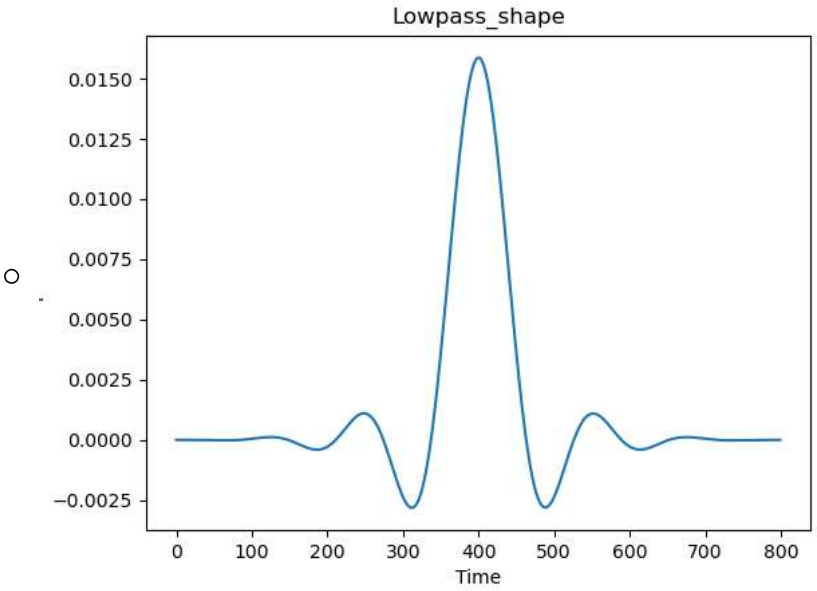
3. Compare spectrum and shape of the filters.

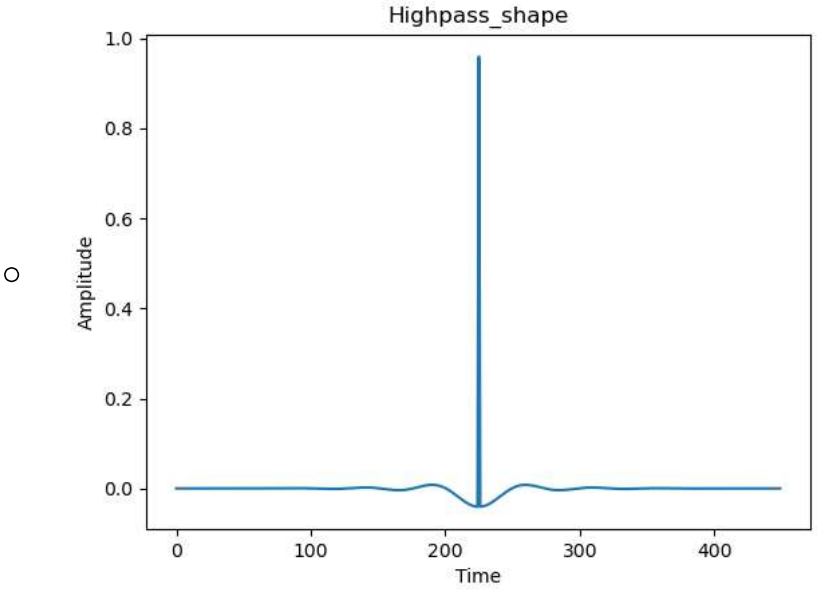
- 首先是對於spectrum的比較(frequency domain):我發現low pass/high pass他其實不適一個硬性filter，他是以一個加成比率得方式將超過f_cut得頻率壓成較低得數值，而非是0。這樣可以是訊號不會那麼直接的被抹去，進而使整體音訊不會突然少了一部分的頻譜進而使其聽起來怪怪的。此外，我發現到他們的最大家成數值是1.0，這代表完全接受那個頻率得訊號。此外，我還觀察到N(filter window size)會對cut off frequency的切割角度有明顯的影響，N越大則切割效果越明顯。
    - 

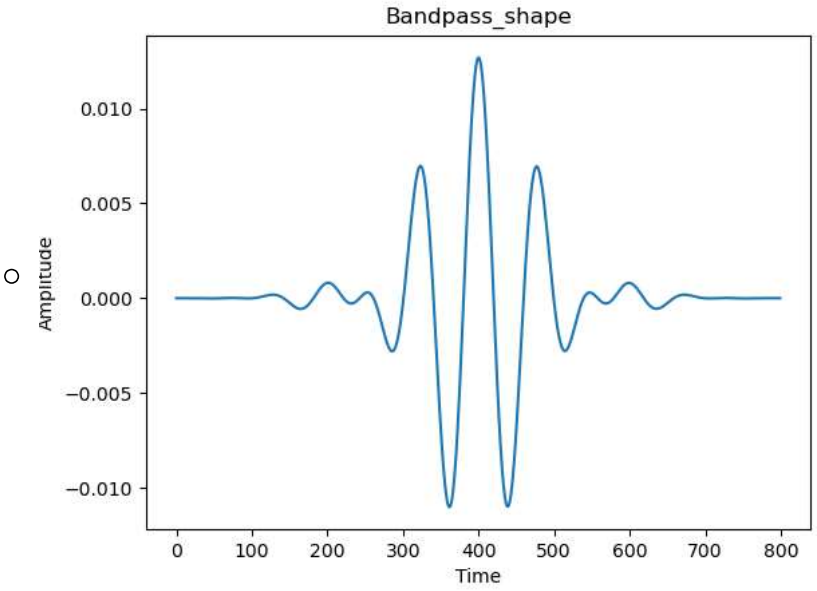




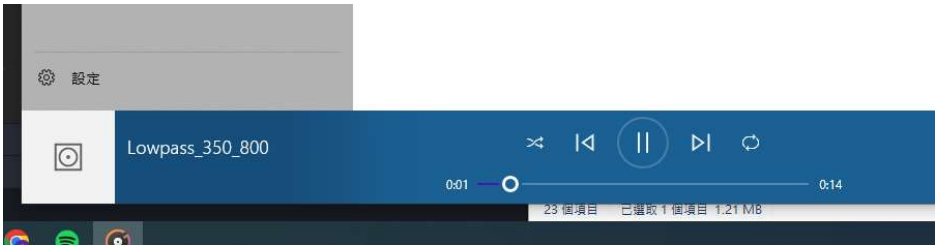
- 其次是對於shape的比較(time domain)。我數學不太好，我覺得low pass長得像是一個於t=400處被拉伸的cosin波，至於highpass則像是一個pulse，長期在0附近，於t=240左右突然衝到很高的地方(1.0)，然後一過t=240又立刻回到0附近。至於bandpass則是如同一個被拉扯了sin波。

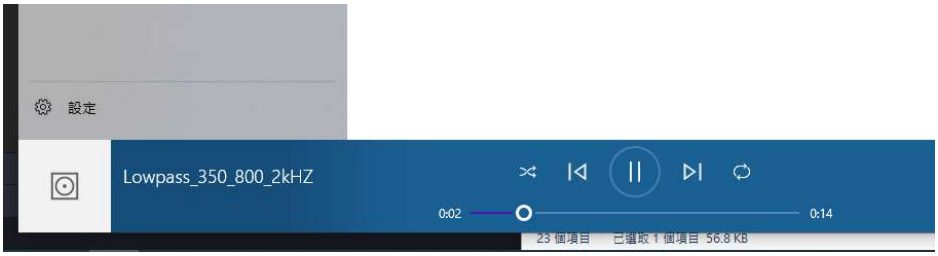






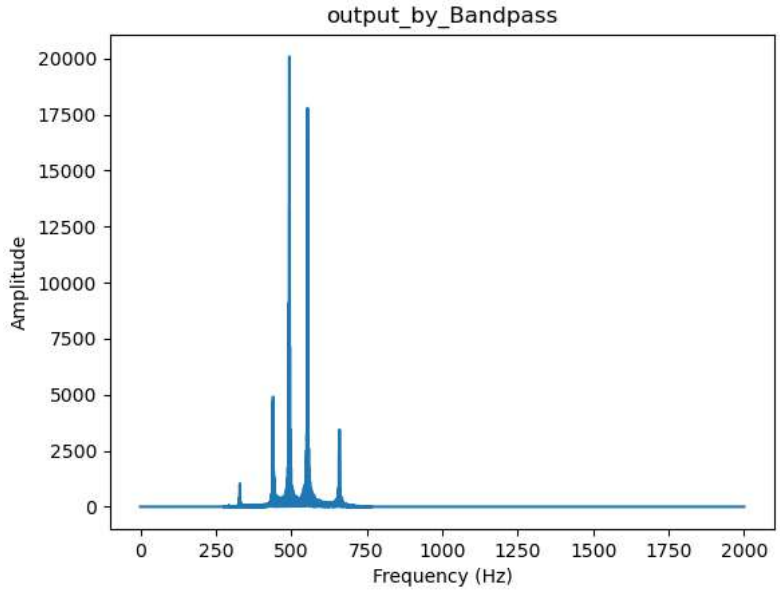
4. Briefly compare the difference between signals before and after reducing the sampling rates.

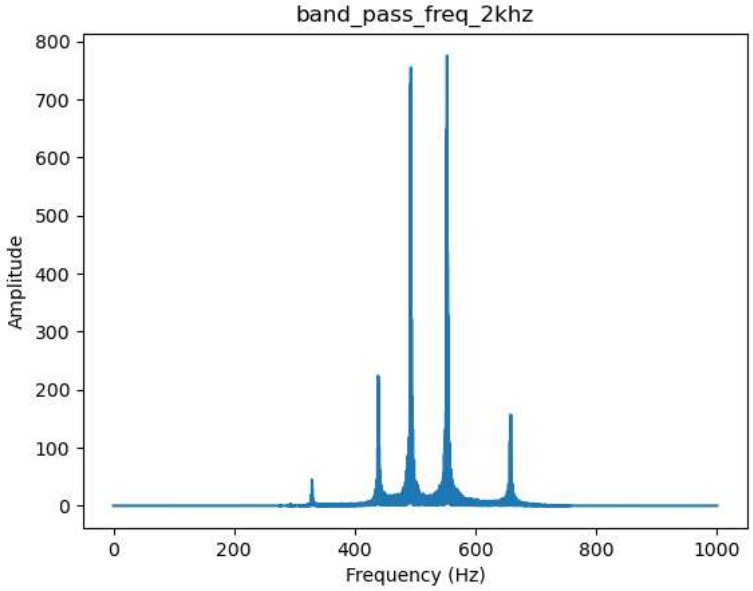
- 經過downsampling歌曲，數據量一定會比原來的歌曲短 `sampling_rate/new_sampling_rate` 倍，這是因為向下採樣本來就會loss一定量的數據。是故我可以猜測經過downsampling得歌曲其內容得變化細緻度會有下降，畢竟sampling rate下降了。
- 但是整體執行長度反倒是沒有什麼改變，就以小蜜蜂為例:2k sampling rate的執行時間和44100 sampling rate的執行時間差不多。我想原因在於我們在存檔案的時候，有對這個即將要存進去得wav檔特別指明說要以指定的sampling rate存入。這樣可以使整體執行時間維持(詳見 `video_saver`)
- 

- 


```
def video_saver(input_audio, filename, sampling_rate):
    data = input_audio
    scaled = np.int16(data / np.max(np.abs(data)) * 32767)
    write(filename, sampling_rate, scaled)
```

- 以frequency domain而言(以Bandpass filter為例)，我觀察到2k版本的頻率整體的採集到次數都較低，就以500hz那個峰值而言，2k版本的是750而44100版本的則是20000左右，我想20000/750=26約莫就是44100/2000的倍率。

- 

- 

- 而就我這個木耳，我其實只覺得downsampling版本的小蜜蜂聽起來基本上沒有甚麼區別...

(d) execution

- 按下 `clear all output`
- 接著按下 `run all`
- 會將結果存放在output這個資料夾中