DIGITAL FILTERS AND APPLICATIONS

In this lab, we will experiment with two types of digital linear time-invariant (LTI) filters: finite impulse response (FIR) filters and infinite impulse response (IIR) filters. By completing this lab, you will gain insights into the properties of different types of digital filters and gain familiarity with the designs and implementations of them for a variety of applications.

LAB INSTRUCTIONS

1. This is a three-week lab.
2. You might have to spend time outside the allocated lab hours to finish the lab. In doing so, you can approach any of the course instructors, or TA(s).
3. You may work in teams or groups of 1-3 members and are not allowed to collaborate with anyone outside your group except the instructors of the course. You may change group or team members for different labs but you cannot change group members for the given lab you are working on.
4. Please document your code well by using appropriate comments, variable names, spacing, indentation, etc.
5. The starter code is not binding on you. Feel free to modify it as you wish. Everything is fine so long as you're getting the right results.
6. Please upload the `.ipynb` file to canvas. One notebook per team is fine and any one team member can upload the file. The required file(s) must be uploaded by the deadline.

1 Digital Filters and Frequency Response (5 Points)

To start things off, we will learn how to define and use the LTI filters with python as they are the building blocks for a variety of filtering applications.

1.1 LTI Filters in Python

Every casual LTI digital filter can be described by a linear difference equation that relates the output of the filter $y[n]$ to the input of the filter $x[n]$:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

The time-domain difference equation presented above is equivalent to the frequency-domain transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}}$$

where $z = e^{j\omega}$. In the equation, a_k and b_k are the filter coefficients. a_k is an array of length $N+1$ and b_k is an array of length $M+1$. The length of the filter, also called the number of taps, is the maximum of the lengths of the two coefficient arrays, i.e. `max(N,M)+1`. The order of the filter is the length of the filter minus 1, i.e. `max(N,M)`

By rearranging the equation, we can isolate the current output $y[n]$ to the left-hand side:

$$y[n] = \frac{1}{a_0}\left(\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right)$$

This gives us the formula for calculating the current output of the filter from the current input, past inputs and past outputs of the filter. The leading denominator coefficient a_0 is usually set to 1, since the effect of a non-unit a_0 is equivalent to dividing all other coefficients by a_0.

In Python, LTI digital filters can be implemented using the `lfilter` function from the `signal` module:

```
y = signal.lfilter(b,a,x)
```

In the example code above, x is the input signal, y is the output signal, and a and b are the filter coefficients. All of these 4 are stored as 1D ndarrays.

1.2 FIR Filters in Python

LTI digital filters can also be classified into two types by structure: finite impulse response (FIR) filter, and infinite impulse response (IIR) filter. A FIR filter is a filter with no denominator coefficients a_k, except for a_0, which we set to 1. Therefore, by definition, the output of a FIR filter can be expressed as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

Shown by the equation, the output of a FIR filter is simply a weighted average of the current and past inputs, with the weights being the nominator coefficients b_k. For this reason, people also call it as the moving average filter. Also notice that the equation exactly describes the convolution operation: $y[n] = b[n] * x[n]$, with the impulse response being the nominator coefficients b_k. Because there can only be a finite number of nominator coefficients, the length of the impulse response is also finite; hence the name "finite impulse response".

Since the impulse response has finite length, you can implement the filter by directly computing the convolution instead of using `signal.lfilter`. To calculate convolution, use `signal.convolve` as follows:

```
y = signal.convolve(x,h)
```

where h is the impulse response of the filter, stored as a 1D ndarray. The advantage of convolve is that it can leverage the circular convolution theorem to use FFT to speed-up the calculation.

Now, proceed to finishing the first set of tasks:

(a) Load the data in the text file `signal-data.txt` to an ndarray. The data is a 100-second sinusoidal signal with random noise. Plot the data.

(b) Consider the following 30-second moving average filter:

$$y[n] = \frac{1}{30}(x[n] + x[n-1] + \dot{+}x[n-29])$$

which corresponds to the impulse response:

$$h[n] = \frac{1}{30}(u[n] - u[n-30])$$

Apply the moving average filter mentioned above to the 100-second signal using both `lfilter` and `convolve`, respectively.

(c) For each the two implementations, plot the original data and the filtered data in the same plot.

(d) Plot the magnitude of the frequency response of the filter to verify that this system corresponds to a low-pass filter. Hint: A system's frequency response is the Fourier Transform of its impulse response. For Fourier Transform, you can use Python's default function `np.fft.fft`.

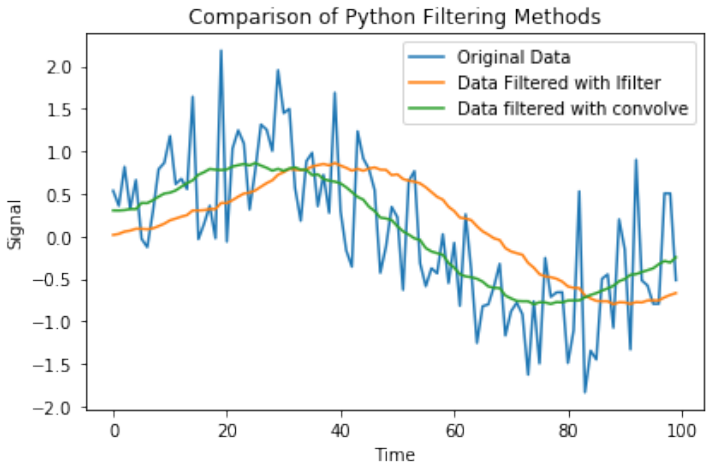


Figure 1: Expected result for 1.2 (c)

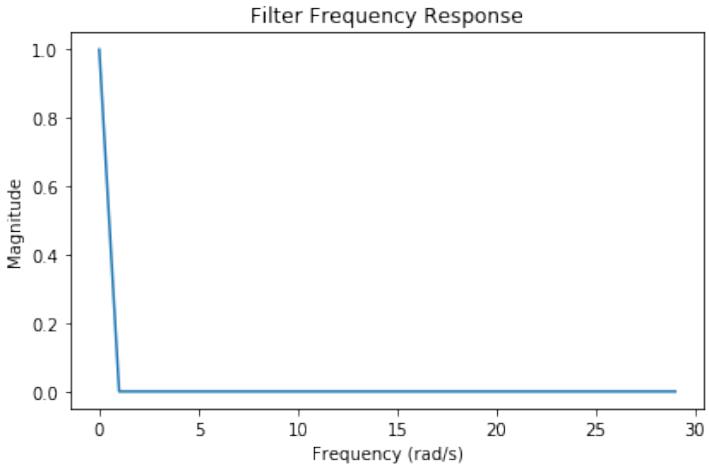


Figure 2: Expected result for 1.2 (d)

Discussion (3 Points)

- Does the filtered result look smoother?
- Is the filter low pass?
- How would you adjust the implementations so that they give identical results?

2 Analyze the Characteristics of FIR and IIR Filters (5 Points)

2.1 IIR Filter in Python

We already know that FIR filters don't have any denominator coefficients a_k for $k \geq 1$. IIR filters, in contrast, have some non-trivial denominator coefficients. Therefore, by definition, the output of an IIR filter can be expressed as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

As shown by the equation, the output of an IIR filter is the weighted average of not only the current and past inputs, but also past outputs. Since the current value of y depends on the previous values of y, IIR filter is also known as the recursive filter. Because of the recursion, the impulse response

of a stable IIR filter decays over time but never dies off, making it infinite length, and hence the name "infinite impulse response".

2.2 Using Classic Filter Design

Instead of designing filters by ourselves, we can use some classic filter designs provided by the `SciPy` package. For IIR filter, we can use the Butterworth design, which is a design that has no ripple in the frequency response. To create a Butterworth filter, use the `signal.butter` function:

```
b, a = signal.butter(order, W, type)
```

where `b` and `a` are the filter coefficients, `order` is the order of the filter, `freq` is the cutoff frequency and `type` can be either `'lowpass'` or `'highpass'`. The unit of `W` is the normalized frequency, which has the range $0 \leq W \leq 1$ where 1 corresponds to the Nyquist frequency (half of the sampling rate).

For FIR filter, we can use the window-method design, which approximates the impulse response of an ideal filter. To create a window-method filter, use the `signal.firwin` function:

```
# low-pass:
b = signal.firwin(num_taps, W)
# high-pass:
b = signal.firwin(num_taps, W, pass_zero = False)
```

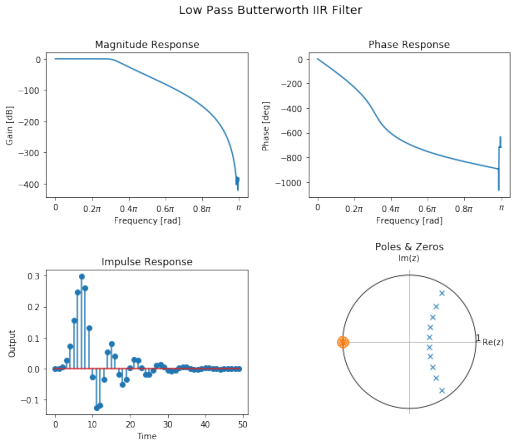
where `b` is the numerator coefficient or impulse response, `num_taps` is the length of the filter and `W` is the cutoff frequency which has the unit of the normalized frequency.

For this task, you will use the `analyze` function provided in the starter code to analyze the characteristics of 4 different filters. In particular, we will look at how FIR and IIR filters behave differently in terms of frequency response, impulse response and poles / zeros.

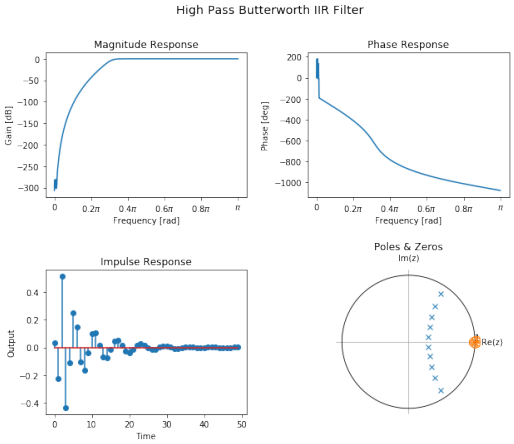
Now, let's proceed to finish the following tasks:

(a) Implement the 4 following filters. Each filter should have an order of 10 and a cutoff frequency of $\omega = 0.1\pi$.

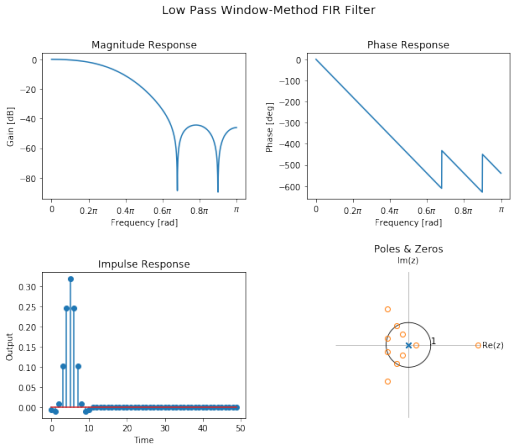
- Low-pass IIR filter using Butterworth design.



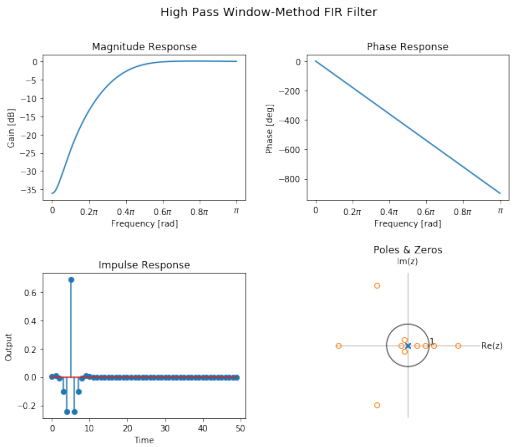
(a) Expected result of low-pass IIR Butterworth



(b) Expected result of high-pass IIR Butterworth



(c) Expected result of low-pass FIR window method



(d) Expected result of high-pass FIR window method

Figure 3: Expected results of 2.2 (b)

- Low-pass FIR filter using window-method design.
- High-pass IIR filter using Butterworth design.
- High-pass FIR filter using window-method design.

(b) Then for each filter, apply the `analyze` function, which will produce plots of the frequency response, impulse response and poles / zeros.

(c) Finally, apply each filter to the each of the following signals:

- The data in `microsoft-stock.txt`, which is the daily stock price of Microsoft over 4 years.
- $x[n] = u[n] - u[n-20]$, where $x[n]$ is of total length 60 (so append 40 zeros to the end).

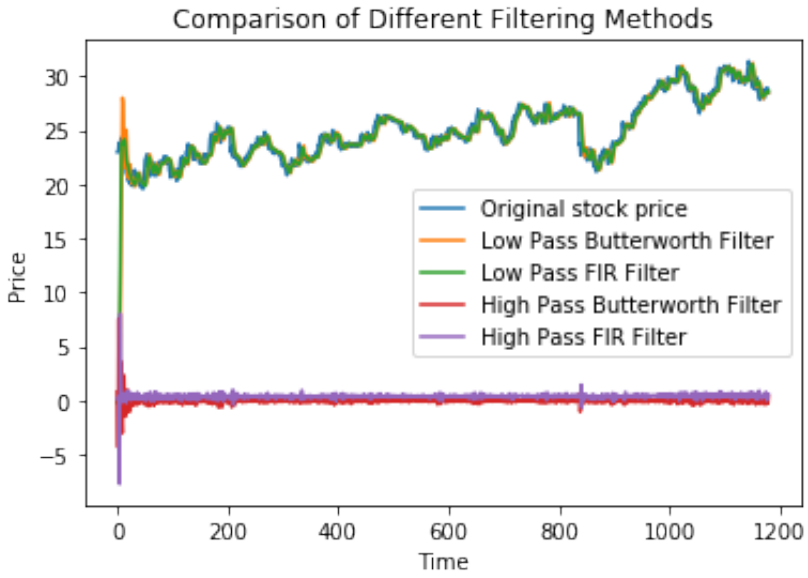


Figure 4: Expected result for 2.2 (c) `Microsoft-stock.txt`

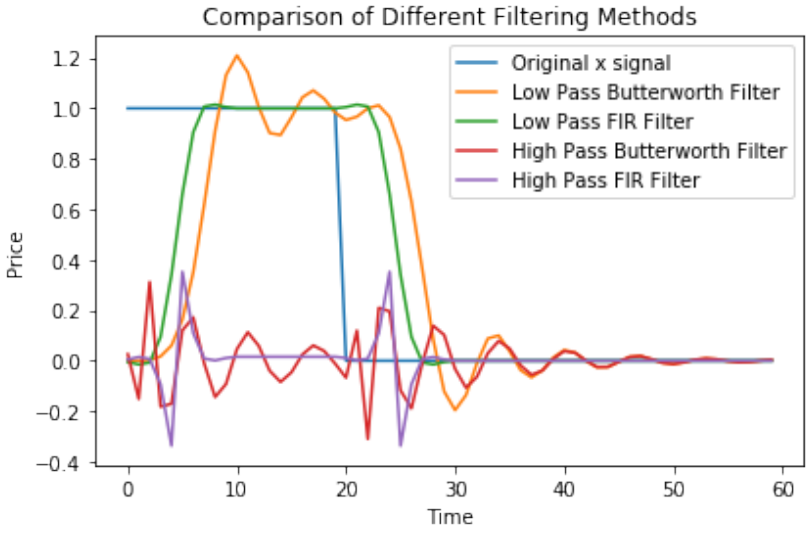


Figure 5: Expected result for 2.2 (c) $x[n] = u[n] - u[n-20]$

Discussion (1 Points)

- Comment on the differences in the frequency response of the two filters (magnitude and phase) and how this impacts the outputs. When commenting on frequency responses, consider how close the filter matches an ideal filter.

3 The 3-Band Audio Equalizer (5 Points)

In this section, you will implement an audio equalizer using three different filters. Equalizer works by first breaking down the audio signal into multiple frequency bands using filters. Each band will then receive a different gain. Finally, all bands are summed together to produce the output signal. Audio mixing boards do exactly this, though with many more bands than the three bands we have here. The term "equalizer" comes from the fact that people often want to boost the gain of low-energy frequency ranges of the sound or to reduce the gain of high-energy frequency ranges of the sound. By doing so, each frequency range of the sound will have about equal loudness, allowing them to mix together nicely.

Figure 6 shows the signal flow diagram of the 3-band equalizer you will implement. G_1, G_2 and G_3 are correspondingly the gains of the bass, mid

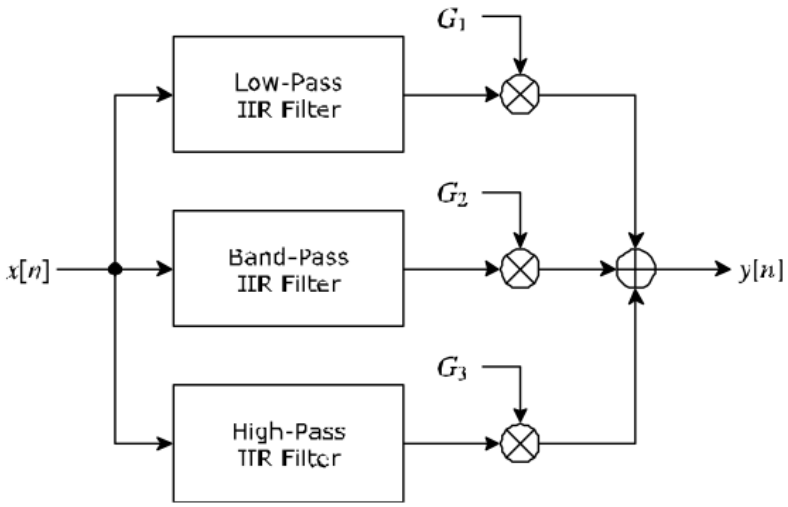


Figure 6: Signal flow of the equalizer

and treble frequency ranges. The coefficients of each filter are provided in Table 1. However, you need to figure out by yourself which filter does each set of coefficients correspond to.

Table 1: Filter coefficients of the equalizer

Filter Coefficients	Which filter?
b_1=[11.713, 0, -23.4257, 0, 11.713]	
a_1=[1800, -6656.7, 9341, -5896.1, 1413.4]	
b_2=[0.2982, 0.89471, 0.89471, 0.2982]	
a_2=[1800, -4989.8, 4625.2, -1433]	
b_3=[688.1, -2752.5, 4128.71, -2752.47, 688.12]	
a_3=[1000, -3256.6, 4033.77, -2246, 473.51]	

In this section, your tasks are:

(a) Identify which filter does each set of coefficients in Table 1 correspond to. Include your answers to the blanks in Table 1. Hint: Recall Sec 2.2, you can use `analyze` function to analyze the characteristics of these three filters.

(b) Implement the equalizer by writing a function that takes a sound signal and the gain terms G_1, G_2 and G_3 as inputs, applies the filters,

multiplies the filter outputs by the gain terms, and sums the results. The gain terms supplied to the function should be in decibels (dB).

Hint: To convert the gain term to the unit of decibels (dB), you can use the formula

$$G_{dB} = 20\log(\frac{x \times G}{x})$$

where x is the input signal, and $x \times G$ denotes the signal multiplied by the gain term.

(c) After you implemented your equalizer, load the sound file `music.wav`. Apply the equalizer to the music with $G_1 = G_2 = G_3 = 0$ dB, and play the output. Verify that it sounds the same as the original input.

(d) Experiment with different sets of gains, and filter out the high frequency content from the signal.

(e) Experiment with different sets of gains, and filter out the low frequency content from the signal.

Discussion (1 Points)

- Discuss your design choices for 3(d) and 3(e). How the filtered sound sounds different.

4 Spectrum Analysis with Short-time Fourier Transform (10 Points)

Listen to this snippet of the nightingale birdsong by running:

```
from IPython.display import Audio
Audio('nightingale.wav')
```

If you are just reading the lab document, you'll have to use your imagination! It goes something like this: chee-chee-woorrrr-hee-hee cheet-wheet-hoorrr-chirrr-whi-wheo-wheo-wheo-wheo-wheo-wheo... Since we realize that not everyone is fluent in bird-speak, perhaps it's best if we visualize the signal instead. By running the given starter code, you should be able to get the visualization in time domain.

Well, that's not very satisfying, is it? If I sent this voltage to a speaker, I might hear a bird chirping, but I can't very well imagine how it would sound in my head. Is there a better way of seeing what is going on?

There is, and it is exactly the discrete Fourier transform (DFT). The DFT is often computed using the FFT algorithm, a name informally used to refer to the DFT itself. The DFT tells us which frequencies or "notes" to expect in our signal. Of course, a bird sings many notes throughout the song, so we'd also like to know *when* each note occurs. The Fourier transform takes a signal in the time domain and turns it into a spectrum — a set of frequencies with corresponding complex values. The spectrum does not contain any information about time!

So, to find both the frequencies and the time at which they were sung, we'll need to be somewhat clever. Our strategy is as follows: (1) take the audio signal, (2) split it into small, overlapping slices, and (3) apply the Fourier transform to each. This is a technique known as the short time Fourier transform (STFT). We'll split the signal into slices of 1,024 samples — that's about 0.02 seconds of audio. The slices will overlap by 100 samples as shown in Figure 7.

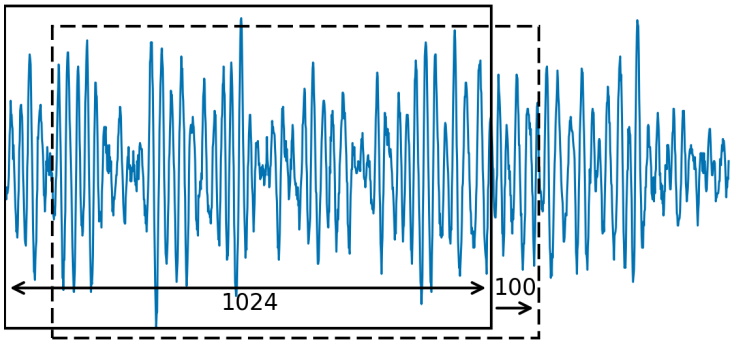


Figure 7: Key idea of short time Fourier transform (STFT). We will need to split the signal into slices of 1,024 samples. The slices will overlap by 100 samples.

Let's implement STFT with the following steps:

(a) Load audio file `nightingale.wav`, and play it.

(b) We will start by chopping up the signal into slices of 1024 samples, each slice overlapping the previous by 100 samples.

Hint: you can use `for` loop or Python's built-in function `util.view_as_windows`. The expected sliced audio shape is $(13446, 1024)$.

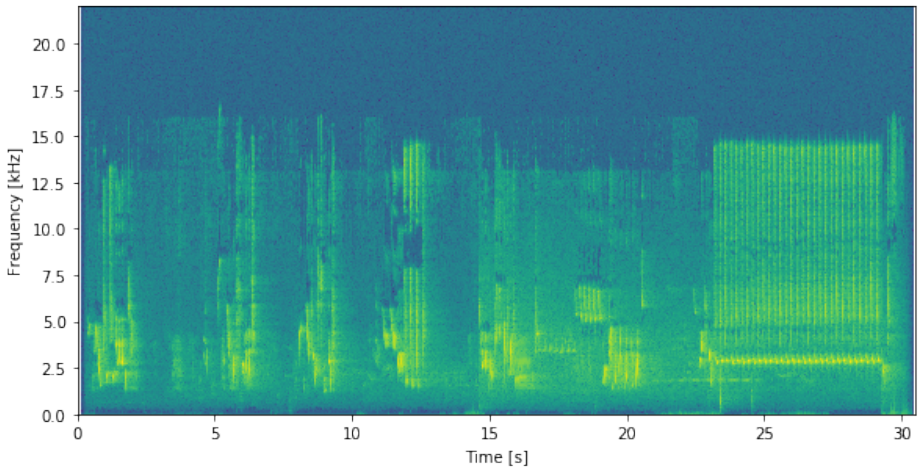


Figure 8: Expected result of 4(h)

(c) Create a windowing function

$$w[n] = 0.5 - 0.5cos(\frac{2\pi n}{1024})$$

where n is the time stamp from 0 to 1023. Next, multiply the window function with the sliced signal.

(d) It's more convenient to have one slice per column for DFT computation, so take the transpose for the slices.

(e) For each slice, calculate the DFT using `np.fft.fft`

(f) After DFT, we will get both positive and negative frequencies. Slice out the positive frequencies for now.

(g) Find absolute value of the spectrum

(h) Do a log plot of the ratio of the signal divided by the maximum signal. The specific unit used for the ratio is the decibel, $20 \log 10$ (amplitude ratio).

Discussion (1 Points)

- Comment the difference between your results and the results obtained by using Python's default function.

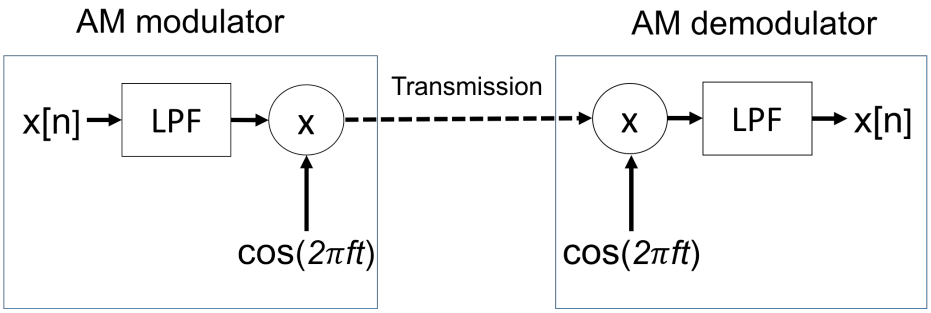


Figure 9: Standard modulation/demodulation scheme.

5 Audio Recovery Challenge (10 Points)

In this section, we will apply the newly learned digital filter design and spectrum analysis skills to recover the secrete messages in a digital audio transmission file `lab3-data.csv`.

This file has $441,000$ data samples, taken at a sampling rate of $Fs = 44100$. The high sample rate permits users to encode multiple audio message channels within the Nyquist bandwidth (22050 Hz) by selecting among the following modulation schemes: (1) Amplitude Modulation (AM), (2) Phase Modulation (PM), and (3) Quadrature amplitude modulation (QAM).

In the transmission file, the transmission consists of an unorthodox frequency-division multiplexing (FDM) arrangement in which mixtures of message channels are each arbitrarily embedded into the transmission using 1 of the 3 modulation schemes, with an arbitrary modulation frequency and an arbitrary audio message bandwidth per message channel.

Due to the loss of the decoding instructions, you will need to apply spectrum visualization to determine the bandwidth, modulated frequency location, and probable AM modulation scheme of each message channel. The recovery process will then require design of digital low-pass, band-pass, and high-pass filters that are applied, together with appropriate demodulation sinusoidal frequencies, to reconstruct all the audio messages within the digital audio transmission.

5.1 Modulation & Demodulation Schemes

Recall EE235, Modulation is an important component in the communication systems. Modulation transforms a low frequency signal into a signal of

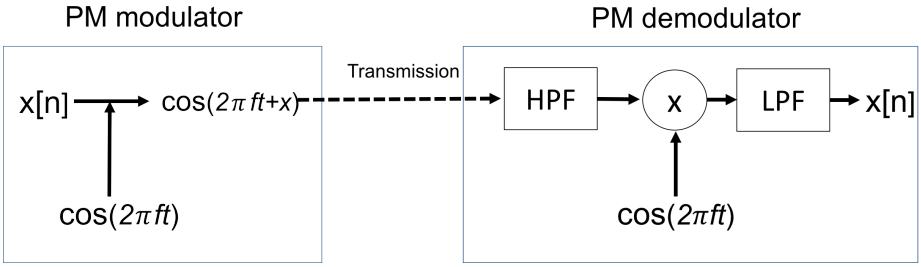


Figure 10: Basic PM modulation/demodulation scheme in this lab.

higher frequency. It is useful for broadcasting audio signals because signals with higher frequency can travel long distance over the air. On the other hand, Demodulation is important to the receiver to capture the transmitted signal. It can be seen as an inverse process of modulation, which is used to recover the original signal. This section introduces you the background of different demodulation schemes including (1) AM demodulation, (2) PM demodulation, and (3) QAM demodulation.

5.1.1 AM Demodulation

Assume for simplicity that the receiver captures the transmitted signal, $t[n]$, with no distortion, noise, or delay; that's about as perfect as things can get. Let's see how to demodulate the received signal to extract $x[n]$. As shown in Figure 9, the trick is to multiply the received signal by a local sinusoidal signal that is identical to the carrier.

5.1.2 PM Demodulation

Phase modulation (PM) is widely used for transmitting radio waves and is important to many digital transmission coding schemes. In short, it encodes a message signal as variations in the instantaneous phase of a carrier wave, and the basic processing flow is shown in Figure 10.

Note that for PM modulation this lab, we consider narrowband modulation that involves a limited modulating bandwidth and allows for easier analysis. That is, we'll begin with a high-pass filter to remove the carrier signal, and recover the signal by using the same strategy as AM modulation.

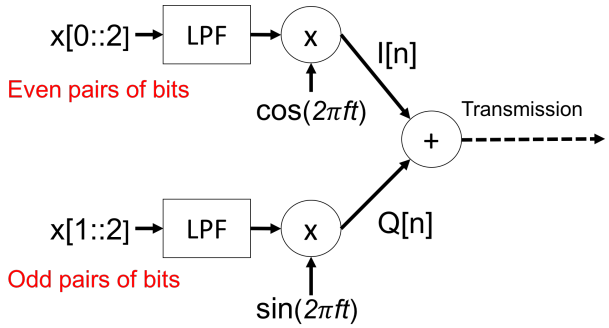


Figure 11: Standard QAM modulation scheme.

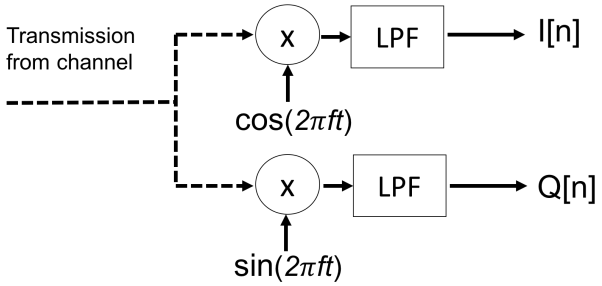


Figure 12: Standard QAM demodulation scheme.

5.1.3 QAM Demodulation

Quadrature amplitude modulation (QAM) is widely used in modern telecommunications to transmit information (such as Wi-Fi). The QAM modulator conveys two analog message signals, or two digital bit streams, by modulating the amplitudes of two carrier waves using AM modulation or other advanced modulation scheme. The two carrier waves of the same frequency are out of phase with each other by 90 degree. Being the same frequency, the modulated carriers add together, but can be coherently demodulated because of their orthogonality property.

In this lab, as shown in Figure 11, we consider modulating the input signal by separating the input to the even, and odd parts. Figure 12 shows the QAM demodulation scheme. To demodulate the signal, we will multiply the transmitted signal by two carrier waves of the same frequency, respectively. After that, you need to figure out how to combine $I[n]$ and $Q[n]$ to recover $x[n]$. Hint: it is similar to how we modulate them.

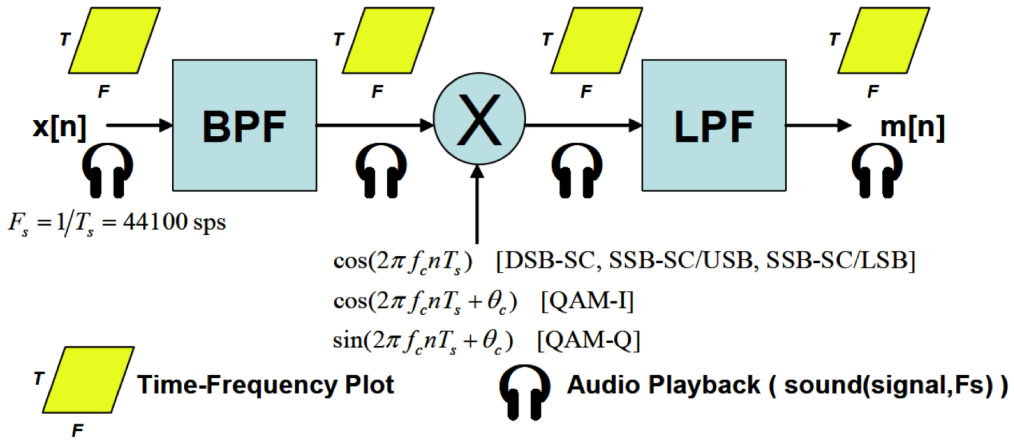


Figure 13: Receiver processing steps to recover the audio message.

5.2 Summary of Message Recovery Processing

Figure 13 summarizes the basic processing flow to recover the messages in `lab3-data.csv`. First of all, a band-pass filter is required to extract out and isolate an individual message channel in the frequency-division multiplexing (FDM) signal. It also reduces the amount of noise in the signal. Secondly, multiply the received signal by a local sinusoidal signal that is identical to the carrier. Finally, a low-pass filter is required to filter out undesirable byproducts of channel demodulation. There are many other clever ways to recover some of the message signals by using different combinations of filters, but these will not be covered here; you may use such alternative recovery processing, but be sure to provide your TA with a processing flow to describe your strategy.

Now, let's proceed to finish the rest of this lab!

(a) Load transmission file `lab3-data.csv`. Play it, and show its spectrum. Warning! Please turn down the volume. It is very noisy.

(b) Implement low-pass, high-pass, and band-pass Butterworth filters. We will use them to extract the signal of interest. Hint: Recall Task2!

(c) Test the bandpass filter you created. Here, we assume `lowcut=5000.0`

and `highcut=10500.0`. Visualize the frequency response of the created filter.

(d) Considering the signal in `data[Fs*2:Fs*8]`, apply the bandpass filter on the audio file, and visualize the spectrum.

(e) Implement AM demodulation (also called cosine demodulation). Plot the modulated and demodulated signals. Plot the spectrum of the demodulated signal.

(f) Apply the low-pass filter on the demodulated signal. Plot the signal before and after low-pass filtering in time domain. Also plot the spectrum of the filtered signal.

(g) Play the result, and find the secrete message.

(h) Recover another secrete message in `data[Fs*2:Fs*8]`. Hint: AM demodulation.

(i) Recover another secrete message in `data[Fs*8::]`. Hint: QAM demodulation.

(j) Recover another secrete message in `data[Fs*0:Fs*2]`. Hint: PM demodulation.

Discussion (4 Points)

- What messages do you recover? Report and discuss your results.

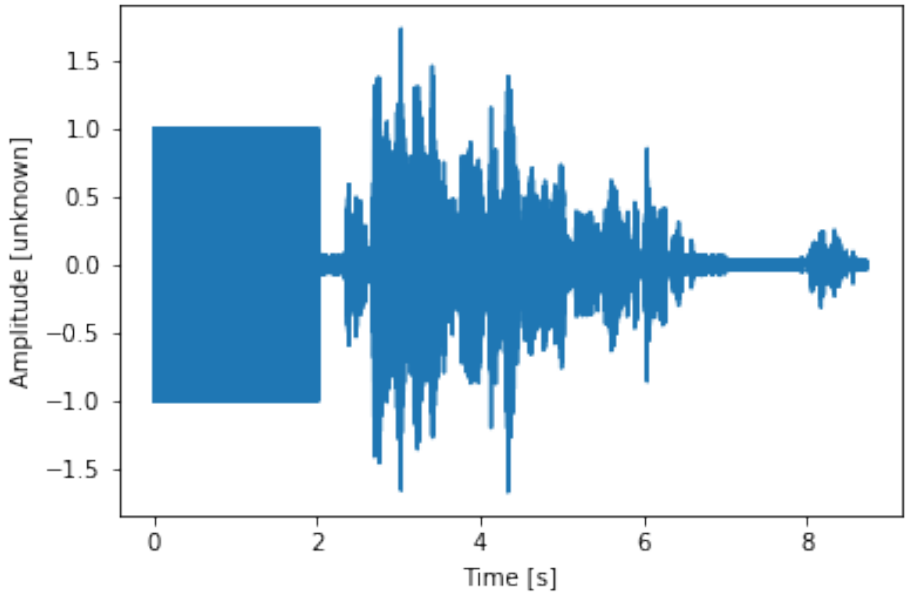


Figure 14: Expected result of 5.2(a).

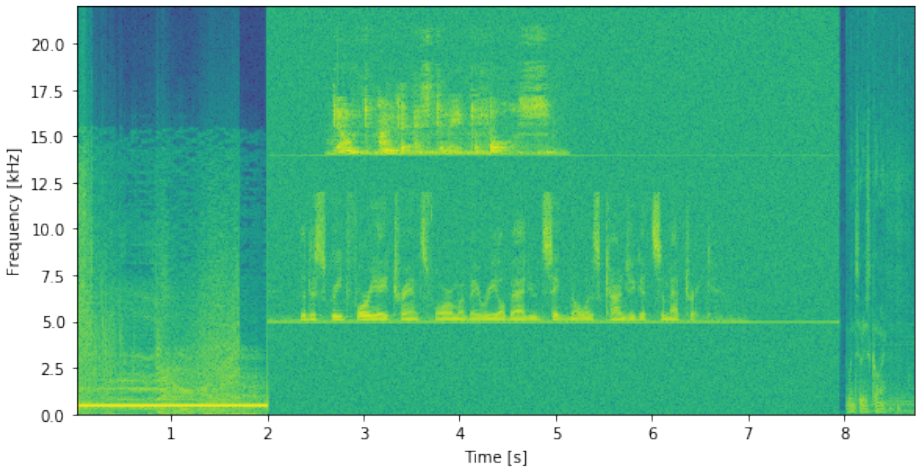


Figure 15: Expected result of 5.2(a).

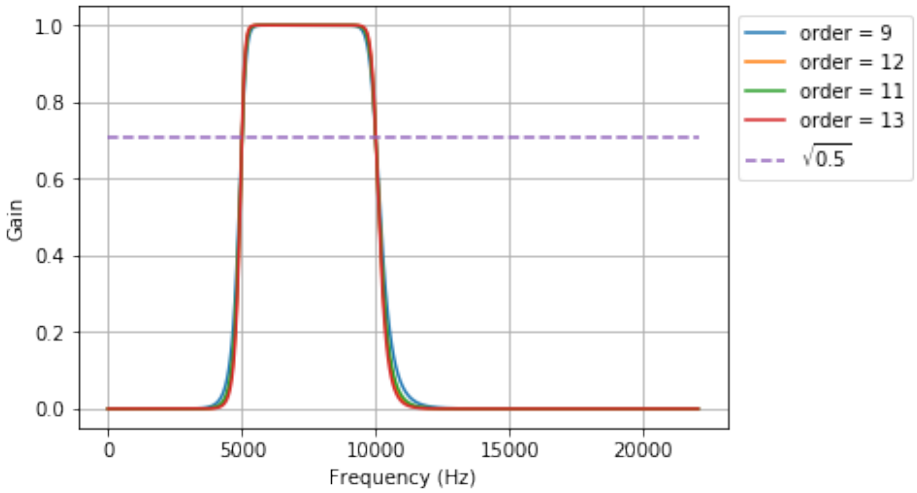


Figure 16: Expected result of 5.2(c).

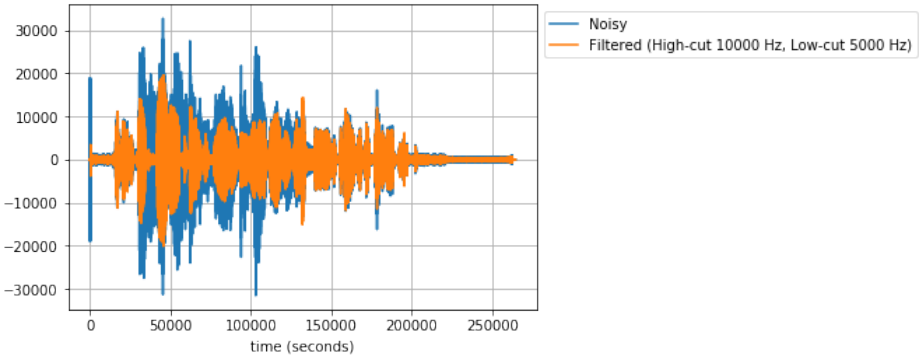


Figure 17: Expected result of 5.2(d).

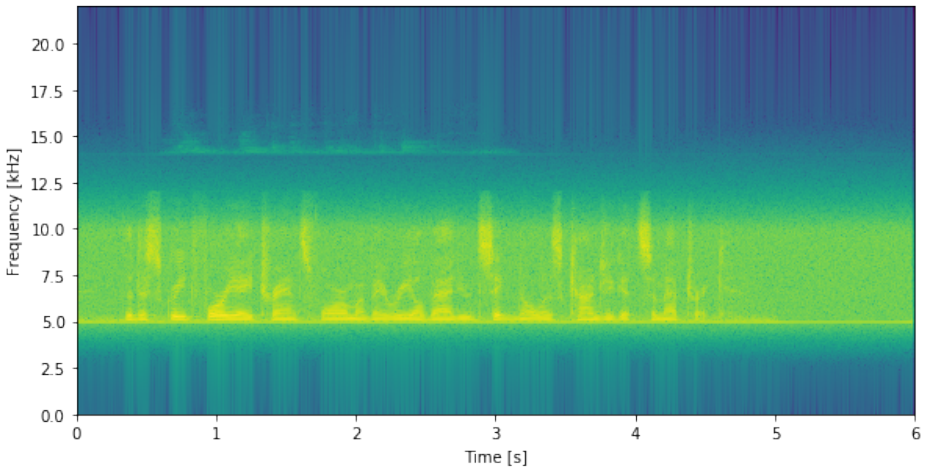


Figure 18: Expected result of 5.2(d).

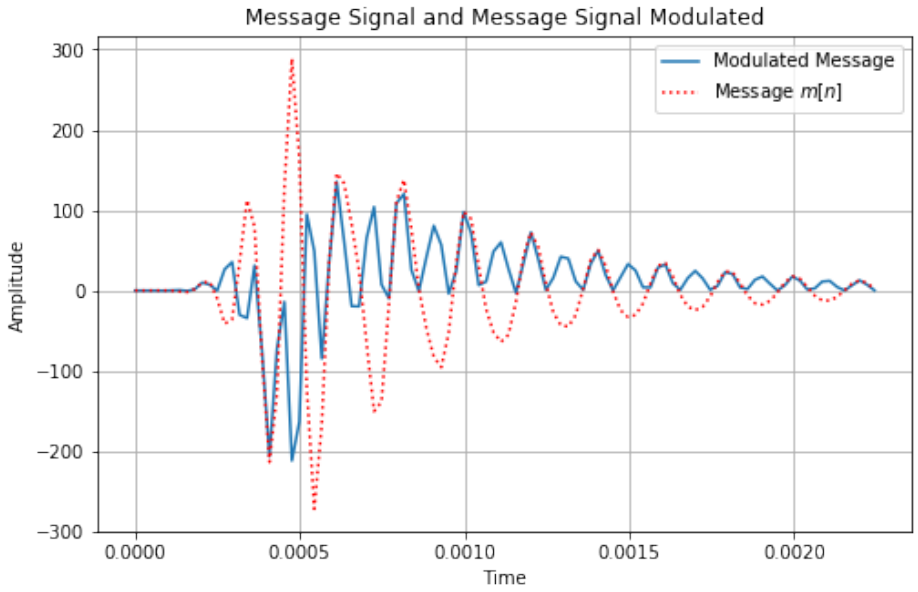


Figure 19: Expected result of 5.2(e).

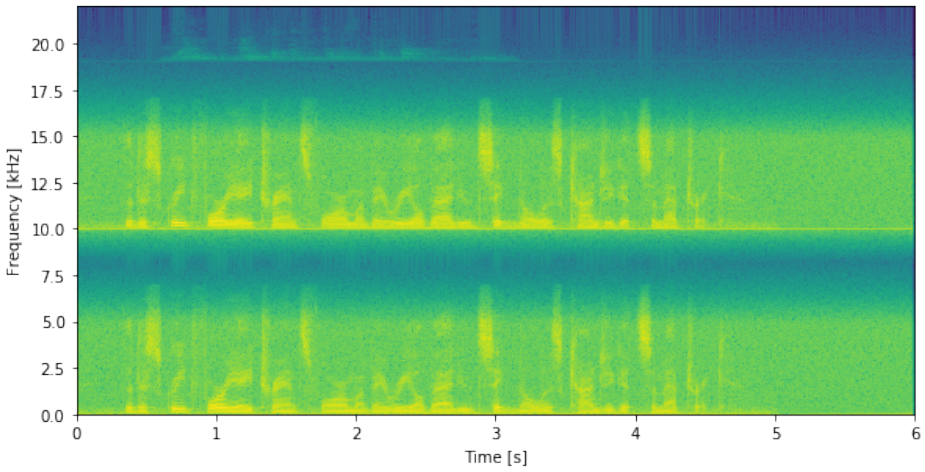


Figure 20: Expected result of 5.2(e).

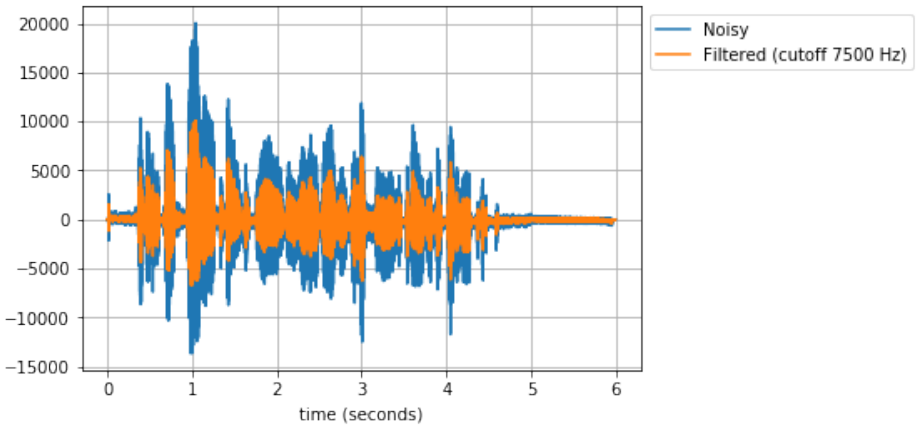


Figure 21: Expected result of 5.2(f).

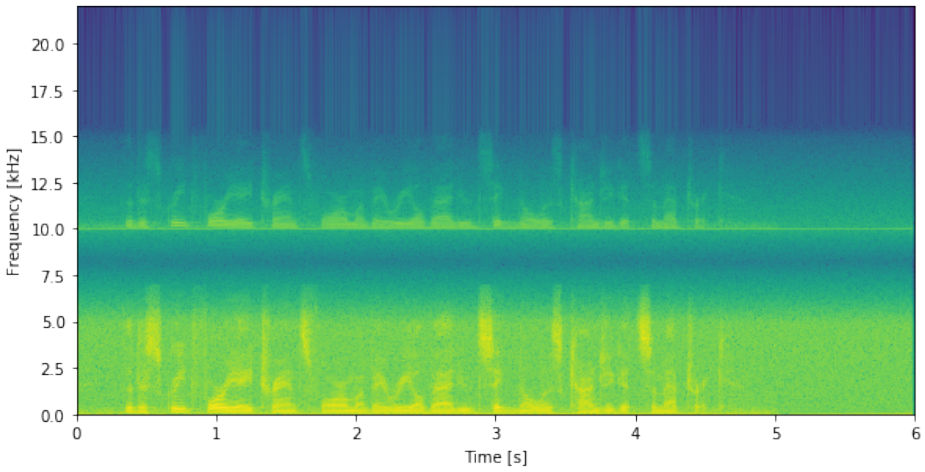


Figure 22: Expected result of 5.2(f).