MATLAB Project Document Report U7310067

Q1.1)

1.1.1

Number of bits per sample is 16

1.1.2

Sampling rate: 16000 Hz

1.1.3

Number of channels: 1

1.1.4

Duration: 9.8438 seconds

1.1.5

Total number of samples: 157500

Q1.2)

The one-line code to chop the audio clip from the 4th to the 9th second using audioread() is:
[chopped_audio, fs] = audioread('DSP_Speech.wav', [4 * sampling_rate + 1, 9 * sampling_rate]);

Q1.3)

To play the chopped audio clip, the two-line code used is:

player = audioplayer(startSample, sampling_rate);

play(player);

Q1.4)

The command to stop the audio playback is stop(player);

Q2.1)

I used five main steps to complete the recording process:

- Initialization of the audiorecorder object with the required specifications.
- Displaying a message to indicate that the recording is about to begin.
- Starting the recording session for 12 seconds.
- Stopping the recording and displaying a message to indicate the recording has stopped.
- Saving the recorded audio data in a .wav file format.

The following built-in MATLAB functions were employed:

audiorecorder() to initialize an audiorecorder object with the desired specifications.

recordblocking() to begin recording voice data for a specific duration.

getaudiodata() to retrieve audio data from the audiorecorder object.

audiowrite() to save the audio data as a .wav file.

audioplayer() and play() to double-check the quality of the recorded audio.

How did you choose the appropriate input parameter(s) for each function?

For audiorecorder(), the input parameters were set to a sampling frequency of 48000Hz, bit-depth of 16 bits, and one audio channel as per the requirements.

recordblocking() was used with the duration of 12 seconds as specified.

audiowrite() was called with a filename ('DSP_Aedan.wav') and the sampling rate (48000Hz) to save the audio clip.

audioplayer() was initialized with the recorded audio data and the same sampling frequency (48000Hz) to double-check the quality of the recording.

```
% Initialize audiorecorder object
recObj = audiorecorder(48000, 16, 1);

% Display message indicating recording is about to begin
disp('Starting recording. Make sure the background noise is playing.')

% Start recording
recordblocking(recObj, 12);

% Stop recording
disp('Recording stopped.');

% Save the audio data
audioData = getaudiodata(recObj);
audiowrite('DSP_Aedan.wav', audioData, 48000);

% Double-check the recording quality
player = audioplayer(audioData, 48000);
play(player);
```

Figure 1: Code for Q2.1

Q3.1)

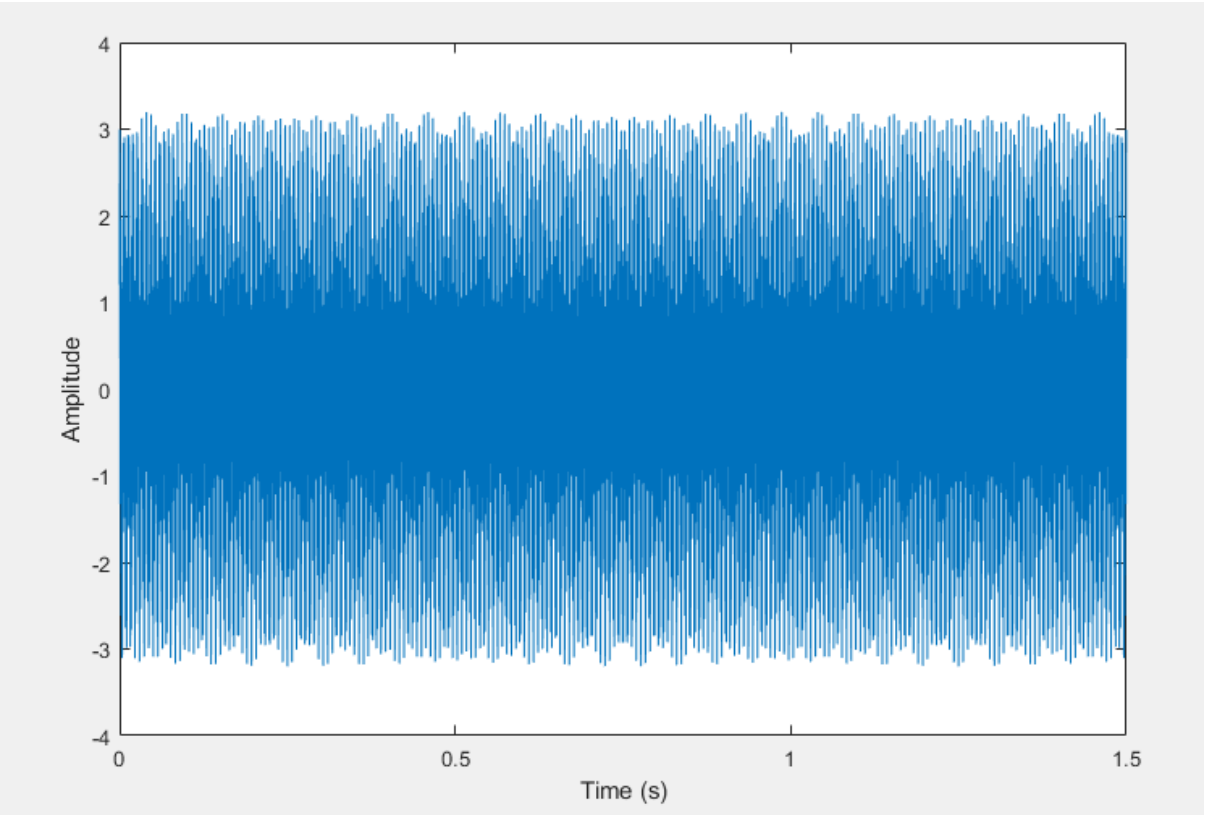

Figure 2: 7200 samples of x(t)

I generated a continuous-time signal x(t) that is a combination of a cosine wave with frequency f1 =150 Hz and amplitude of 1.2, and a sine wave with frequency f2=731 Hz and amplitude of 2. I used a sampling rate of 4800 Hz to discretise the signal over a period defined by 7200 samples. The signal was then plotted against time to visualize its waveform, with time on the x-axis and amplitude on the y-axis.

Q3.2)

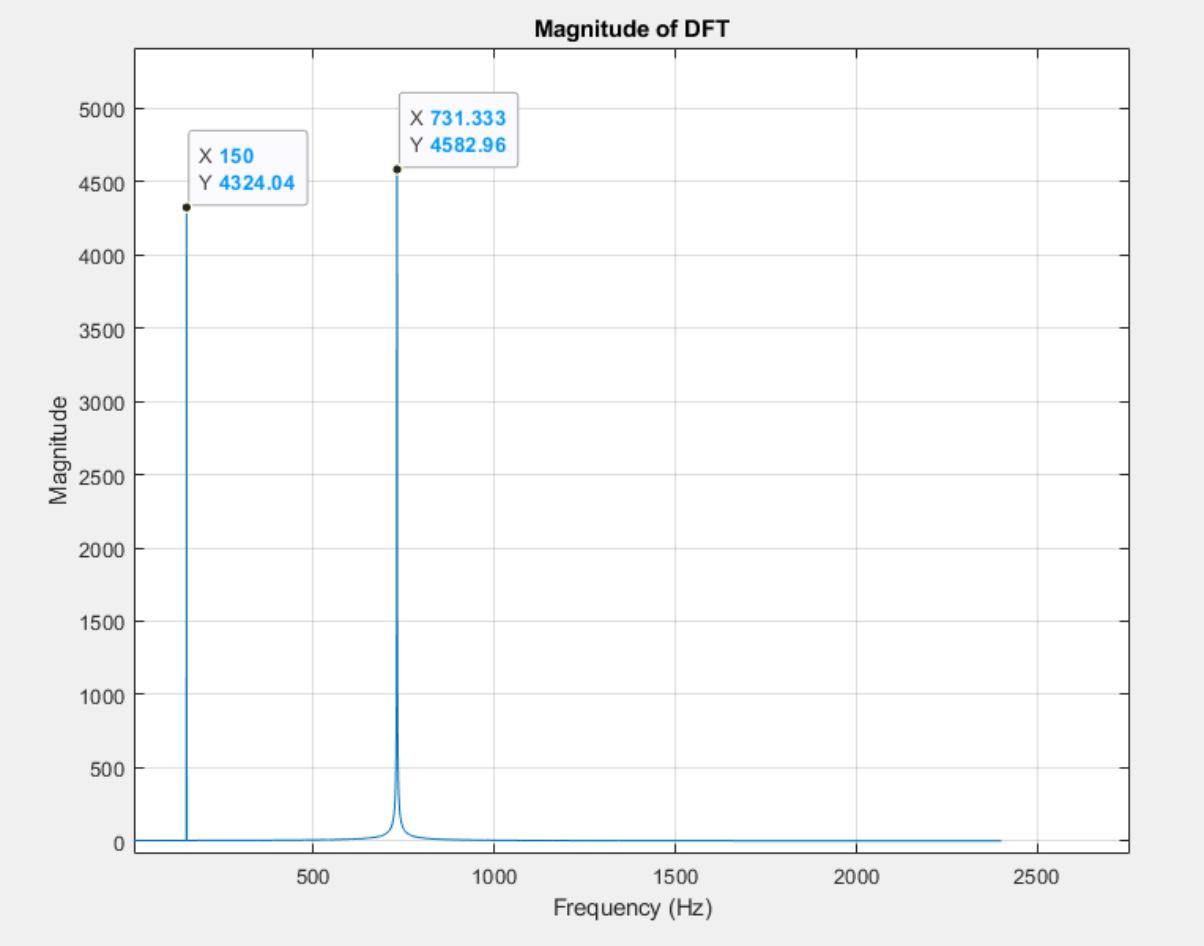

Figure 3: DFT Magnitude Plot

In Part 3.2 of my analysis onwards, I aimed to evaluate and compare the performance of two distinct methods of computing the Discrete Fourier Transform (DFT): a manual implementation (dft1) and an in-built MATLAB function (dft2).

Initially, I set up the main function (main_function) to execute various parts of my analysis, including Parts 3.1 and 3.2.

In dft1, I manually computed the DFT of the given signal. I used nested loops to compute the summation for each value of k (from 0 to N−1), where N is the length of the signal. The magnitude of this manually computed DFT was then plotted against frequency to visualize its spectrum.

For dft2, I leveraged MATLAB's fft function to efficiently compute the DFT. To ensure that the magnitudes represented the actual amplitude of the original time-domain signal, I scaled the output. The magnitude of this scaled DFT was plotted against frequency. I also saved this plot as an image (DFT_Magnitude_Plot.png).

To further evaluate the computational efficiency of these two methods, I measured the average running time of both dft1 and dft2 over multiple runs using the compare_running_times function. This allowed me to determine which method was faster on average, providing insights into the computational cost associated with each method.

Q3.3

```
function [DFT_mag_half_scaled] = dft2(signal, fs, plotData)

    % Determine the number of samples of the signal (N)
    N = length(signal);

    % Compute the DFT of the signal using the MATLAB in-built fft function
    Xk = fft(signal);

    % Apply scaling factor to restore the amplitude of the original signal in the time domain
    Xk_scaled = 2 * Xk / N;

    % Compute the magnitude of the scaled DFT of the signal
    DFT_mag_scaled = abs(Xk_scaled);

    % Determine the frequency resolution of the plot, delta_f = fs / N
    delta_f = fs / N;

    % Create a frequency vector from 0 Hz to fs / 2 Hz, using an increment of delta_f Hz
    freq_vector = 0:delta_f:fs/2;

    % Determine the length of this frequency vector (call it N1)
    N1 = length(freq_vector);

    % Create a new vector that consists of the first N1 values of the scaled DFT magnitude
    DFT_mag_half_scaled = DFT_mag_scaled(1:N1);
```

Figure 4: DFT2 Function

Q3.4)

The average running time of dft1: 4.9356 s

The average running time of dft2: 0.0002 s

dft2 has the lower running time because it uses an in-built FFT algorithm.

This was accomplished by the use of tic and toc

Q3.5)

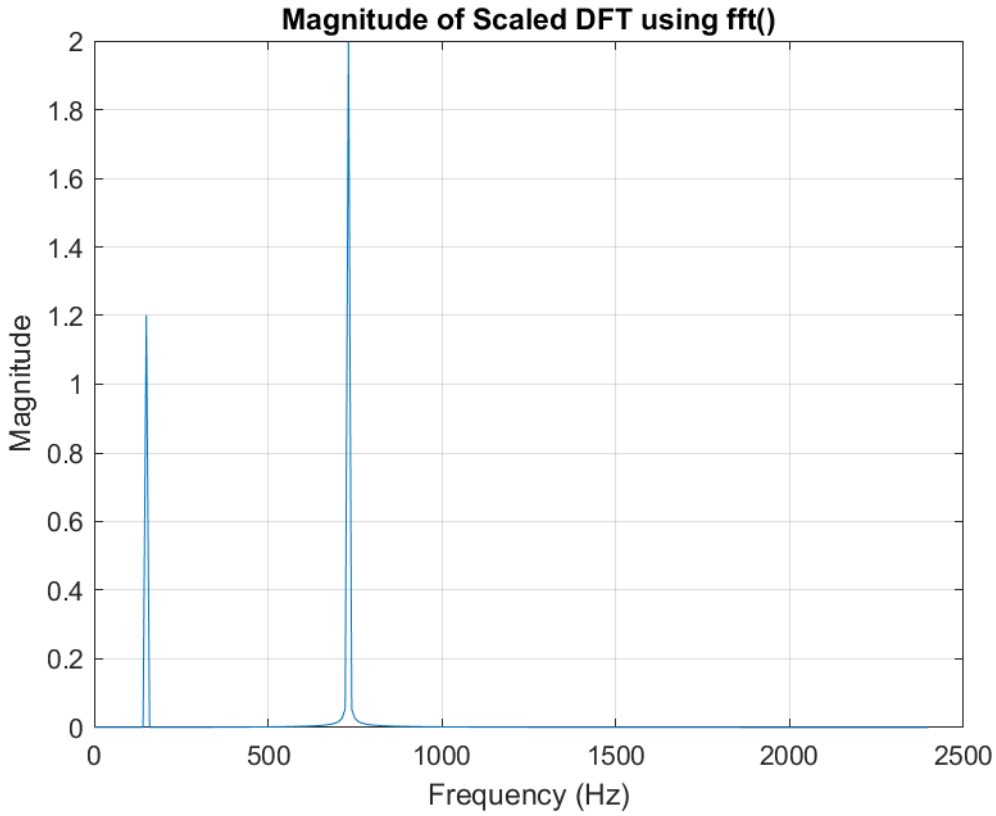

Figure 5: DFT Magnitude Plot

Q3.6)

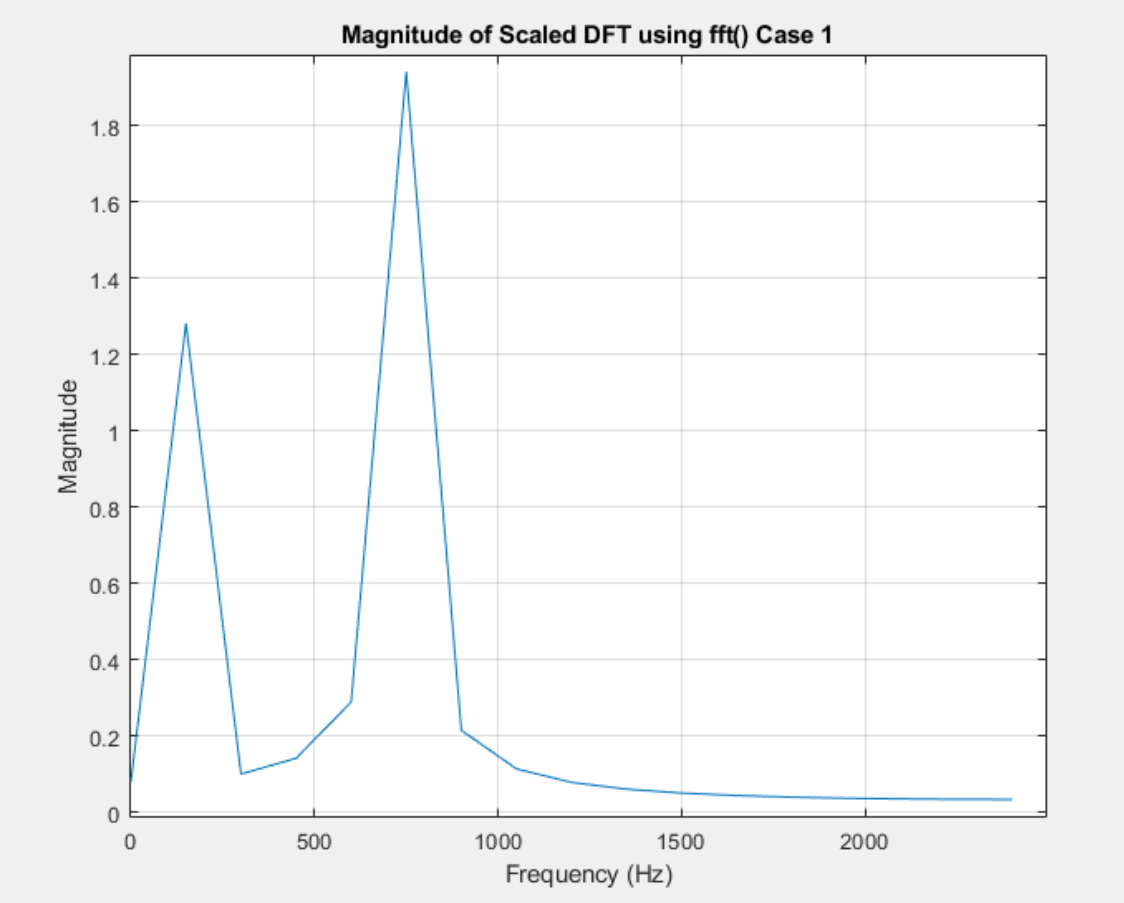

Figure 6: x(t) 32-point DFT

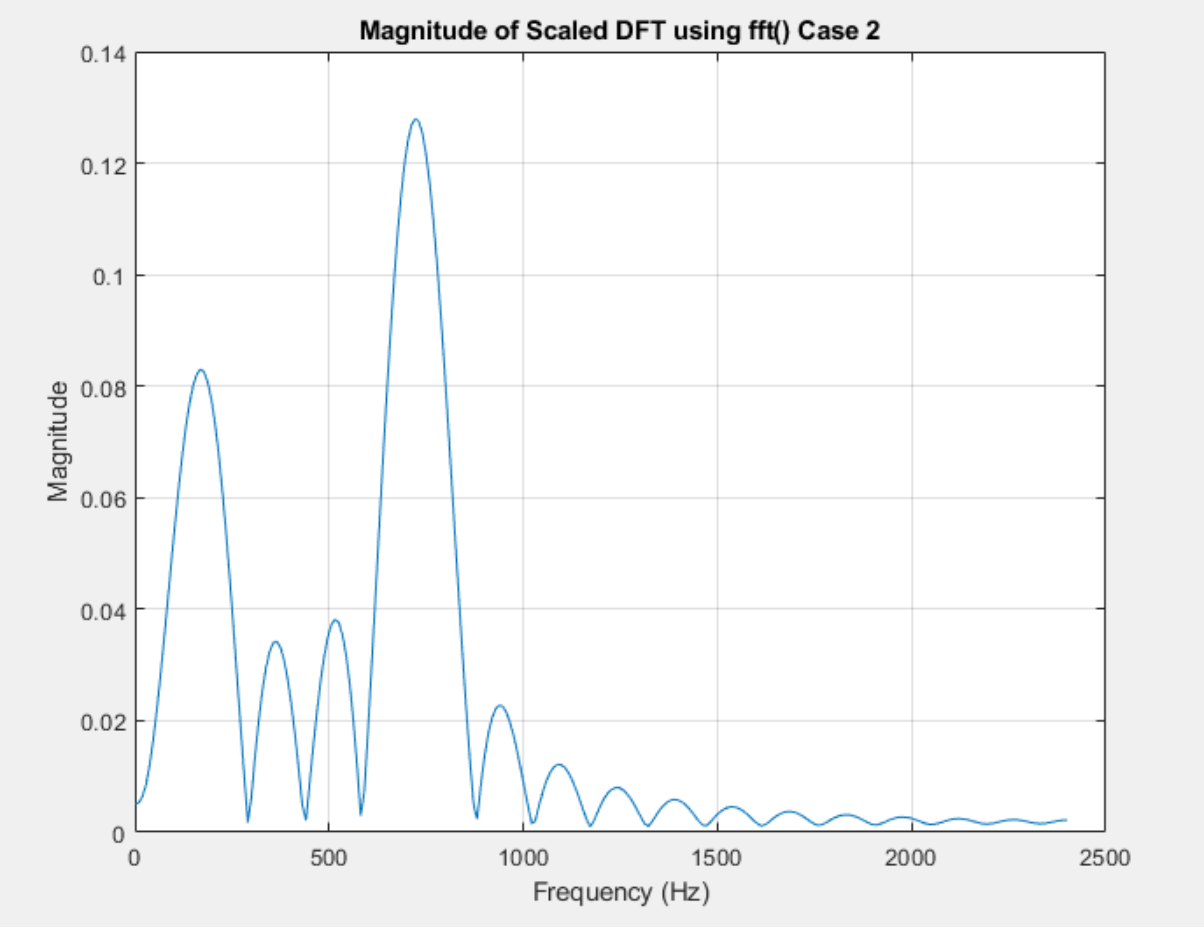

Figure 7: Zero-padded 32-sample x(t) 512 point DFT

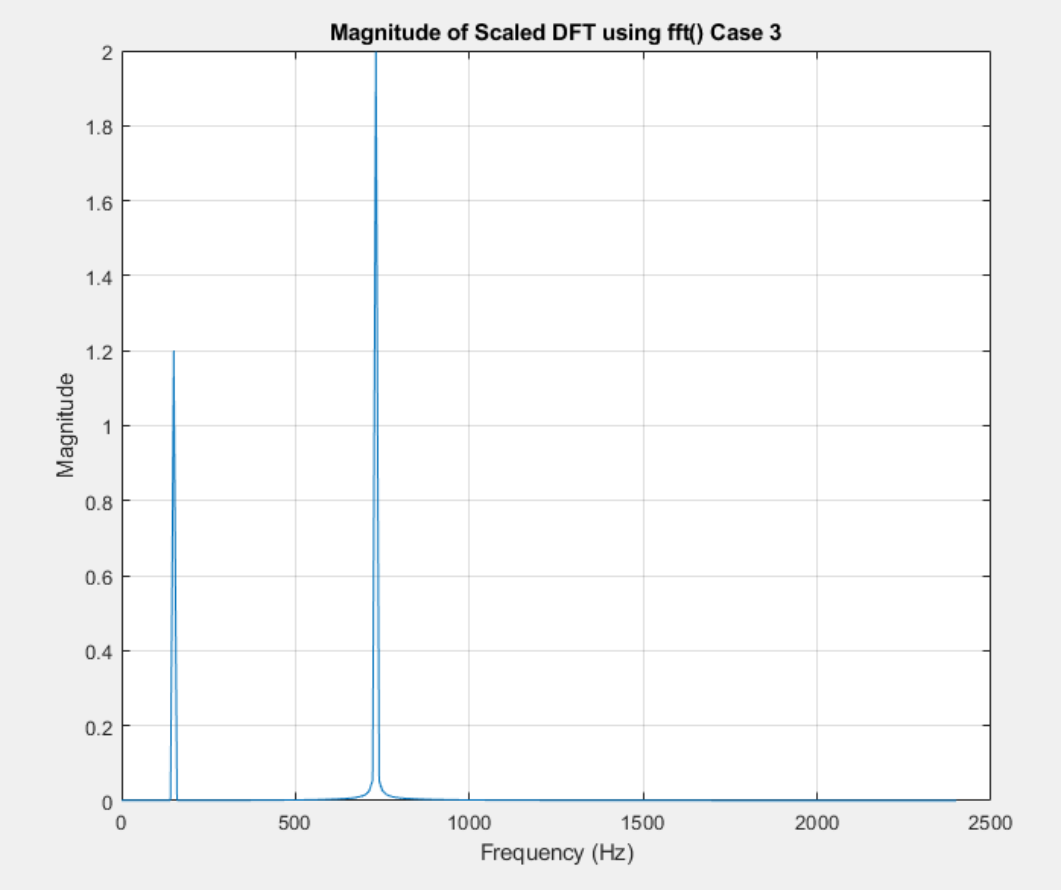

Figure 8: 512 sample x(t), 512-point DFT

Case 3 is the most accurate, closely followed by the plot in 3.5, given that the peaks are almost exactly at 150 and 731, with the correct magnitudes. Case 1 and Case 2 are less accurate, with Case 1 being the least accurate due to its limited frequency resolution.

The sample size affects the frequency resolution and hence the accuracy of the representation of the signal's frequency content. The larger the sample size, the better the frequency resolution. Zero padding makes the graph smoother but doesn't improve frequency resolution. Also, zero-padding affects the magnitude as it distributes the same energy over more points, which can be misleading if trying to measure the actual signal energy at frequencies.

Q4.1)

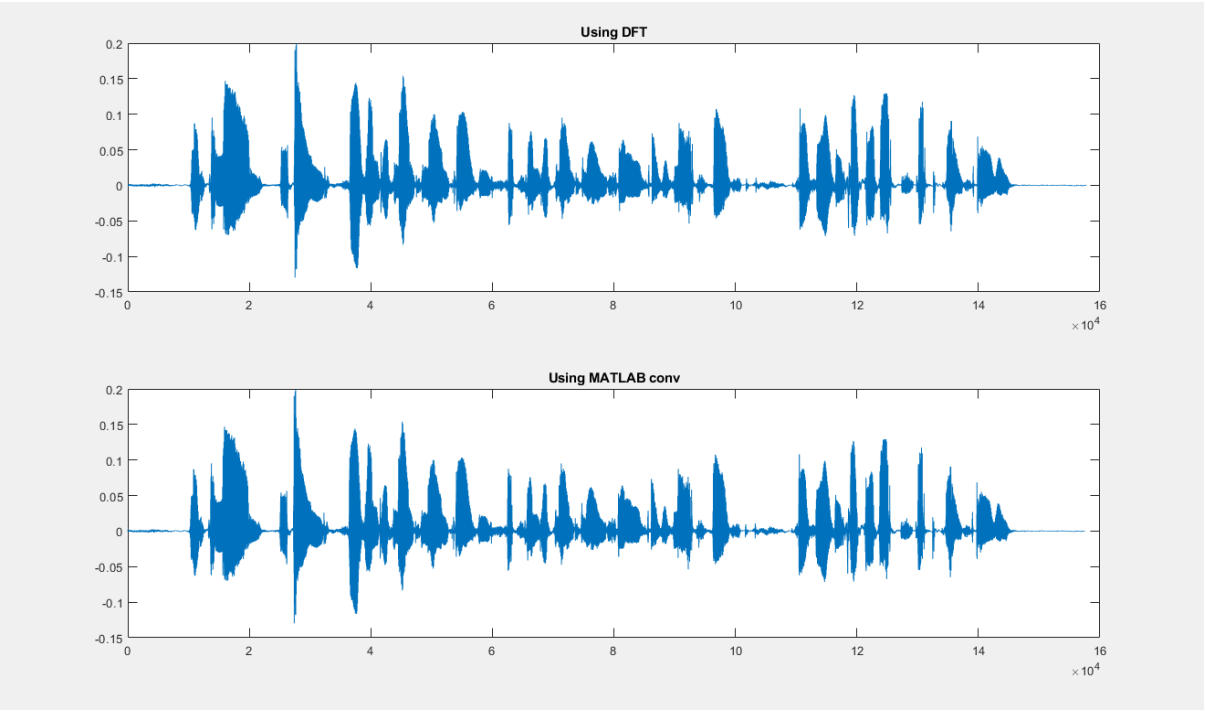

Figure 9: DFT conv vs conv

In the MATLAB code for Q 4.1, I initially loaded the speech signal "DSP_Speech.wav" and the impulse response from "HRIR_1.mat". To carry out linear convolution using DFT, I first zero-padded both signals to a length of length(x1) + length(x2) - 1. I then performed the DFT on these zero-padded signals using MATLAB's fft function. After obtaining the DFTs, pointwise multiplication was carried out to get the DFT of the resulting convolution. Finally, an inverse DFT (ifft) was performed to obtain the convolution in the time domain. To verify the accuracy of this method, I also utilized MATLAB's built-in conv function and compared the outputs. The results were then plotted for visual comparison.

Q4.2)

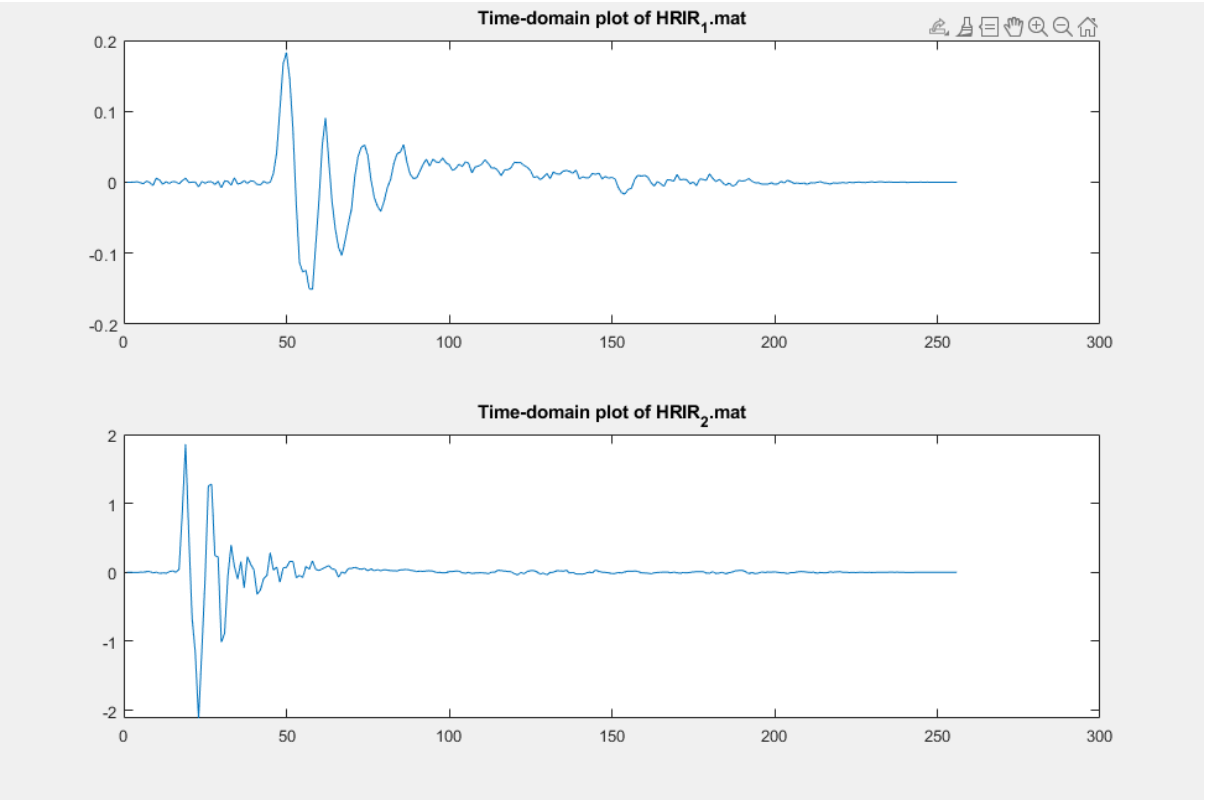

Figure 10: Time-Domain plots of HRIR_mat and HRIR_2.mat

After listening to the 'Binaural_Surname.wav' file, I noticed that it was distinctly different from the original 'DSP_Speech.wav'. The binaural audio gave me a more spatial auditory experience, making it easier to identify the directionality of the sound. This was primarily due to the convolution with (HRIRs) from 'HRIR_1.mat' and 'HRIR_2.mat'.

I generated time-domain plots for both 'HRIR_1.mat' and 'HRIR_2.mat' to further analyse their characteristics. Comparing these plots with my listening experience, it became clear that the impulse response from 'HRIR_2.mat' corresponds to the right ear. This was evident from the spatial cues and the directionality of the sound.

Q5.1)

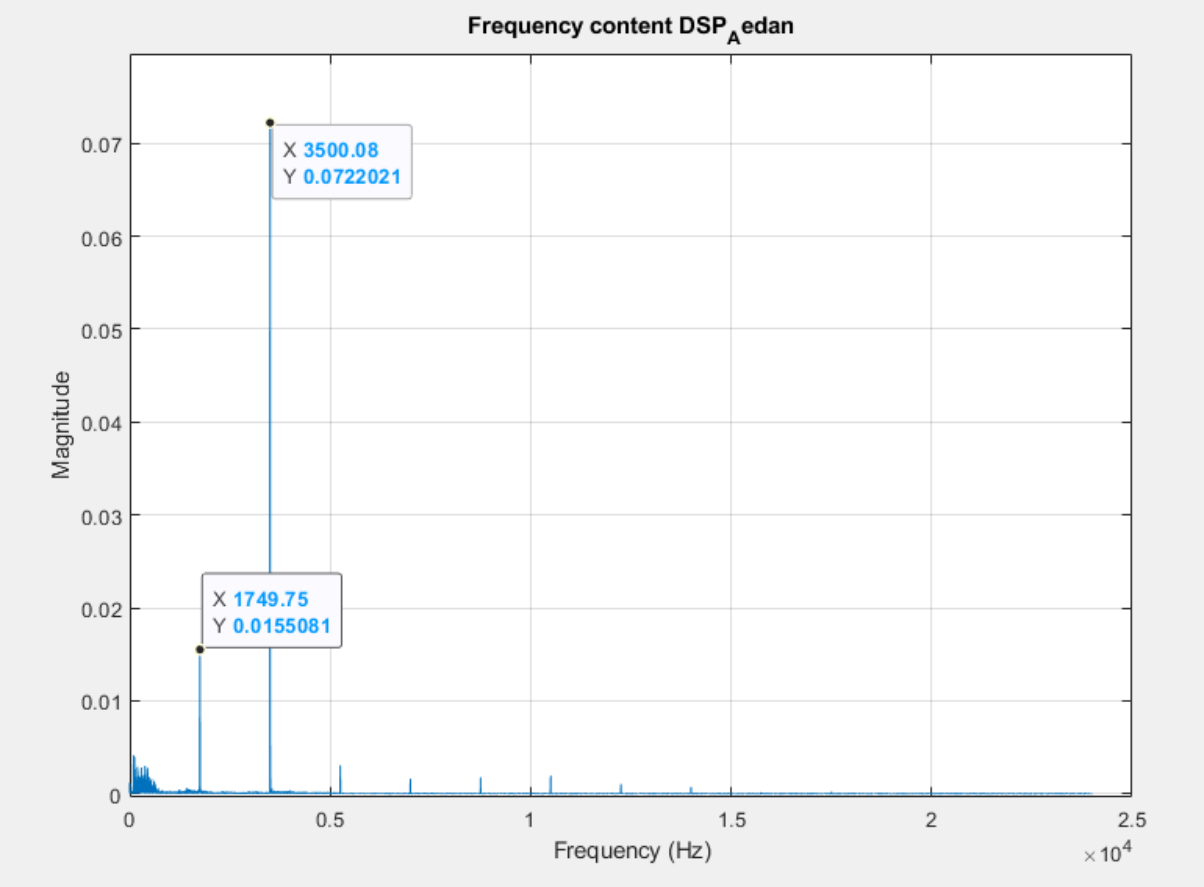

Figure 11: Frequency content of DSP_Aedan

- Loaded the voice recording from 'DSP_Aedan.wav'.
- Applied the dft2 function to the voice signal to visualize its frequency content.
- Saved the resulting frequency plot as 'Frequency_Content_DSP_Aedan.png' for report documentation.

  Repeated for 5.2, 5.3

Q5.2)

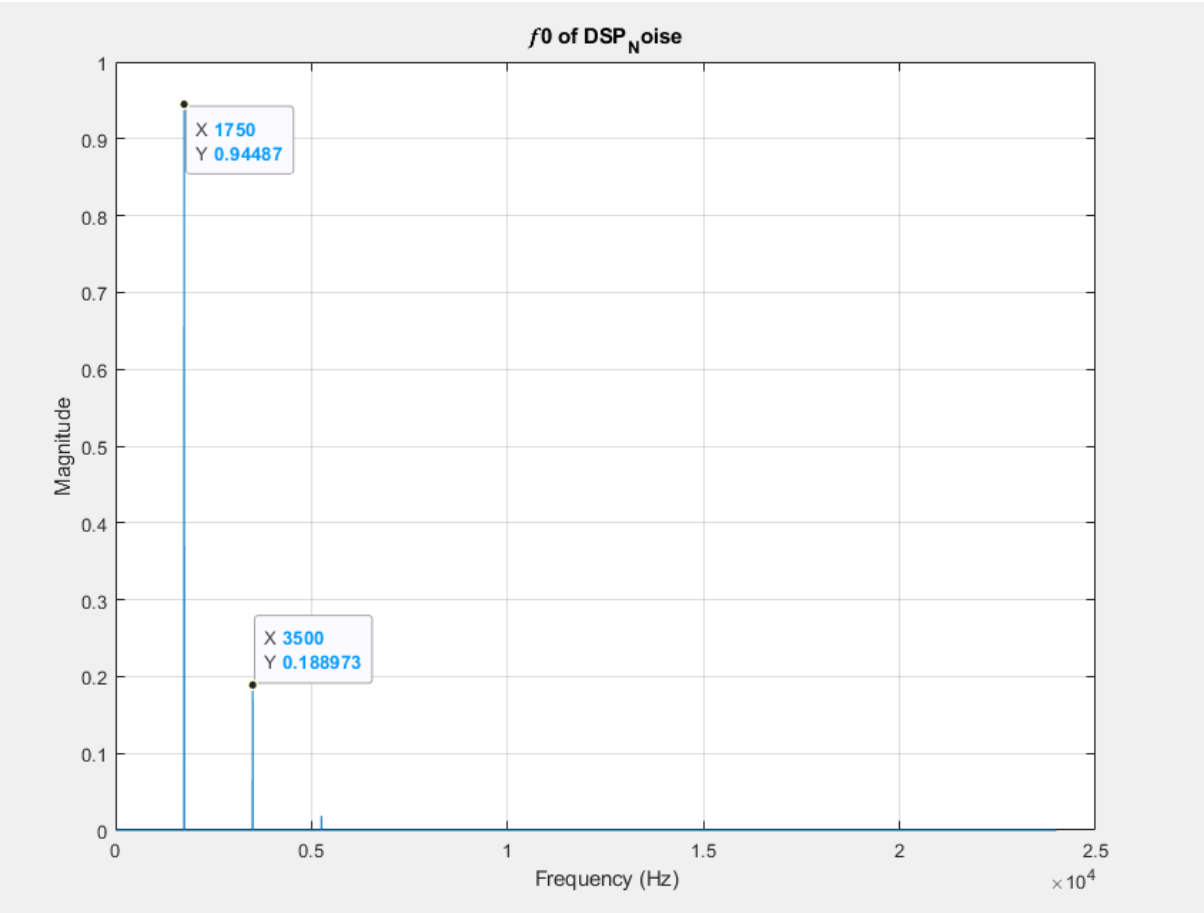

Figure 12: f0 spectrum

F_0 is 1750

Q5.3)

In the graph corresponding to my voice, the frequency components are more concentrated in specific bands. This suggests that my vocal emissions predominantly occupy these frequency ranges.

On the other hand, the frequency spectrum of the noise signal (dsp_noise.png) appears to be more uniformly distributed across various frequencies. This indicates that the noise in the system is more of a 'white noise' type, spreading its energy across a wide range of frequencies.

These observations are instrumental in separating the voice and noise signals during any filtering or processing tasks. The concentrated frequency bands of my voice can be targeted for preservation, while the broader frequency range of the noise can be attenuated to improve the signal-to-noise ratio.

Q5.4)

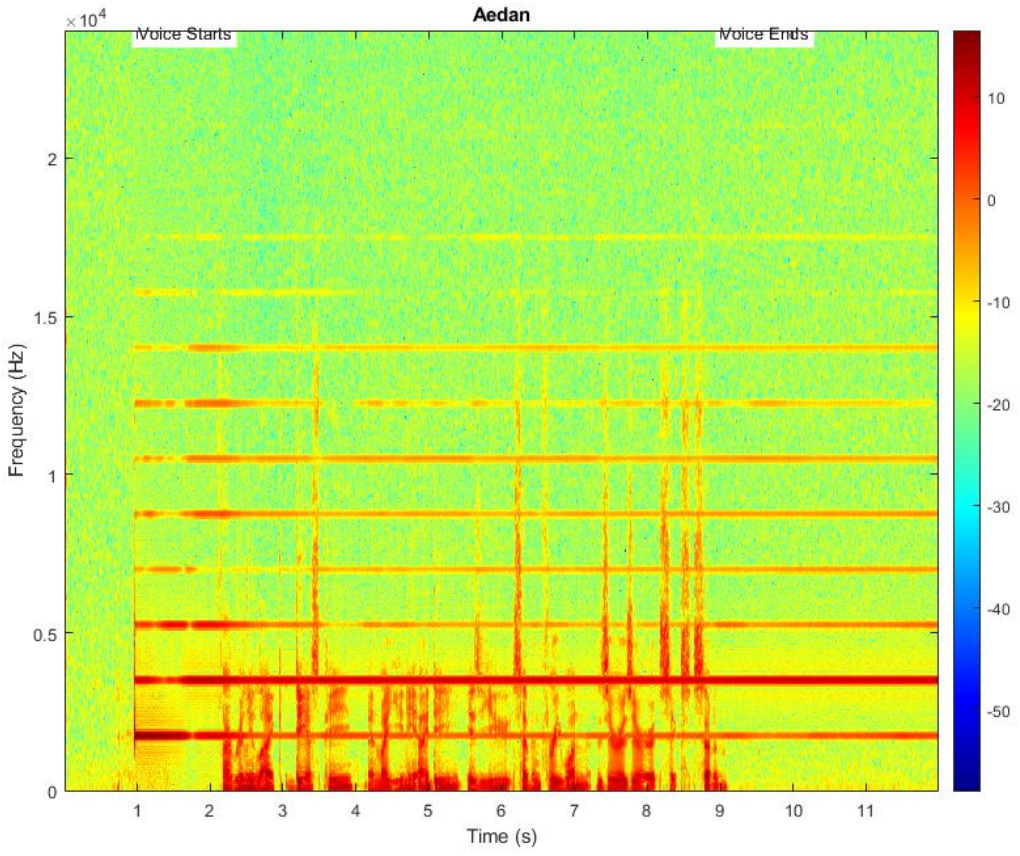

Figure 13: DSP_Aedan spectrogram:

- Loaded the audio recording from 'DSP_Aedan.wav'.
- Defined the parameters for the spectrogram - chose a Hamming window of size 512, set overlap to 256 samples, and specified a DFT size of 1024.
- Generated the spectrogram of the audio using the specified parameters.
- Plotted the spectrogram with a logarithmic scale for better visibility.
- Annotated the spectrogram to indicate where the voice starts and stops (using example times of 1.0s for start and 9.0s for end).
- Saved the plotted spectrogram as 'Spectrogram_DSP_Aedan.png' for documentation purposes.

Q5.5)

Which of the mentioned input parameter(s) directly influence the size of vector?

The size of the vector F is directly influenced by '**nfft**', the number of points I used for the DFT. I set '**nfft'** to 1024, and as a result, the length of F turned out to be $\frac{1024}{2} + 1 = 513$.

Which of the mentioned input parameter(s) directly influence the size of vector?

The size of vector **T** is influenced by a combination of parameters: the total length of my audio signal ('my_audio'), the size of the window ('window'), and the amount of overlap ('noverlap'). In my case, the approximate relationship for the size of **T** is:

$$\frac{Length\ of\ my_{audio} - noverlap}{Length\ of\ window - noverlap}$$

If the overlap length is 50% of the window size, find the relation between the input parameter(s) and the resolution of vector **T.**

When the overlap length is 50% of the window size, '**noverlap'** would be $\frac{Length\ of\ window}{2}$. This means that each step would effectively be half the window size, thus refining the time resolution of T. I could determine the time resolution as:

$$Resolution\ of\ T = \frac{Length\ of\ window}{2 * fs}$$

Q5.6)

In my implementation of the function FindSignalStart, I chose to work in the frequency domain to more easily identify when the speech begins in a recorded audio signal. To achieve this, I employed a Short-Time Fourier Transform (STFT) via MATLAB's spectrogram function. I specified the window size, overlap, and number of DFT points according to my requirements. The idea was to break down the complex audio signal into smaller "chunks" and analyze their frequency content over time.

I targeted a specific frequency range that corresponds to the human voice, specifically focusing on the range of a male voice (85 Hz to 180 Hz in my case). Then, I calculated the energy in this frequency range for each time bin generated by the STFT. This gave me a clear idea of when there's a significant amount of energy in the voice frequency range, which, in turn, indicates the start of speech.

To pinpoint the exact start, I set a threshold at 10% of the maximum energy observed within the speech frequency range. Using MATLAB's find function, I located the first instance where the energy surpasses this threshold. This point is deemed as the start time of the speech, and I sliced the input audio signal from this point onward.

The function returns the truncated audio signal along with the start time of the speech in seconds. I also included a visualization plot to show the energy over time and where the algorithm detected the start of the speech, marked by a red vertical line. This helped me verify the accuracy of my function by listening to the truncated audio and comparing it to the visual plot.

Q5.7)

In the FindSignalStop function, I flipped the audio signal and performed a Short-Time Fourier Transform (STFT) on it. I calculated its energy profile and set a threshold to pinpoint where the speech ends in this flipped version. I then mapped this point back to the original signal's timeframe to truncate the non-speaking tail, effectively finding where the speech ends. This approach mirrors the logic used in FindSignalStart, making it efficient for this task.

Q5.8)

I first loaded my original audio file and then utilized the FindSignalStart and FindSignalStop functions to precisely trim the non-speaking portions at the beginning and end of the clip. I then saved this trimmed audio as 'DSP_Chopped_Aedan.wav'. To validate the changes, I played the newly trimmed audio file. Finally, I plotted its spectrogram to visualize the frequency components of the edited audio.

Q6.1)

- Defined the coefficients for filter H1.
- Defined the coefficients for filter H2.
- Defined the coefficients for filter H3.
- Visualized the magnitude response of filter H1 using fvtool and labeled it as 'H1 Filter Magnitude Response'.
- Visualized the magnitude response of filter H2 using fvtool and labeled it as 'H2 Filter Magnitude Response'.
- Visualized the magnitude response of filter H3 using fvtool and labeled it as 'H3 Filter Magnitude Response'.

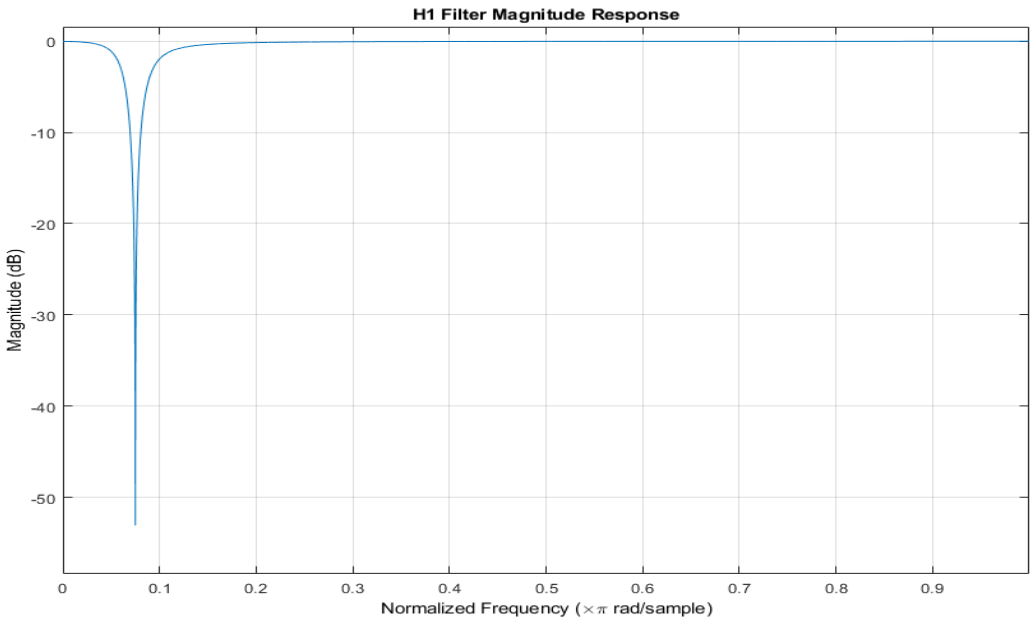

Figure 14: H1 Filter Magnitude Response

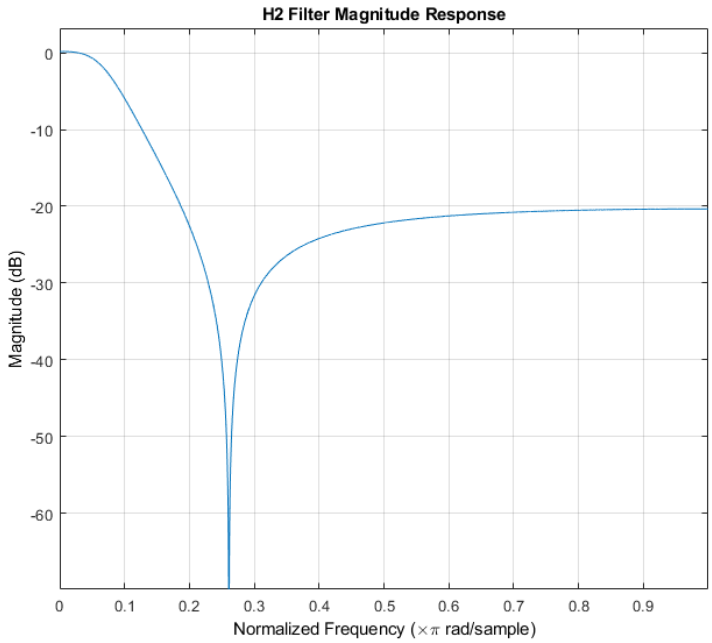

Figure 15: H2 Filter Magnitude Response

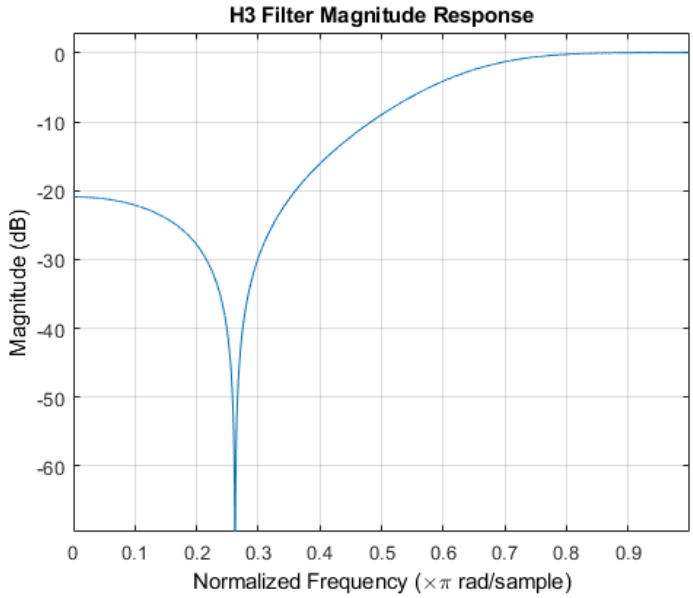

Figure 16: H3 Filter Magnitude Response

Filter H1 would likely be the most precise in filtering out a specific frequency range. If the noise's fundamental frequency falls within this range, H1 would be the most effective at isolating and removing it while preserving the voice signal.

Filter H2 would probably let through more than just the desired voice signal but would also be more forgiving of slight variations in the noise and voice frequency components.

Filter H3, with its irregular response, would be the least predictable in this application.

Based on the plots, I choose H1, especially since the fundamental component of the noise is narrow and well-defined. This would allow for the most precise noise removal while maintaining the integrity of my voice signal.

Q6.2)

- Loaded the original voice recording 'DSP_Aedan.wav'.
- Defined the coefficients for filter H1.
- Applied filter H1 to the original voice recording.
- Saved the filtered voice signal as 'Filtered_Voice.wav'.
- Visualized the frequency spectrum of the filtered signal and labelled it as 'Frequency Spectrum of Filtered Signal'.

Upon applying the selected efficient filter to the voice recording, it was observed that there was minimal noticeable change to the audio quality. The frequency response of the filtered signal was plotted to understand the changes introduced by the filter. Visually, the spectrum displayed only minor alterations, validating the auditory assessment.

Listening to the filtered signal, it's clear that this filter has minimal impact on the voice recording, effectively maintaining its original quality.

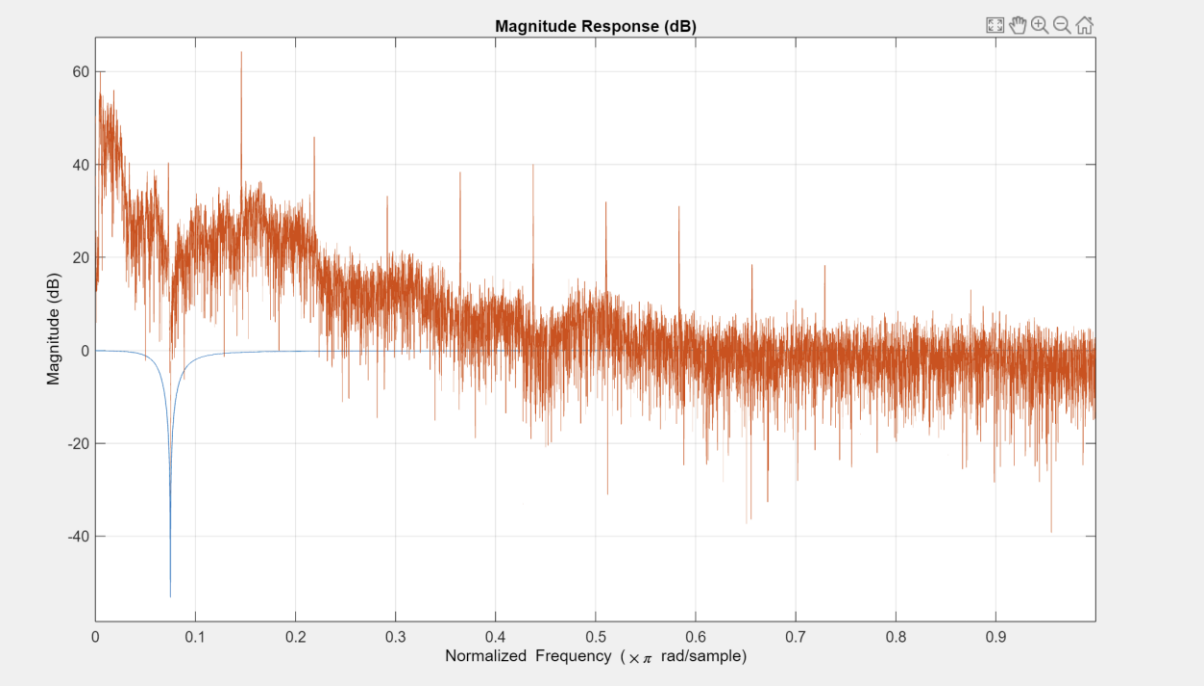

Figure 17: H1 filtering the audio

Q6.3)

- Loaded the original voice recording 'DSP_Aedan.wav' again.
- Defined the coefficients for filter H4.
- Applied filter H4 to the original voice recording.

- Saved the filtered voice signal as 'Filtered_Voice_H4.wav'.
- Visualized the frequency spectrum of the signal after applying filter H4.
- Displayed the Pole-Zero plot for the H4 filter and labeled it as 'Pole-Zero Plot of H4'.

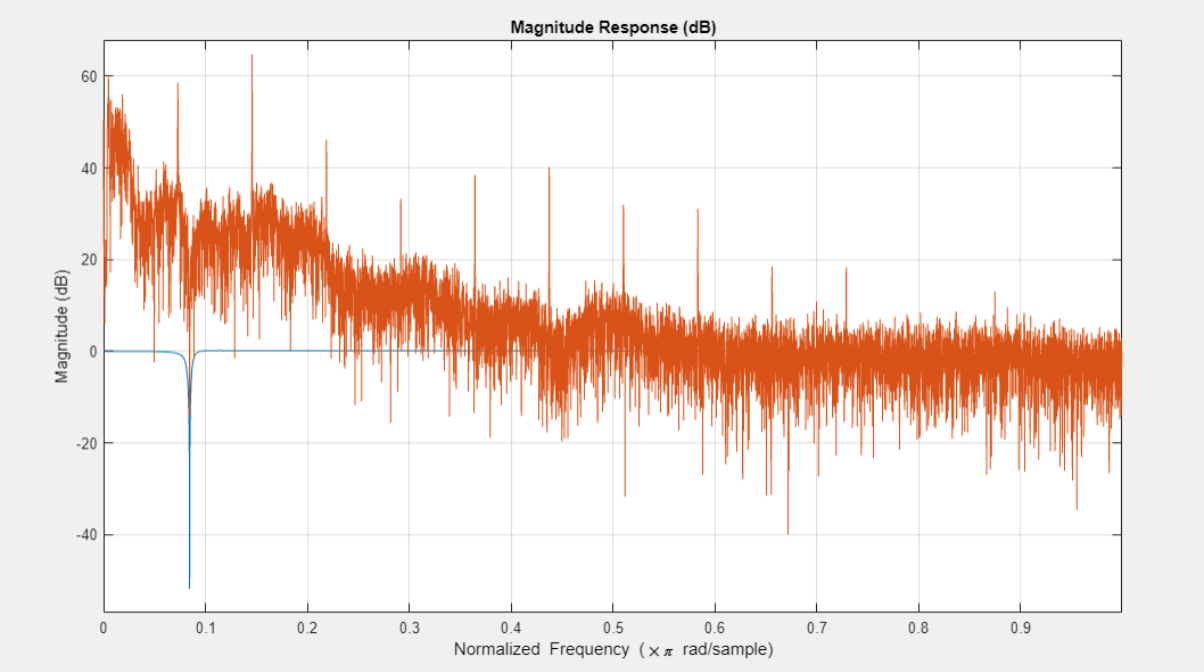

Figure 18: Plot of H4 filtering DSP_Aedan

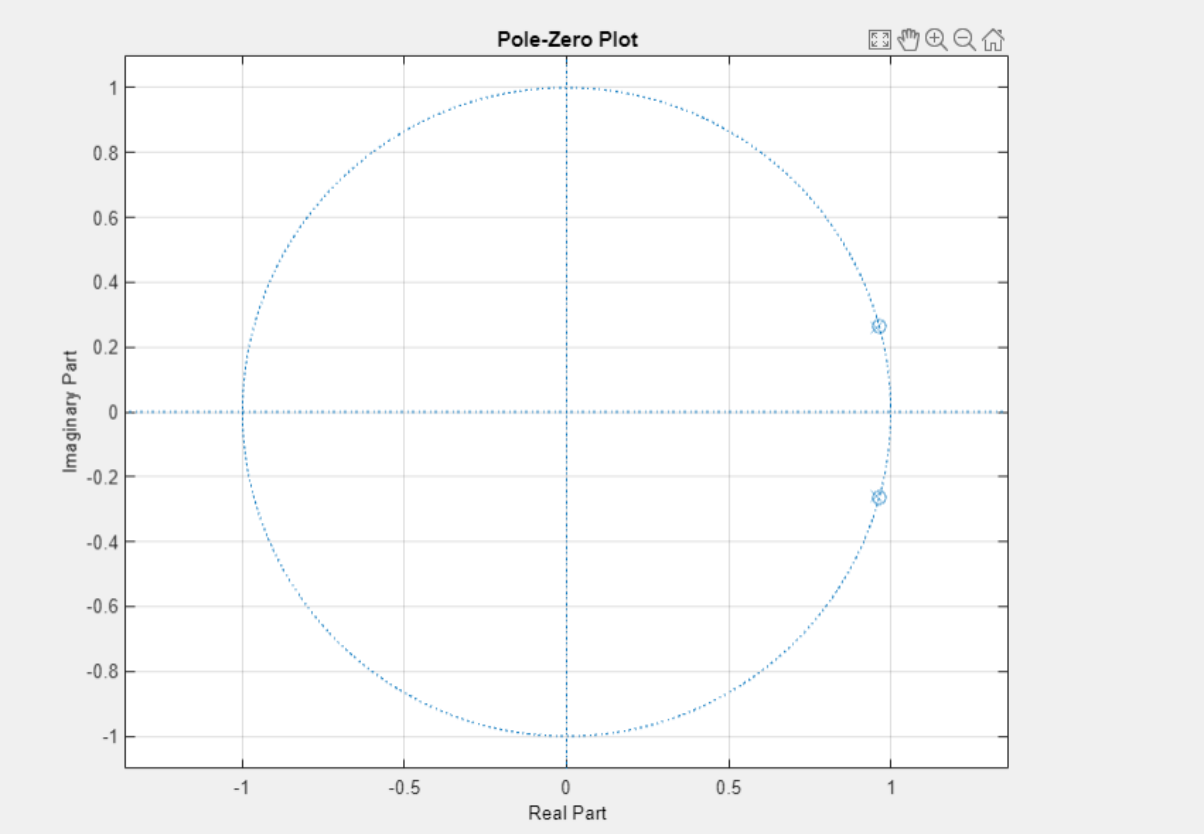

Figure 19: Pole-zero plot of H4

Upon listening to the filtered audio, the filter's effect was discernible but subtle. It mildly muffled the audio, which led to a slight muting of the voice's clarity.

The pole-zero plot for $H_4(\Omega)$ reveals important characteristics about the filter. The poles, marked by 'x', are located close to the unit circle. This signifies that the system's response will be more pronounced or resonant near those frequencies. The zeros, on the other hand, determine the frequencies which the filter will attenuate. From the plot, it's clear that there are no zeros on the unit circle, explaining the slight muffled effect rather than any significant frequency nullification.

The proximity of the poles to the unit circle suggests that the filter has a sharp response at those frequencies, which might explain the slight muffling of the audio. The frequencies around the poles are being amplified, while others remain relatively unchanged, leading to the described audio output.

Q7.1.1)

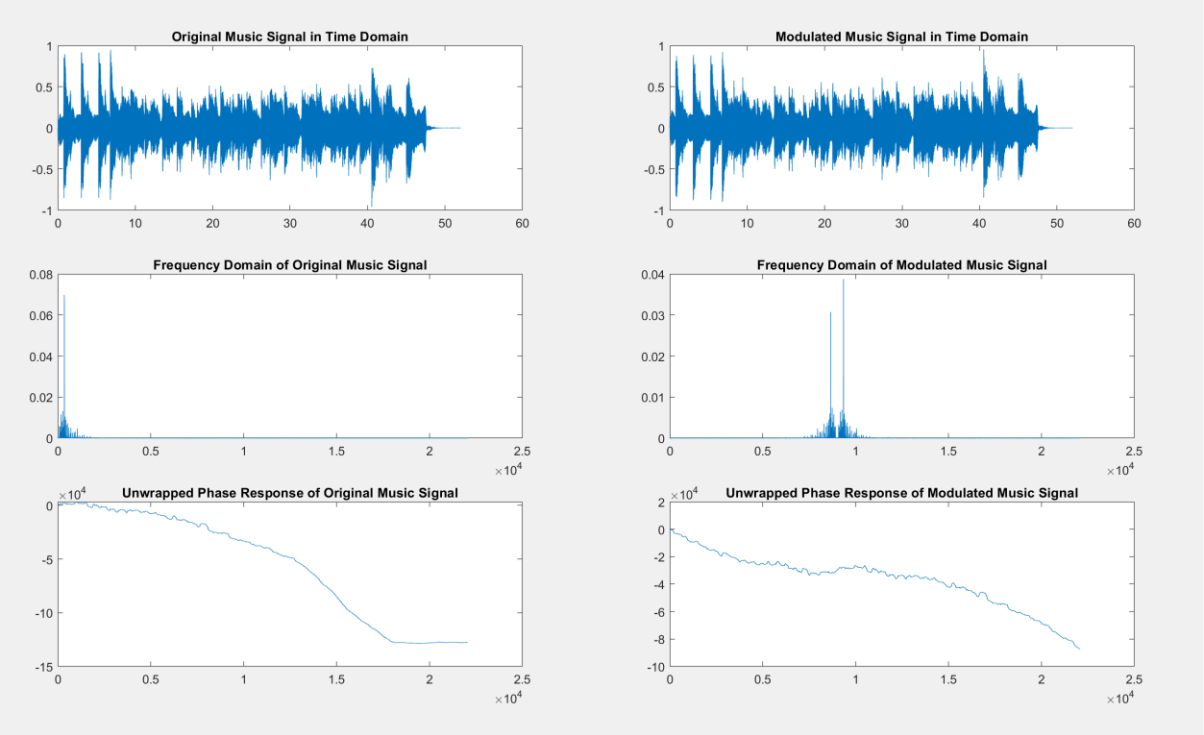

Figure 20: Original and modulated signal plots

Q7.1.2)

After amplitude modulating my original music signal with a 9 kHz sine wave, I noticed the frequency components shifted by ±9 kHz. This introduced both upper and lower sidebands. The phase of the original music clip's components, however, remained unchanged.

Q7.1.2) a)

The Fourier Transform of the sine wave I used for modulation resulted in two impulses in the frequency spectrum: one at +9 kHz and another at -9 kHz.

Q7.1.2) b)

When I multiplied the music signal with the sine wave in the time domain, their spectra convolved in the frequency domain. This explained the appearance of sidebands at f+9 kHz and f−9 kHz in my modulated signal.

Q7.1.3)

Upon comparing the original and modulated music signals in the time domain, the modulated signal had a clear high-frequency oscillation due to the 9 kHz sine wave. When I listened to the modulated music, it sounded screechy and high-pitched.

Q7.2)

Multiplying by the same sine wave effectively restores the original music because amplitude modulation (AM) shifts the frequency content of the original signal up and down the frequency spectrum by the frequency of the sine wave. Multiplying again by the same sine wave brings it back to the original frequency range. It will likely have some high-frequency components that will need to be filtered out, which brings us to the low-pass filter.

The magnitude of the restored music would depend on how well the demodulation and subsequent filtering have been executed. If done perfectly, it should be very close to the original signal.

Q7.3)

To design the desired filter, the guidelines provided in the handout were followed. Initially, the desired cutoff frequency for our low-pass filter was set at $0.3 * \frac{fs}{2} Hz$, where fs represents the sampling frequency. This frequency also marked the desired edge of our passband. To determine the transition width, I utilised the formula $3.44 * \frac{fs}{199}$. With this transition, I calculated the pass-band edge frequency, f1, by adding half of the transition width to our desired pass band edge frequency.

The next step involved calculating the impulse response of the ideal low-pass filter. Using our determined f1, I found the corresponding digital frequency, Ω1, using the relation $2\pi * \frac{\mathrm{f1}}{fs}$. The impulse response, h1[n], of my low-pass filter was then derived using the sinc function, specifically through the formula $\frac{\Omega 1}{\pi} * sinc(\frac{\Omega}{\pi})$.

For windowing, I opted for the hamming window due to its balanced trade-off between main lobe width and side lobe attenuation. This window was generated with 199 points, adhering to our design constraints. The filter's impulse response, h2[n], was then achieved by multiplying our ideal impulse response, h1[n], with the Hamming window. This multiplication is crucial as it tapers the ideal impulse response, which effectively reduces the Hibbs phenomenon or ringing in the frequency response.

Lastly, to ensure our filter was causal and thus physically realisable for real-time applications, I shifted the impulse response, h2[n], to the right by $\frac{N-1}{2}$ samples. This resulted in my final causal FIR low-pass filter's impulse response, defined as $h[n] = h2[n - \frac{N-1}{2}]$. Once our filter was designed, it was applied to the demodulated signal, effectively filtering out undesired frequencies and restoring the original music signal.

Q7.4.1)

Use a low-pass filter. The reason is to preserve the original music while eliminating the high-frequency noise introduced by the modulation and demodulation process.

Q7.4.2)

FIR filters are generally preferred over IIR for this kind of application because they are inherently stable and have a linear phase response, meaning they won't distort the phase of the signal. IIR filters can be more efficient but are more prone to instability and may have a nonlinear phase response.

Q7.4.3)

Causal filters are used because they only depend on the current and past inputs, not future inputs. Non-causal filters are not physically realisable because they require future information.

Q7.4.4)

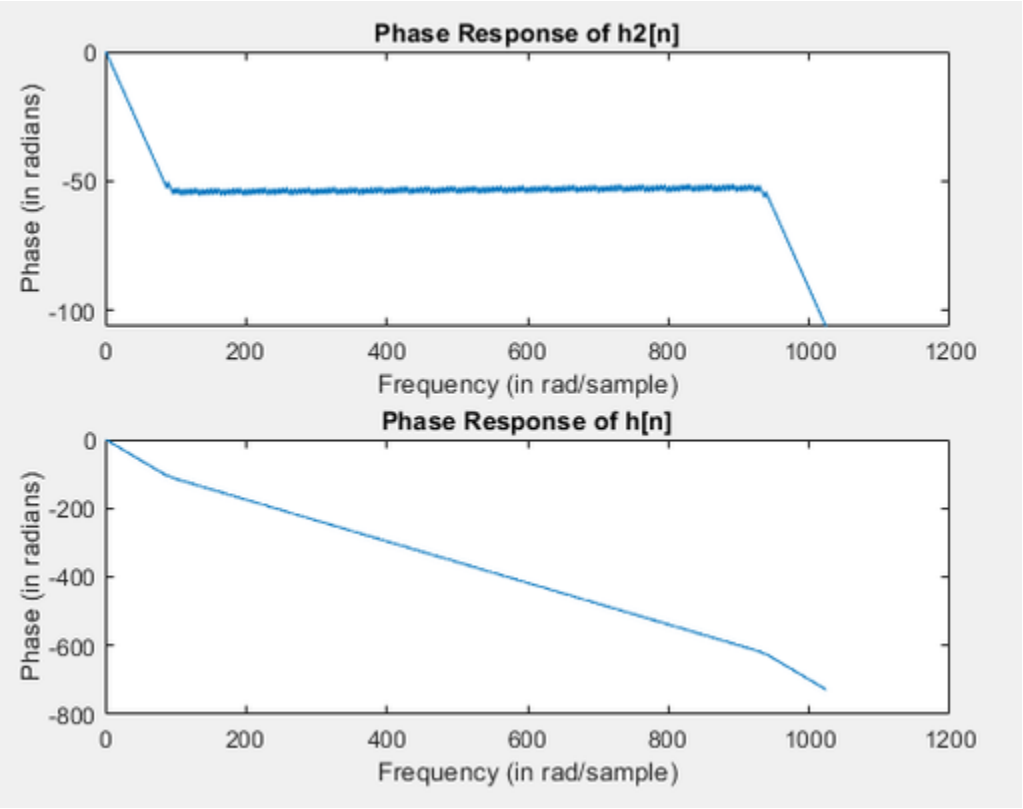

Figure 21: Phase response plots

Q7.4.5)

Frequency Domain: In the frequency domain, the act of shifting an impulse response doesn't effect the magnitude of its frequency response but does affect its phase. For FIR filters, the phase response is typically linear when the impulse response is symmetric. The phase response of the shifted h[n] wraps around multiple times, which is a consequence of making the filter causal. This wrapping or "phase distortion" is characteristic of causal systems, as they can't have output before they have input.

Time Domain: In the time domain, the shift made the filter causal. Initially, h2[n] was a non-causal impulse response, meaning it started "responding" before n=0. Shifting it to the right by 2(N−1)/2 samples moved its centre n=0, making it causal. This is essential for real-world implementations because non-causal systems, theoretically, require knowledge of future inputs to compute the current output.

Justification for Phase Difference: The difference between the two phase-response plots is due to the shift of (N−1)/2 samples. A shift in the time domain corresponds to a linear phase change in the frequency domain. Given that the filter was made causal, the linear phase response gets modified, and this modification is seen as the phase wrapping in the plot for h[n].

Q7.5)

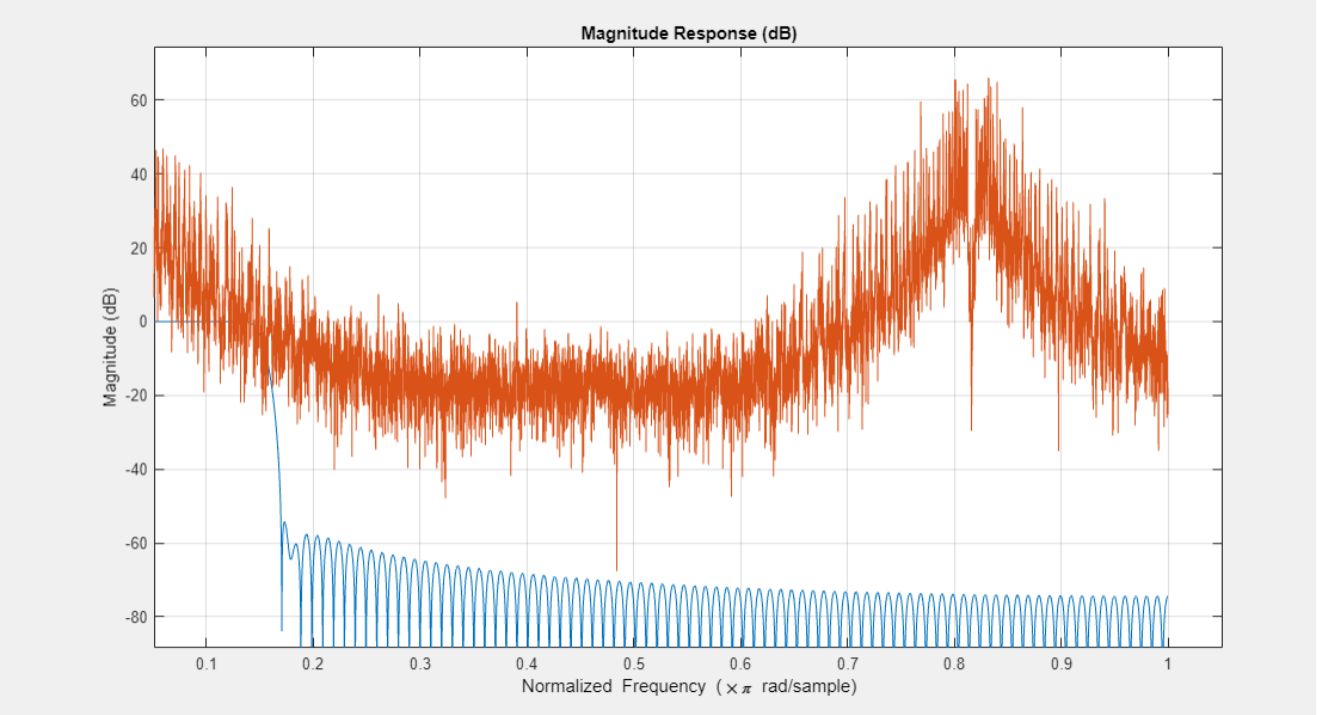

Figure 22: Frequency response of filter and demodulated signal plot

Q7.6)

Passband edge frequency

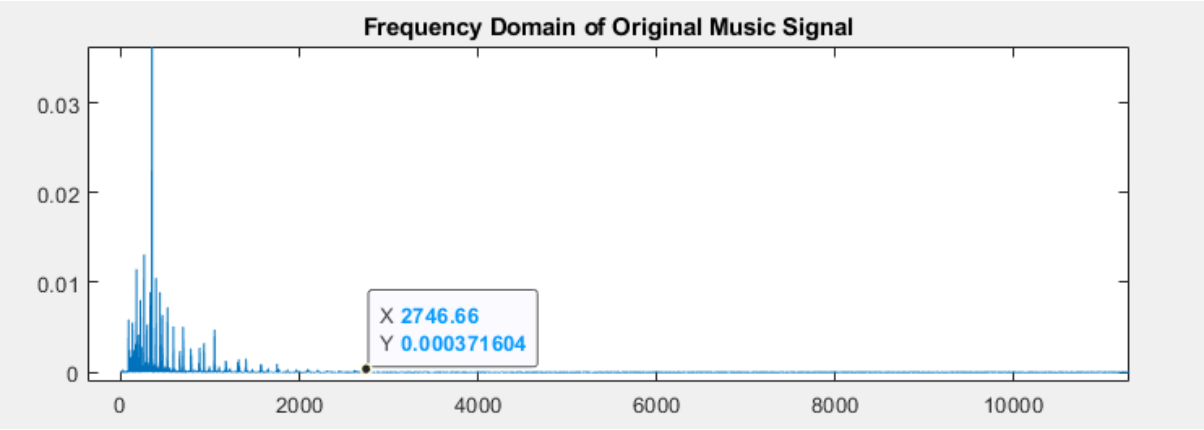

Figure 23: Frequency range limit

3000hz as the frequencies of the original music signal is covered by that range.

Transition width:

This is given by the formula in the handout: $3.44 * \frac{fs}{TW}$

Where TW = 199

$$= 3.44 * \frac{44100}{199}$$

$$= 762.331$$

Window Type: I used the hamming window for my design.

Number of Coefficients: This is 199, as per our design constraints.

Stop Band Attenuation: The Hamming window provides a stopband attenuation of about 55dB as per the handout.

Q7.7)

Stopband Attenuation:

0.1707 * 22050 = 3763.94Hz

-3dB: 0.1491 * 22050 = 3285.45Hz

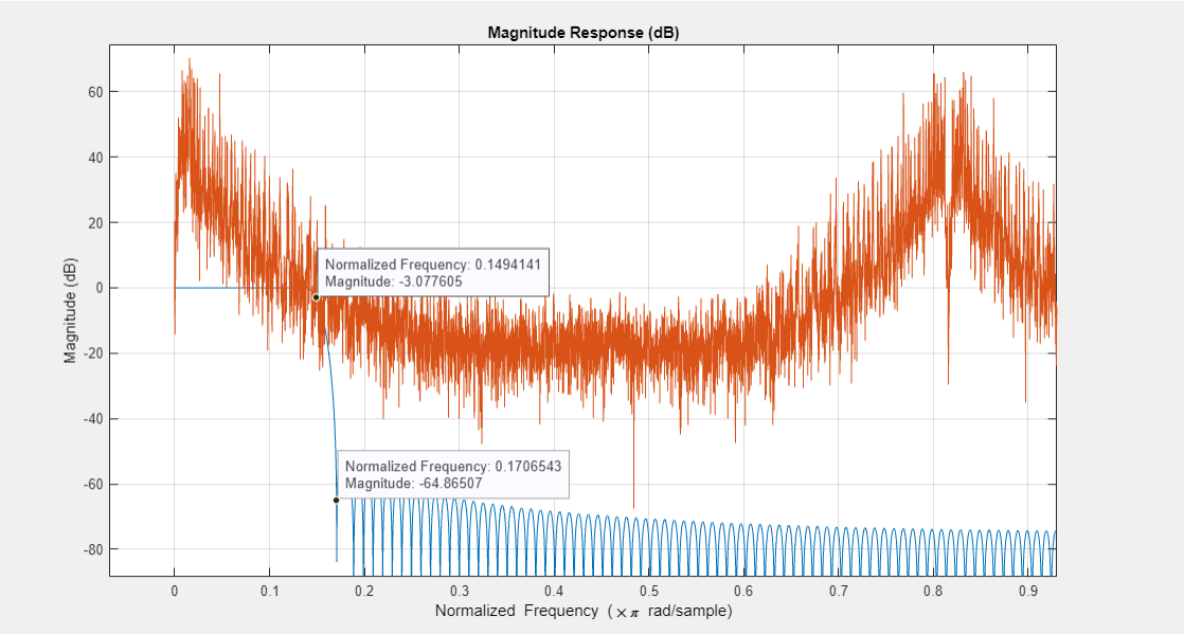

Figure 24: Stopband Attenuation and -3dB

Q7.8)

```
seveneight = fvtool(h, 1, filtered_signal, 'MagnitudeDisplay', 'Magnitude (dB)');
```

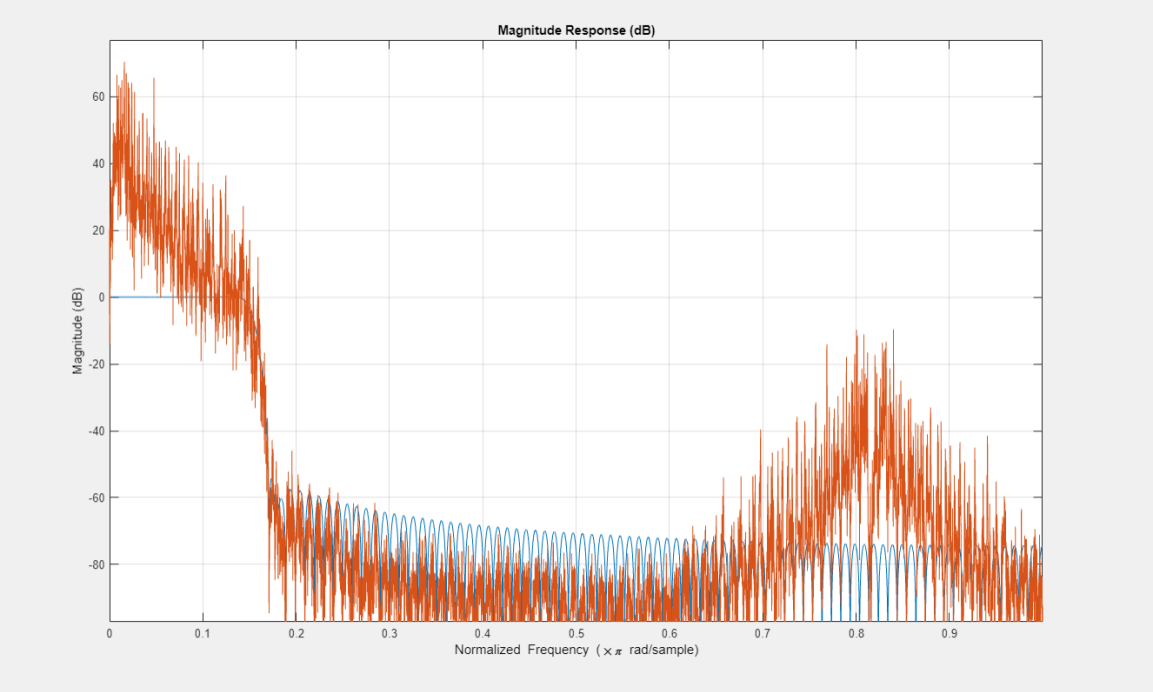

Figure 25: Filtered signal and frequency response

Q7.9)

Audibly, the filter works well. The annoying high-pitched sound (often associated with higher frequencies) has been significantly reduced, implying that the filter effectively attenuated those unwanted frequencies.

In the provided magnitude response plot, the filter demonstrates a sharp transition from the passband to the stopband, ensuring that the unwanted higher frequencies are attenuated effectively. This observation aligns with the audible assessment where the filter successfully diminishes the screechy high-pitched sounds from the demodulated signal. Hence, from a subjective standpoint, the listening experience is enhanced with the application of this filter. The combined evidence from the plot and auditory feedback confirms that the filter is adept at its intended task, providing a clear and improved audio output.