MATLAB Project

Question 1

Question 1.1

The `audioinfo()` function was used to extract the desired information from the provided audio file, *"DSP_Speech.wav"* (**Table 1.1.1**).

Table 1.1.1: *DSP_Speech.wav information extracted using the audioinfo() function.*

	DSP_Speech.wav
Bit-depth (bits)	16
Sampling rate (Hz)	16000
Channels	1
Duration (seconds)	9.8438
Total number of samples	157500

Question 1.2

The `audioread()` command was used to read and chop the provided audio file, *"DSP_Speech.wav"*, from the 4th-9th second:

```
DSP_Speech = audioread("Resource Files/DSP_Speech.wav", [64000 144000]);
```

Question 1.3

The `audioplayer()` and `play()` commands were used to play the chopped audio clip from Question 1.2:

```
clip = audioplayer(DSP_Speech, 16000, 16);
play(clip)
```

Question 1.4

The stop(clip) command can be used to prematurely stop playback of the clip. For example:

```
play(clip)
pause(1);
stop(clip)
```

Will result in the clip only playing for 1-second.

Question 2

Question 2.1

A voice recording was created in MATLAB using the `record()` function. Firstly, an `audiorecorder()` object was created with the specified sampling-rate (48000), bit-depth

(16) and number of channels (1). Secondly, the `record()` function was called and a 12-second pause was initiated. Thirdly, the `stop()` command was used to stop the recording after the 12-second pause. Finally, the `audiowrite()` function was used to save the recording as a .wav file. The full code for this task can be found in Appendix A.

Question 3

$$x(t) = 1.2\cos(2\pi \times 150t) + 2\sin(2\pi \times 728t)$$

Question 3.1

A time-space was created containing 7200 points each spaced T_s $\left(\frac{1}{4800}\right)$ seconds apart. The desired signal, x(t), was then defined as follows:

```
x_t = 1.2*cos(2*pi*150*t)+2*sin(2*pi*728*t);
```

A plot of the defined signal can be seen in **Figure 3.1.1**.

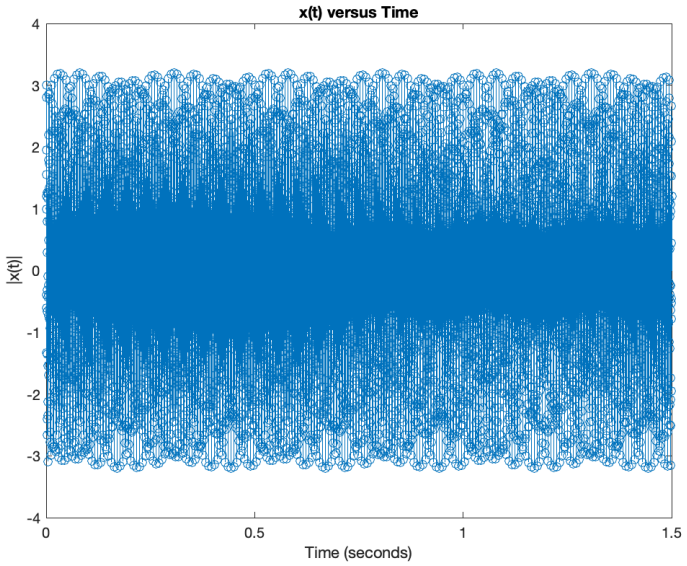


Figure 3.1.1: *Stem plot of the discrete signal x(t).*

Although the signal is discrete, and thus should be plotted using the `stem()` command, the `plot()` command provides a clearer view of the signal (**Figure 3.1.2**).

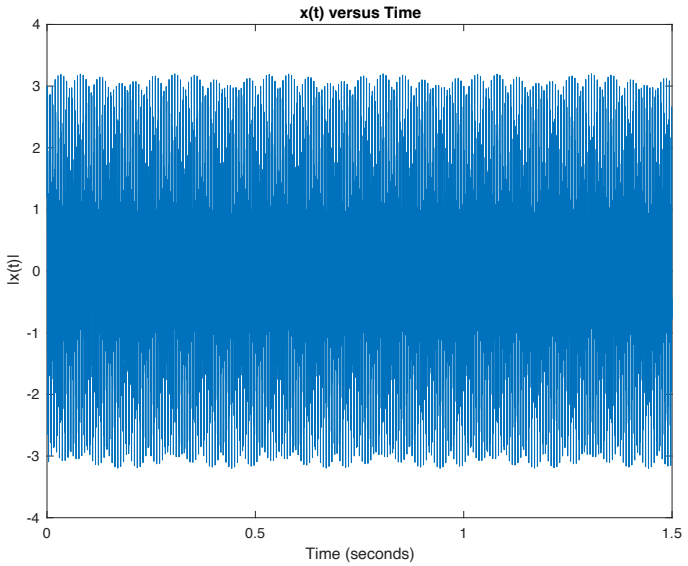


Figure 3.1.1: *Continuous plot of the discrete signal x(t).*

Question 3.2

The discrete Fourier transform (DFT) of x(t) was computed using the following equation:

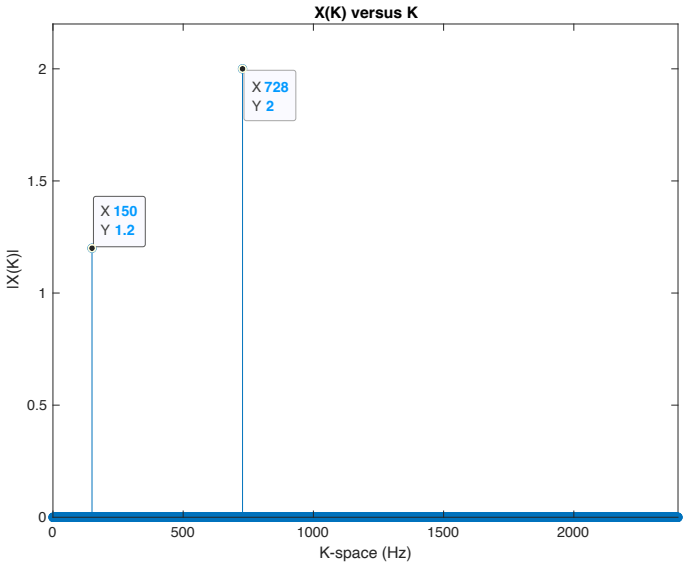
$$X(K) = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\Pi}{N}kn}$$

This was implemented in MATLAB code using nested for loops:

```
for k=k_space
    for n=n_space
            x_k(k+1) = x_k(k+1)+(x_t(n+1)*exp(-1i*2*pi/size(x_t,2)*n*k));
    end
end
```

This was abstracted into a function, called `dft1()`, that takes a discrete input signal and its sampling-frequency, computes the associated DFT, and plots the result (**Figure 3.2.1**). The full code for this task can be found in **Appendix B**.

Figure 3.2.1: *The computed discrete Fourier transform of x(t). Two clearly defined peaks are*



observed at 150Hz and 728Hz with magnitudes corresponding to their time-domain amplitudes.

Question 3.3

The fast Fourier transform (FFT) of x(t) was computed using the MATLAB function `fft()`. Similarly, the function was abstracted into a function called `dft2()`, that takes a discrete input signal and its sampling-frequency, computes the associated FFT, and plots the result (**Figure 3.3.1**). The full code for this task can be found in **Appendix C**.

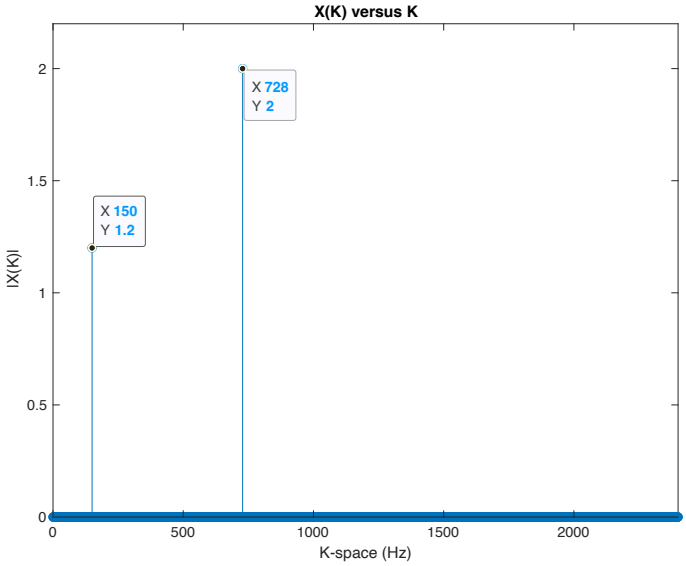


Figure 3.3.1: *The computed discrete Fourier transform of x(t). The result is identical to that seen in Figure 3.2.1.*

Question 3.4

The MATLAB functions `tic()` and `toc()` were used to measure the runtime of `dft1()` and `dft2()`. For each function, the average and minimum runtime across five repetitions was recorded.

Table 3.4.1: *The average and minimum runtime results of dft1() and dft2() when analysing the signal defined in Question 3.1. A total of five trials were performed.*

	Average Runtime (seconds)	**Minimum Runtime (seconds)**
`dft1()`	4.534952	4.451561
`dft2()`	0.080372	0.054390

The recorded average runtime of `dft2()` was over 56-times faster than the recorded average runtime of `dft1()`. This result highlights the difference in computational intensity of the radix-two FFT and DFT for large N, $O\left(Nlog_2(N)\right)$ versus $O(N^2)$, respectively. Such improvements are accomplished through increased algorithmic efficiency (e.g. decimation in frequency and time), efficiency in memory and CPU usage (e.g. the FFT can be evaluated in a parallelisable manner on multicore processors) and optimised hardware implementations.

For example decimation in time works to decompose the DFT into successively smaller computations (e.g. N/2, N/4, N/8, etc.) until the original sequence is represented by N/2 DFTs of size two (**Figure 3.4.1**). Note that this technique requires that the signal length be a power of two. If the signal length is not a power of two, it can be zero-padded to the next closest power of two to realise these gains in efficiency.

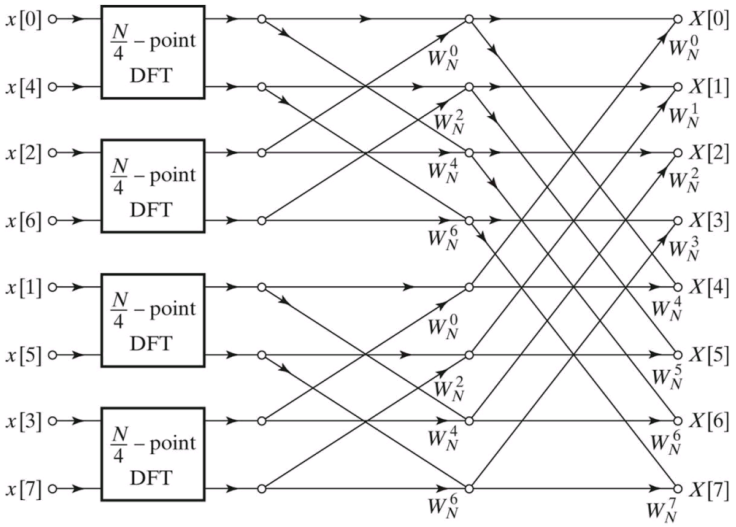


Figure 3.4.1: *Decomposition of an 8-point DFT into four (N/4)-point DFTs.*

It is important to note that this result is not a direct comparison of the two methods of computing the Fourier transform, as both functions contain additional code to plot the resultant frequency magnitudes. However, seeing as this additional code is identical in both functions, this comparison still provides valuable insight into the difference between the two methods.

Question 3.5

The `dft2()` function was used to analyse the frequency content of 7200 samples of x(t) (i.e. the signal defined in Question 3.1). The associated output of `dft2()` can be seen in **Figure 3.3.1**.

Question 3.6

For this question, three variants of the signal defined in Question 3.1 were analysed using the `dft2()` function. Firstly, the first 32 samples of the signal were passed into `dft2()` and the result was observed (**Figure 3.6.1**).

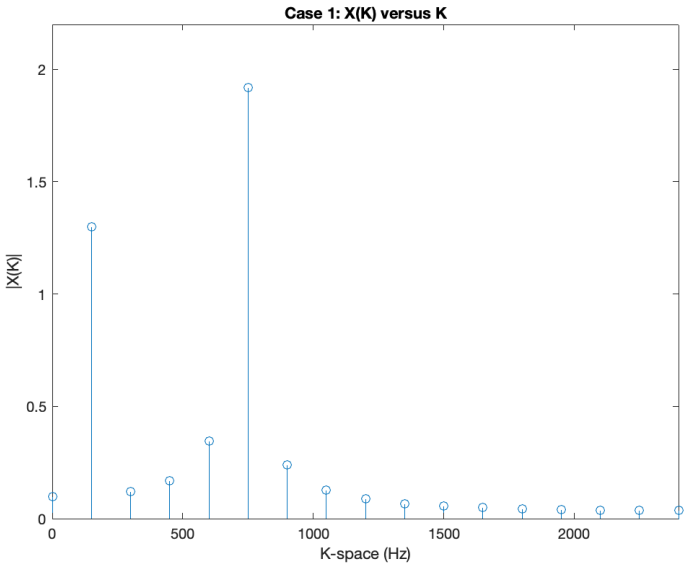


Figure 3.6.1: *The computed discrete Fourier transform of the first 32 samples of the signal. The spacing between samples in the frequency domain is* $\frac{4800}{32} = 150\ Hz$*, meaning there is no single sample indicating the frequency content at 728Hz. As a result, a single, strong peak is observed at 750 Hz, which leaks into surrounding frequencies presenting the illusion of additional frequency content.*

Secondly, the first 32 samples of the signal were again passed, but with additional zero padding to bring the total input signal size to 512 (**Figure 3.6.2**).

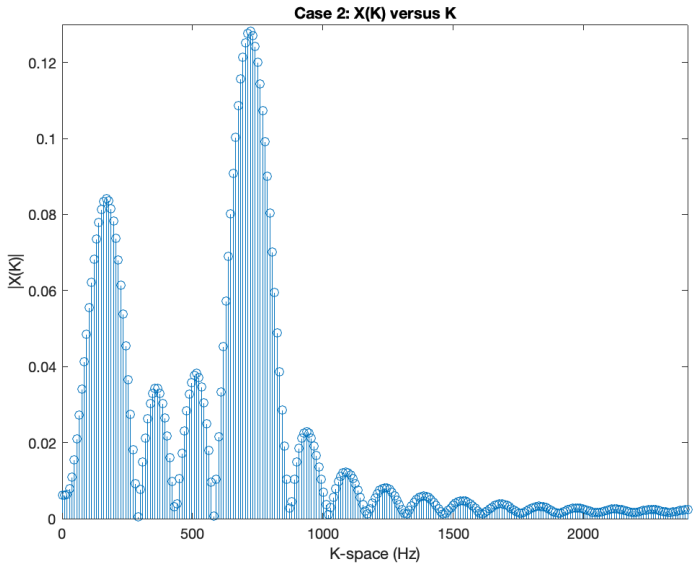


Figure 3.6.2: *The computed discrete Fourier transform of the first 32 samples of the signal zero-padded to a total size of 512. The same bleeding of the peaks is observed, but the higher resolution of the frequency-space effectively interpolates between the peaks seen in Figure 3.6.1. Additionally, Parseval's Theorem—coupled with the increased frequency resolution and the bleeding of the peaks—results in a much smaller global peak magnitude, such that the total power of the frequency-domain signal remains equivalent to the total power of the time-domain signal.*

Finally, the first 512 samples of the signal were passed into `dft2()` and the result was observed (**Figure 3.6.3**).

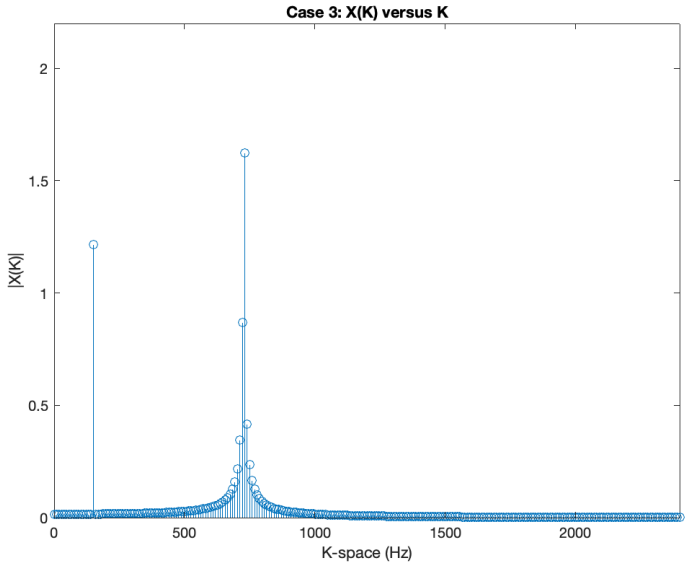


Figure 3.6.3: *The computed discrete Fourier transform of the first 512 samples of the signal. Some bleeding of the 728 Hz peak remains, however, significantly less than what is seen in Figure 3.6.1.*

These three cases provide valuable insight into the effect of sample size and zero padding on the resultant DFT. As shown comparing **Figures 3.6.1-2**, zero padding the input signal increases the frequency resolution through interpolation. Comparatively, as seen in **Figure 3.6.3**, increasing sample size increases the spacing between frequency domain samples. Hence, both of these are related to frequency resolution, but zero-padding the signal artificially increases frequency resolution while increasing sample size increases the 'true' frequency resolution.

Comparing these three plots to the DFT seen in **Figure 3.3.1**, they do not provide an equally clear representation of the frequency content of x(t). Again, this is resultant of the smaller sample size increasing the spacing between samples in the frequency domain, causing leaking of the observed frequency magnitude peaks. Despite this, however, it is still possible to get a faithful reconstruction of the original signal, it will just involve the addition of many sinusoids—versus just two sinusoids when reconstructing from the magnitude spectrum shown in **Figure 3.3.1**.

Question 4

Question 4.1

The linear convolution of the provided audio clip, *"DSP_Speech.wav"* with the impulse response *"HRIR_1.mat"* was computed using two methods. Firstly, the convolution was performed by computing the DFT of both signals (using `fft()`), multiplying their respective frequency spectrums, and taking the inverse Fourier transform of the result (using `ifft()`). Note that an N-point DFT was performed where N was determined as the sum of the size of both signals minus one. This is necessary to force the frequency domain multiplication to replicate linear convolution in the time domain, versus circular convolution. The result of this calculation can be seen in **Figure 4.1.1**.

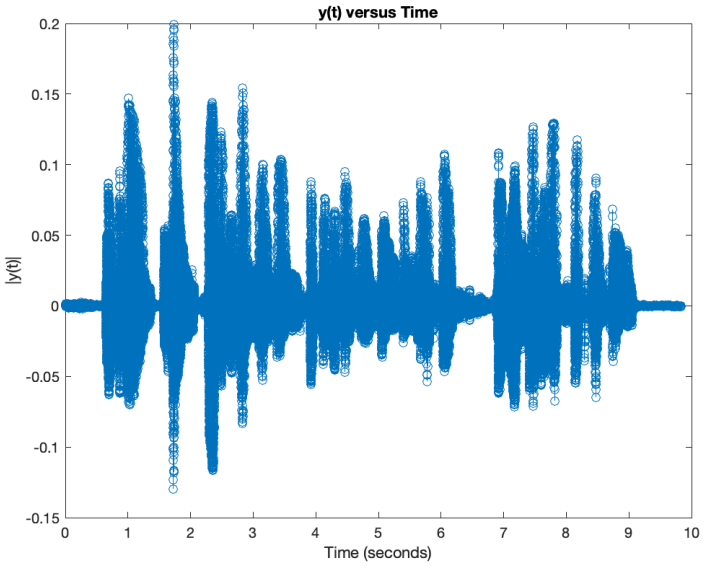


Figure 4.1.1: *The results of the linear convolution of "DSP_Speech.wav" with "HRIR_1.mat" using frequency-domain multiplication.*

Secondly, the linear convolution was performed using the in-built `conv()` function (**Figure 4.1.2**).

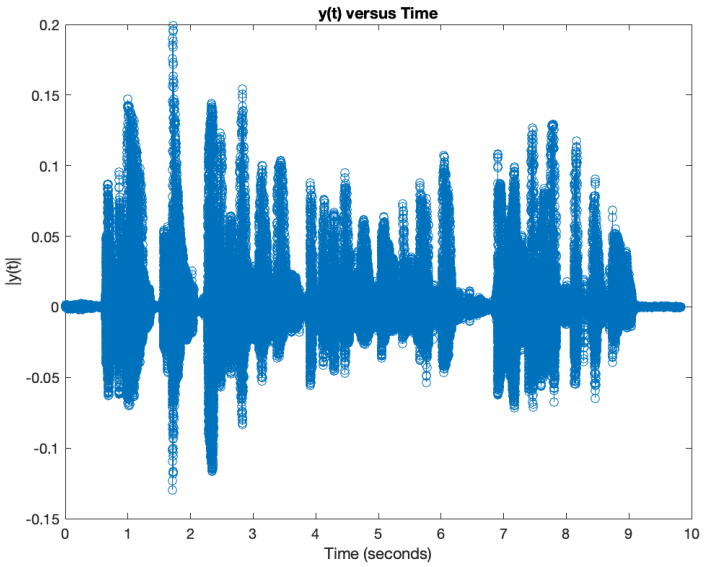


Figure 4.1.2: *The results of the linear convolution of "DSP_Speech.wav" with "HRIR_1.mat" using the* `conv()` *function.*

Question 4.2

The convolution of the provided audio clip, *"DSP_Speech.wav"* with the impulse response *"HRIR_2.mat"* was computed using the built-in `conv()` function. The result of this was combined with the signal depicted in Figure 4.1.2 to form a 2-channel audio vector. The `audiowrite()` function was used to save the 2-channel vector as a 2-channel .wav file

The head-related transfer function is the frequency response function of the human ear. It characterises how an ear receives sound at a point in space. Convolution of an arbitrary source sound with the head-related impulse response (HRIR) has been used to produce virtual surround sounds [1]. This is evident in the produced *"Binarual_Alder.wav"* files, which is convolved with the HRIR such that it appears to originate to the right of the listener, in comparison to the original *"DSP_Speech"* clip. Plots of the utilised HRIRs are seen in **Figure 4.2.1**.

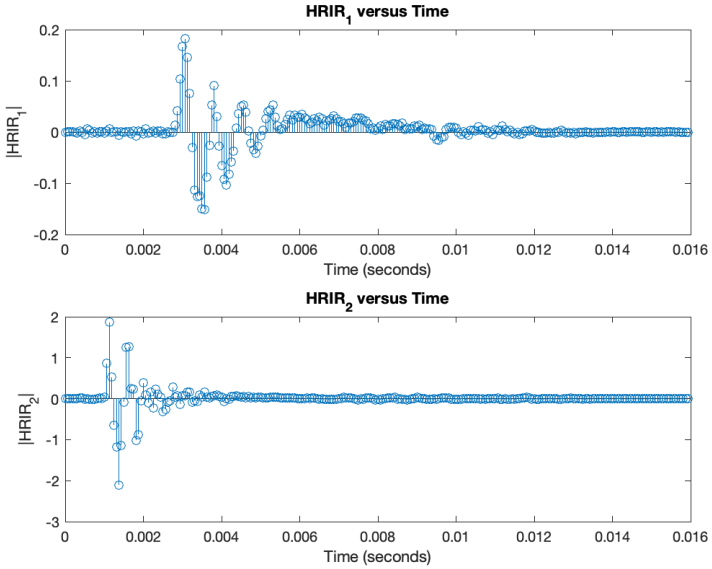


Figure 4.2.1: *The time-domain plots of "HRIR_1.mat" and "HRIR_2.mat".*

Based on the listening experience, and considering these plots, it is evident that *"HRIR_2.mat"* corresponds to the right ear due to its larger magnitude. This analysis is supported by the additional time delay present in *"HRIR_1.mat"*, which serves to mimic the effect of the sound travelling further. This suggests that the HRIR matrices serve to transform the original recorded signal such that it appears to originate to the right of the listener; travels directly into the right ear; and travels around the listener, reflecting off of some surface (e.g. a wall) and into the left ear.

Question 5

Question 5.1

The `dft2()` function was used to analyse the frequency content of the voice recording performed in Question 2. The magnitude plot of the frequency spectrum produced by the `dft2()` function can be seen in **Figure 5.1.1**.

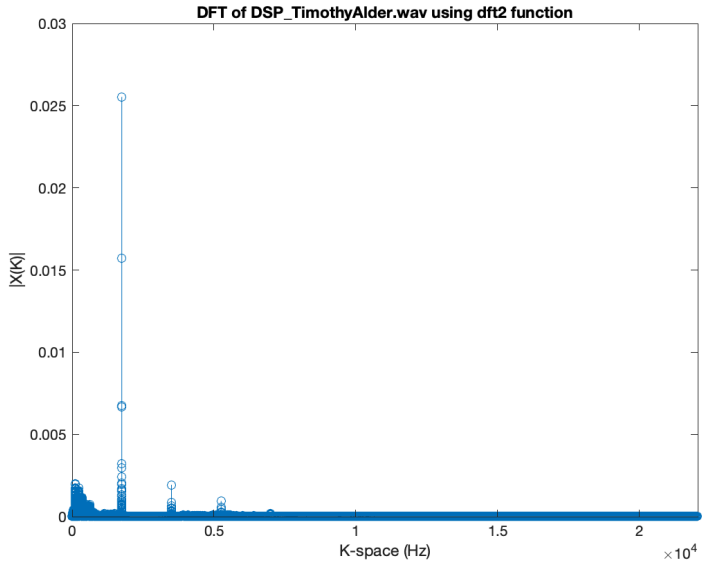


Figure 5.1.1: *Magnitude of the DFT of the recording DSP_TimothyAlder.wav performed in Question 2. The frequency information of the recorded speech is primarily seen in lower frequencies, while three distinct peaks are observed corresponding to the first, second and third-order harmonics of DSP_Noise.wav.*

Question 5.2

The `dft2()` function was used to analyse the frequency content of the provided *"DSP_Noise.wav"* file. The magnitude plot of the frequency spectrum produced by the dft2() function can be seen in **Figure 5.2.1**.

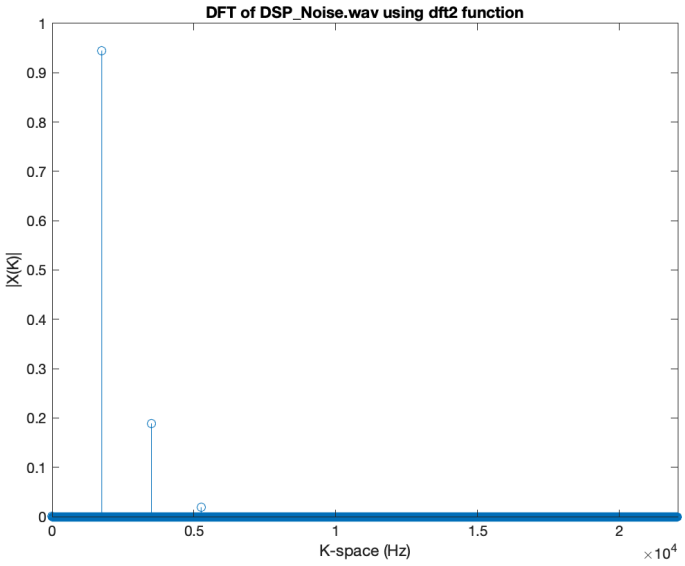


Figure 5.2.1: *Magnitude of the DFT of the provided DSP_Noise.wav. Three distinct peaks are observed corresponding to the first, second and third-order harmonics of noise.*

Three distinct peaks can be seen in the magnitude spectrum of the DFT. The first of these peaks, seen at 1750Hz, corresponds to the fundamental frequency of the signal (f_0) while the next two peaks, at 3500Hz and 5250Hz respectively, correspond to the second and third-order harmonics of the signal.

Question 5.3

The magnitude of the DFT of the recorded signal DSP_TimothyAlder.wav (**Figure 5.1.1**) contains both the frequency content of DSP_Noise.wav and the recorded speech. Accordingly, the frequency content corresponding to the recorded speech is that in **Figure 5.1.1** which is independent of **5.2.1**. Observing the two figures, this content is primarily contained within lower frequency's ($K < 750Hz$).

Question 5.4

A spectrogram of the recorded signal DSP_TimothyAlder.wav was produced using MATLABs spectrogram() function (**Figure 5.4.1**). The produced spectrogram contains thirty-two windows with a visualisation of the DFT of each window (i.e. a short-time Fourier transform of the signal using thirty-two windows). Zero overlap is used for each window.

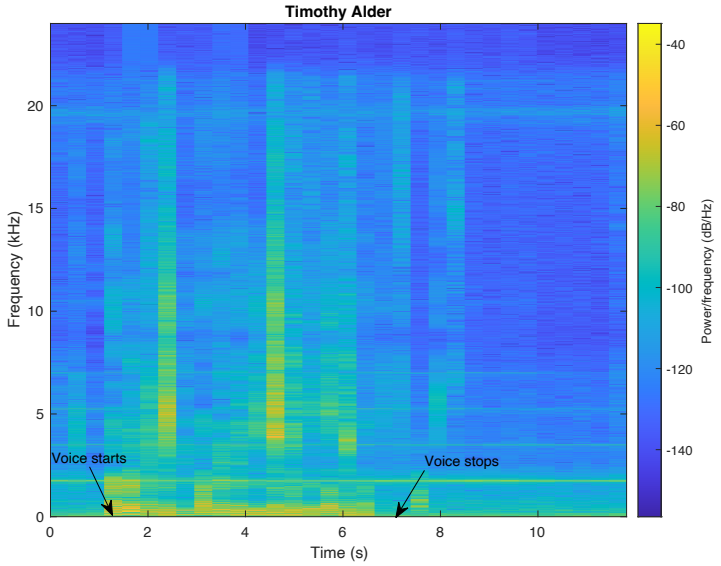


Figure 5.4.1: *Spectrogram of the recorded signal DSP_TimothyAlder.wav.*

Another way of visualising the output of the spectrogram function is using the waterplot function [2]. The equivalent waterplot for the spectrogram shown in **Figure 5.4.1** is shown in **Figure 5.4.2**.

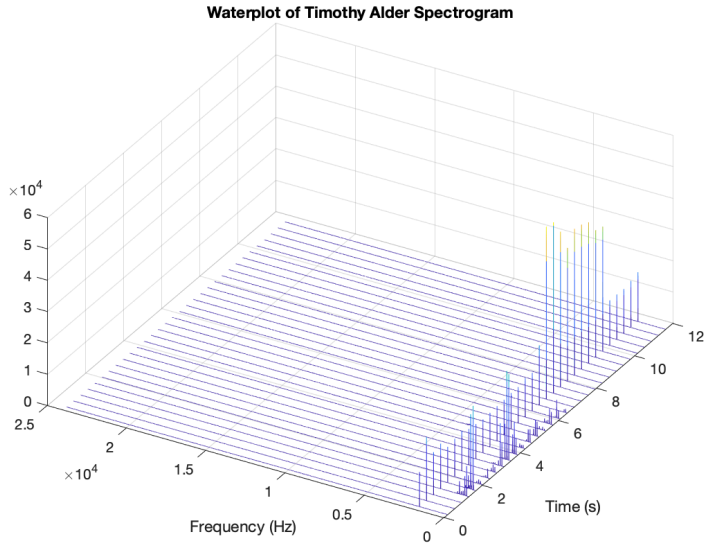


Figure 5.4.2: *Waterplot corresponding to the spectrogram seen in* ***Figure 5.4.1****.*

The waterplot clearly demonstrates that the spectrogram function has computed thirty-two distinct DFTs of the input signal – one for each window. Furthermore, the frequency content corresponding to the recorded speech, seen from 1.29 to 6.46 seconds, is clearly distinguishable from the added noise.

Question 5.5

The spectrogram() function has the input arguments X, WINDOW, NOVERLAP, NFFT, and Fs. The input arguments are summarised in **Table 5.5.1**.

Table 5.5.1: *Input arguments of the spectrogram function.*

Argument	***Description***
X	*The input signal.*
WINDOW	*The desired number of intervals into which the input signal will be divided (i.e. the window length of the STFT). For each interval, the DFT will be computed.*
NOVERLAP	*The overlap between each window, specified in samples.*
NFFT	*The number of frequency points used to compute the DFTs*
Fs	*The sampling frequency of the input signal.*

Accordingly, the NFFT parameter directly influences the size of the computed frequency vector, F; and the WINDOW and NOVERLAP parameter directly influences the size of the computed time vector, T. For example, if a 50% overlap is specified, then the resolution of the T vector is given by:

$$T_{res} = \frac{WINDOW - NOVERLAP}{Fs}$$

Question 5.6

A function called `FindSignalStart()` was written to detect the start time of the speech signal in the audio recording *"DSP_TimothyAlder.wav"*. This function takes the *"DSP_TimothyAlder.wav"* sample vector as input, clips the signal to begin at the speech onset, and provides the speech signal's start time. It employs the `spectrogram()` as seen in **Figure 5.4.1** to locate the first frequency magnitude component surpassing a specified threshold in each time window. The resulting indices vector contain two clusters: one for high-frequency noise and the other for low-frequency speech. The indices vector was binarized and the low-frequency cluster's first window index determines the speech start time and the input signal is clipped to begin at this new start time. The full code for this function can be found in **Appendix D**.

Using NOVERLAP=0 gives finer time resolution but means we won't be able to perfectly reconstruct signal from the STFT (as the Hamming Window will scale some frequency amplitudes in such a way that the original signal cannot be reconstructed). However, seeing as reconstruction isn't important for our application, NOVERLAP=0 is used for the increased time resolution.

Question 5.7

A function called `FindSignalStop()` was written that uses the `spectrogram()` function to determine the time at which the speech signal ends in the recorded audio signal *"DSP_TimothyAlder.wav"*. This was accomplished using the same workflow outlined in Question 5.6, except the binarized vector of indices is now flipped such that **the last** time interval (i.e. window) corresponding to the low-frequency cluster is determined. The full code for this function can be found in **Appendix E**.

Question 5.8

The recorded audio DSP_TimothyAlder.wav was passed through both the `FindSignalStart()` and `FindSignalStop()` functions such that the recording was clipped of any content not containing the speech signal. The resulting signal was saved as *"DSP_Chopped_TimothyAlder.wav"*. The spectrogram of this chopped signal is shown in **Figure 5.8.1**.

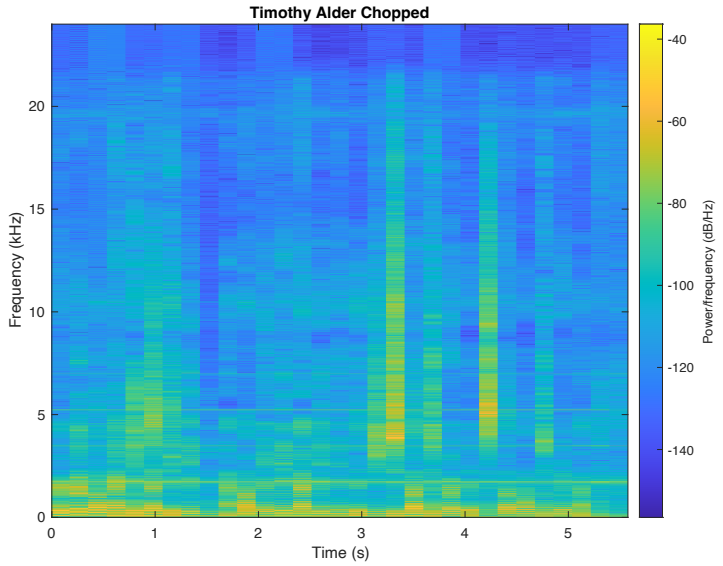


Figure 5.8.1: *Spectrogram of the recorded signal DSP_TimothyAlder.wav. Thirty-two windows are used with zero overlap.*

Question 6

Question 6.1

The fvtool() function was used to generate filter magnitude response plots for the provided three filters (**Figures 6.1.1-3**).

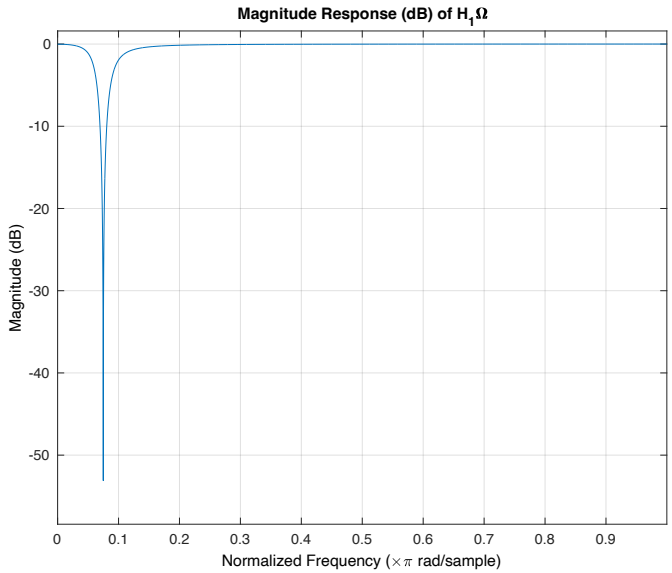


Figure 6.1.1: *Magnitude response (dB) of $H_1\Omega$. The magnitude response is mostly uniform (i.e. $gain = 1$) with significant attenuation of normalised frequency $0.075\pi \times \frac{rad}{sample}$ (Corresponding to 1800 Hz). Thus, the filter is an implementation of a stopband filter.*

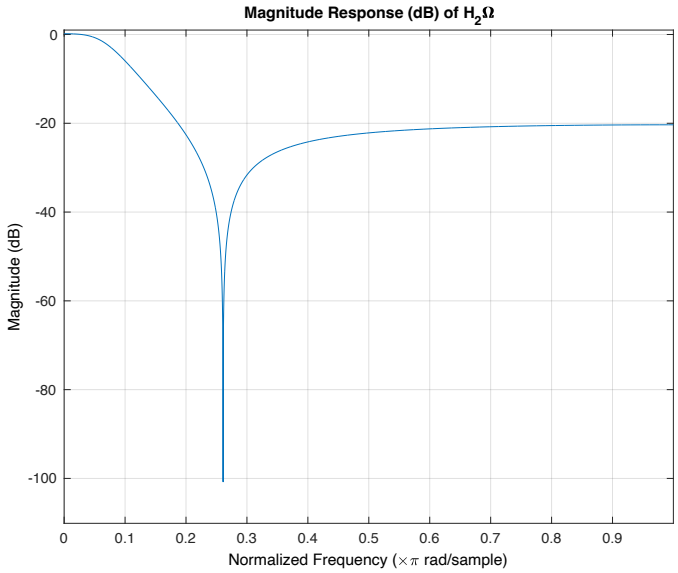


Figure 6.1.2: *Magnitude response (dB) of $H_2\Omega$. The magnitude response is uniform for low frequencies, while attenuating high frequencies. Thus, the filter is an implementation of a low-pass filter. Significant attenuation is observed for normalised frequency* $0.26\pi \times \frac{rad}{sample}$ *(corresponding to 6240 Hz), with the high-frequency attenuation converging to -20dB.*

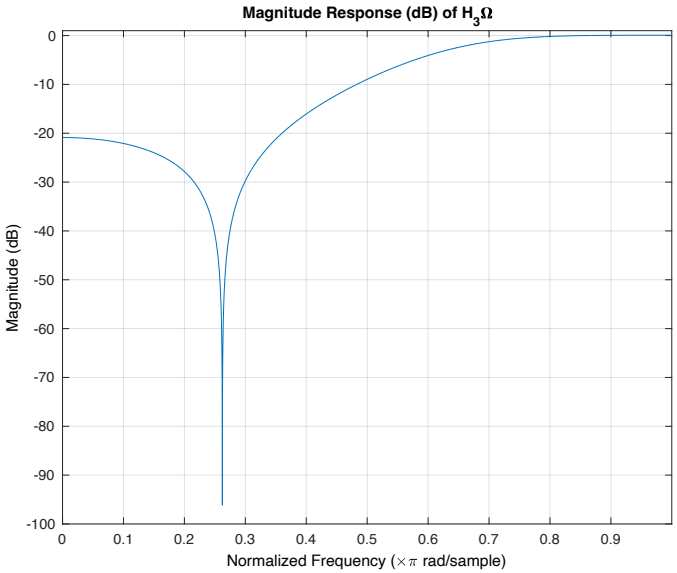


Figure 6.1.3: *Magnitude response (dB) of $H_3\Omega$. The magnitude response is uniform for high frequencies, while attenuating low frequencies. Thus, the filter is an implementation of a high-pass filter. Significant attenuation is observed for normalised frequency* $0.26\pi \times \frac{rad}{sample}$ *(corresponding to 6240 Hz), with the low-frequency attenuation converging to -20dB.*

Question 6.2

Filter $H_2\Omega$ was chosen as the most suitable filter for removing the noise present in *"DSP_TimothyAlder.wav"* due to its attenuation of all high-frequency content.

Comparatively, filter $H_3\Omega$ would attenuate the low-frequency speech content while amplifying the undesired *"DSP_Noise.wav"* signal, and filter $H_1\Omega$ would provide excellent attenuation of the noise observed at normalised frequency $0.075\pi \times rad/sample$, but would fail to attenuate any of the higher-frequency noise content. The shape of the frequency spectrum of the filtered signal is shown in **Figure 6.2.1**.

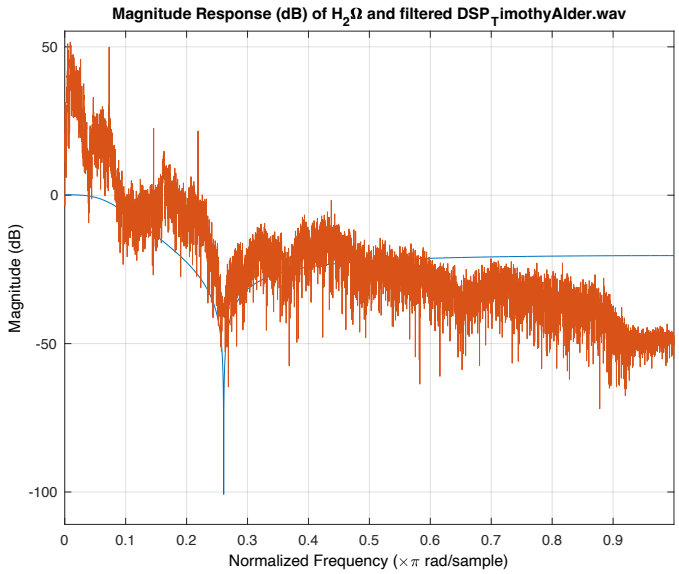


Figure 6.2.1: *Magnitude response (dB) of $H_2\Omega$ and the filtered DSP_TimothyAlder.wav signal. Attenuation of the high-frequency content of DSP_TimothyAlder.wav is observed with significant attenuation of the signal at normalised frequency* $0.26\pi \times \frac{rad}{sample}$.

Listening to the filtered signal, it is clear the high-frequency noise content has been attenuated, although it is still audible in the signal. This is to be expected, as the filter provides minimal attenuation of the fundamental frequency of the noise signal at 1750 Hz. Furthermore, attenuation of the speech signal is clear as the recorded voice is not as clear as when listening to the original *"DSP_TimothyAlder.wav"* signal.

Question 6.3

Next the recorded speech signal was filtered with the provided filter $H_4\Omega$. A pole-zero plot of $H_4\Omega$ is show in **Figure 6.3.1**.

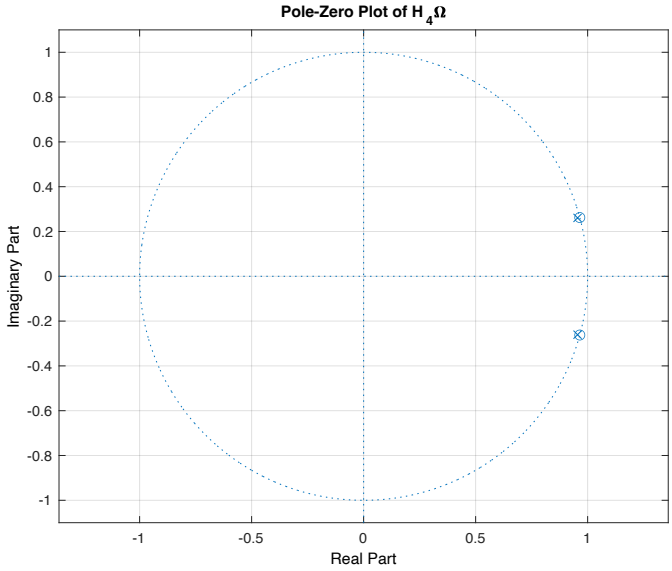


Figure 6.3.1: *Pole-zero plot of* $H_4\Omega$

Listening to the filtered speech signal, virtually no difference is heard in comparison to the original recording. This observation can be explained through the pole-zero plot in **Figure 6.3.1**. The pole-zero plot has a pair of complex conjugate poles and zeros. A pair of complex conjugate poles or zeros means the transfer function (frequency response) will have a peak or notch, respectively. The frequencies (i.e. phase) corresponding to the complex conjugate zeros is given by:

$$\tan^{-1}\left(\frac{0.262499}{0.965}\right) = 0.2656\, rads$$

Comparatively, the frequency (i.e. phase) corresponding to the complex conjugate poles is given by:

$$\tan^{-1}\left(\frac{0.2607202}{0.955}\right) = 0.2665\, rads$$

Thus, considering the pair of complex conjugate zeros, we expect the numerator of the transfer function to have a notch at frequency $0.2656\, rads$. Furthermore, considering the pair of complex conjugate poles, we expect the denominator to have a peak at $0.2665\, rads$. Considering the numerator and denominator together, we expect the transfer function to have a notch around $0.2656\, rads$, whose sharpness is determined by how close together the poles and zeros are. Plotting the frequency response verifies this analysis, with a notch observed at normalised frequency $0.08447266\pi \times \frac{rad}{sample}$, corresponding to $0.2654\, rads$. Translating this to Hz yields 2027.3 Hz. Thus, we expect the filtered signal to have significant attenuation at 2027.3 Hz. Seeing as there is no noise content and minimal speech content at this frequency in the original signal, we expect the filtered signal to sound virtually identical to the original signal, which explains the observed result.

Question 7

Question 7.1

Question 7.1.1

The provided music signal *"DSP_Music.wav"* was modulated with a 9000 Hz sine wave. The time domain, frequency magnitude and phase response plots of the original signal and resulting amplitude-modulated signal are shown in **Figures 7.1.1-2**.

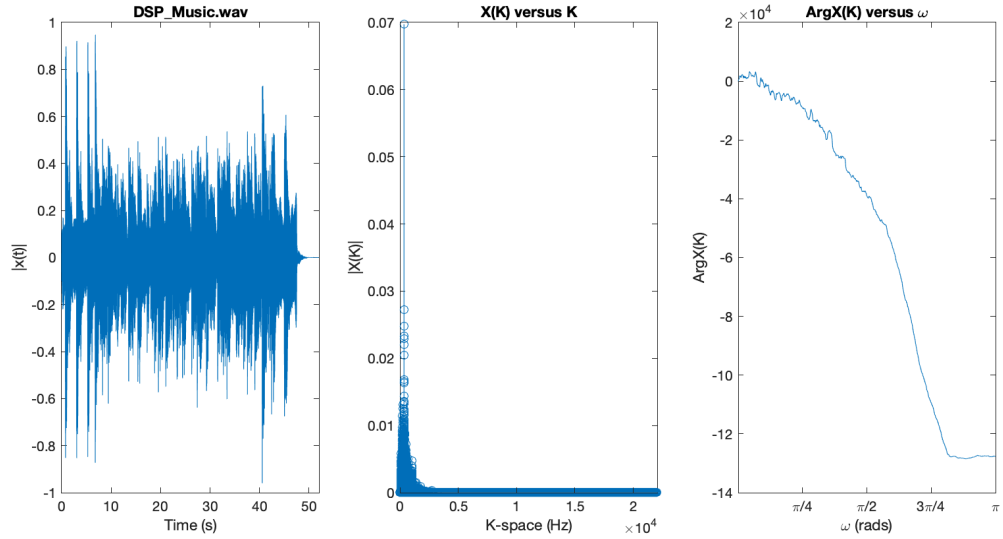


Figure 7.1.1: *Plot of the original music signal and its corresponding frequency magnitude and unwrapped phase response plots.*

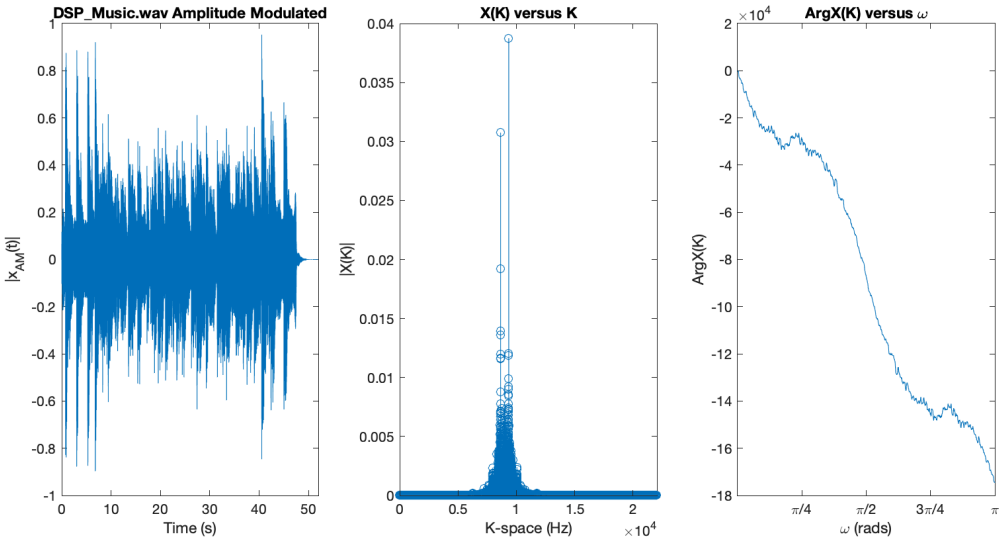


Figure 7.1.2: *Plot of the amplitude-modulated music signal and its corresponding frequency magnitude and unwrapped phase response plots.*

Question 7.1.2

Comparing the frequency magnitude plots in **Figure 7.1.1** with **Figure 7.1.2**, the frequency magnitude has been shifted by 9000 Hz, and the magnitude has been scaled down by a factor of roughly ½. Notably, the mirrored frequency component of the signal (i.e. from $[-\pi\ 0]$) can be seen in the shifted signal.

Comparatively, comparing the phase components, a duplication and shift in the phase response is observed. This is better illustrated by observing the phase response of the original signal on the interval $[-\pi\ \pi]$ (**Figure 7.1.3**).

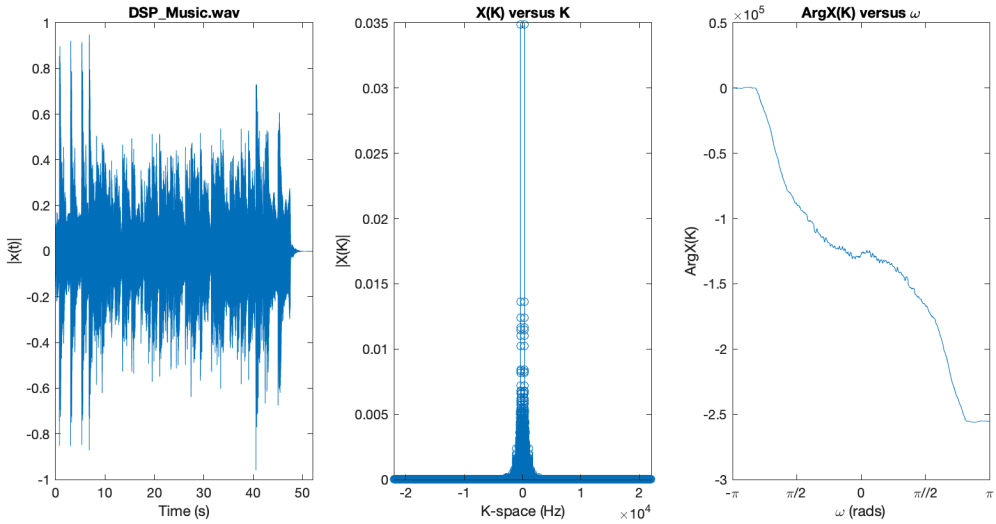


Figure 7.1.3: *Plot of the original music signal and its corresponding frequency magnitude and unwrapped phase response plots on the interval* $[-\pi\ \pi]$.

Considering this plot, it is evident a duplication in the phase response is observed in **Figure 7.1.2** centred at $\frac{\pi}{4}$ and $\frac{3\pi}{4}$. Furthermore, it is clear we have observed a shift in the phase response in comparison to **Figure 7.1.1**.

Question 7.1.2 Part A

The DFT of the modulation signal is shown in **Figure 7.1.4**.

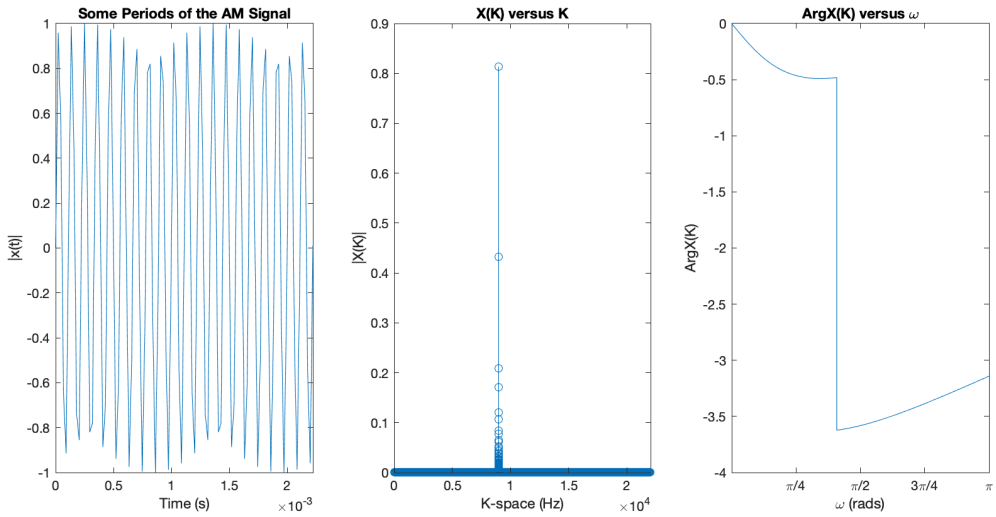


Figure 7.1.4: *Plot of the modulation signal and its corresponding frequency magnitude and unwrapped phase response plots. Similar to Question 3.6, sample size can be used to explain the observed spectral leakage.*

Question 7.1.2 Part B

Figure 7.1.4 can be used to elaborate on the discussion presented in Question 7.1.2. More specifically, the shifting of the frequency spectrum can be explained through the Fourier transform property relating multiplication in the time domain with convolution in the frequency domain. The scaling of the amplitude of the frequency spectrum by ½ can be explained using Parseval's theorem, where the total signal energy should remain conserved despite now considering the DFT of the original signal on interval $[-\pi\ \pi]$ (due to the shift) compared to $[0\ \pi]$. This analysis is supported by **Figure 7.1.3**. As explained in Question 3.6, some spectral leakage is observed in the 9000 Hz peak due to the chosen sample size. This explains the remaining observed change in magnitude (e.g. why the symmetrical peaks around 9000 Hz do not have equal amplitudes).

Thus, if a sample size was used that provides a 9000 Hz frequency sample, one would expect to see a perfect copy of the original signal shifted by 9000 Hz, with an amplitude scaling of ½. This is demonstrated in **Figure 7.1.5**, where the same operations as above are performed, but the $x(t)$ signal is first clipped to the closest integer multiple of the sampling frequency (in this case, $52 \times Fs$).

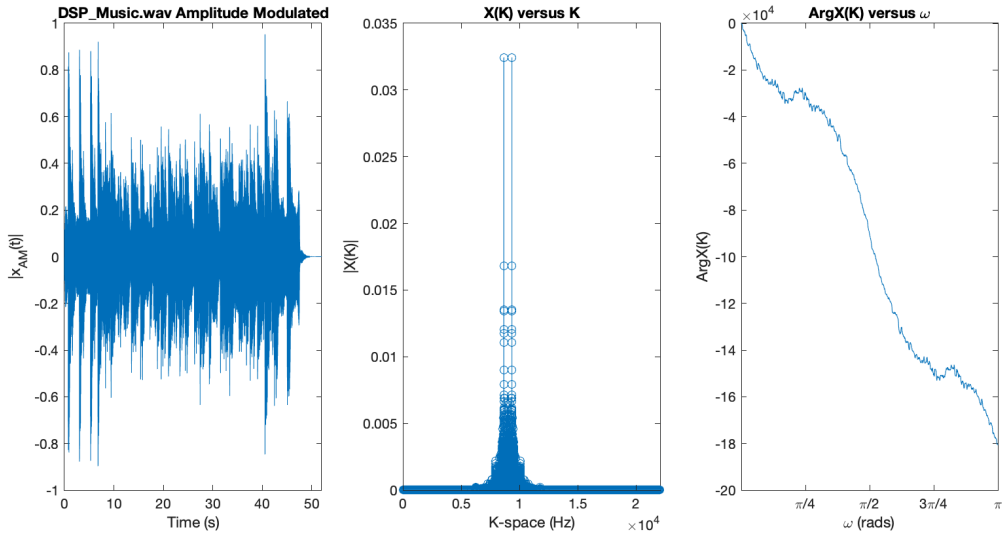


Figure 7.1.5: *Plot of the amplitude modulated music signal clipped to the nearest integer multiple of the sampling frequency and its corresponding frequency magnitude and unwrapped phase response plots.*

Furthermore, comparing the phase components, **Figure 7.1.4** suggests that the modulation signal has a non-linear phase response, however, if we examine the DFT of the same modulation signal without spectral leakage (by clipping the modulation signal to the closest integer multiple of the sampling frequency), we can see the phase response is in fact linear (**Figure 7.1.6**).

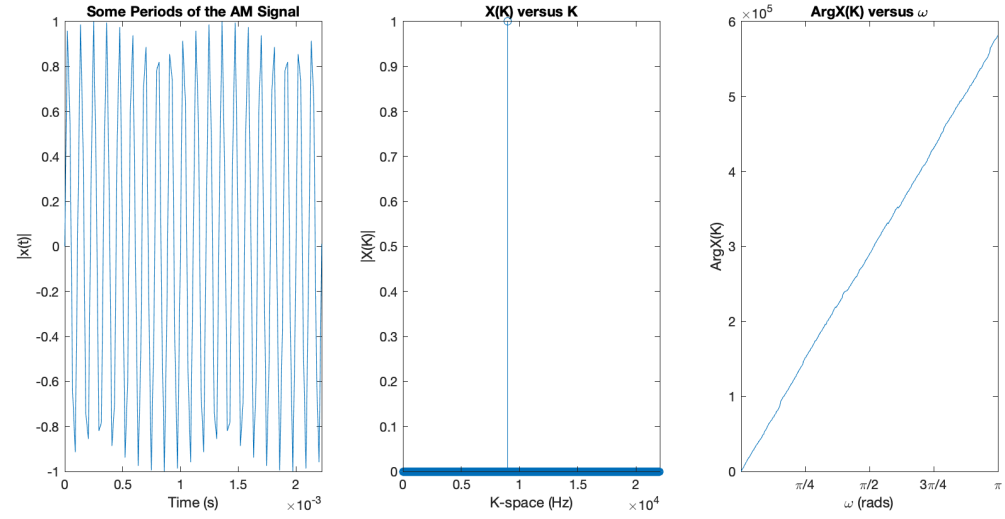


Figure 7.1.6: *Plot of the modulation signal and its corresponding frequency magnitude and unwrapped phase response plots. Without spectral leakage, a linear phase response is observed.*

This is expected, as the DFT of a single sinusoid is an impulse at $\pm$ the frequency of the sinusoid. The group delay of a linear phase response is a constant. This indicates that the modulation signal imparts a constant delay to all input frequencies, corresponding to a shift in the phase response with no distortion. This explains the results we obtained in **Figure 7.1.1**.

Question 7.1.3

Comparing the time domain plots between the original and the amplitude modulated *"DSP_Music.wav"* signal, the form of the two signals is largely identical with some variation in amplitude observed for the amplitude modulated signal.

Listening to the two signals provides more insight into difference; the amplitude modulated signal has the same chords as *"DSP_Music.wav"* but at a significantly higher octave (i.e. frequency). This analysis is supported by **Figure 7.1.2**.

Question 7.2

The original signal can be reconstructed by multiplying the modulated signal with the original modulating carrier wave. This can be explained using trigonometric identities. The modulated signal is given by:

$$x(t)\sin\left(9000 \times 2\pi \times t\right)$$

And the demodulated signal is given by:

$$\begin{aligned} & x(t)\sin(9000 \times 2\pi \times t)\sin(9000 \times 2\pi \times t) \\ & = x(t)\frac{(\cos(0) - \cos(18000 \times 2\pi \times t))}{2} \\ & = \frac{1}{2}x(t) - \frac{1}{2}x(t)\cos(2 \times 9000 \times 2\pi \times t) \end{aligned}$$

Thus, by multiplying the demodulated signal with a gain of two, we perfectly recover the source signal, although with some residual high-frequency content from the modulation process remaining centred around two times the carrier frequency. Demodulating the modulated signal from Question 7.1.1 and computing the DFT, we can confirm this analysis (**Figure 7.1.5**).

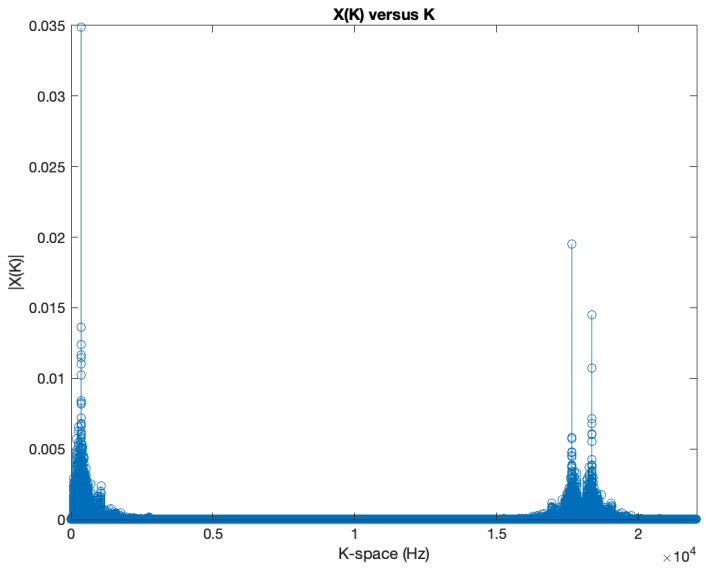


Figure 7.1.4: *Plot of the frequency magnitude spectrum of the demodulated music signal. The frequency content of the original signal is restored with magnitude scaled by a factor of ½, and residual high-frequency content from the modulation process remains centred around 18000 Hz, two times the carrier frequency of 9000 Hz.*

Listening to the restored signal, the music is quieter due to the amplitude scaling of ½. The higher frequency content in the restored signal is barely discernible to the human ear.

Multiplying the demodulated signal by a gain of two to undo the amplitude scaling, the music is the same as the original signal, although the high-frequency content present in the signal causes speakerphone artifacts that are uncomfortable to listen to.

Question 7.3

A finite impulse response (FIR) filter was designed to remove the residual high-frequency components within the restored signal. The design methodology specified in the Handout for Windowed FIR LPF Design [2] was followed.

Firstly, the passband edge frequency was determined by analysing the DFT of *"DSP_Music.wav"* in **Figure 7.1.1**. Specifically, the desired passband edge frequency was identified as 3000 Hz.

Secondly, a windowing function was chosen that achieved the minimum required stop-band attenuation frequency.

Thirdly, as per the design specifications, the minimum obtainable transition width was calculated using the window-associated formula. The transition width formulae are inversely proportional to the number of filter coefficients used. Accordingly, the maximum odd number of allowed filter coefficients was used, 199. Importantly, an odd number of filter coefficients was used so that the impulse response of the filter is perfectly symmetrical, ensuring no phase distortion.

Fourthly, the transition width and desired passband edge were used to calculate the passband edge used in the filter design, f_1. The corresponding digital frequency was then calculated.

Fifthly, the digital frequency was used to define the IIR of the ideal lowpass filter (LPF) using a sinc function. Sixthly, the inbuilt Hamming window equation MATLAB function, `hamming()`, was used to compute the relevant filter coefficients which were then element-wise multiplied with the specified sinc function. This produced the non-causal FIR of the desired filter.

Finally, the causal FIR of the desired filter was created simply by shifting the non-causal FIR by $\frac{N-1}{2}$ samples.

Question 7.4

Question 7.4.1

The optimal filter for removing the residual high-frequency content in the demodulated signal is a low-pass filter. This is because the desired signal (i.e. the original signal) is contained within lower frequencies, while the undesirable residual signal from the amplitude modulation is contained within higher frequencies. Thus, a low-pass filter can be used to preserve the original signal content while attenuating the undesirable residual modulated high-frequency signal.

Question 7.4.2

A finite impulse response (FIR) filter is chosen over an infinite impulse response (IIR) filter for this process as FIR filters have a linear phase response. This is critical in audio processing applications, where a non-linear phase response will potentially heavily distort the signal in the time domain, introduce undesirable signal artifacts and degrade the overall sound quality of the signal.

Question 7.4.3

A causal filter should be used for the filtering process as it has no inherent delay and will have a linear phase response. Both of these features are attractive for audio-processing applications, where real-time performance is often desirable and a linear phase response is crucial to ensure no distortion of the original input audio signal.

Question 7.4.4

In **Figure 7.4.**1, the causal and non-causal variants of the designed filter are plotted.

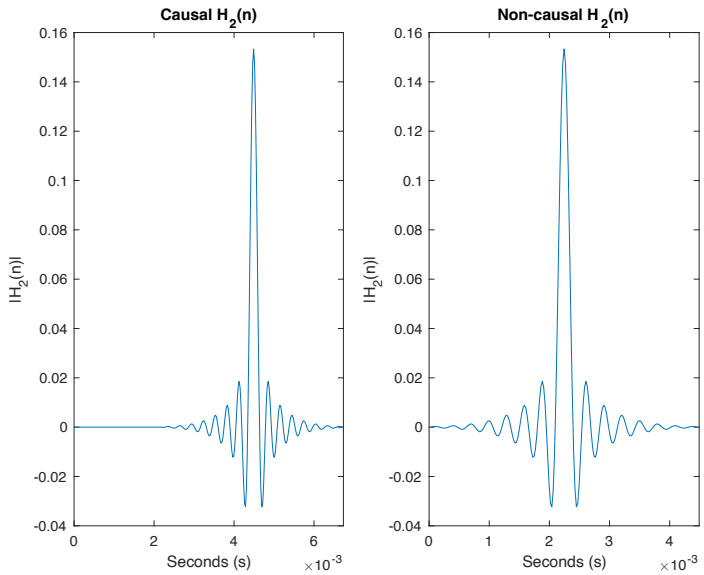


Figure 7.4.1: *Plot of the causal and non-causal impulse response of the designed filter.*

It is not possible to produce a phase response of the non-causal filter. This is because there is no way of specifying to the `fft()` function that the signal is non-causal. Instead, the `fft()` function assumes the first sample in the input vector corresponds to the initial time index (i.e. $t = 0$). The phase response of the causal filter is plotted in **Figure 7.4.2**.

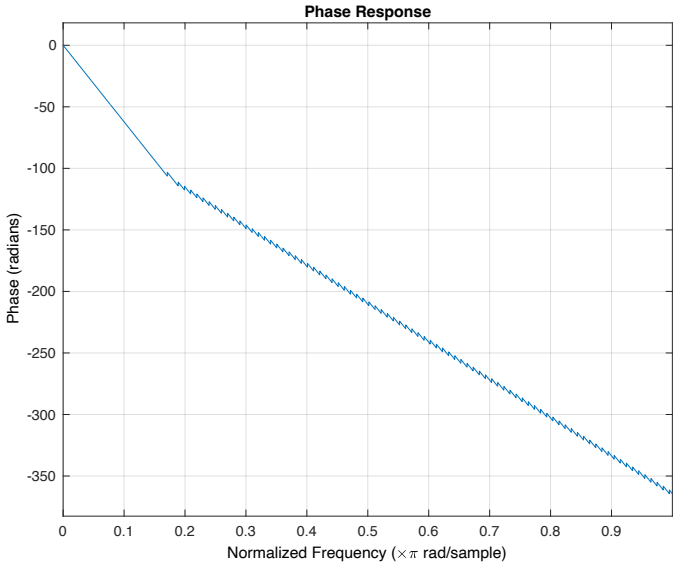


Figure 7.4.1: *Phase response of the designed causal filter. The phase response is linear in both the passband and stopband, with a steeper gradient in the stopband.*

Question 7.4.5

Step five of the 'Handout for Windowed FIR LPF Design' document states to shift the impulse response of the produced filter by $\frac{N-1}{2}$ samples such that the impulse response is made to be causal.

In the time domain, making the filter causal means that the output of the filter depends only on past and present values of the signal, not future values. This means the filter is capable of operating in real-time and any delay introduced by the filter to the output signal is predictable and known in advance. In the frequency domain, making the filter causal ensures it has a linear phase response. This means the relative time relationships of different frequency components of the input signal is preserved in the output of the filter. This is crucial for preserving signal integrity in audio processing applications.

Comparatively, the output of the non-causal filter depends on future values of the signal which inherently introduces varying delays in the time domain. In the frequency domain, the non-causal filter has a non-linear phase response which can introduce unpredictable phase shifts that vary with frequency.

Question 7.5

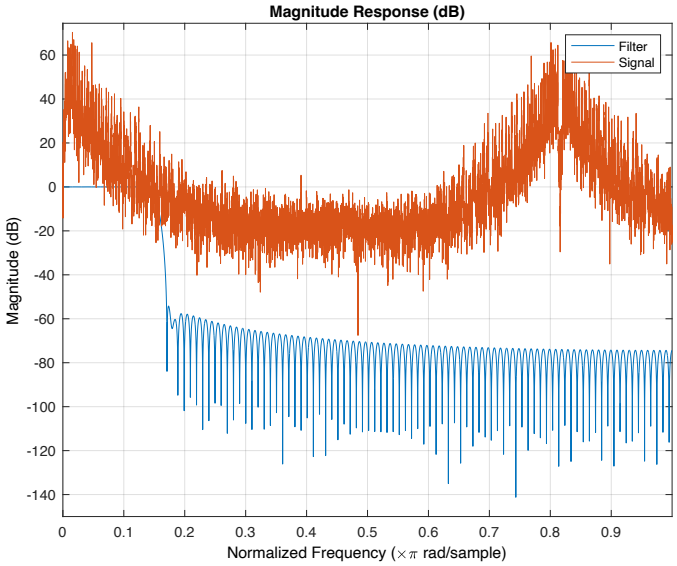


Figure 7.5.1: *Plot of the frequency response magnitude of the designed FIR filter and the corrupted signal.*

Question 7.6

The following parameters (**Table 7.6.1**) were used in the filter design process.

Table 7.6.1: *Parameters used in the FIR filter design process.*

Parameter	**Value**
Passband edge frequency	3381.2 Hz
Transition width	762.3317 Hz
Window type	Hamming
Number of filter coefficients	199
Stop band attenuation	55 dB

The design specification stated that a stop band attenuation greater than or equal to 50 dB should be used. Thus, a Hamming window was chosen as it provides a stop band attenuation of 55 dB.

The filter design specifications stated that the minimum possible transition width should be used and provided a maximum of 200 filter coefficients. Thus, seeing as the transition width is inversely proportional to the number of filter coefficients, the maximum allowable number of filter coefficients was used. Importantly, an odd number of filter coefficients was used so that the impulse response of the filter is perfectly symmetrical ensuring no phase distortion.

For a Hamming window, the transition width of the filter is given by:

$$3.44 \times \frac{f_s}{\#\ of\ filter\ coefficients} = 3.44 \times \frac{44100}{199} = 762.3317\ Hz$$

Finally, the passband edge frequency was determined by taking the desired cutoff frequency (3000 Hz) and adding half of the transition width.

Question 7.7

Considering the frequency response magnitude plot of the actual designed FIR filter, the observed actual stop band attenuation is -57.6827 dB and the -3 dB bandwidth is 3316.1 Hz. The actual stop band attenuation was obtained programmatically by computing the frequency response of the filter, converting it to dB scale, and using the find() command to find the magnitude of the first index greater than the stop band edge (i.e. the desired passband edge plus the transition width). Similarly, the -3 dB bandwidth (i.e. the cutoff frequency) was found by using the `find()` command to find the first instance below -3dB. The code for this task is included below.

```
% Compute the frequency response of the filter
[H, w] = freqz(h2_n, 1);

% Find the -3dB frequency programmatically
magdB = 20 * log10(abs(H));  % Convert magnitude to decibels
idx = find(magdB <= -3, 1, 'first');  % Find the first frequency point below -3dB
cutoff_frequency = w(idx) * (Fs / (2 * pi));  % Convert to Hz

% Find the stopband attenuation programmatically
idx = find(w*Fs/(2*pi) >= f_c+TW, 1, 'first');  % Find the first frequency point
above 3000Hz
stopband_attenuation = magdB(idx);
```

Question 7.8

A plot of the frequency response magnitude of the designed FIR filter as well as the filtered, demodulated signal is shown in **Figure 7.8.1**.

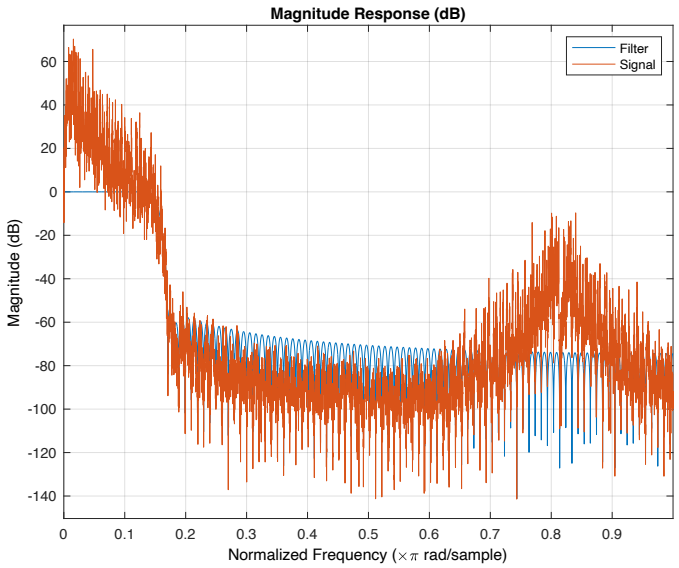


Figure 7.8.1: *Plot of the frequency response magnitude of the designed FIR filter and the filtered signal.*

Question 7.9

The DFT of the resulting filtered version of the corrupted signal is shown in Figure 7.9.1.

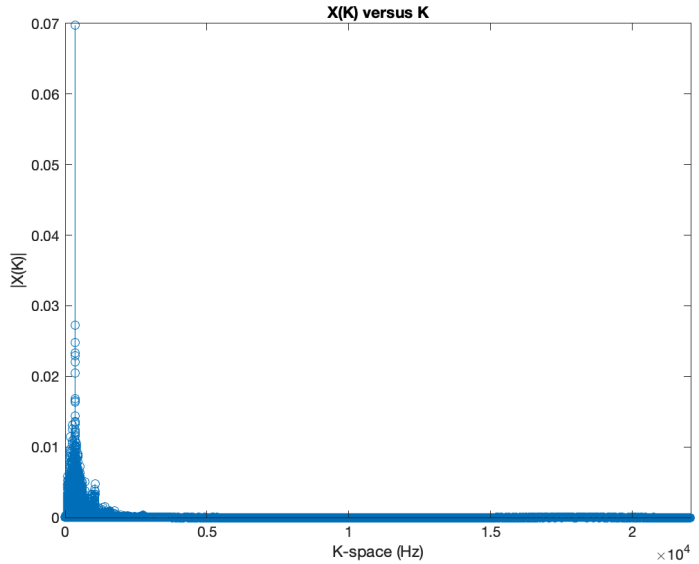


Figure 7.9.1: *Plot of the frequency response magnitude of the filtered signal.*

As indicated by the DFT, the high-frequency artifacts left by the modulation process have successfully been removed. The resulting signal is audibly identical to the original signal, though is not a perfect replica due to the non-ideal implementation of an LPF. This is illustrated in **Figure 7.8.1**, where we can see some of the high-frequency artifacts remain in the filtered signal (although their magnitude is miniscule). Finally, it is arguable the signal has lost some of its 'crispness' due to the attenuation of small high-frequency components present in the original signal. This could be improved by increasing the desired passband of the designed filter. In retrospect, a 3000 Hz passband edge was much smaller than necessary due to the small transition width of the filter and the undesirable high-frequency content beginning at ~14000 Hz. Despite this, however, one can say with confidence that >99% of the original signal has been recovered.

References

[1] Sima, S. (2008) *HRTF Measurement and Filter Design for a Headphone-Based 3D-Audio System*. HAW Hamburg. [Online]. Available: https://reposit.haw-hamburg.de/bitstream/20.500.12738/9618/1/BA_Sylvia_Sima.pdf [Accessed 10 Oct. 2022].

[2] MATLAB. (2023) *spectrogram*. [Online]. Available: https://au.mathworks.com/help/signal/ref/spectrogram.html [Accessed 13 Oct. 2022].

[3] College of Cybernetics, Engineering and Computer Science. (2023) *Handout for Windowed FIR LPF design*. ANU. [Online]. Available: https://wattlecourses.anu.edu.au/pluginfile.php/3490639/mod_folder/content/0/Resource%20Files/Handout%20for%20Windowed%20FIR%20LPF%20Design.pdf?forcedownload=1 [Accessed 15 Oct. 2022].

Appendices

Appendix A

```
% create an audiorecorder object with the correct fs and resolution
mic = audiorecorder(48000,16,1);

% take a 12 second recording
record(mic);
pause(12);
stop(mic);

% get the recording data
% recording = getaudiodata(mic);

% play the recording
clip = audioplayer(recording,48000,16);
play(clip);

% write recording to wav
audiowrite("Results/DSP_TimothyAlder2.wav",recording,48000);
```

Appendix B

```
function x_k=dft1(x_t,fs)

    % computing the DFT based on the number of samples in x_t
    x_k = zeros(1,fs);
    k_space = [0:fs-1];
    n_space = [0:size(x_t,2)-1];
    for k=k_space
        for n=n_space
            x_k(k+1) = x_k(k+1)+(x_t(n+1)*exp(-1i*2*pi/size(x_t,2)*n*k));
        end
    end
    x_k = x_k*2/size(x_t,2);

    % computing the magnitude
    x_k = abs(x_k);

    % computing the freq res
    freq_res = fs/size(x_t,2);
    % use freq res to create scaled k-space
    adj_k_space = [0:freq_res:fs/2];

    % get DFT magnitude values from 0 to Fs/2
    x_k = x_k(1:size(adj_k_space,2));

    % plot the result
    figure()
    stem(adj_k_space,x_k)
    xlim([0,fs/2])
    xlabel('K-space (Hz)')
    ylabel('|X(K)|')
    title('X(K) versus K')

end
```

Appendix C

```
function x_k=dft2(x_t,fs)

    % computing the DFT
    x_k = fft(x_t);
    x_k = x_k*2/size(x_t,2);

    % computing the magnitude
    x_k = abs(x_k);

    % computing the freq res
    freq_res = fs/size(x_t,2);
    % use freq res to create scaled k-space
    adj_k_space = [0:freq_res:fs/2];

    % get DFT magnitude values from 0 to Fs/2
    x_k = x_k(1:size(adj_k_space,2));

    % plot the result
    figure()
    stem(adj_k_space,x_k)
    xlim([0,fs/2])
    xlabel('K-space (Hz)')
    ylabel('|X(K)|')
    title('X(K) versus K')

end
```

Appendix D

```
function [y,StartTime]=FindSignalStart(x_t)

    % Variable initialisation
    Fs = 48000;

    % Computing the spectrogram of the signal
    time_steps = 32; % Number of desired separate intervals of x_t
    WINDOW = size(x_t,1)./time_steps; % Corresponding window size
    NOVERLAP = 0; % Overlap between windows
    [S,F,T] = spectrogram(x_t,WINDOW,NOVERLAP,[],Fs);

    % Threshold all values in each signal power window less than 3,000 to 0
    thresholded = abs(S.^2)>3000;
    % Return the index of the first non-zero value of each row
    [~, first_indices] = max(thresholded, [], 1);
    % At this point, first_indices contains two clusterings (high freqs and
    % low freqs). We want to isolate the lower freqs.

    % binary threshold all the higher indices to be zero
    first_indices(first_indices>(max(first_indices)-min(first_indices))) = 0;
    % Get the corresponding StartTime
    StartTime = T(find(first_indices~=0,1));
    % Trim the input signal using the StartTime
    y = x_t((StartTime*Fs):end,:);

end
```

Appendix E

```
function [y,EndTime]=FindSignalStop(x_t)

    % Variable initialisation
    Fs = 48000;

    % Computing the spectrogram of the signal
    time_steps = 32; % Number of desired separate intervals of x_t
    WINDOW = size(x_t,1)./time_steps; % Corresponding window size
    NOVERLAP = 0; % Overlap between windows
    [S,F,T] = spectrogram(x_t,WINDOW,NOVERLAP,[],Fs);

    % Threshold all values in each signal power window less than 3,000 to 0
    thresholded = abs(S.^2)>3000;
    % Return the index of the first non-zero value of each row
    [~, first_indices] = max(thresholded, [], 1);
    % At this point, first_indices contains two clusterings (high freqs and
    % low freqs). We want to isolate the lower freqs.

    % binary threshold all the higher indices to be zero
    first_indices(first_indices>(max(first_indices)-min(first_indices))) = 0;
    % same as FindSignalStart but now we are interested in the LAST
    % non-zero window so flip the indices to find the EndTime
    EndTime = T(find(fliplr(first_indices)~=0,1));
    % trim the input signal using the EndTime
    y = x_t(1:(EndTime*Fs),:);

end
```