ESE 531: Project 1, Filter Design and Filter Banks

Noah Schwab

March 29, 2022

All code for this project was written in Python with the aid of NumPy and SciPy packages.

1 PART A: FIR FILTER DESIGN 2 WAYS

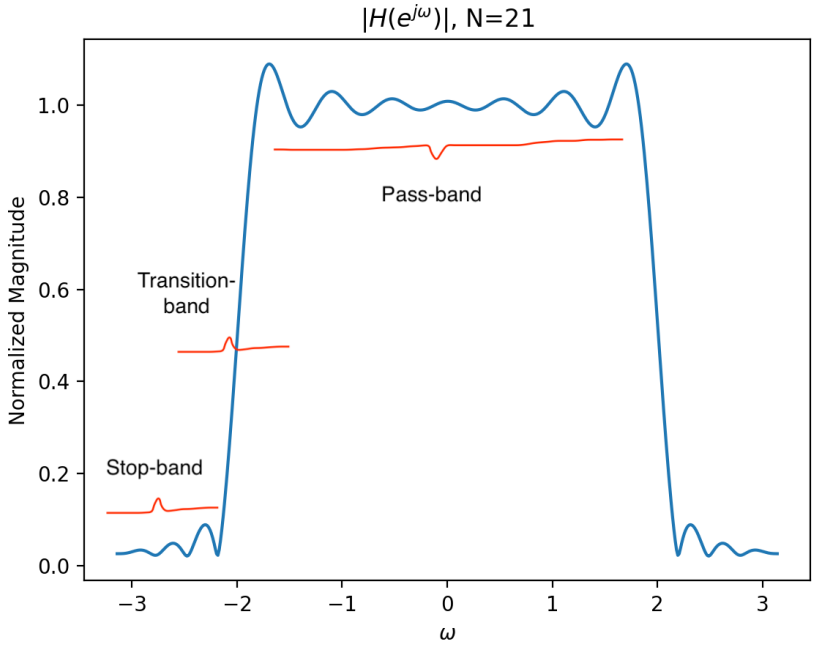
1.1 Design an FIR Filter using Truncation

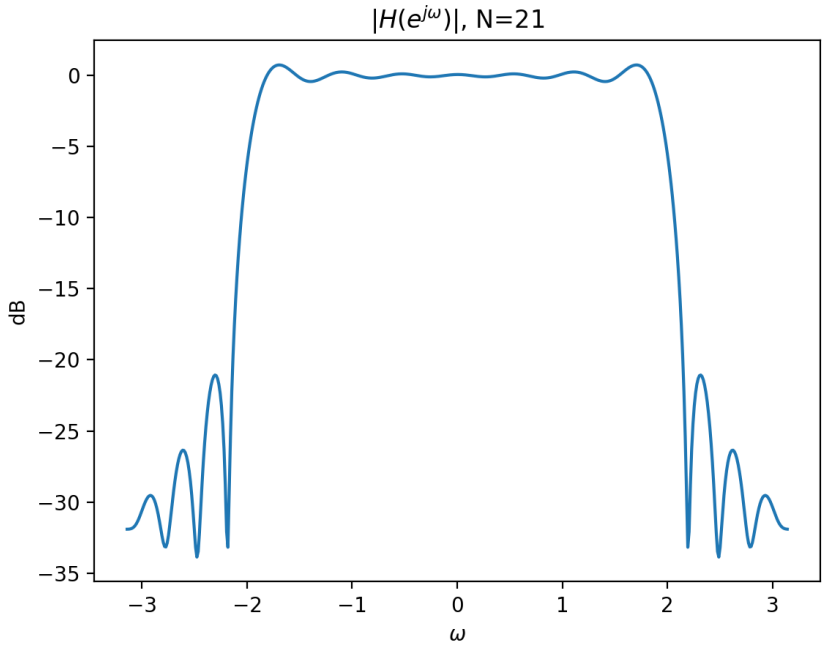
For this problem, I created a simple function LPFtrunc(N) which returns the impulse response associated with the truncation frequency response. In other words, this function returns a periodic sinc according to Equation 1.

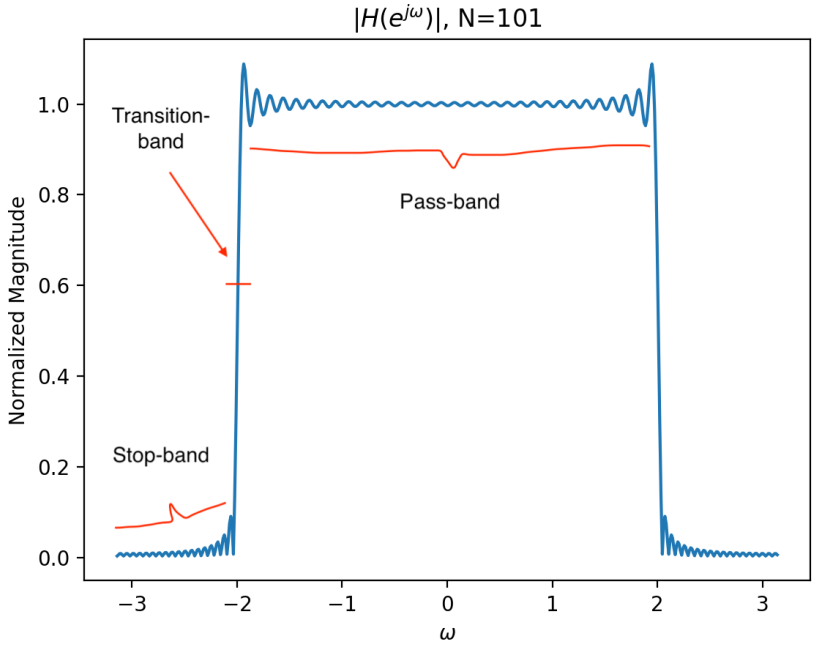
$$h_{\text{ideal}}[n] = \frac{\omega_c}{\pi} \operatorname{sinc} \frac{\omega_c n}{\pi} \tag{1}$$

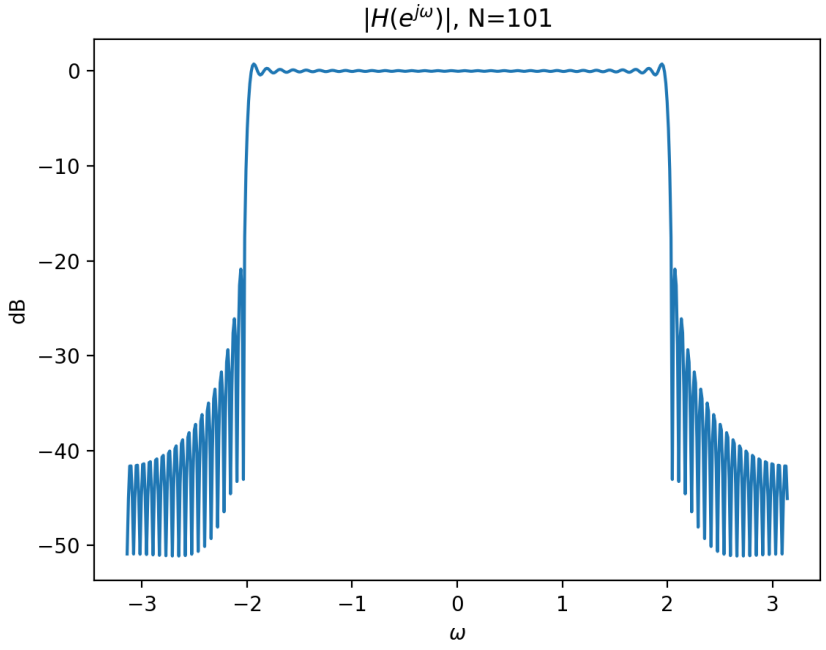
Within the function, I set the cutoff frequency to be $\omega_c = 2.0$ rad/sec.

I then called the function for N=21 and N=101, and used the NumPy function fft, which corresponds to the Fast Fourier Transform, to compute the frequency response for each truncated filter. I plotted the magnitude frequency responses on the normalized scale and in decibels ($20 \log_{10}(|H(e^{j\omega})|)$), as shown in the figures below. In the normalized scaled plots, I have labeled the pass-band, the transition-band, and the stop-band.


$|H(e^{j\omega})|$, N=21
Normalized Magnitude
1.0
0.8
0.6
0.4
0.2
0.0
Pass-band
Transition-
band
Stop-band
−3
−2
−1
0
1
2
3
ω


$|H(e^{j\omega})|$, N=21
dB
0
−5
−10
−15
−20
−25
−30
−35
−3
−2
−1
0
1
2
3
ω





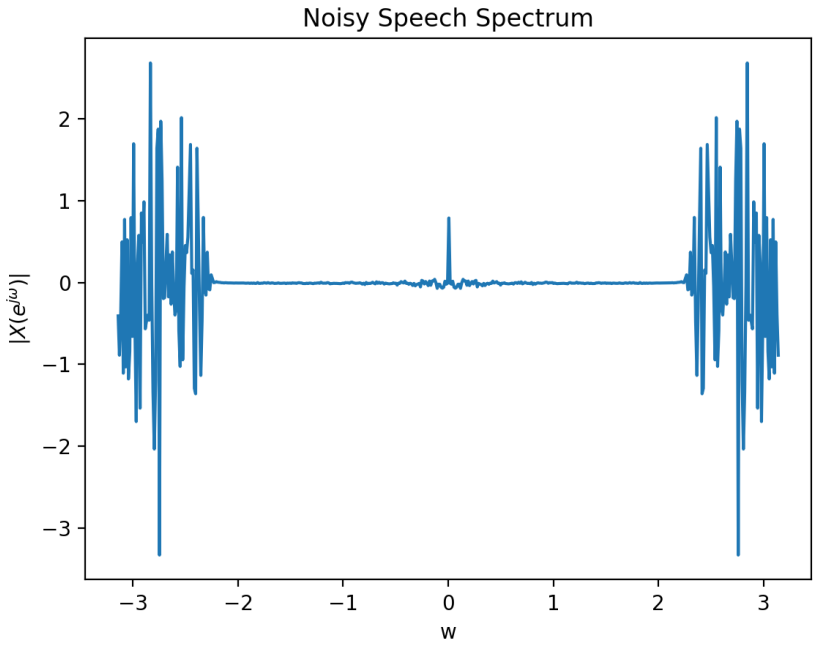
As we can see from the plots above, increasing the length of the truncation window and subsequent FIR filter results in a sharper transition-band and the stop-band ripple is decreased. We can understand this phenomenon by considering the truncated FIR filter as

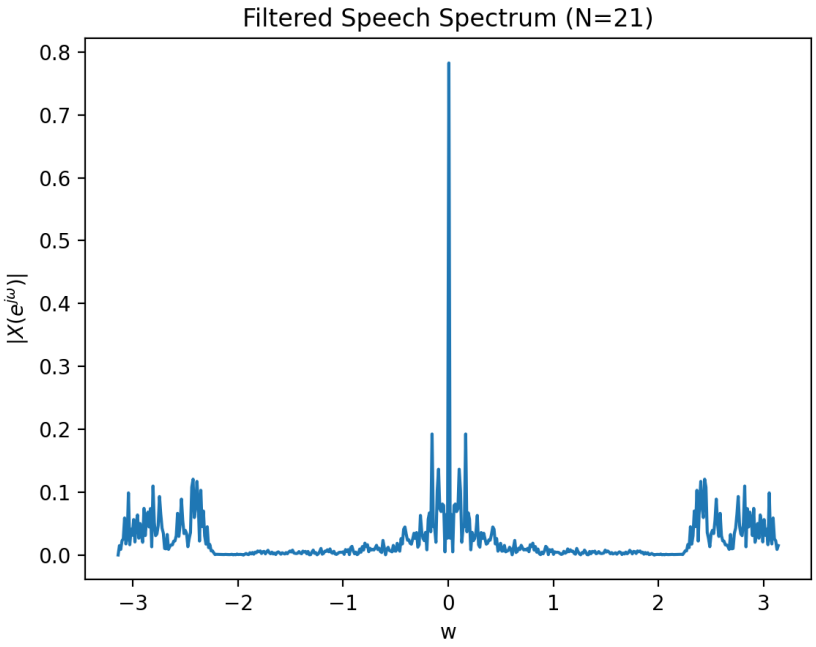
an approximation of an infinitely wide window. In other words, to obtain the ideal low-pass filter, we would essentially by truncating by an infinite window, which would result in the infinite-time sinc function as an impulse response. However, when we design by truncation, we must chose a window size, which corresponds to the resultant filter size. As a result, larger filters are better approximations of the sinc impulse response, and as a result more closely resemble the ideal low-pass filter. This is why N=101 has a sharper transition and smaller ripple as compared to N=21.

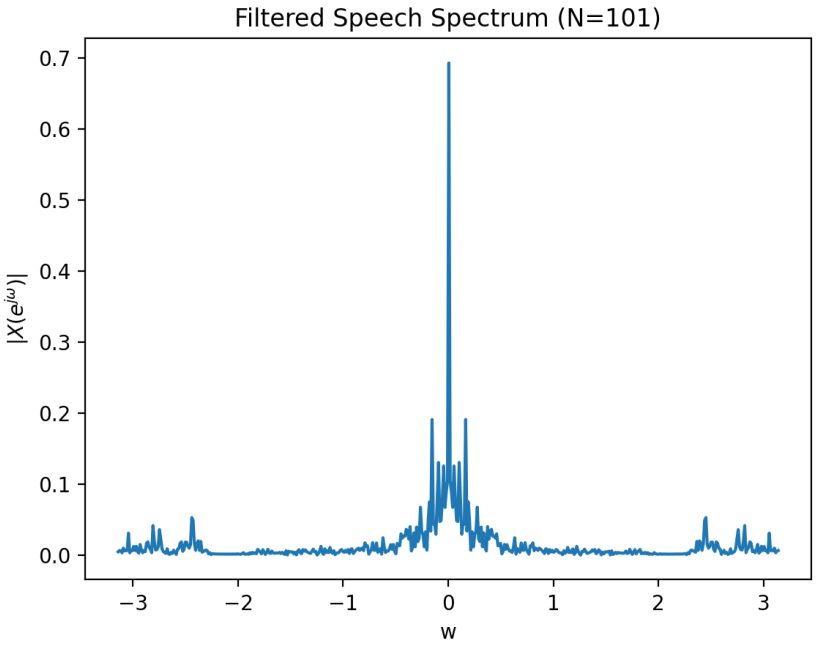
For the next section of this part, I loaded in the provided audio signal by using the loadmat function from SciPy. I wrote provided signal to a .wav file and found the audio file to be understandable, but there is strong, high-pitch background noise.

I used my function LPFtrunc to generate the impulse responses of low-pass filters for N=21 and N=101. After convolving the original noisy signal with both impulse responses, I saved both files as .wav. It is evident that both filters reduce the background noise and make the audio clearer. However, the N=101 filter performs better since it has a smaller ripple in the stop-band and can better attenuate the high-frequency noise.

Evidence for this can be seen in the Fourier Transform of audio signal. Below, I have shown plots for the spectrum of the noisy speech, as well as the spectrum of the audio signal after being filtered by our truncated FIR low-pass systems. As we can see, the original signal is corrupted by high-frequency noise. In order to clean the signal, we can low-pass filter it. When filtering with the N=101 filter, we are more successful at removing high-frequency components, which explains why the audio signal is cleaner when filtered with the N=101 low-pass as opposed to the N=21.

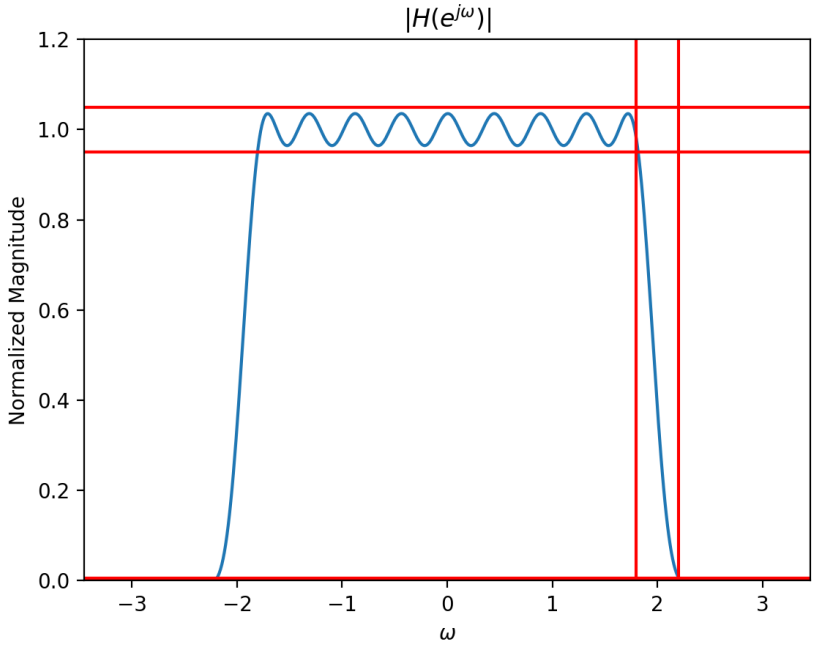

Noisy Speech Spectrum
$|X(e^{j\omega})|$
w
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

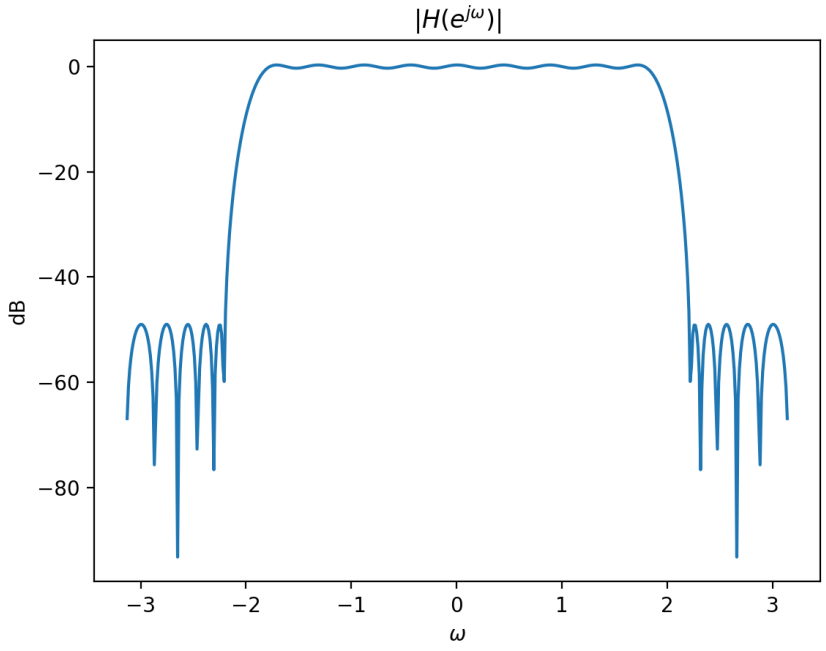

Filtered Speech Spectrum (N=21)
$|X(e^{j\omega})|$
w
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−3
−2
−1
0
1
2
3



1.2 Parks-McClellan Filter Design

I used the function remez from the SciPy Signal package, which is equivalent to Matlab's firpmord function. In my program, I defined the given specs, and I computed the frequency response of the Parks-McClellan/Remez output. Additionally, I specified the weights according to the ration of the desired ripple heights in the pass-band and stop-band. I generated the plots, and continually adjusted the filter order until I found the threshold such that the ripple specs in both the pass-band and stop-band were met. The figures below show the resultant plots.


$|H(e^{j\omega})|$
Normalized Magnitude
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−3
−2
−1
0
1
2
3
ω


$|H(e^{j\omega})|$
dB
0
−20
−40
−60
−80
−3
−2
−1
0
1
2
3
ω

2 Multi-Channel FIR FILTER BANK IN ONE DIMENSION

2.1 4-Channel Octave Band Filter-Bank

In order to build a filter-bank with two low-pass frequency bands each 1/8 of a full band, an intermediate frequency band that is 1/4 of a full band, and a high-pass frequency band that is 1/2 of a full band, I applied the general scheme of a two-cahnnel analysis and synthesis filter bank, as provided by the textbook in Chapter 4. The general idea is to design a low-pass filter and a high-pass filer that are both half of the full-band, and analyze the lower and higher frequency components of signal separately by employing downsampling and upsampling. We can extrapolate this framework to create the specified 4-channel octave band filter-bank by cascading the low-frequency analysis and synthesis channel three times. The sketch below shows the diagram of such a filter bank.

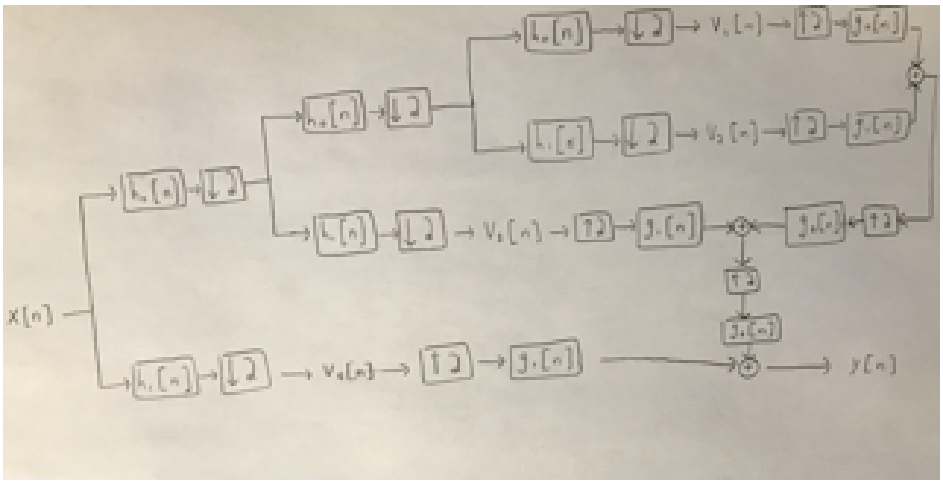


Figure 1: 4-Channel Filter Bank Design

In the above diagram, $h_0[n]$ is a low-pass frequency impulse response that occupies the lower-half of the full band, and $h_1[n]$ is a high-pass frequency impulse response that occupies the upper-half of the full band. By cascading the low-pass filter and downsampling, we are effectively recasting the lower-half band as a new complete band, and restarting the whole

process. This system results in four subsystems that can process the frequency components within each band, as specified by the problem. These resultant sub-signals are represented by $v_1[n], v_2[n], v_3[n], v_4[n]$. $g_0[n]$ and $g_1[n]$ are the synthesis filters that are associated with the low-pass and high-pass analysis filters. In theory, $y[n]$ should be a perfect-reconstruction of the input, $x[n]$.

I could not find an equivalent Python function for firpr2chfb, so I instead designed the synthesis and analysis filters according to the alias cancellation condition of the quadrature mirror system described below:

$$h_1[n] = e^{j\pi n} h_0[n] \tag{2}$$

$$g_0[n] = 2h_0[n] \tag{3}$$

$$g_1[n] = -2h_1[n] \tag{4}$$

In my program, I defined $h_0[n]$ as a Hamming window with order M=50 and cutoff frequency $\omega_s = \frac{\pi}{2}$ rad/sec. The figure below shows the magnitude frequency responses of the mirror filters.

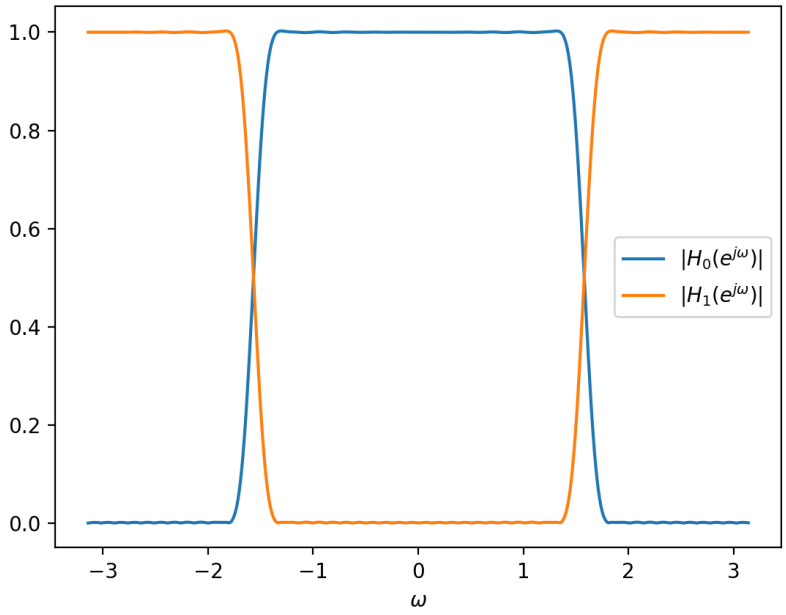


Figure 2: Analysis Filters

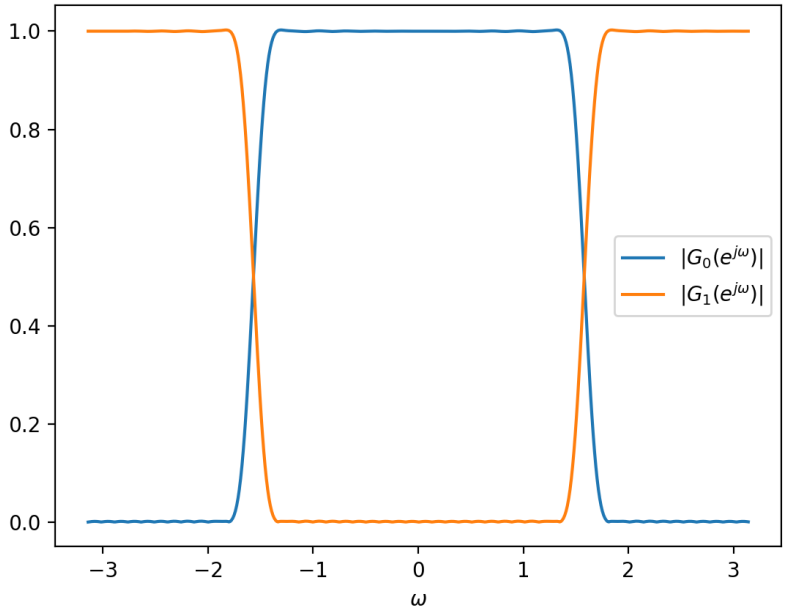
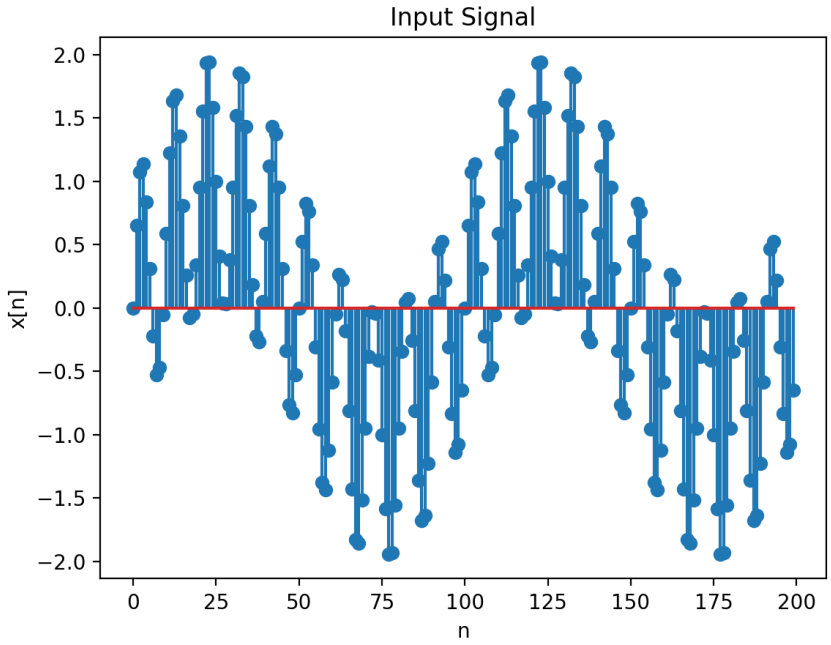
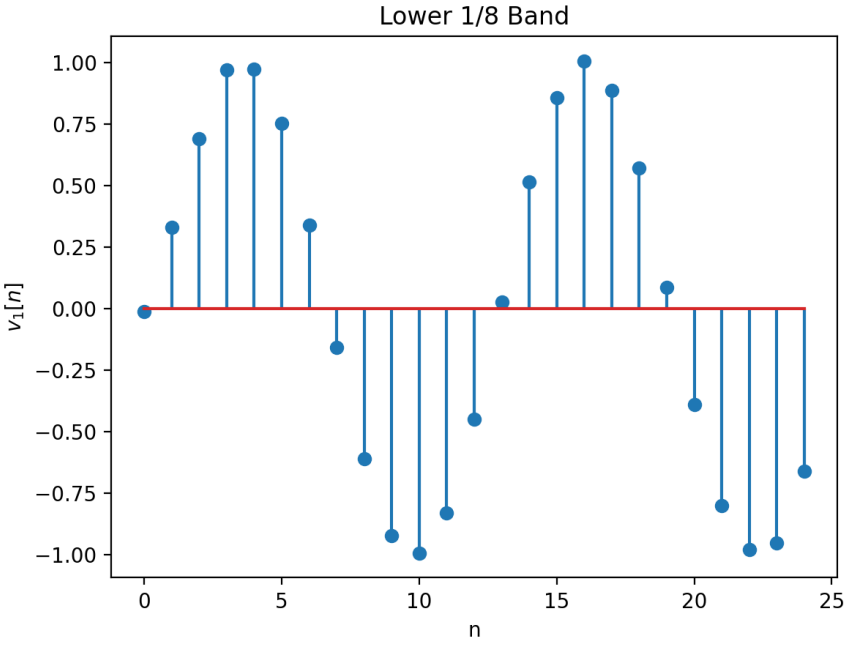


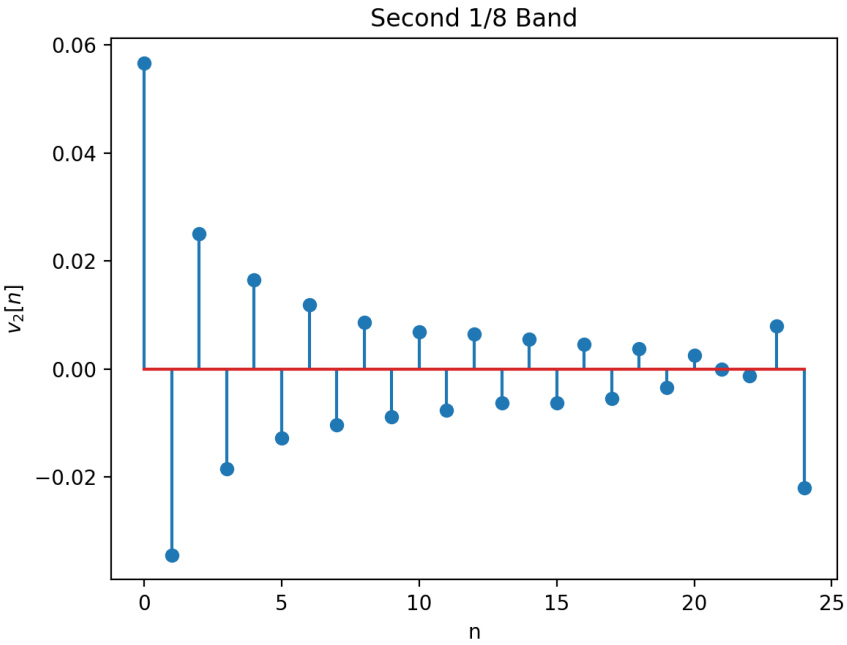
Figure 3: Synthesis Filters

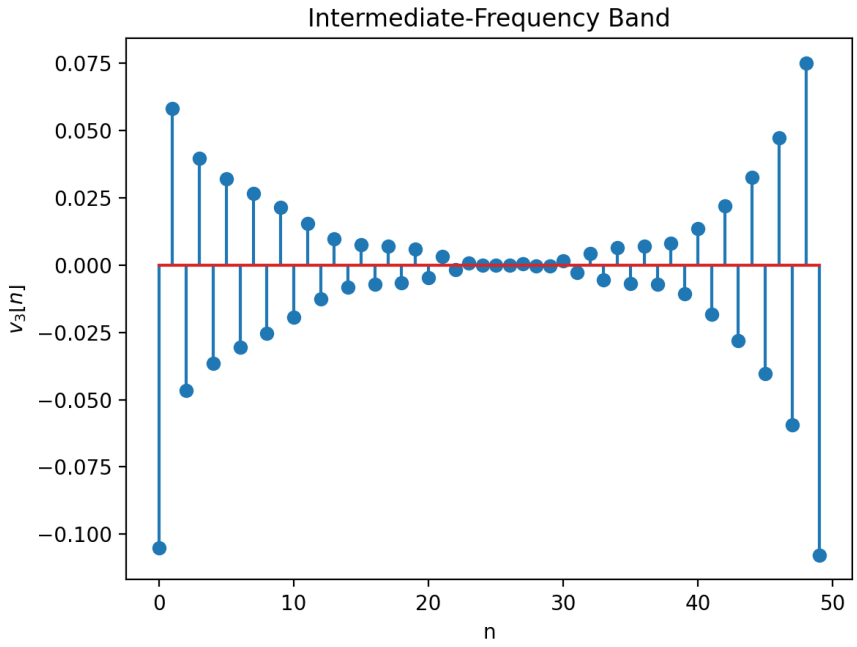
To demonstrate that the filter bank works, I entered an input signal of $x[n] = \sin 0.02\pi n +$

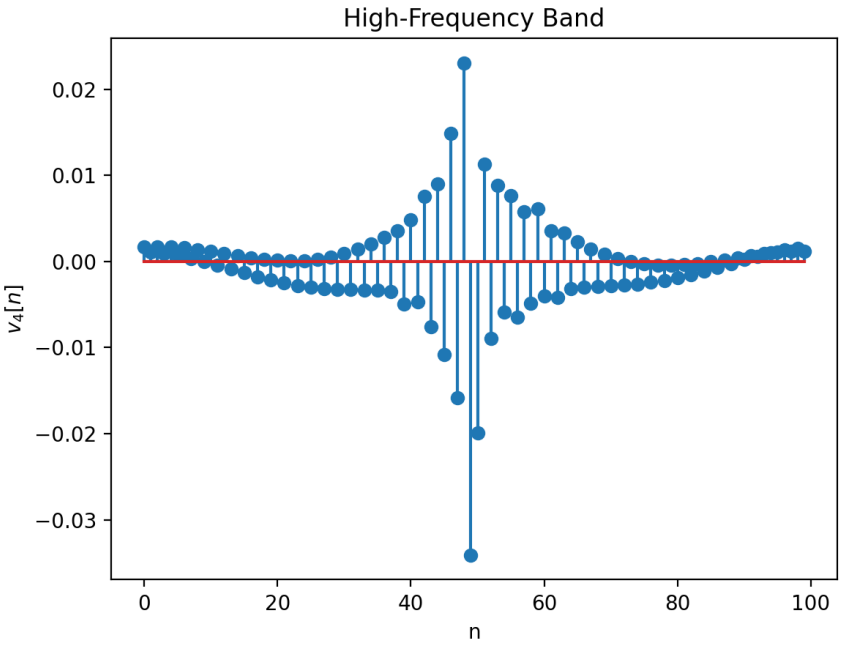
$\sin 10\pi n$. As the plots show the below, the filter bank separates out the frequency components, as the lowest band analyzes the low-passed sine and the high-frequency band analyzes the high-passed sine. Due to phase delay and aliasing introduced by the imperfect filter design, the high-frequency component is slightly attenuated. As suggested by instructors, I empirically determined an acceptable delay value for each sub-band within the filter bank. As a result, we get near-perfect reconstruction. See the figures below.

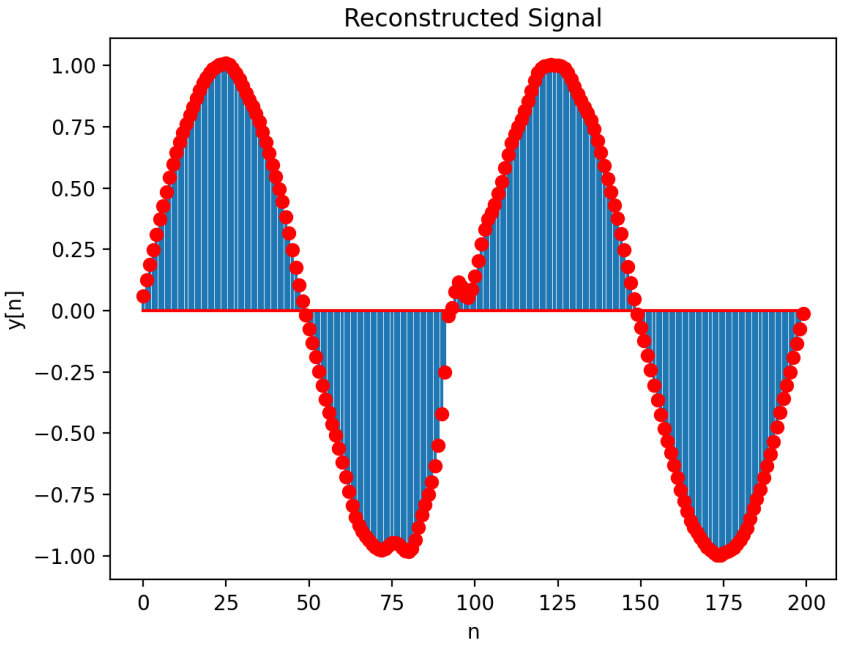



Lower 1/8 Band
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
$v_1[n]$
0
5
10
15
20
25
n


Second 1/8 Band
0.06
0.04
0.02
0.00
−0.02
$v_2[n]$
0
5
10
15
20
25
n


Intermediate-Frequency Band
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
$v_3[n]$
0
10
20
30
40
50
n


High-Frequency Band
0.02
0.01
0.00
−0.01
−0.02
−0.03
$v_4[n]$
0
20
40
60
80
100
n


Reconstructed Signal
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y[n]
0
25
50
75
100
125
150
175
200
n

```python
# PART A: FIR FILTER DESIGN USING TRUNCATION
# author: Noah Schwab

# import libraries
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.io.wavfile import write

'''
LPFtrunc(N)

arguments:

    N: size of impulse response

return:

    h: truncated and shifted impulse response of size N for a LPF filter w/ cutoff wc = 2.0

'''
def LPFtrunc(N):

    # cutoff frequency value
    wc = 2.0

    # generate an impulse response for values -N/2 to N/2
    h = np.array([(wc/np.pi)*np.sinc(wc*n/np.pi) for n in np.arange(-N/2, N/2)])

    # h = np.concatenate((np.zeros(N), h))
    return h


# main function for part a
def part_a1():

    # use LPFtrunc(N) to generate an impulse response for N=21 and N=101
    # compute the DTFT and plot the magnitude response normalized and in dB

    N_list = [21, 101]

    for N in N_list:
        h = LPFtrunc(N)

        # use numpy to compute DTFT and frequency samples of impulse response
        H = np.roll(np.fft.fft(h, 512), int(512/2))
        w = np.linspace(-np.pi, np.pi, 512)

        # plot the normalized magnitude response and dB response
        plt.plot(w, np.abs(H))
        plt.title(r'$|H(e^{j\omega})|$, N=%d' % (N))
        plt.xlabel(r'$\omega$')
        plt.ylabel('Normalized Magnitude')
        plt.show()

        plt.plot(w, 20*np.log10(np.abs(H)))
        plt.title(r'$|H(e^{j\omega})|$, N=%d' % (N))
        plt.xlabel(r'$\omega$')
        plt.ylabel('dB')
        plt.show()

# main function for part b
def part_a2():

    # read in noisy speech file
    noisy_speech = scipy.io.loadmat('Project1/nspeech2.mat')['nspeech2'].flatten()
```

```
    noisy_speech_spec = np.roll(np.fft.fft(noisy_speech, 512), int(512/2))
    w = np.linspace(-np.pi, np.pi, 512)

    # plot noisy speech
    plt.plot(w, noisy_speech_spec)
    plt.title('Noisy Speech Spectrum')
    plt.xlabel('w')
    plt.ylabel(r'$|X(e^{j\omega})|$')
    plt.show()

    # generate two LPF impulse responses
    h21 = LPFtrunc(21)
    h101 = LPFtrunc(101)

    # convolve noisy speech with both impulse responses
    filtered_speech21 = np.convolve(noisy_speech, h21)
    filtered_speech101 = np.convolve(noisy_speech, h101)

    # plot the filtered speech
    filtered_speech21_spec = np.roll(np.fft.fft(filtered_speech21, 512), int(512/2))
    plt.plot(w, np.abs(filtered_speech21_spec))
    plt.title('Filtered Speech Spectrum (N=21)')
    plt.xlabel('w')
    plt.ylabel(r'$|X(e^{j\omega})|$')
    plt.show()

    filtered_speech101_spec = np.roll(np.fft.fft(filtered_speech101, 512), int(512/2))
    plt.plot(w, np.abs(filtered_speech101_spec))
    plt.title('Filtered Speech Spectrum (N=101)')
    plt.xlabel('w')
    plt.ylabel(r'$|X(e^{j\omega})|$')
    plt.show()

    # write the speech to .wav files
    write('noisy_speech.wav', 10000, 3*noisy_speech)
    write('filtered_speech21.wav', 10000, 3*filtered_speech21)
    write('filtered_speech101.wav', 10000, 3*filtered_speech101)


if __name__ == "__main__":
    part_a2()
```

```python
# PART A: FIR DESIGN USING PARKS-McCLELLAN
# author: Noah Schwab

# import libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import remez as remez
from scipy.signal import freqz as freqz


# main function
def part_b():

    # default sampling frequency is 2pi
    fs = 2 * np.pi

    # design specs
    omega_p = 1.8
    omega_s = 2.2
    delta_p = 0.05
    delta_s = 0.005

    # numtaps, parameter should be altered until optimized
    numtaps = 30

    # array of frequencies
    bands = [0, omega_p, omega_s, np.pi]

    # desired values in frequency ranges
    des_response = [1, 0]

    # weighting
    wgt = [delta_s/delta_p, 1]

    # use Parks-McClellan algo to compute impulse response of filter
    h = remez(numtaps, bands, des_response, weight=wgt, fs=fs)

    # compute DTFT of h
    w, H = freqz(h, [1], whole=True)
    H = np.roll(H, int(512/2))
    w = np.linspace(-np.pi, np.pi, 512)

    # plot the frequency response and specs on same graph
    plt.plot(w, np.abs(H))
    plt.axhline(y=1+delta_p, color='r')
    plt.axhline(y=1-delta_p, color='r')
    plt.axhline(y=delta_s, color='r')
    plt.axvline(x=omega_p, color='r')
    plt.axvline(x=omega_s, color='r')
    plt.ylim(0, 1.2)
    plt.title(r'$|H(e^{j\omega})|$')
    plt.xlabel(r'$\omega$')
    plt.ylabel('Normalized Magnitude')
    plt.show()

    # plot the frequency response on dB scale
    plt.plot(w, 20*np.log10(np.abs(H)))
    plt.title(r'$|H(e^{j\omega})|$')
    plt.xlabel(r'$\omega$')
    plt.ylabel('dB')
    plt.show()

    # compute and print filter length, passband ripple, and stopband ripple
    print(f'Filter length: {numtaps}')

    passband_ripple = max(np.abs(H)) - 1
    print(f'Passband ripple: {passband_ripple}')
```

```
    stopband_vals = np.abs(H)[np.where(w > omega_s)[0]]
    stopband_ripple = max(stopband_vals)
    print(f'Stopband ripple: {stopband_ripple}')


if __name__ == "__main__":
    part_b()
```

```
# PART B: MULTI-CHANNEL FIR FILTER-BANK IN ONE DIMENSION
# author: Noah Schwab

# import libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin as firwin
from scipy.signal import freqz as freqz
from scipy.signal import resample as resample
from scipy.signal import butter, lfilter, dlti, dimpulse

'''
filt_and_down(x, h, m)

arguments:

    x: input signal

    h: impulse response

    m: downsampling rate

return:

    y: filtered and downsampled signal
'''
def filt_and_down(x, h, m):

    N = len(x)
    M = len(h)
    y = resample(np.convolve(h, x, mode='same'), num=int(N/m))

    return y

'''
up_and_filt(x, h, l)

arguments:

    x: input signal

    h: impulse response

    l: upsampling rate

return:

    y: upsampled and filtered signal
'''
def up_and_filt(x, h, l):

    N = len(x)
    M = len(h)
    y = np.convolve(h, resample(x, num=l*N), mode='same')

    return y

'''
pr_fb(x)

arguments:

    x: input signal

return:
```

```
    y: perfectly reconstructed signal
'''
def pr_fb(x):

    # first design h0, the impulse response of a low-pass filter which passes
    # frequncies 0 to pi/2

    fs = 2*np.pi
    cutoff = np.pi / 2
    numtaps = 50 # must be odd to be Type I FIR Filter

    # low-pass FIR filter using Hamming window
    # _, h0 = dimpulse(dlti(*butter(N=numtaps, Wn=cutoff, fs=fs)), n=numtaps)
    # h0 =np.squeeze(h0)
    h0 = firwin(numtaps, cutoff, fs=fs)
    w, H0 = freqz(h0, [1], worN=512, whole=True)
    H0 = np.roll(H0, int(512/2))
    w = np.linspace(-np.pi, np.pi, 512)
    plt.plot(w, np.abs(H0), label=r'$|H_{0}(e^{j\omega})|$')
    # plt.show()

    # define high-pass filter impulse response h1 according to quadrature mirror design
    h1 = np.exp([1j * np.pi * n for n in range(len(h0))]) * h0
    _, H1 = freqz(h1, [1], worN=512, whole=True)
    H1 = np.roll(H1, int(512/2))
    plt.plot(w, np.abs(H1),  label=r'$|H_{1}(e^{j\omega})|$')

    plt.xlabel(r'$\omega$')
    plt.legend()
    plt.show()

    # define synthesis filters
    g0 = h0
    g1 = h1

    _, G0 = freqz(g0, [1], worN=512, whole=True)
    G0 = np.roll(G0, int(512/2))

    _, G1 = freqz(g1, [1], worN=512, whole=True)
    G1 = np.roll(G1, int(512/2))

    plt.plot(w, np.abs(G0), label=r'$|G_{0}(e^{j\omega})|$')
    plt.plot(w, np.abs(G1), label=r'$|G_{1}(e^{j\omega})|$')

    plt.xlabel(r'$\omega$')
    plt.legend()
    plt.show()

    # apply filters in cascade to reconstruct signal
    v1 = np.roll(filt_and_down(filt_and_down(filt_and_down(x, h0, 2), h0, 2), h0, 2), 700)
    v2 = np.roll(filt_and_down(filt_and_down(filt_and_down(x, h0, 2), h0, 2), h1, 2), 700)
    v3 = np.roll(filt_and_down(filt_and_down(x, h0, 2), h1, 2), 400)
    v4 = np.roll(filt_and_down(x, h1, 2), 50)

    plt.stem(v1)
    plt.title('Lower 1/8 Band')
    plt.xlabel('n')
    plt.ylabel(r'$v_1[n]$')
    plt.show()

    plt.stem(v2)
    plt.title('Second 1/8 Band')
    plt.xlabel('n')
    plt.ylabel(r'$v_2[n]$')
    plt.show()

    plt.stem(v3)
```

```
    plt.title('Intermediate-Frequency Band')
    plt.xlabel('n')
    plt.ylabel(r'$v_3[n]$')
    plt.show()

    plt.stem(v4)
    plt.title('High-Frequency Band')
    plt.xlabel('n')
    plt.ylabel(r'$v_4[n]$')
    plt.show()

    # upsample and synthesize the sub-bands

    y1 = up_and_filt(v1, g0, 2)
    y2 = up_and_filt(v2, g1, 2)
    y12 = y1 + y2

    y3 = up_and_filt(v3, g1, 2)
    y123 = y3 + up_and_filt(y12, g0, 2)

    y4 = up_and_filt(v4, g1, 2)

    y = up_and_filt(y123, g0, 2) + y4

    plt.stem(np.roll(y, 92), markerfmt='ro', basefmt='r-')
    plt.title('Reconstructed Signal')
    plt.xlabel('n')
    plt.ylabel('y[n]')
    plt.show()

    return


if __name__ == "__main__":
    x = np.sin([.02 * np.pi * n for n in np.arange(0, 200)]) + np.sin([0.2 * np.pi * n for n in np.arange(0, 200)])
    plt.stem(x)
    plt.title('Input Signal')
    plt.xlabel('n')
    plt.ylabel('x[n]')
    plt.show()
    pr_fb(x)
```