DSP: Spectral analysis and Filter design of Fayrouz

Yassin Ahmed 202300304

Part 1: Spectral analysis

We select our samples from the mp3 given from 3.5 to 6.5 seconds, giving us a total of 3 seconds. The code for reading and plotting the signal is attached, afterwards we graph it in Figure 1.

```
% (1)
[y, Fs] = audioread("D:\college\y2s2\DSP\project\music_test_fayrouz.
    mp3");

% (2)
% the piano sounds really nice here :)
samples = y(3.5*Fs:6.5*Fs, 1);

% Create time vector, from 0 to (N-1)/Fs with step size 1/Fs
t = (0:1/Fs:(size(samples,1)-1)/Fs)';

% (3)
figure;
plot(t, samples);
title("Audio samples");
xlabel("Time");
ylabel("Amplitude");
xlim("tight");
ylim([-1 1]);
```

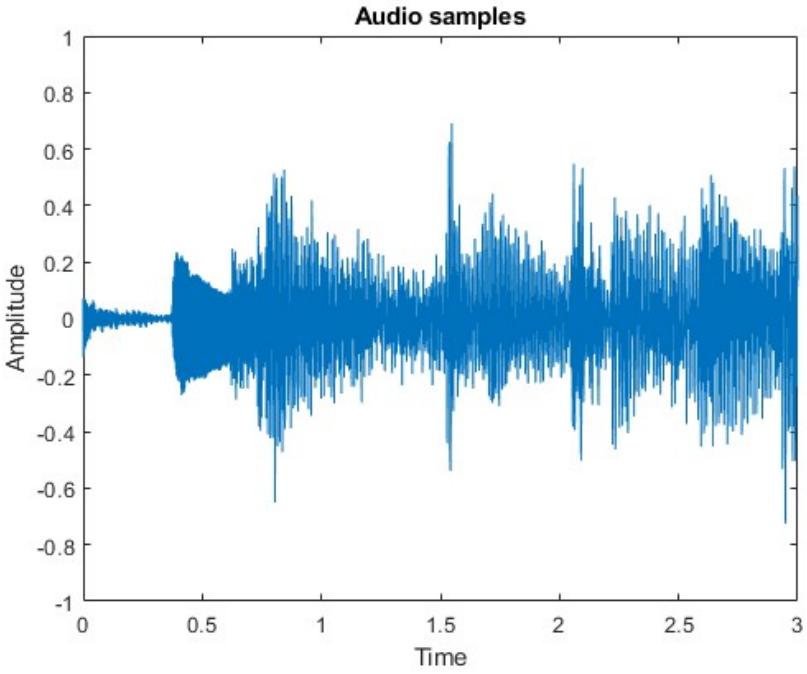

Figure 1: Amplitude of samples taken versus time

The time vector we have constructed above will benefit us in our next step. We now aim to add the following interference tone to our samples:

$$1.8\sin(2\pi(15.2\cdot 10^3)t)$$

To do this, we must have the time nt_s for which each sample in the variable `samples` maps to. Luckily, `t` contains this information and we can use the power of matlab to carry this interference out and listen to it using the following code.

```
% (4)
A = 1.8; tone_freq = 15.2*10^3;
interference = A*cos(2*pi*tone_freq*t);

% (5)
samples_interfered = samples + interference;
sound(samples, Fs);
pause(3.5);
sound(samples_interfered, Fs);
```

To plot the single sided power spectrum density, with the following parameters: (1) FFT Size, (2) Window size, (3) Window type, (4) Percentage overlap between segments taken), (5) Sampling frequency of the input signal; We simply make a function that takes these parameters as the inputs and uses the `pwelch` function in MATLAB.

```
% (6)
function plot_spectrum(signal, Fs, Nfft, WS, WT, percentage_overlap)
    % Put all window function handles in a dictionar then call that
    % dictionary and instantiate a window with the size the user has
        defined
    window_types = ["Rect" "Tri" "Hanning" "Hamming" "Kaiser"];
    window_functions = {@rectwin, @triang, @hann, @hamming, @kaiser
       };
    windows = dictionary(window_types, window_functions);
    window = windows{WT}(WS);


    PSD = pwelch(signal, window, ceil(percentage_overlap*Nfft), Nfft
       , Fs);
    f = 0:Fs/Nfft:Fs/2;
    % frequency in KHz
    f = f/1e3;

    % Plot results
    figure;
    title("Spectral analysis using " + WT + " window");
    plot(f, 10*log10(PSD), 'linewidth', 1.5)
    grid on; xlabel('Frequency (KHz)'); ylabel('PSD (db/Hz)');
end

% Example usage:
Nfft = 2^ceil(log2(length(samples)/10));
percentage_overlap = 0.6;
WS = Nfft;
WT = 'Rect';
plot_spectrum(samples_interfered, Fs, Nfft, WS, WT,
   percentage_overlap)
```

We can see the output of the example in Figure 2. One thing to note is how destructive the spectral leakage here is, making the analysis difficult. Changing the window type to another one can show us how much it affects our output. This is shown in Figure 3 (no other parameters were changed).

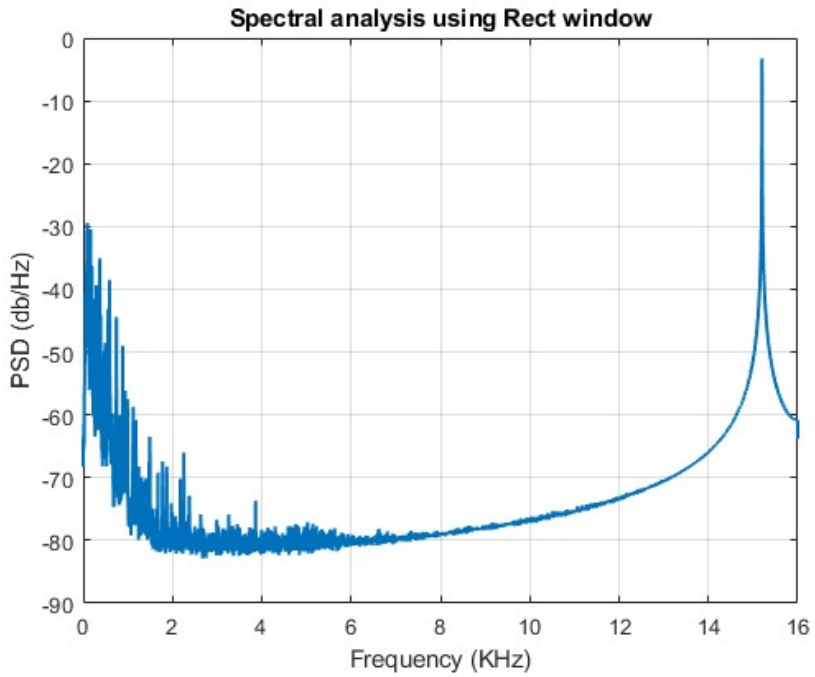

Figure 2: Spectral analysis example

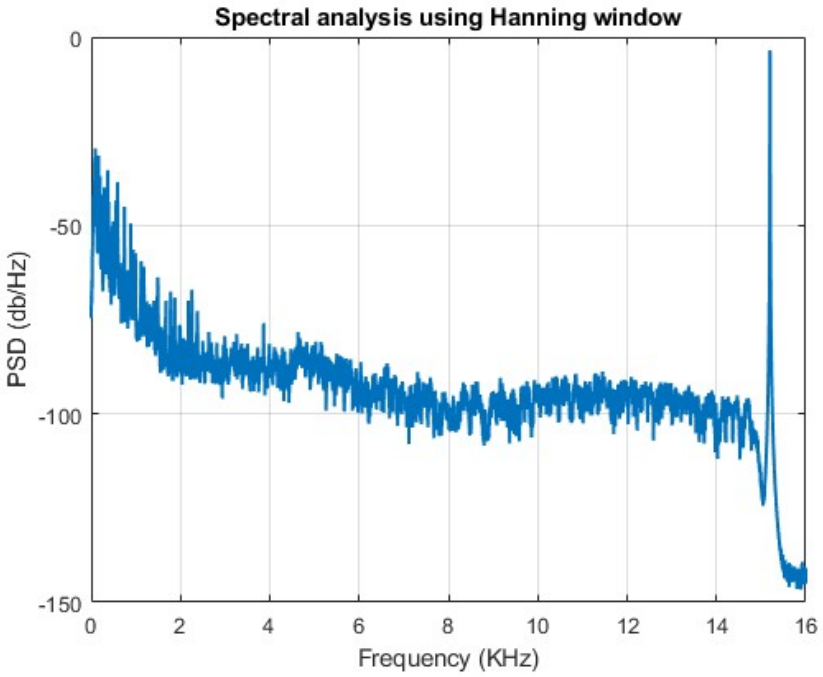

Figure 3: Spectral analysis of signal using a Hanning window

The windows for these examples are plotted in both the time and frequency domain in Figure 4. This plot is achieved by using the following code:

```
wvtool(hann(Nfft), rectwin(Nfft))
```

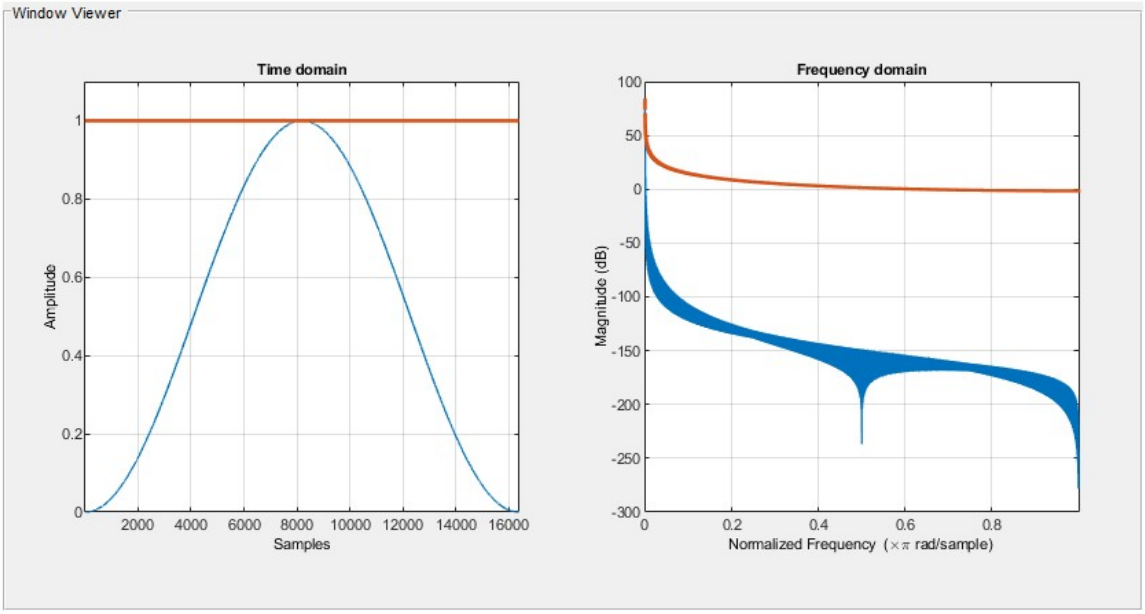

Figure 4: Window comparison of Rect and Hanning windows in time and frequency domain

Now, we ask, how does increasing/decreasing the size of the segments taken before our FFT change the output? This statement is equivalent to changing the FFT size of the segments taken. Before we plot, we can predict that the noise may get reduced if the size of the FFT decreases as we now average a higher number of FFT's since we take more segments from the signal. On the other hand, this means that increasing the size of the FFT reduces the number of FFT's taken and thus may increase the noise, but this is only true if we assert that the window size is also equal to the FFT size. If the window size is held constant, the FFT size increase only means more zero padding.

```
    Nfft1 = 2^10;
    Nfft2 = 2^ceil(log2(length(samples)/5));
```

The result is plotted in the next page in Figures 5 and 6. As we can see, this agrees with our predictions. One thing we notice immediately other than the noise is the resolution of the peaks we have. Both signals suffer from `okay` resolution at 15.2 `Khz`. This is because a higher FFT size, with the same WS does not increase resolution, it simply descritizes more of the DTFT. The resolution we refer to here is the ability to distinguish between closely spaced frequencies. To increase this, we aim for a higher WS in order for the FT to have more samples to determine the frequencies available to a higher resolution. This is shown in 7 and 8. Another way to look at this is that a higher WS decreases the width of the window's lobes, resulting in higher resolution. As we can see though, this is at the expense of more noise because the pwelch takes less averages since we cut the signal into larger pieces for the FFT.

Moving on to the next parameter, we will fix the FFT size to be 1024 samples. When a window size less than the FFT size is used, the `pwelch` function in MATLAB pads the remaining samples till the FFT size is reached. As an example, for an FFT size of 1024, if a window size of 924 is used, the signal is cut into 924 sample segments and each segment is padded with 100 extra zeros. This is essentially reducing the number of samples we are analyzing per segment so there must be greater leakage as covered before. These effects are shown in Figures 9 and 10. Again, we repeat that both figures have an FFT size of 1024, and all other parameters are the same as well.

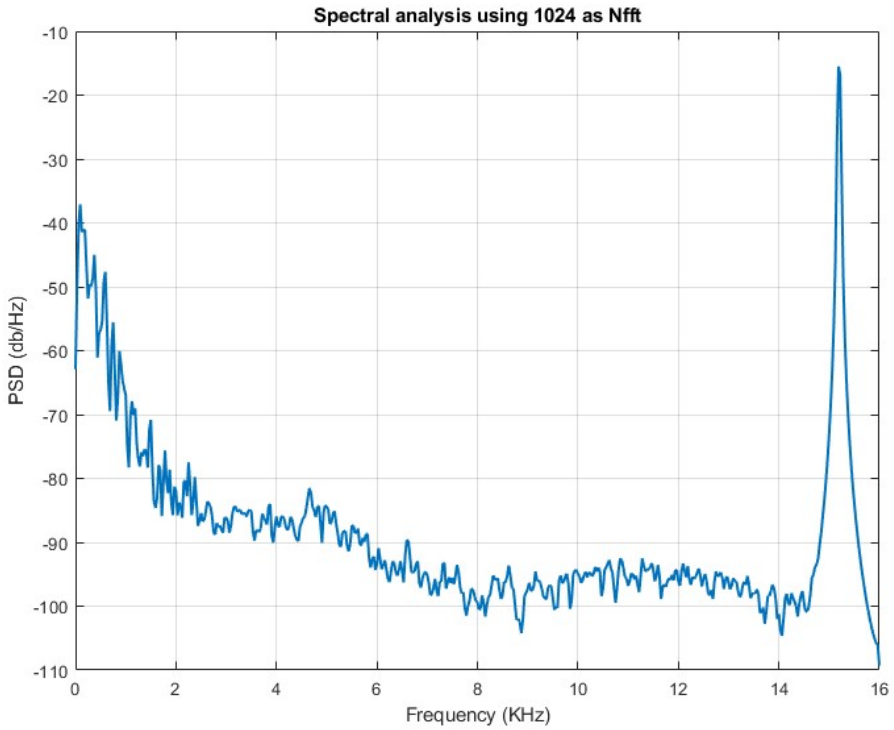

Figure 5: 1024 Samples per FFT (WS = 1024)

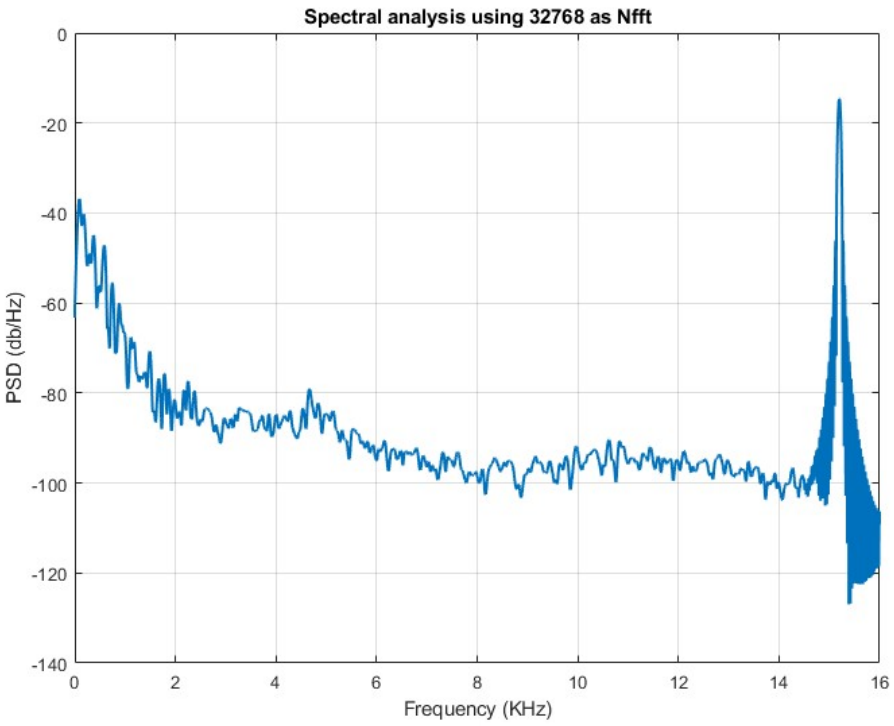

Figure 6: 32768 Samples per FFT (WS = 1024)

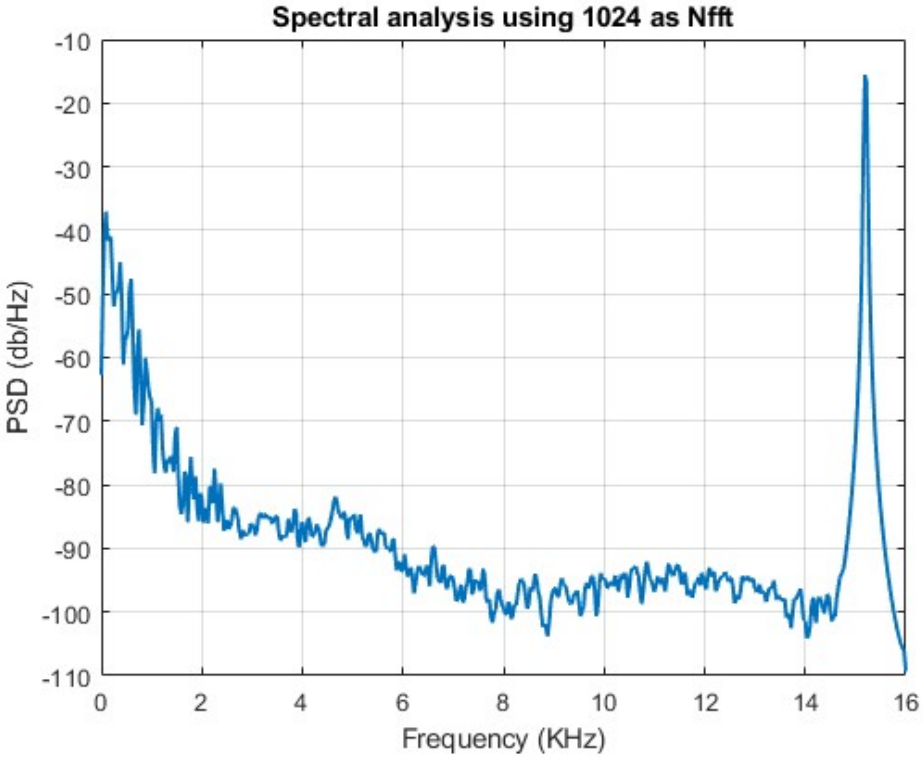

Figure 7: 1024 Samples per FFT (WS = 1024)

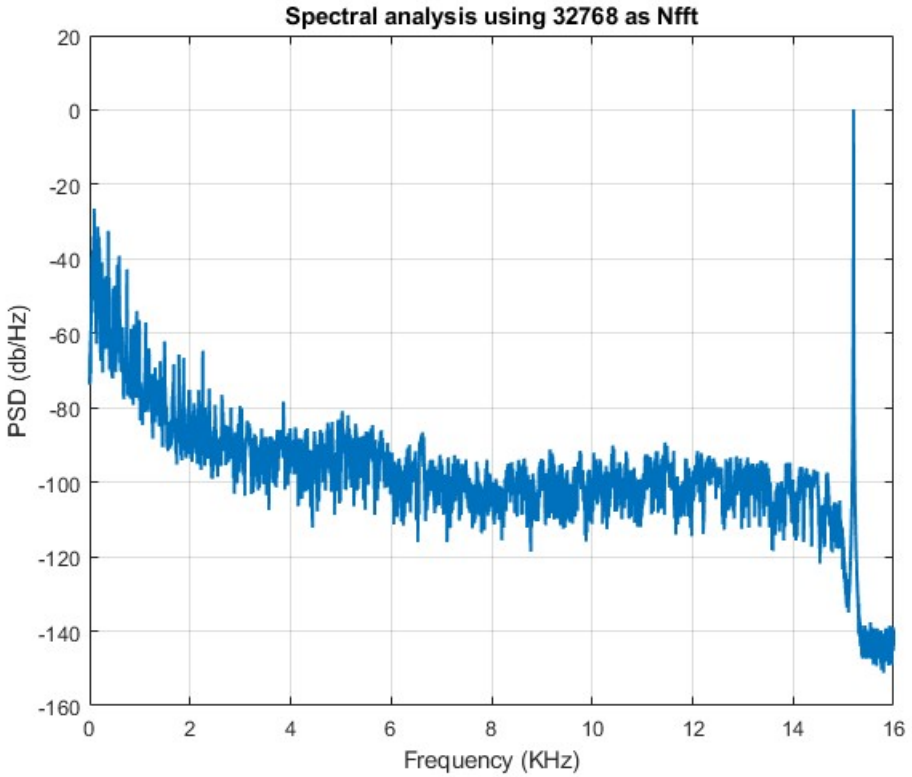

Figure 8: 32768 Samples per FFT (WS = 32768)

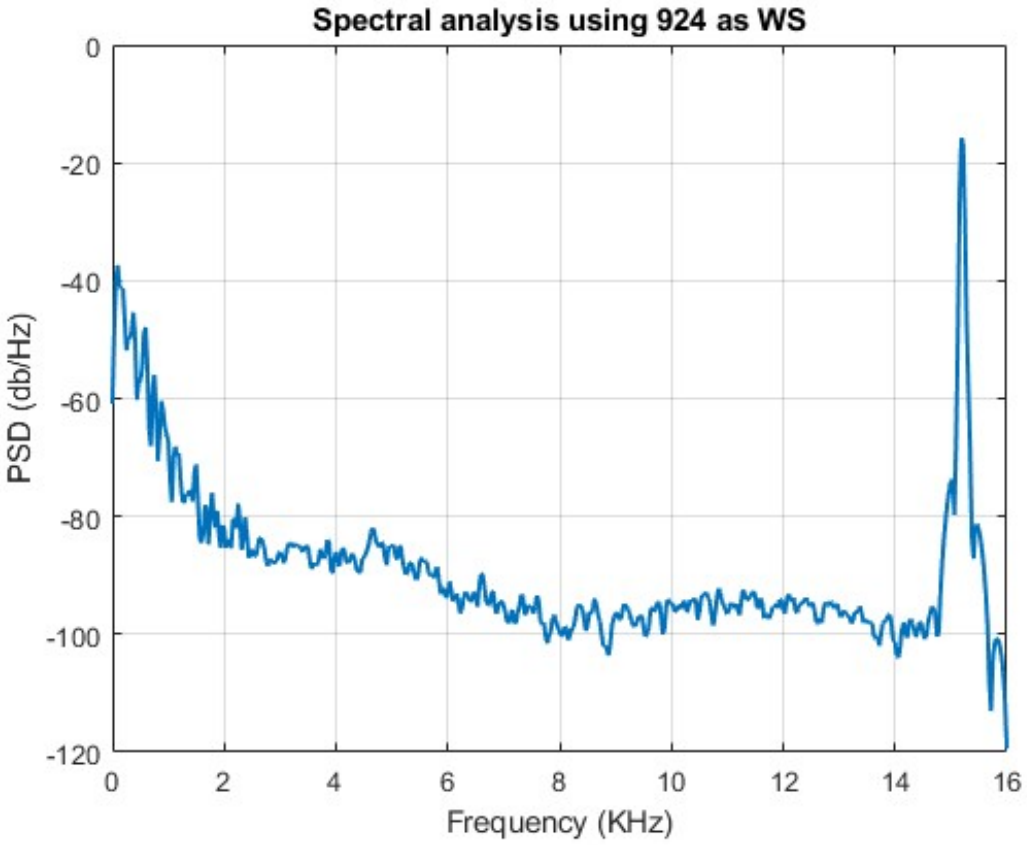

Figure 9: Window size of 924 samples

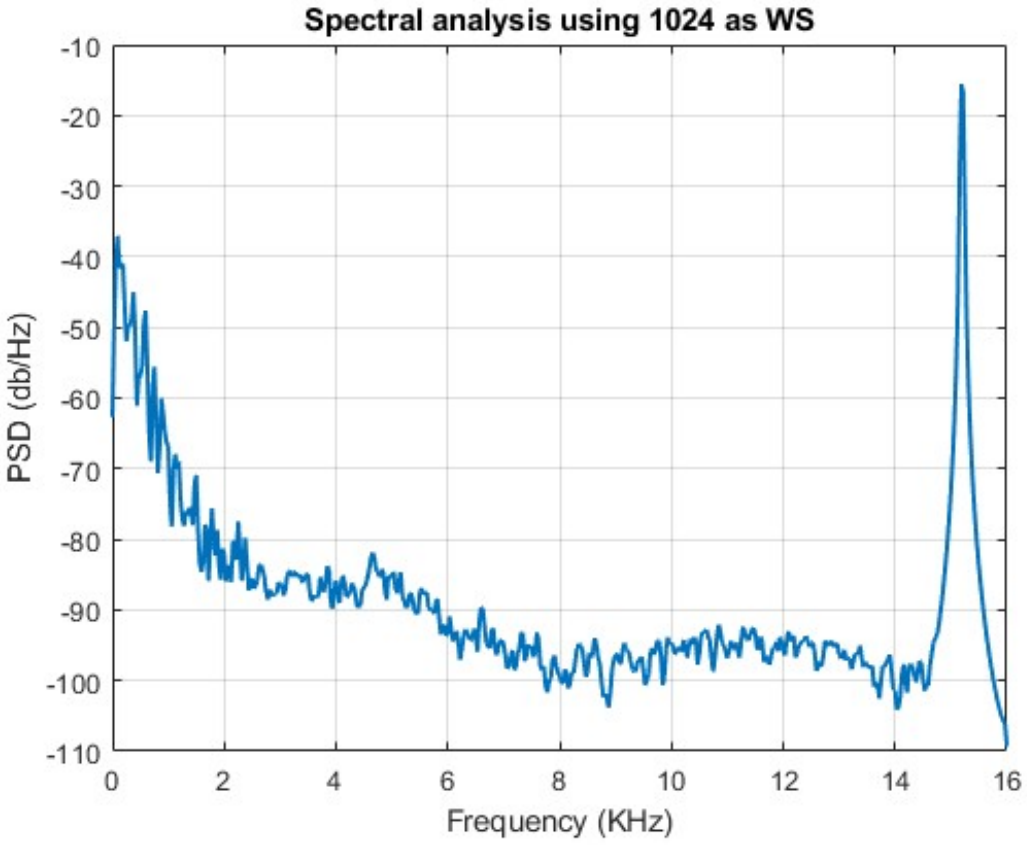

Figure 10: Window size of 1024 samples

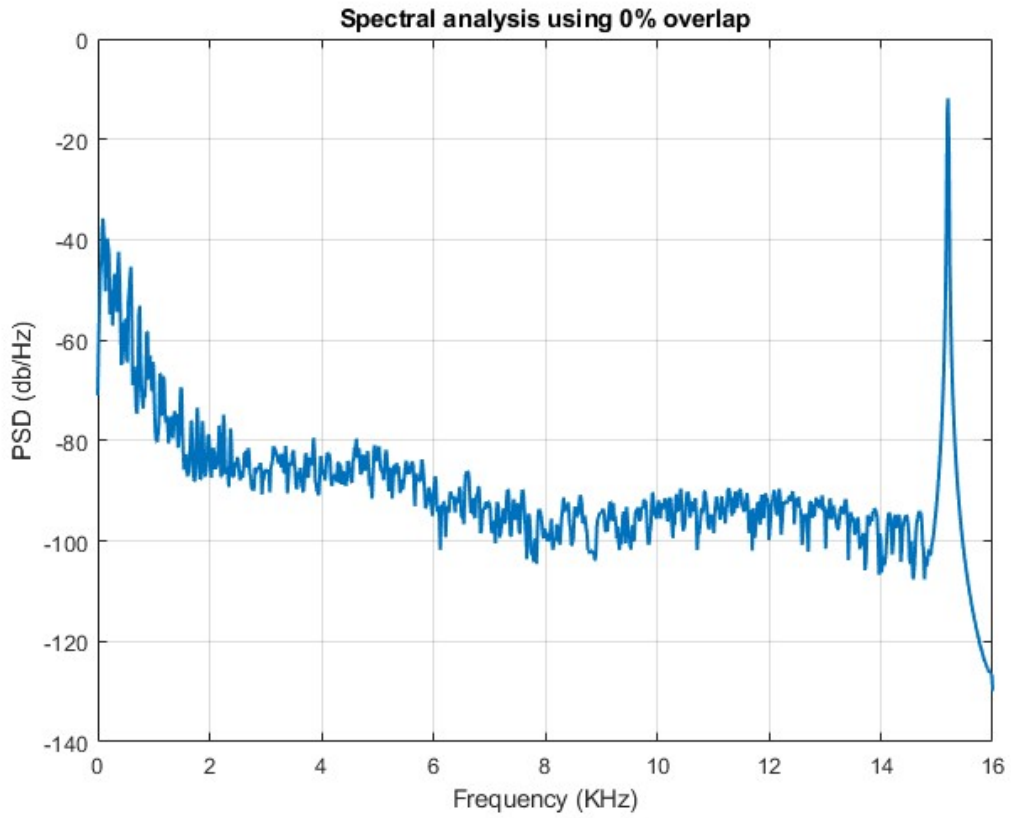

Figure 11: 0% segment overlap

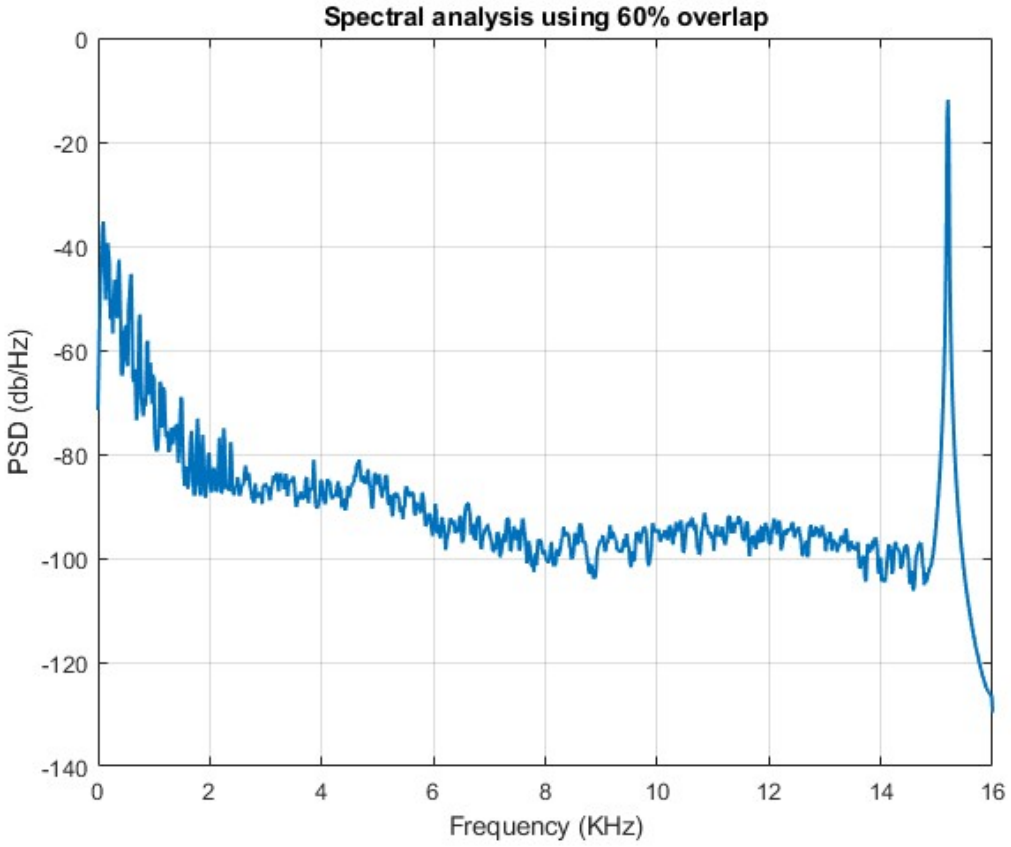

Figure 12: 60% segment overlap

The last parameter we play with and observe its effect is the percentage overlap between the segments. The percentage overlap is simply the percentage by which two segments overlap each other allowing us to take more overlapping segments. If we set this to zero, there is no overlap between the segments and for a signal of size 1000, if the FFT size is 100, then we segment the signal into 10 parts. On the other hand, if it is set to 50% with the same FFT size then we have 19 segments! This greater segmentation allows us to take more FFTs, hence more averages, and so the overall noise of the signal is reduced. Using an FFT size of 2048, and WS = NFFT, we see the - admittedly, small - noise reduction between Figures 11 and 12.

Part 2: Filter Design

Using the myriad of analyses we have above, we can very nicely define our digital filtering problem. The problem is as follows: We have a signal with frequencies ranging up to 15 `KHz` with a very strong signal with frequency 15.2 `KHz` interfering with it. As such, we need to design a low pass filter that removes this interference. Of course, as we see in the spectral analysis, the closer it is to 15 `KHz` the better as more of our original signal will be heard. Immediately, we can define the cutoff frequency and transition region specification. We would like to have a cutoff frequency of 14 `KHz` and a transition region ≤ 0.5 `KHz`. This is chosen because if our transition region or cutoff frequency are any higher, the side lobes of our filter will bleed into the interference and it will be heard, which will not solve our problem. A ripple band ≤ 0.3 dB is also chosen in order to not distort the original audio that much. We also aim for a relatively high stop-band attenuation, the magnitude of the side lobes should not exceed 40 dB. To summarize:

- Transition region ≤ 0.5 `KHz`
- Ripple band ≤ 0.3 `dB`
- Magnitude of side lobes ≤ -40 `dB`

As for our constraints, we imagine that this is a casual problem and so will be carried out by an engineer in an evening on his home computer. This computer has a GTX 1660 TI GPU, which is capable of 170 64 bit GFLOPS. This means that it is capable of $170 \cdot 10^9$ 64 bit floating point operations per second (FLOPS). Assuming we have the filter ready, we want the filteration process with our original signal to take less than a second. We know that our signal contains 96001 samples. Therefore our constraint can be written as:

$$\frac{96001 \cdot \text{filter order}}{170 \cdot 10^9} \leq 1$$

$$\text{filter order} \leq 1770815$$

We obviously see that this constraint will not affect our design this much, as a filter of order 1770815 is incredibly large. We therefore specify an arbitrary constraint of filter order ≤ 400. This arbitrary constraint is also needed as we do not want our signal to be greatly delayed after filtering, as 400 is already comparable to 96001. So, we might even aim for lower order to achieve less delay. A delay of more than 10 *milliseconds* is not accepted. We can calculate the length of the filter that will result in that delay. Our sampling frequency is 32 `KHz`, Therefore:

$$n \cdot \frac{1}{32 \cdot 10^3} \leq 10 \cdot 10^{-3}$$

$$n \leq 320$$

As such, we aim for a filter with order ≤ 320. Now we begin to explore possible filters that meet these constraints and specifications. We showcase 3 filters that easily pass the order constraint and showcase how they meet our specifications. All three are Finite Impulse Response (FIR) filters. The filters we investigate are: a Blackman windowed filter with order 300, a Hanning windowed filter with order 250, and an equiripple filter with order 270. The frequency response of each filter respectively is plotted in Figures 13 & 14 & 15. We develop these filters using MATLAB's filter design tool, which basically runs the code below. The code for the other filters is attached in the appendices.

```
function Hd = han250
Fs = 32;  % Sampling Frequency
N    = 250;       % Order
Fc   = 14.2;      % Cutoff Frequency
flag = 'scale';  % Sampling Flag
% Create the window vector for the design algorithm.
win = hann(N+1);

% Calculate the coefficients using the FIR1 function.
b  = fir1(N, Fc/(Fs/2), 'low', win, flag);
Hd = dfilt.dffir(b);
end
```

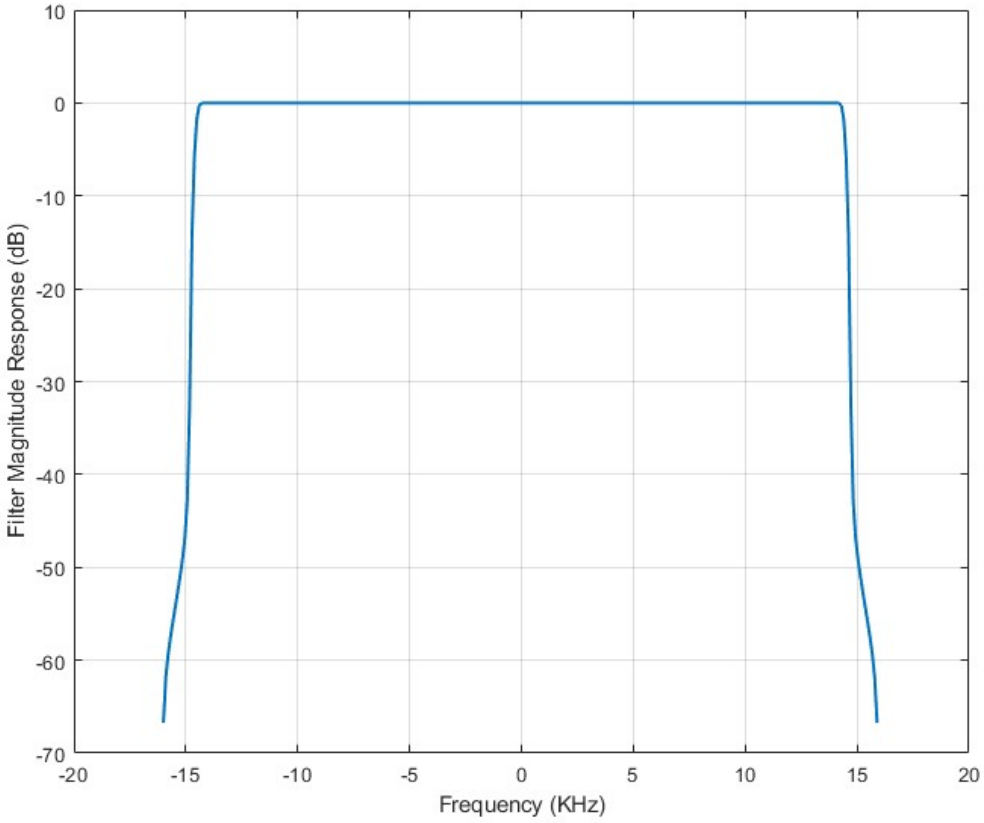

Figure 13: Frequency response of Blackman filter of order 300

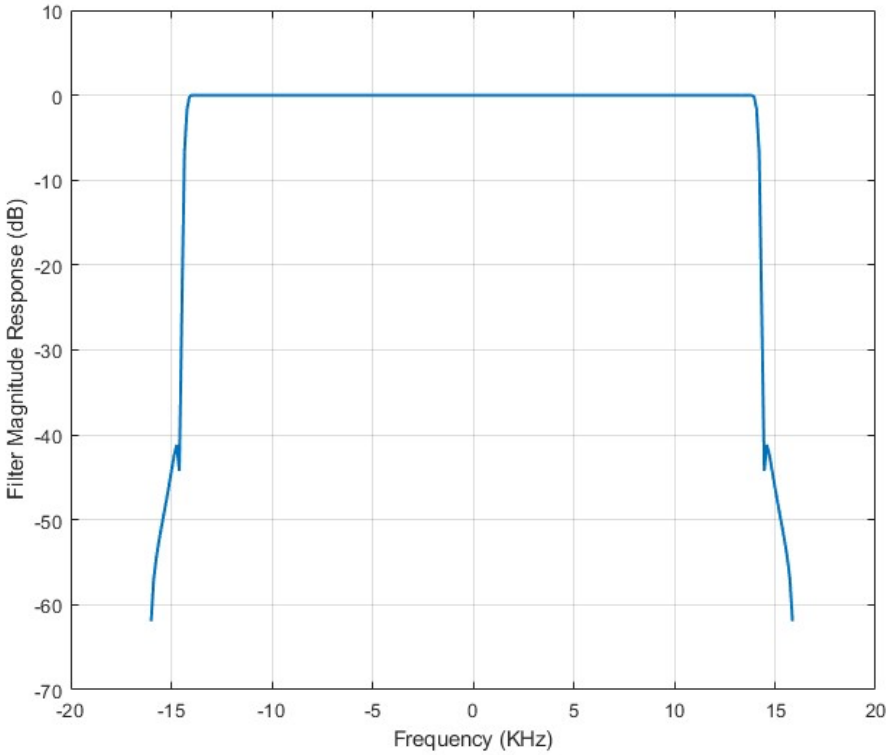

Figure 14: Frequency response of Hanning filter of order 250

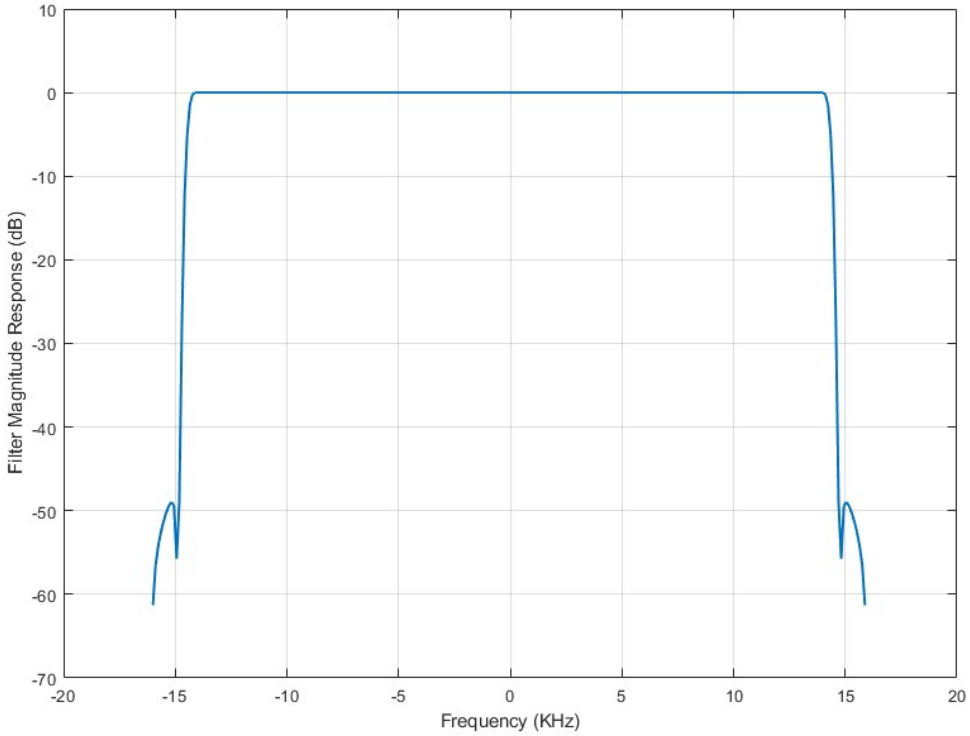

Figure 15: Frequency response of Equiripple filter of order 270

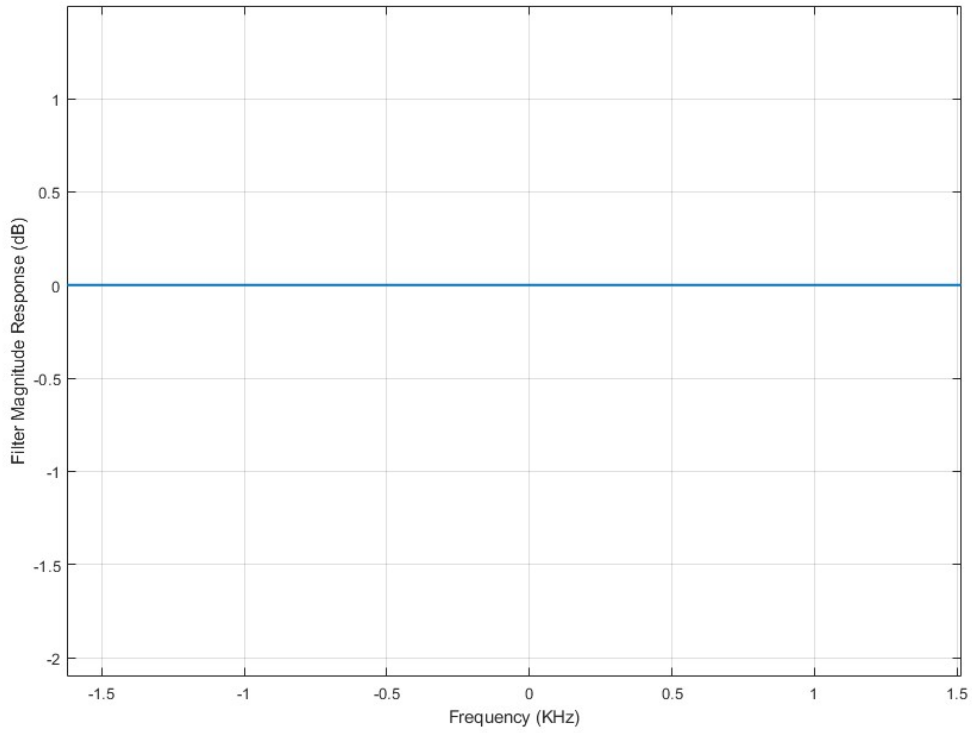

Figure 16: Blackman of order 300 zoomed in on ripples

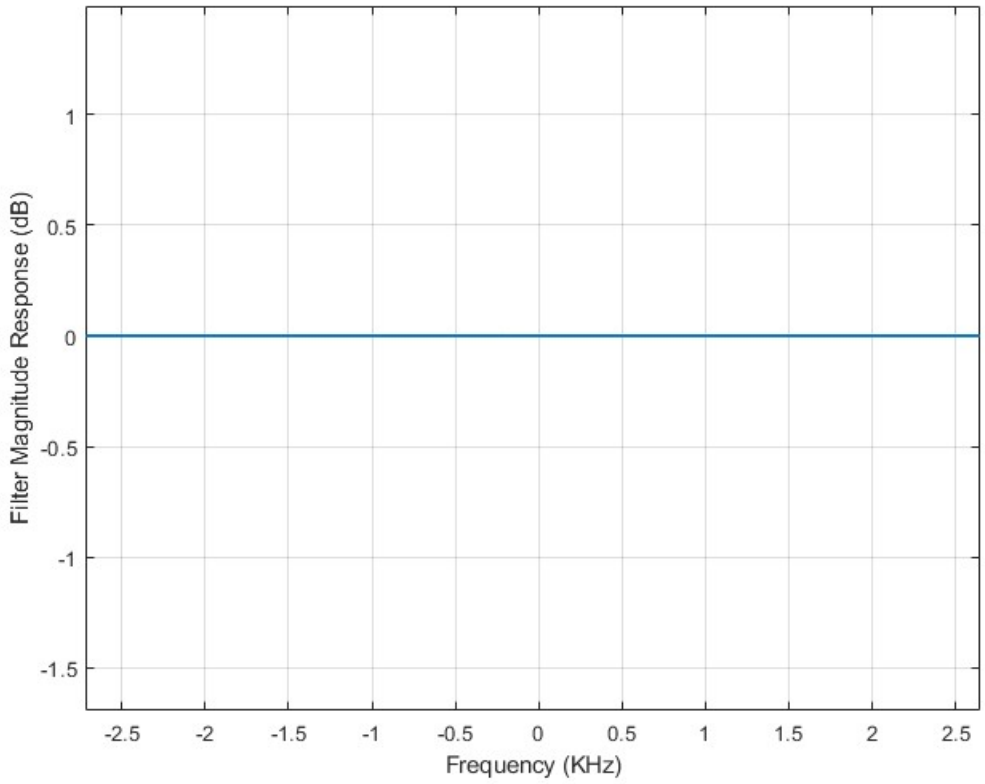

Figure 17: Hanning filter of order 250 zoomed in on ripples

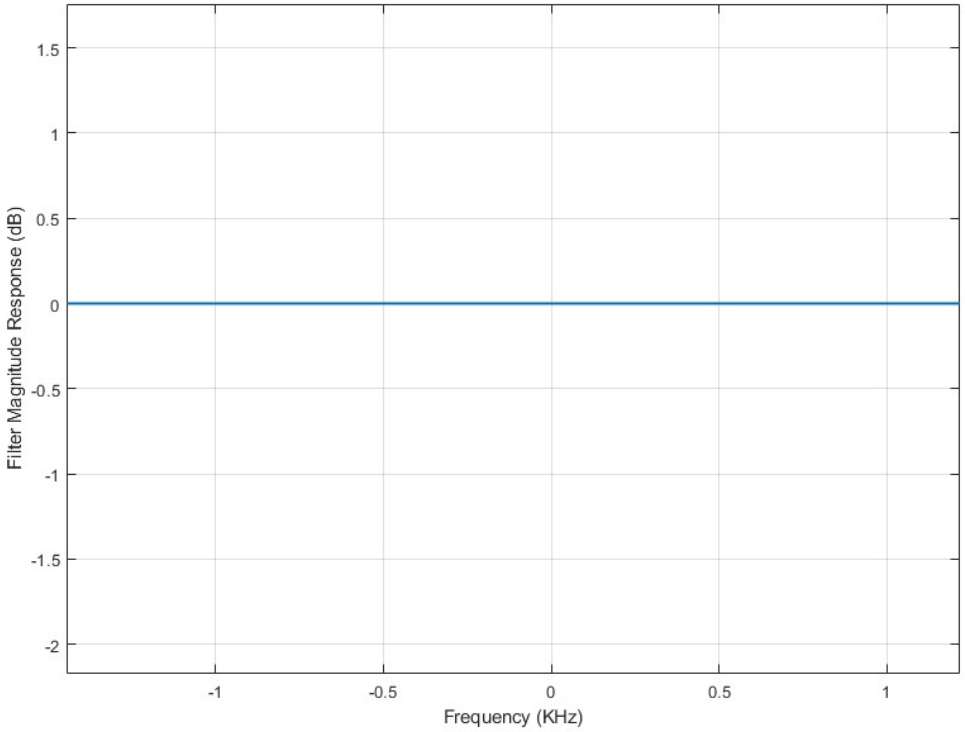

Figure 18: Equiripple filter of order 270 zoomed in on ripples

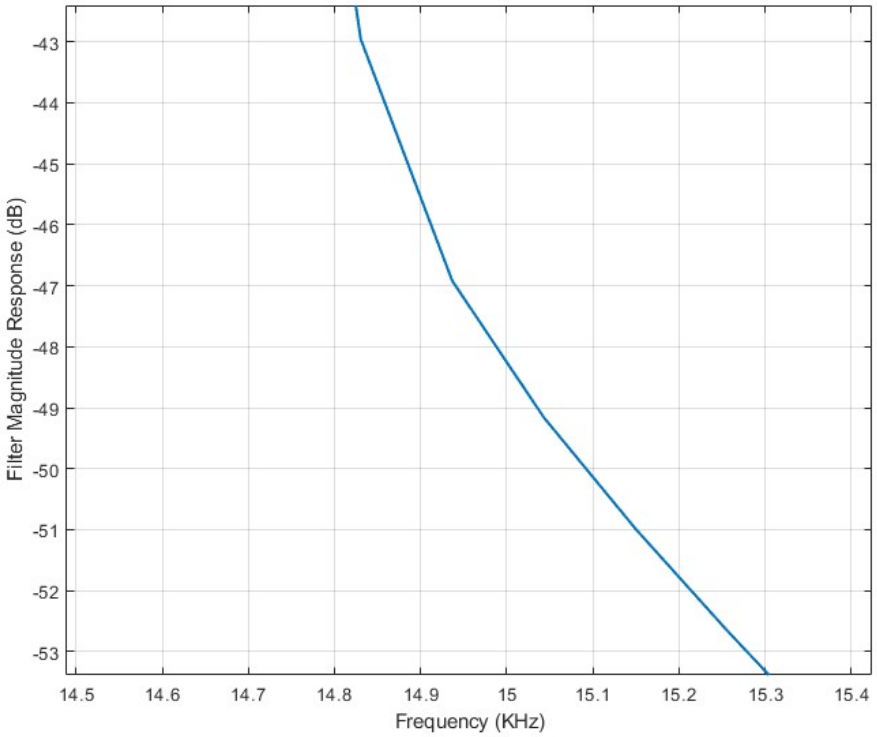

Figure 19: Blackman of order 300 zoomed in on transition

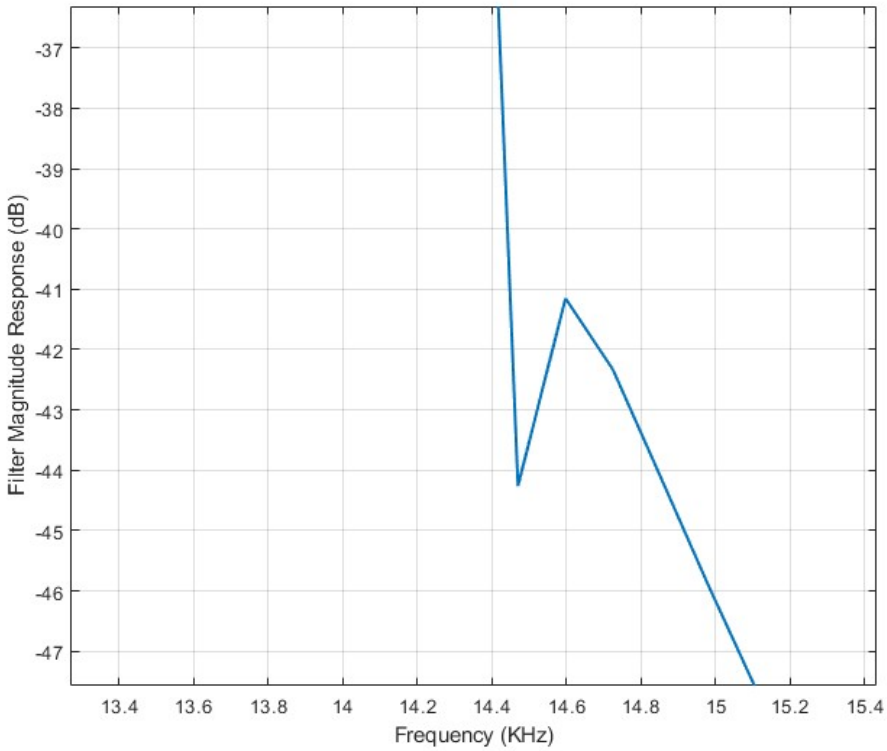

Figure 20: Hanning filter of order 250 zoomed in on transition

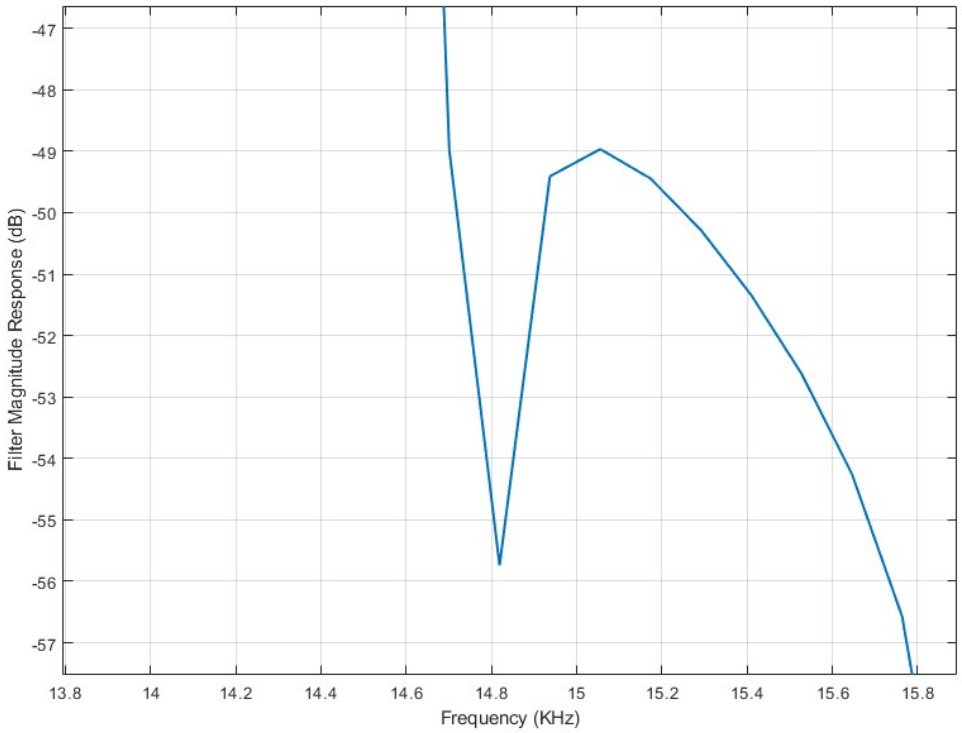

Figure 21: Equiripple filter of order 270 zoomed in on transition

In Figures 16 & 17 & 18 we see how the ripples of our filters are almost nonexistent, and they all meet our specification of ripples being ≤ 0.5 dB. This is due to the high order of our filters, which makes the pass-band ripples extremely low. In Figures 19 & 20 & 21 we also see that the transition region is ≤ 0.5 KHz, as per the specification. The blackman filter is set to have $\omega_c = 14.5$ KHz and we see that it transitions to a magnitude of ≤ -40 dB way before 15 KHz as per our specification. The hanning filter is set to have $\omega_c = 14.2$ KHz and we see that it takes only 0.2 KHz to transition. The Equiripple filter is set to have a stopband frequency of 14.7 KHz and it finishes transitioning at 14.8 KHz which is incredibly quick relative to our specification. We also see that all the side lobes are ≤ -40 dB. We now organize these thoughts in a decision analysis matrix.

Filter	**Criteria (Weight)**					**Total Score**
	Transition region (3)	**Ripple band** (1)	**Stopband attenuation** (4)	**Complexity (n coeff.)** (4)	**Signal recovery** (3)	
Blackman	4	5	4	3	5	60
Hanning	5	5	4	4	4	64
Equiripple	5	5	5	4	5	**71**

Table 1: Decision analysis of proposed filters

All the scores are given out of 5. The weights for each criterion is enclosed in brackets after each criterion's name. We see that the highest score is given to the Equiripple's, and so we use that to filter our interfered signal. We do so by running the following code: filter = equiripple270().numerator;

```
filtered_samples = conv(samples_interfered, filter);
pause(3.5); sound(samples_interfered, Fs); pause(3.5)
sound(filtered_samples, Fs);

Nfft = 2^10;
percentage_overlap = 0.5;
WS = Nfft;
WT = 'Hanning';
plot_spectrum(filtered_samples, Fs, Nfft, WS, WT, percentage_overlap
    )
```

This results in the plot in Figure 22. Mission accomplished! we have now filtered our signal of that loud interference that was ruining it and can listen to fayrouz happily.

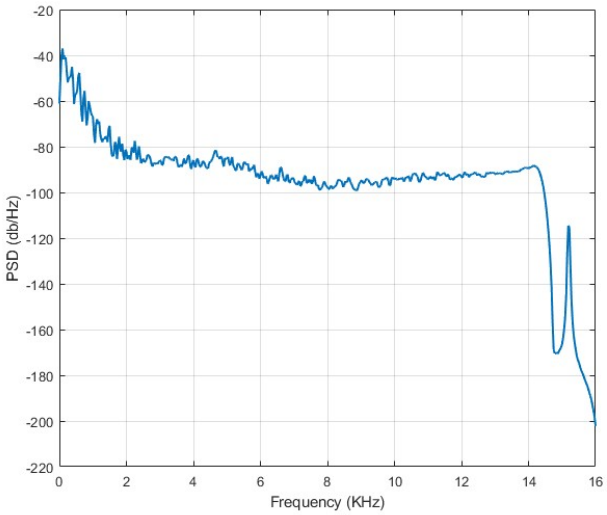

Figure 22: Power spectrum density of filtered samples

Appendices

```
function Hd = blackman300
% FIR Window Lowpass filter designed using the FIR1 function.

% All frequency values are in kHz.
Fs = 32;  % Sampling Frequency

N    = 300;       % Order
Fc   = 14.5;      % Cutoff Frequency
flag = 'scale';  % Sampling Flag

% Create the window vector for the design algorithm.
win = blackman(N+1);

% Calculate the coefficients using the FIR1 function.
b  = fir1(N, Fc/(Fs/2), 'low', win, flag);
Hd = dfilt.dffir(b);
end
```

```
function Hd = equiripple270
    % All frequency values are in kHz.
    Fs = 32;  % Sampling Frequency

    N     = 270;    % Order
    Fpass = 14;     % Passband Frequency
    Fstop = 14.7;   % Stopband Frequency
    Wpass = 1;      % Passband Weight
    Wstop = 1;      % Stopband Weight
    dens  = 20;     % Density Factor

    % Calculate the coefficients using the FIRPM function.
    b  = firpm(N, [0 Fpass Fstop Fs/2]/(Fs/2), [1 1 0 0], [Wpass
        Wstop], ...
                {dens});
    Hd = dfilt.dffir(b);
end
```