Audio Manipulations

Rishi Nandha V - EE21B111

EE6311 - Multirate Digital Signal Processing

October 20, 2023

All codes used in these Experiments can be found here.

1 Magnitude Spectrums

The following code can be used to visualize the Magnitude Spectrum of any given slice of audio. Since the audio is real valued we only visualize the positive half of the spectrum for convenience. Note here that a Semi-Logarithmic plot is taken because it's more intuitive.

```python
import wave
import numpy as np
import matplotlib.pyplot as plt

audio_file = wave.open("music16khz.wav", 'rb')

channels = audio_file.getnchannels()
bit_depth = 8*audio_file.getsampwidth()
sampling_rate = int(audio_file.getframerate())
length = audio_file.getnframes()/sampling_rate

audio = audio_file.readframes(audio_file.getnframes())
audio = np.frombuffer(audio, dtype=np.int16)

print(f"No. of Channels: {channels}\nBit Depth: {bit_depth} bits")
print(f"Sampling Rate: {sampling_rate/1000} kHz\nLength: {length} sec")

y = np.absolute(np.fft.fft(audio))
y_dB = 20*np.log10(y)
halfbracket = int((length*sampling_rate)//2+1)
x = np.linspace(0,sampling_rate//2,halfbracket)

plt.plot(x,y_dB[:halfbracket])
plt.xlabel("Frequencies")
plt.ylabel("FFT Coefficients (in dB)")
plt.show()
```

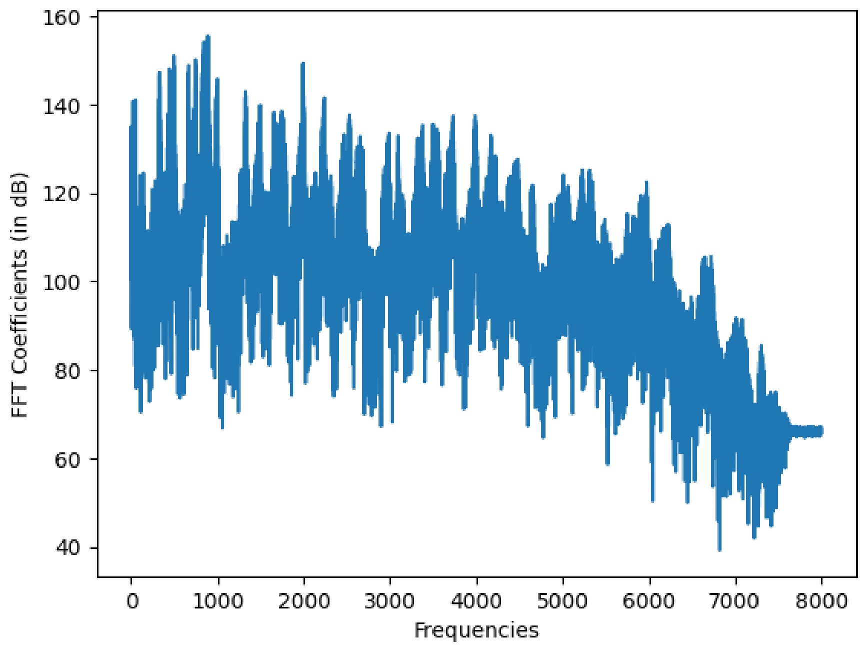

Figure 1: FFT of music16kHz.wav as a whole

But note that this has no physical significance in the way we hear audio because we took the entire audio file and ran an FFT on it. In reality, the magnitude spectrum we hear and perceive also varies with time. Thus a better visualization can be obtained by plotting a Spectrogram.

A Spectrogram is a contour plot of magnitude spectrums evaluated with a sliding window of short slices of the audio and plotted against time. The number of samples in each slice is known as the FFT Size. The frequency axis is on a logarithmic scale to make it more intuitive, since our perception of pitch is logarithmic to base 2. The following code in addition to the previous snippet does the plotting:

```
def openaudio(filepath):
    audio_file = wave.open(filepath, 'rb')
    channels = audio_file.getnchannels()
    bit_depth = 8*audio_file.getsampwidth()
    sampling_rate = int(audio_file.getframerate())
    length = audio_file.getnframes()/sampling_rate
    audio = audio_file.readframes(audio_file.getnframes())
    audio = np.frombuffer(audio, dtype=np.int16)
    print(f"No. of Channels: {channels}\nBit Depth: {bit_depth} bits")
    print(f"Sampling Rate: {sampling_rate/1000} kHz\nLength: {length} sec")

    fft_size = int(sampling_rate//20)
    half_size = int(fft_size//2)+1
    sliced_audio = np.lib.stride_tricks.sliding_window_view(audio, window_
  ↪shape=(fft_size,))
    spectrum = np.absolute(np.fft.fft(sliced_audio))

    X= (np.arange(0,spectrum.shape[0],1))/sampling_rate
    Y= np.log2((np.linspace(0,0.5, half_size)[1:])*sampling_rate)
    Z = 20*np.log10(np.transpose(spectrum[:,1:half_size]))
    return (X,Y,Z, sampling_rate, audio)

def spectrum2(X,Y,Z, savepath):
    _, ax = plt.subplots(1,1)
    cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", ["black",
 ↪"orange"])
    norm=plt.Normalize(np.std(np.power(10,Z/50))*1.5,np.max(np.power(10,Z/
 ↪50))/1.5,clip=True)

    ax.contourf(X, Y, np.power(10,Z/50), cmap = cmap, norm=norm)

    ax.set_title('Discrete Spectrogram')
    ax.set_xlabel('Time (in Seconds)')
    ax.set_ylabel('Frequency (Logarithmic to Base 2)')
    plt.savefig(savepath)
    plt.show()
```

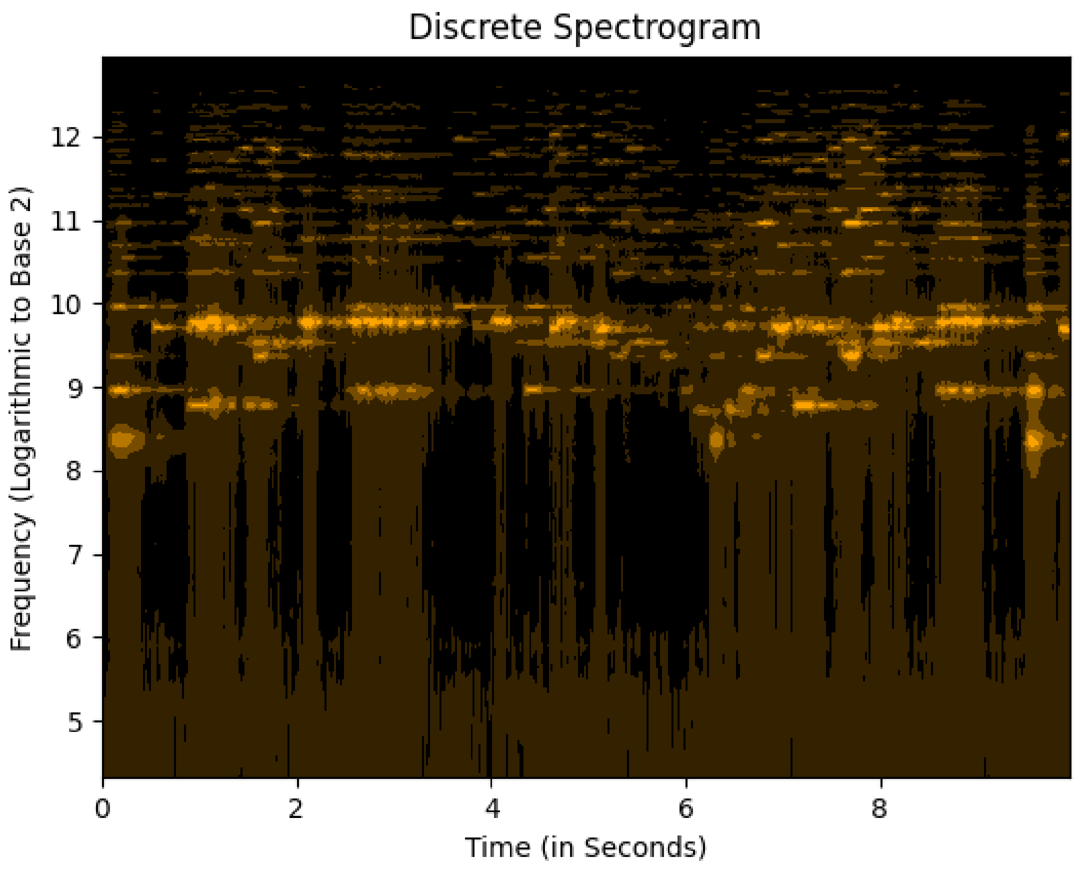

Figure 2: Spectrogram of music16kHz.wav

This is clearly a much more intuitive and more physically significant representation of our Audio's frequency information. We can do the same for our speech8kHz.wav too. The above functions were appropriately made into generalized functions and used as shown below:

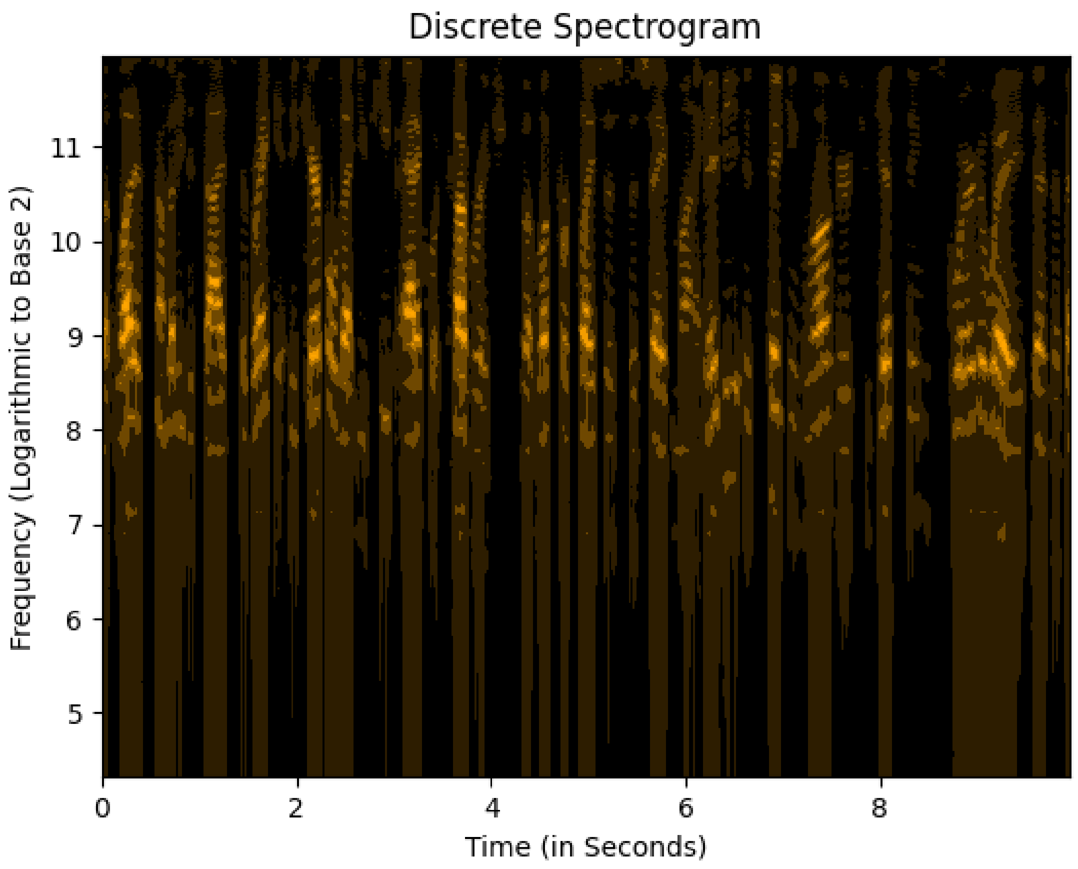

Figure 3: Spectrogram of speech8kHz.wav

2 Downsampling by 2 without AA Filter

Downsampling without an AA Filter can be done appropriately with the following code:

```
audio_1_d = audio_1[::2]
samples = audio_1.size

with wave.open('music8khz.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate(sampling_rate_1//2)
    wav_file.setnframes(samples)
    wav_file.writeframes(audio_1_d.tobytes())
```

This down-sampled audio sounds muffled owing to the fact that at this smaller sampling rate, the higher frequency information is lost. It also sounds distorted because of Aliasing. These artefacts due to distortion is a lot more prominent in the given Speech than Music. The reason is simply that the frequency content and the sampling rate/2 are closer to each other in the given speech audio than the music audio.

Both of these effect are better visualized when we plot a magnitude spectrum of a particular time instead of the entire spectrogram.

The below two figures are the same timestamp from the original and downsampled versions of 'music16khz.wav'. Notice how the downsampled version contains information only until 8kHz and loses the information above that.

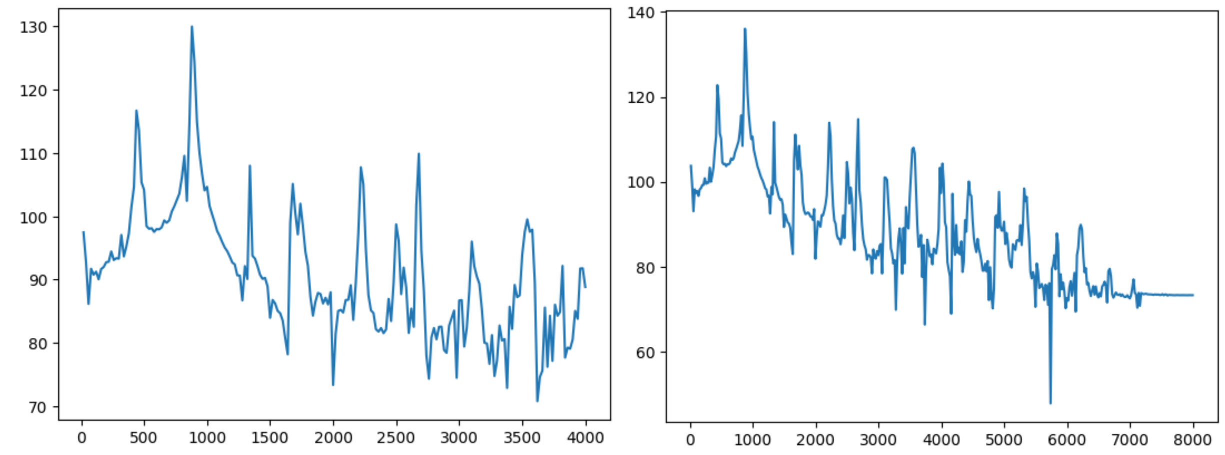

Figure 4: Comparison of Downsampled Music

The below two figures are the same timestamp from the original and downsampled versions of 'speech8kHz'. Notice how the down-sampled version has some very prominent spurs caused by Aliasing that aren't present in the original and also a much smaller resolution between the peak and the floor

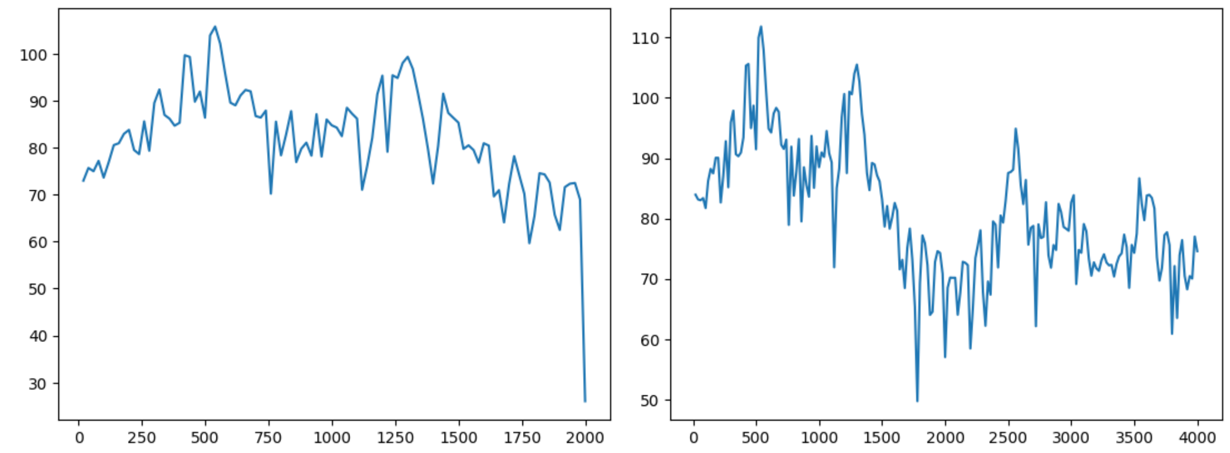

Figure 5: Comparison of Downsampled Speech

3 AA - Filtering using an Equiripple LPF

An Equiripple Low Pass filter with $\omega_p = 0.45\pi$, $\delta_p = 0.02$, $\delta_s = 0.02$, $\omega_s = 0.55\pi$ and an order of 100 can be realized using the following code:

```
from scipy.signal import remez, freqz, lfilter

omega_p = 0.45 * 0.5
omega_s = 0.55 * 0.5
delta_p = 0.02
delta_s = 0.02

filter_coeffs = remez(101, [0, omega_p, omega_s, 0.5], [1,0], [delta_p,␣
 ↪delta_s])

w, h = freqz(filter_coeffs, worN=8000)
```

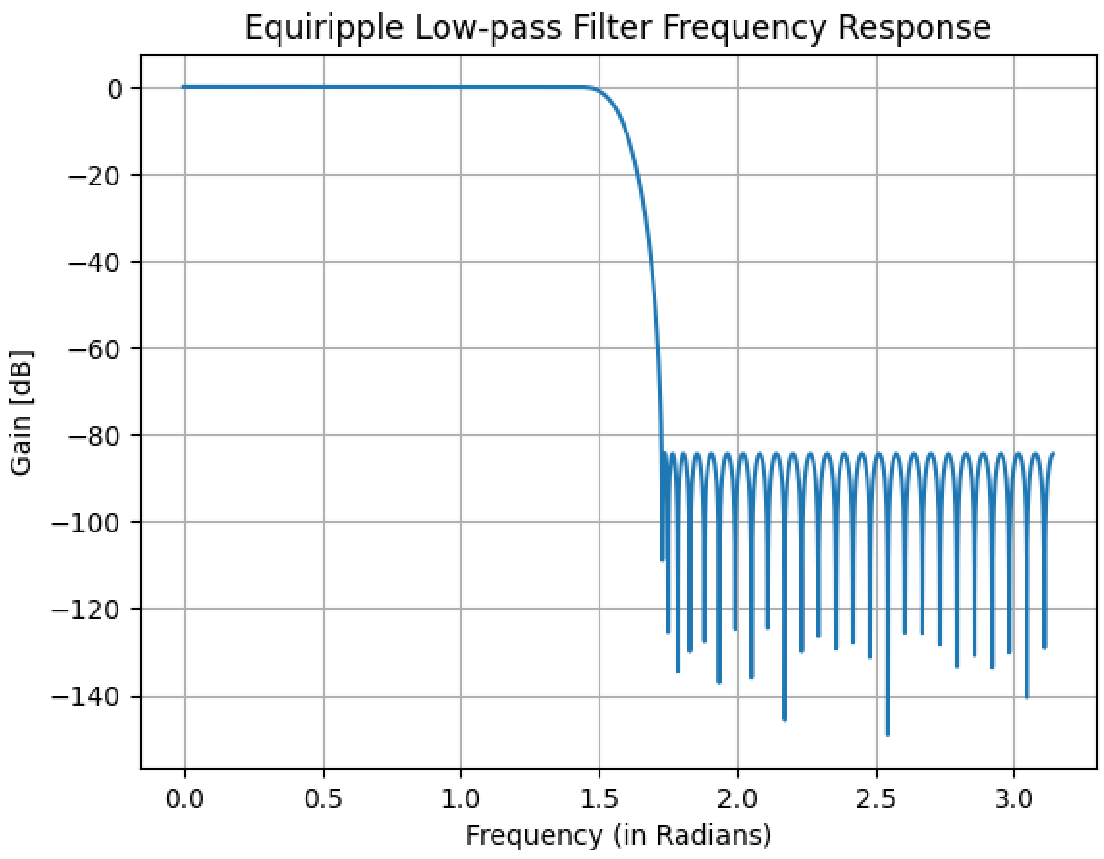

Figure 6: Frequency Response of Equiripple LPF

The filter can be applied on the down-sampled audios using the following code:

```
from scipy.signal import lfilter

audio_1_f = lfilter(filter_coeffs, 1.0, audio_1)[::2].astype(np.uint16)
samples = audio_1_f.size

with wave.open('music8khz_filtered.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate(sampling_rate_1//2)
    wav_file.setnframes(samples)
    wav_file.writeframes(audio_1_f.tobytes())

X_6,Y_6,Z_6, sampling_rate_6, audio_1_f = openaudio("music8khz_filtered.wav")
spectrum(X_6,Y_6,Z_6, 65, 120,'spectrogram_music8khz_filtered.png')
```

Below are the spectrograms of down-sampled music with AA filter and down-sampled without AA filter respectively

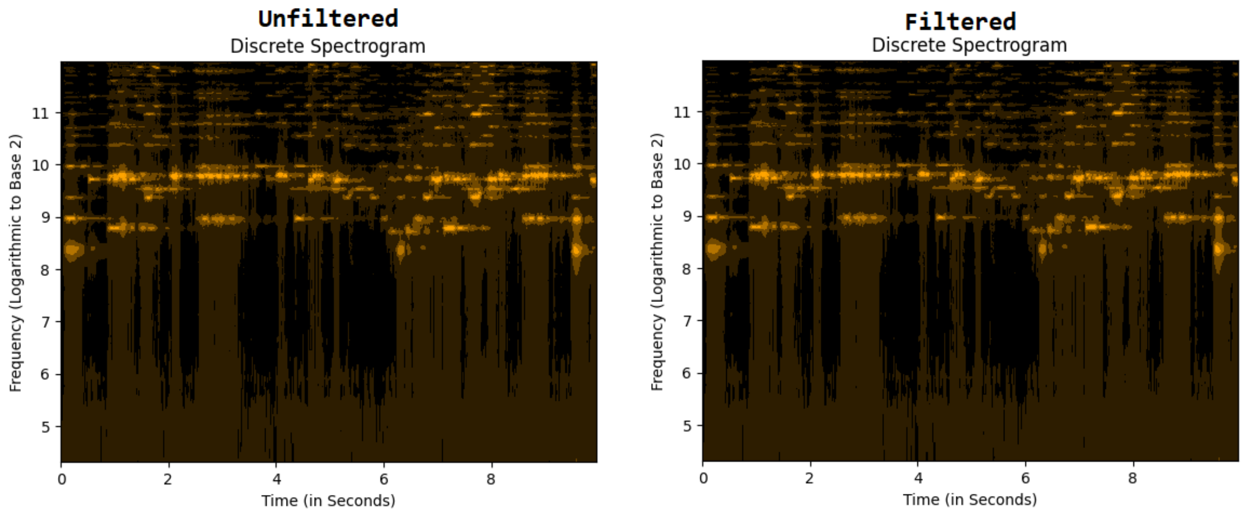

Figure 7: Comparison of with & without AA Filter

The high frequency information is still not recoverable so it still sounds muffled and dull than the original. But the AA-filter definitely has reduced the distortion audibly. Since the given Speech was in-fact more severely Aliasing than Music, we can expect the spectrograms of speech and playback of speech after AA - Filtering to improve more visibly and audibly:

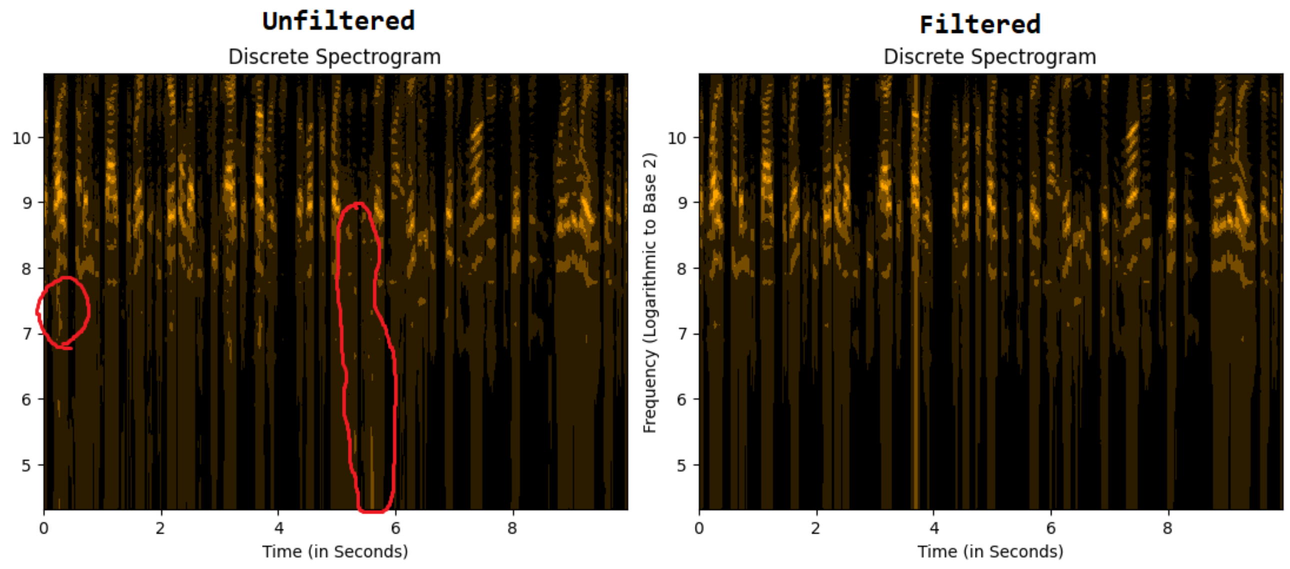

Figure 8: Comparison of with & without AA Filter

Here the impact of the AA-Fitler is much more apparent. Two spots have been circles which are spots caused by Aliasing in Unfiltered Down-sampling which are not occuring in Filtered Down-sampling. In the playback too, the audio sounds just as muffled as before, but much less distorted than before once the filter has been added.

4 Equiripple LPF with Cutoff at $\frac{\pi}{4}$

Same code as the previous LPF is used with different start and stop band frequencies.

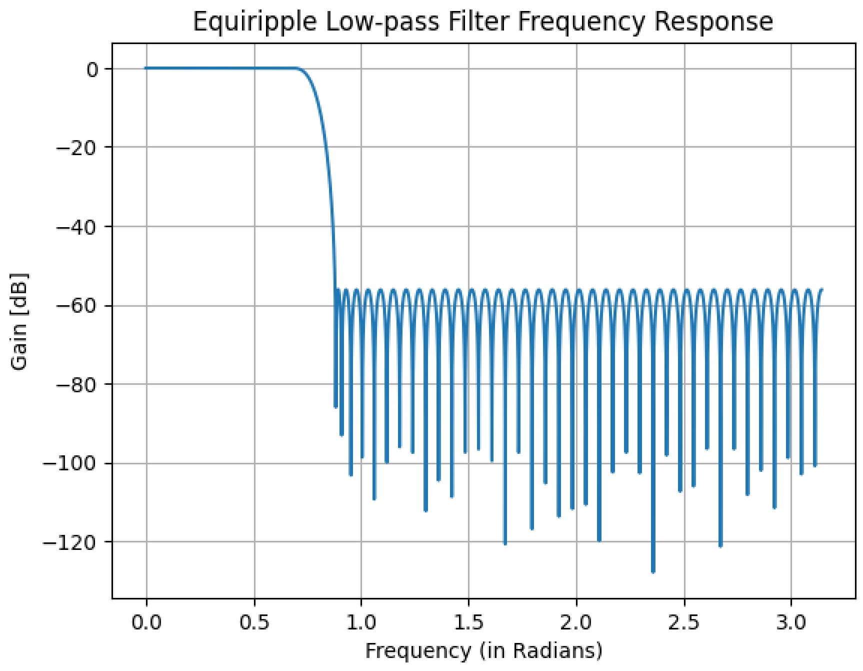

Figure 9: Equiripple LPF 2

5 Up-sampling by 3 without Filtering

The following code can be used to Upsample and Save an Audio File. Note here that a large audio can't be written in one step without consuming a lot of RAM. Thus the writing is done in a loop in chunks of data.

```
audio_1_u = np.zeros(audio_1.size * 3, dtype=np.uint16)
audio_1_u[::3] = audio_1
samples = audio_1_u.size
# Open a WAV file for writing
with wave.open('music48khz.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate(sampling_rate_1*3)
    wav_file.setnframes(samples)
    wav_file.writeframes(b'')
    print(len(np.array_split(audio_1_u,8000)))
    for i in np.array_split(audio_1_u,8000):
        wav_file.writeframes(i.tobytes())
X_8,Y_8,Z_8, sampling_rate_8, audio_1_u = openaudio("music48khz.wav")
```

The playback of the up-sampled versions have a very prominent eeriness. Almost as if there is a high pitched noise following the contour of the the original audio but higher in pitch. This eeriness can be seen below in the spectrogram of the given music up-sampled by 3.

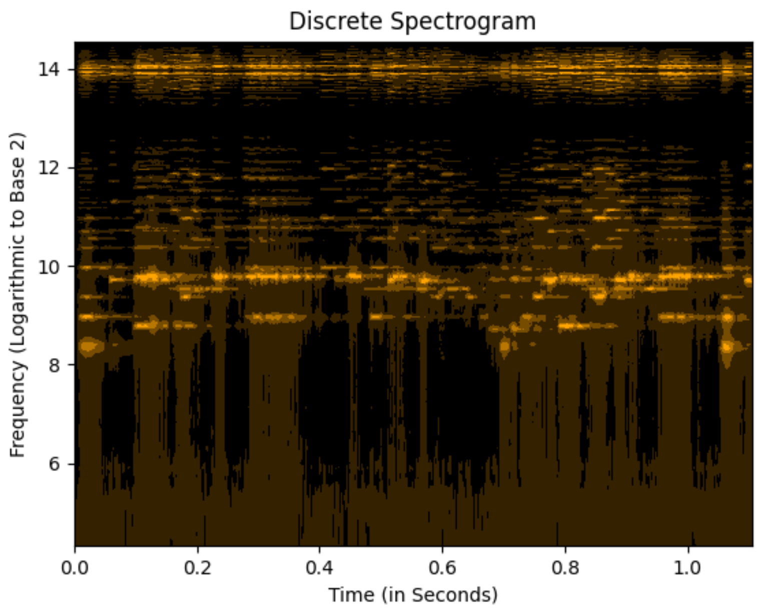

Figure 10: Spectrogram after upsampling Music by 3

The higher frequencies are essentially a copy of the frequencies centred around original. To see this, below is a comparison of original and up-scaled versions' magnitude spectrums of the given speech at a particular time instant. The speech sounds a lot more robotized and as if it as some parallel (non-harmonic) higher frequency components.

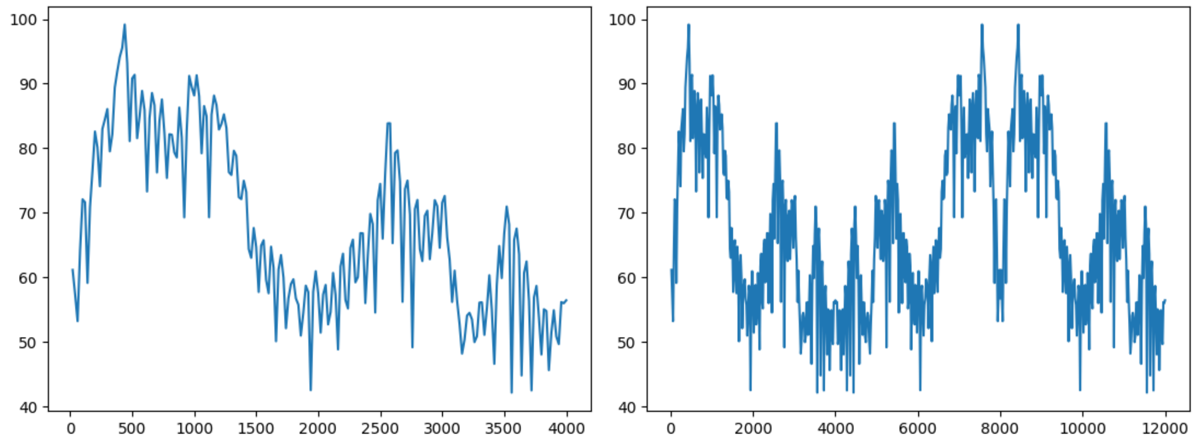

Figure 11: Roll-over effect after up-sampling Speech by 3

6 Up-sampling by 3, AA-Filtering, and Down-sampling by 4

The following code converts the audio files appropriately:

```
audio_1_ud = lfilter(filter_coeffs_2, 1.0, audio_1_u)[::4].astype(np.uint16)
samples = audio_1_ud.size
print(samples)

with wave.open('music_12khz.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate((sampling_rate_1*3)//4)
    wav_file.setnframes(samples)
    wav_file.writeframes(audio_1_ud.tobytes())
X_10,Y_10,Z_10, sampling_rate_10, audio_1_ud = openaudio("music_12khz.wav")
```

The playback now sounds a lot better. Although there is some amount of high frequency information lost, this isn't as noticeable as with down sampling by 2. The LPF Filter has both taken care of interpolating and filtering out the high frequency copy of the spectrum and also anti-aliasing by filtering out frequencies more than the new Nyquist frequency. So the audio is not distorted.

Here are two snapshots of the magnitude spectrum from Music and Speech.

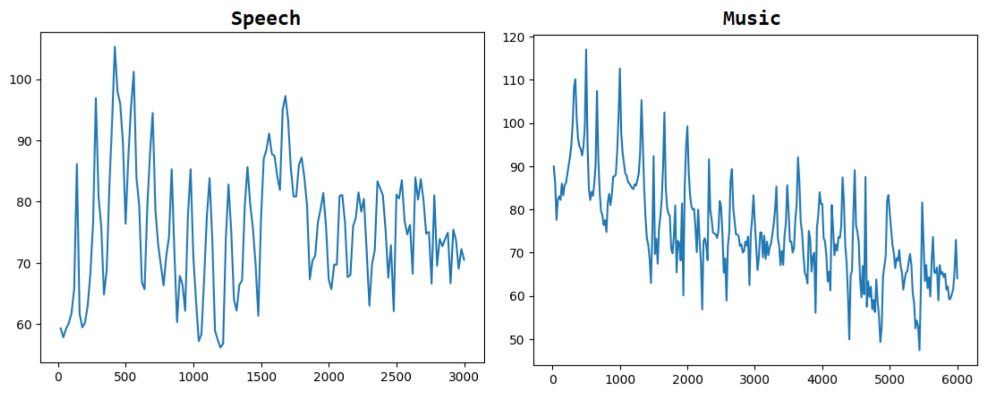

Figure 12: Up-sampling by 3, AA-Filtering, and Down-sampling by 4

We can compare the original spectrograms against these. Notice how they are pretty close to each other, which is why the playback sounds a lot better than any of the other cases we tried so far.

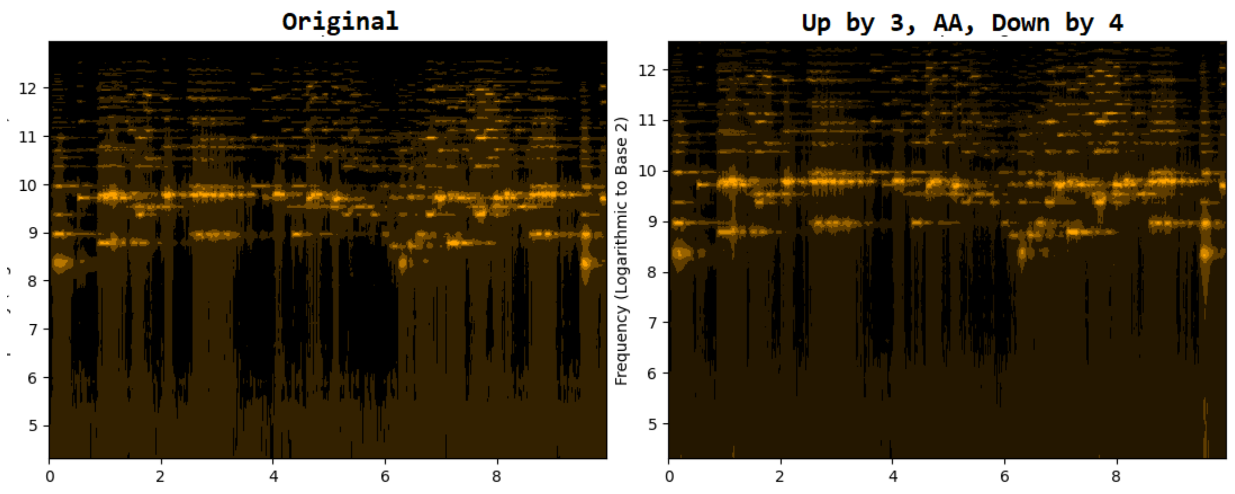

Figure 13: Music Spectrograms Comparison

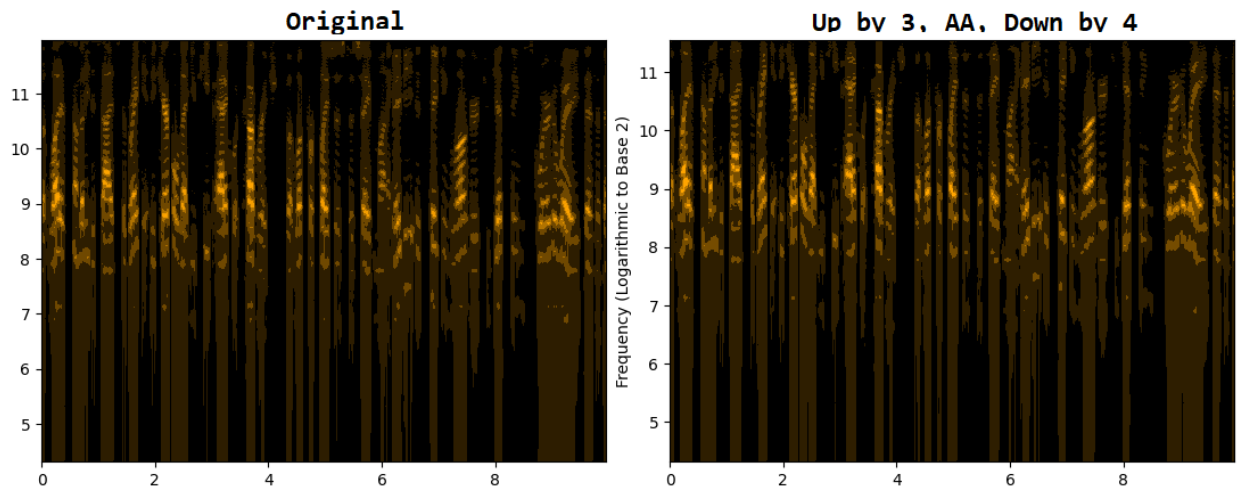

Figure 14: Speech Spectrograms Comparison

7 Swapping Order of Down-sampling & Up-sampling

Finally we try swapping the order of operations and see the effect of it. Before we proceed. For this we first realize a filter of 17 taps for this interpolation.

```
from scipy.signal import firwin
filter_coeffs_3 = firwin(17, 1/6)
```

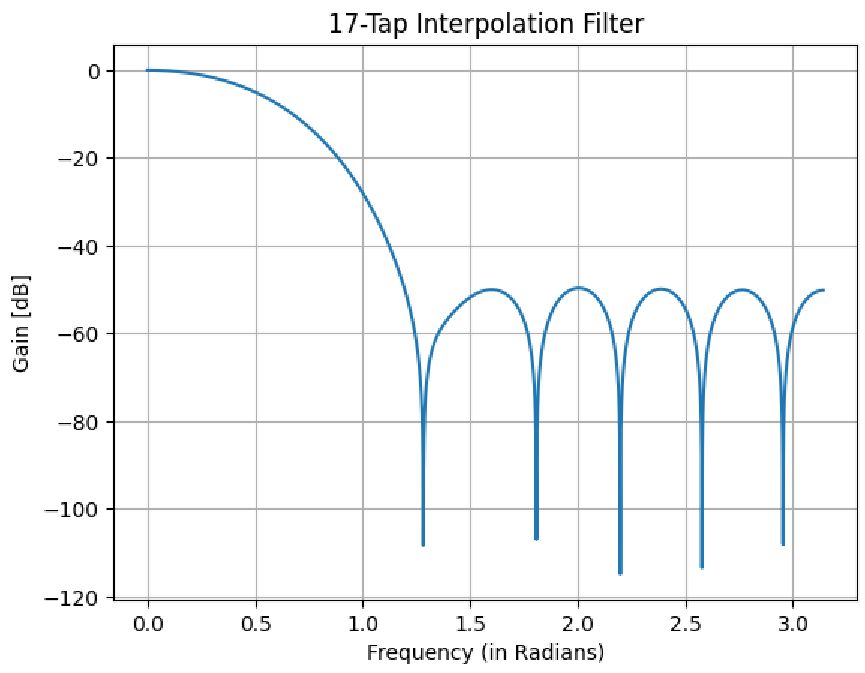

Figure 15: Interpolation Filter

The following code will AA-Filter, Downsample and then Upsample:

```
audio_2_du = np.zeros(audio_2.size*3//4, dtype=np.uint16)
audio_2_du[::3] = lfilter(filter_coeffs_2, 1.0, audio_2)[::4].astype(np.
 ↪uint16)
samples = audio_2_du.size

with wave.open('speech_du.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate((sampling_rate_2*3)//4)
    wav_file.setnframes(samples)
    wav_file.writeframes(audio_2_du.tobytes())
X_13,Y_13,Z_13, sampling_rate_13, audio_2_du = openaudio("speech_du.wav")
```

The resulting audio from doing this sounds severely damaged. The speech has lost all consonant because of the junk frequencies created in the upper half of the spectrum (which is the region that is manipulated by our mouths to produce consonants). The music sounds completely distorted without any harmonic definition because the same. To visualize this, below are their spectrograms.

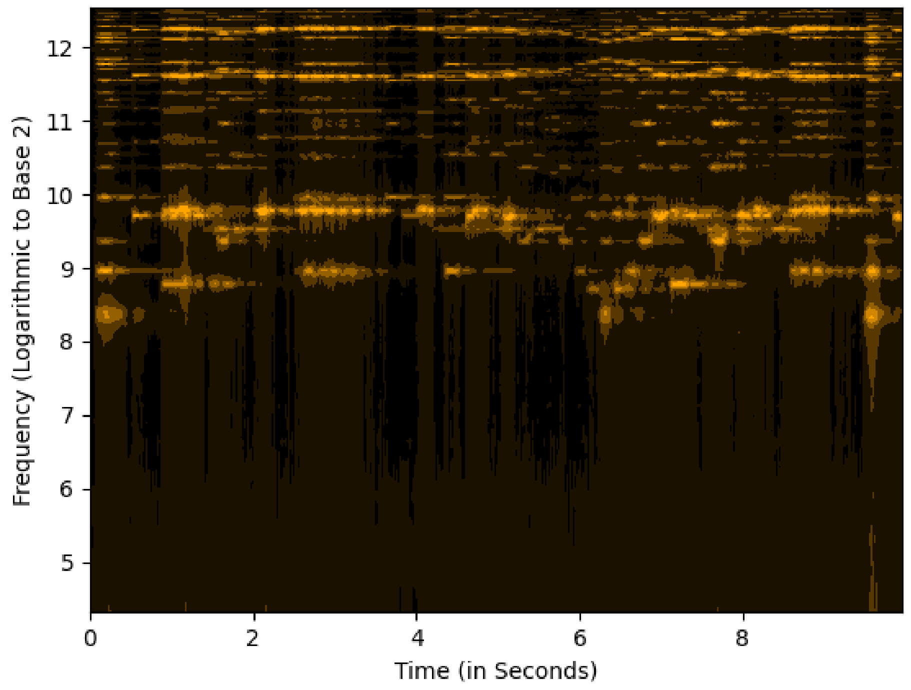

Figure 16: Spectrogram of Music demonstrating severe damage

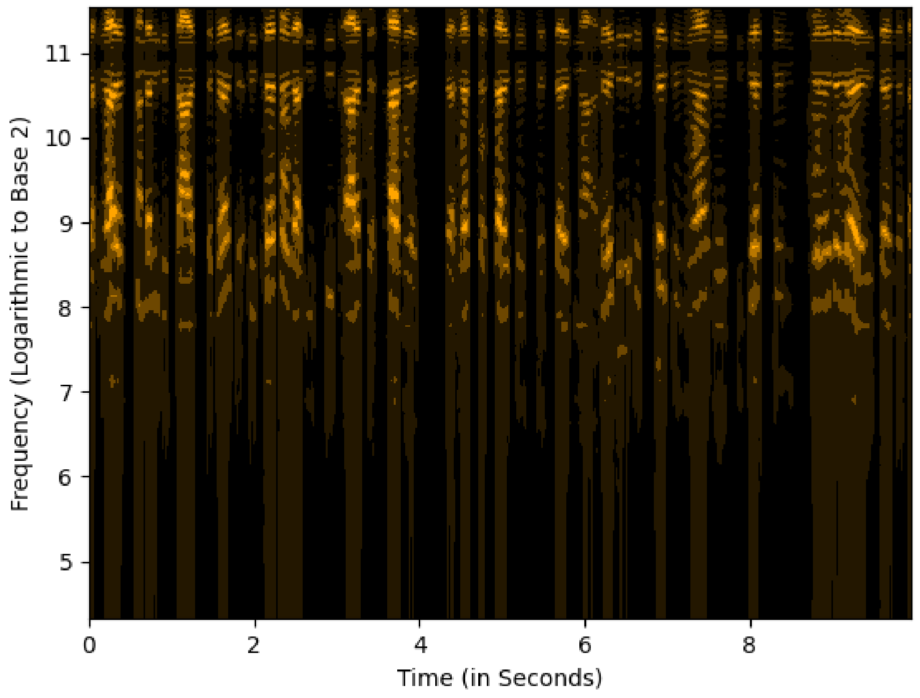

Figure 17: Spectrogram of Speech demonstrating severe damage

Now as discussed earlier, we can fix this with the interpolation filter.

```
audio_1_du_filtered = lfilter(filter_coeffs_3, 1.0, audio_1_du).astype(np.
 ↪uint16)
audio_2_du_filtered = lfilter(filter_coeffs_3, 1.0, audio_2_du).astype(np.
 ↪uint16)
samples_1 = audio_1_du_filtered.size
with wave.open('music_du_filtered.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate((sampling_rate_1*3)//4)
    wav_file.setnframes(samples_1)
    wav_file.writeframes(audio_1_du_filtered.tobytes())
X_14,Y_14,Z_14, sampling_rate_14, audio_1_du_filtered = openaudio('music_du_
 ↪filtered.wav')
samples_2 = audio_2_du_filtered.size
with wave.open('speech_du_filtered.wav', 'wb') as wav_file:
    wav_file.setnchannels(1)
    wav_file.setsampwidth(2)
    wav_file.setframerate((sampling_rate_2*3)//4)
    wav_file.setnframes(samples_2)
    wav_file.writeframes(audio_2_du_filtered.tobytes())
X_15,Y_15,Z_15, sampling_rate_14, audio_2_du_filtered = openaudio('speech_du_
 ↪filtered.wav')
spectrum2(X_14,Y_14,Z_14, "music_du_filtered.png")
spectrum2(X_15,Y_15,Z_15, "speech_du_filtered.png")
```

Here is a snapshot of the magnitude spectrum of Music before and after the Interpolation Filter.

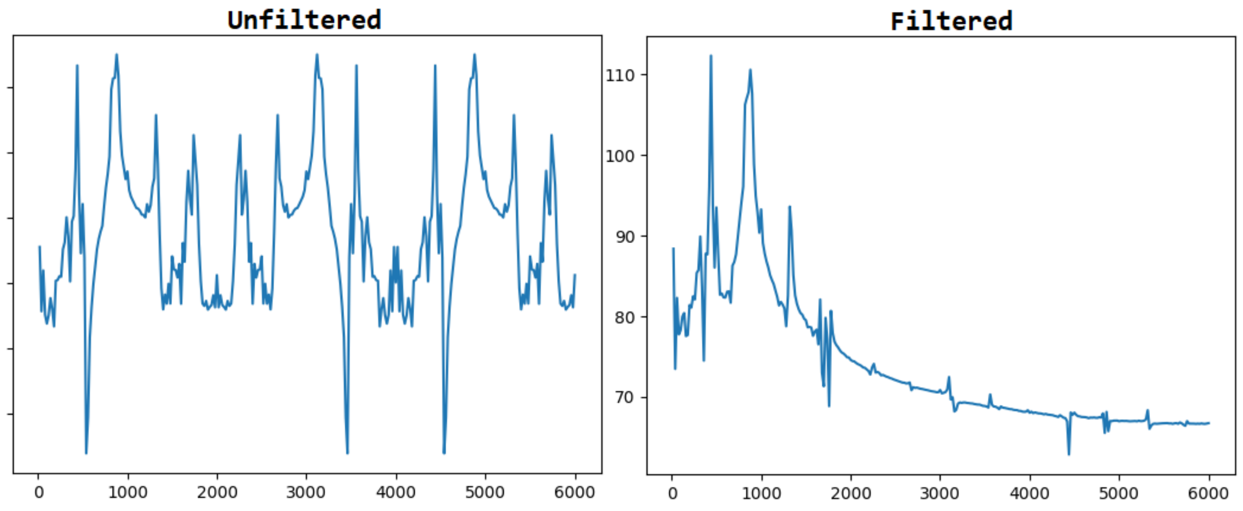

Figure 18: Image Rejection in Magnitude Spectrum

We can clearly see the image rejection in the below spectrograms. The higher frequencies have almost completely been removed. The playback is much better than before the interpolation filter but still is muffled in comparison to how it was when up-sampling was done first due to the loss of higher frequency information.

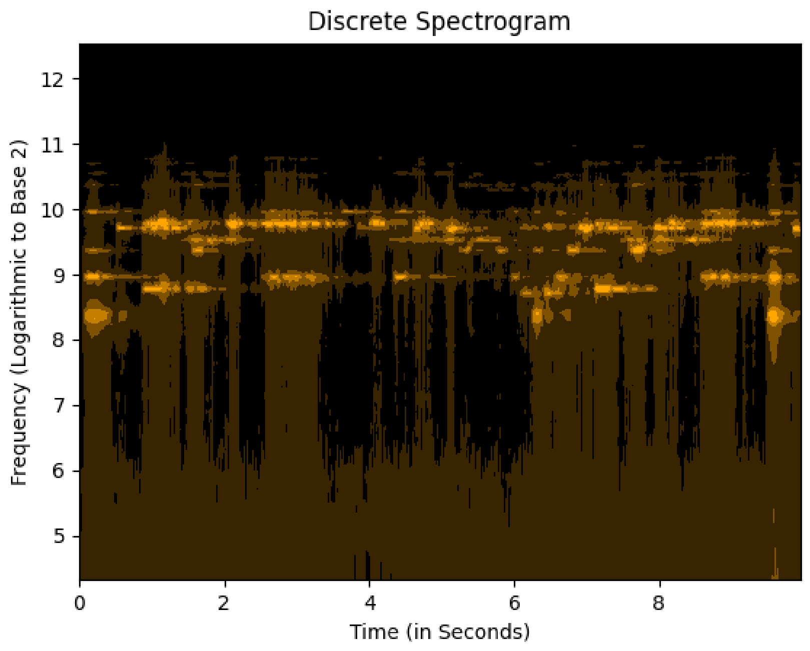

Figure 19: Spectrogram of Music demonstrating image rejection

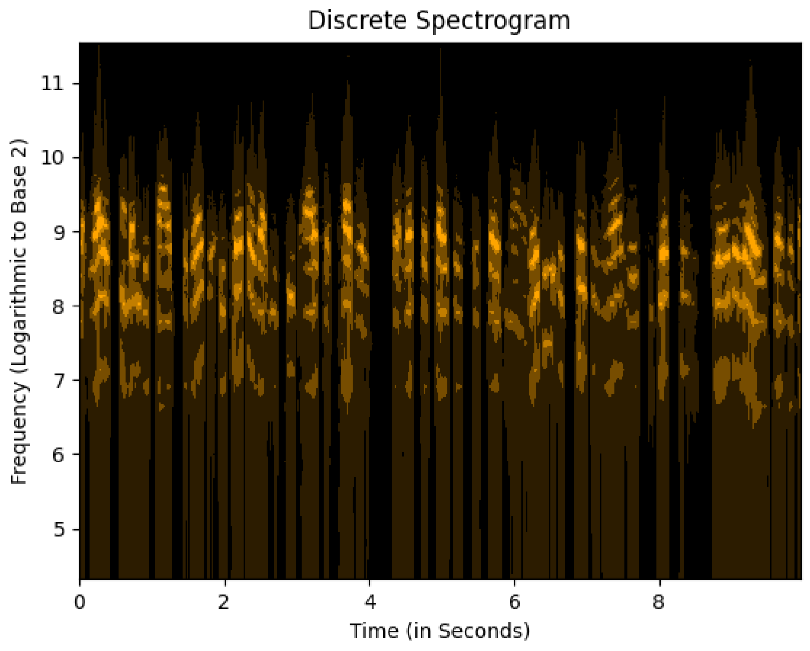

Figure 20: Spectrogram of Speech demonstrating image rejection

8 Links To Delivarables:

1. Magnitude spectrum of the two inputs “music16khz.wav” and “speech8khz.wav”: Figure 2 and Figure 3 in pages 3 and 4
2. Magnitude spectrum of the two outputs after downsampling by 2: Figure 4 and Figure 5 in page 5
3. Filter response of the Equiripple LPF: Figure 6 in page 6
4. Magnitude spectrum of the two outputs after anti-aliasing filter and downsampling by 2: Figure 7 and Figure 8 in pages 7 and 8
5. Compare the two downsampled signals with and without the anti-aliasing filtering of Part 1 and 2 for each audio input: Discussion at the start of page 5 and the discussion on Figures 8 & 9 in pages 7 and 8
6. Filter response of the Equiripple LPF: Figure 9 in page 8
7. Magnitude spectrum of the two outputs after upsampling by 3, to show the images: Figures 10 and 11 in pages 9 and 10
8. Magnitude spectrum of the two outputs after upsampling by 3, anti-aliasing filter and downsampling by 4: Figures 12, 13 and 14 in page 11
9. Compare the outputs with the original (input) signals: Discussion in page 10. Figures 13 and 14 in page 11
10. Magnitude response and impulse response of the interpolation filter: Figure 15 in page 12
11. Magnitude spectrum of the two outputs after anti-aliasing filter, downsampling by 4 and upsampling by 3: Figures 16 and 17 in page 13
12. Magnitude spectrum of the two final outputs after anti-aliasing filter, downsampling by 4, upsampling by 3 and interpolation: Figures 18, 19 and 20 in pages 14 and 15
13. Compare the outputs of Part 4 and Part 5 for each of the two inputs: Discussion and Figures in page 15