Student Name:	**Aaron Dinesh**
Student ID Number:	**20332661**
Assessment Title:	**DSP Lab 02**
Lecturer (s):	**Naomi Harte**
Date Submitted	**16/10/23**

I hereby declare that this assessment submission is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I hereby declare that I have not shared any part of this submission with any other student or person.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at: http://www.tcd.ie/calendar

I have also completed the Online Tutorial on avoiding plagiarism 'Ready, Steady, Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write

I am aware that the module coordinator reserves the right to submit my exam to Turnitin and may follow up with further actions if required should I be found to have breached College policy on plagiarism

Signed: 

DSP Lab 02

Ex 1.2.1

According to the Nyquist Sampling theorem, in order to sample a signal without any aliasing, you need to sample the signal at twice the maximum frequency contained in the signal. Given that the sine wave has a max frequency of 800Hz we would need to sample at 1600Hz. In reality, if we sample at exactly this frequency, we might only sample the zero-crossings of the sine wave, giving us a signal of all zeros. So, we should ideally sample above the Nyquist rate.

Ex 1.2.2

I chose a frequency of 800Hz. By the Nyquist Theorem we know that to properly sample this signal, we need to have a sampling rate greater than 1600Hz. So, I sampled the signal at 500Hz, 1000Hz and 2000Hz. In the 500 Hz signal I found a low frequency component, which from theory we know should lie at 300Hz. The story is similar in the 1000Hz sampled signal, where I heard another low frequency component at 200Hz. However in the 2000Hz signal I found no aliasing and no introduction of any low frequency components.

Ex 1.2.3

To illustrate the Nyquist theorem, I have chosen two signals, one with a frequency of 800Hz and another with a frequency of 7200Hz. Both sampled at a rate of 8000Hz. From sampling theory, we know that the Fourier transform of a sampled signal will contain the original frequency of the signal but will also contain shifted copies of the signal centred at the sampling frequency. Aliasing occurs when the shifted copies of the original signal begin to overlap with the signal centred at 0, thereby introducing new frequency components which weren't there before. From the Nyquist sampling theorem, we know that the 800Hz signal should be perfectly sampled, with a spike at 800Hz and a spike at 7200Hz (8000 - 800). For the 7200 signal, we should expect aliasing since we are not sampling at it's critical sampling rate. We should expect to see the fundamental frequency at 7200Hz but also a new aliased frequency at 800Hz (8000 – 7200). The FFTs of the two sampled signals are shown below:

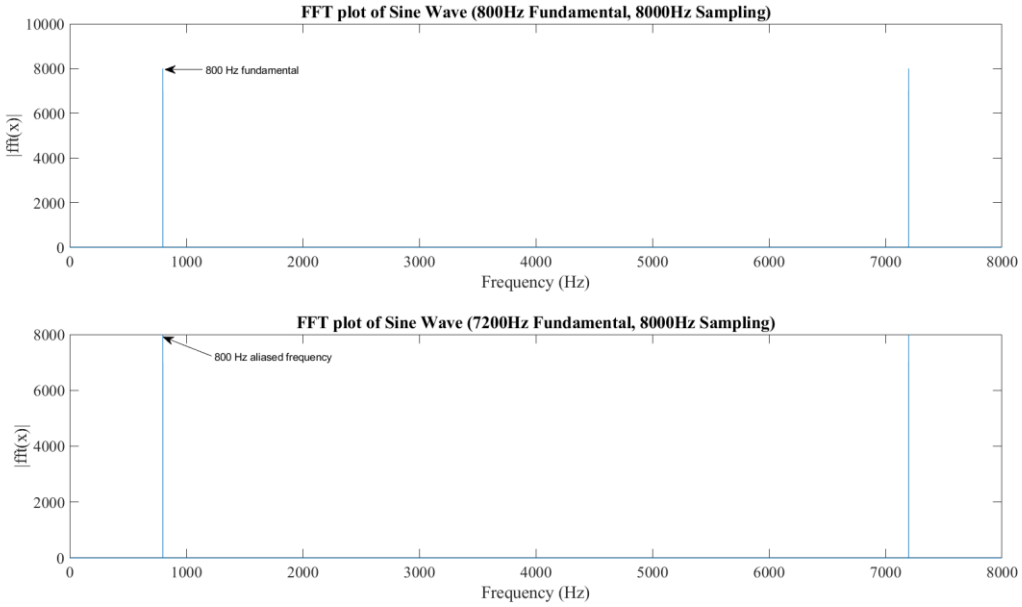


Figure 1 - FFTs of the sampled signals

As we can see in the graphs, our theory perfectly matched what we are seeing. When listening to both signals you hear the new low frequency component that was introduced.

Ex 1.2.4

This part of the lab had us load the handel.mat file available in MATLAB. This audio clip had a sampling rate of 8192Hz. The exercise required us to undersample at various sampling rates until we noticed aliasing.

At the original sample rate, the audio sounded aliased already. The sampling rate is far below what it should be to record music. The industry standard sampling rate for music is 44.1 kHz since it can accurately capture the entire range of frequencies that would be present in a song. But resampling this at lower sample rates increases the aliasing effect and shifts the frequency distribution towards the lower end of the spectrum. This can be seen in the FFT graphs below. The blue FFT is the original audio sampled at 8192Hz, the orange FFT is the audio sampled at 1000Hz and as we can see the resampled audio has a similar frequency distribution to the original audio, but it is more concentrated towards the lower end of the spectrum, which is characteristic of aliasing.

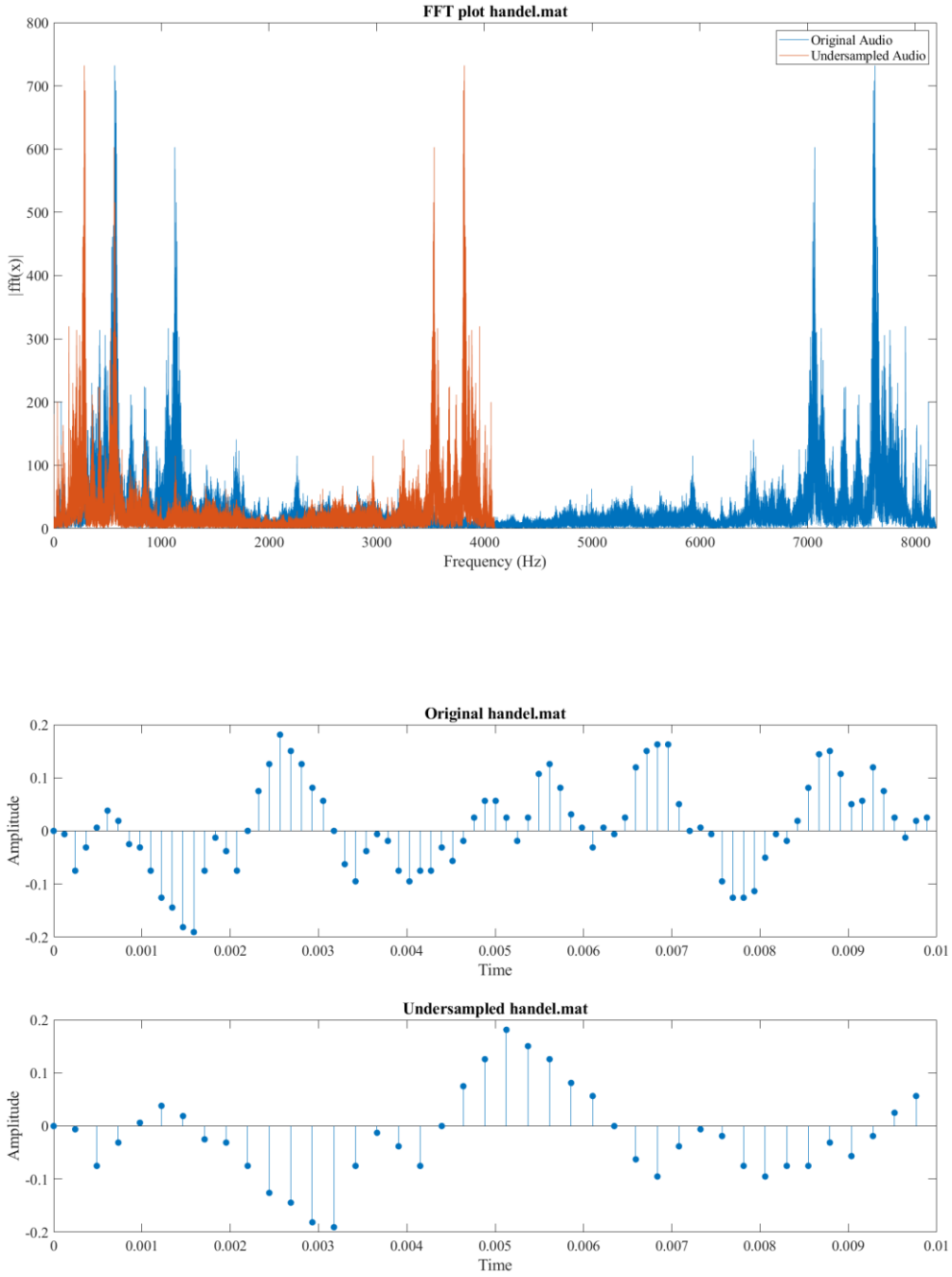


Figure 2 - Handel.mat and the resampled version

Ex 1.3.3 – 1.3.4

This exercise required me to create a function that accepts a note index, its duration, any octave change and then calculate the corresponding frequency for that index. The formula for the frequency of the note is as follows:

$$f_{note} = 440 \times 2^{\frac{note-69}{12}}$$

The note indices were taken from the table below which was given in the lab handout.

```
function [y, nOut] = createNote(note, duration, fs, octave_change)
    nOut = (0:1/fs:duration-1/fs);

    if(note == -1)
        y = zeros(1,length(nOut));
    else
        fNote = round(440 * power(2, (((note+(octave_change*12)) - 69) / 12)), 2);
        y = sin(2*pi*fNote*nOut);
    end
end
```

Figure 3 - createNote function

Using the function above I passed in notes 60, 62, 64 and also the "note" -1. The specification required us to return an array of zeros if the note passed in was -1. Since a time plot of the notes would yield no useful information, a frequency domain plot was generated which can be seen in figure 4.

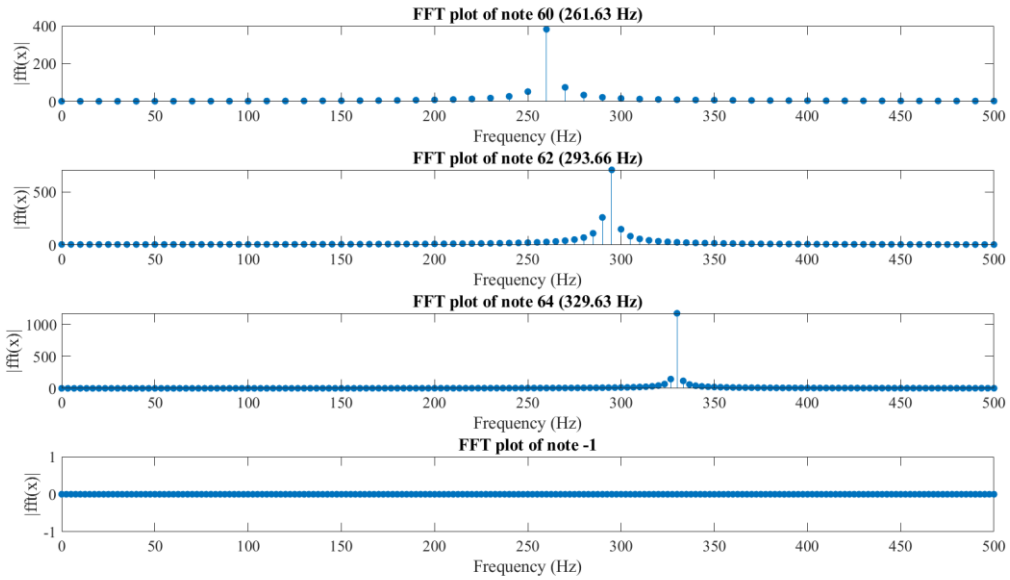


Figure 4 - FFT plot of the notes produced

As evidenced by the plot, the createNote function the correct frequency for the given index. The case where the note is -1 can be seen in the 4th subplot. Since it is an array of 0s there are no frequency components in there.

Ex 1.4.2

For this exercise I had to extend the createNote function to create a createMeoldy function. The implementation of this function is seen below in figure 5.

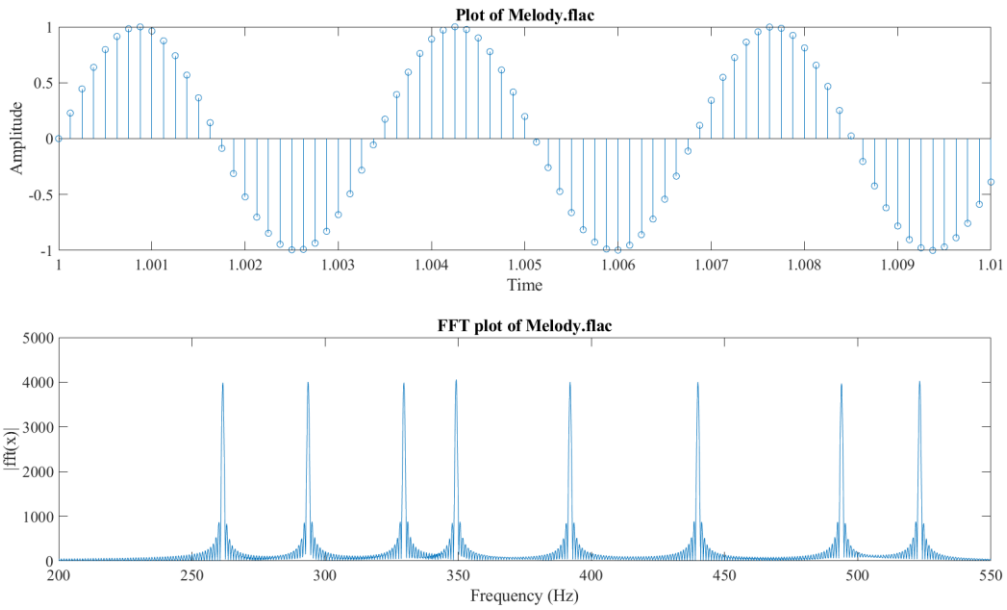
```
function [y, nOut] = createMelody(notes, duration, fs, octave_change)
    y = [];
for i = notes
    yNote = createNote(i,duration,fs, octave_change);
    y = [y yNote];
end

nOut = 0:1/fs:duration-1/fs;
end
```

Figure 5 - createMelody function implementation

Ex 1.4.3 – 1.4.4

In order to test the function I create a simple melody that started at middle C and ran through all the major notes until the C in the next octave. A short time plot and FFT plot of the melody can be seen below. There are 9 notes between middle C and the next octave (including the start and end notes) and in the FFT plot of the melody we can see 8 distinct peaks corresponding to the 8 notes played.



To increase it by an octave, we set the octave change to 1. To make the melody 50% faster, I wrote the original melody to a file, read it back and then rewrote it to a different file at double the sampling rate. Since I'm writing the same number of samples at twice the original sampling rate, I effectively made the melody 50% faster.

Ex 1.5

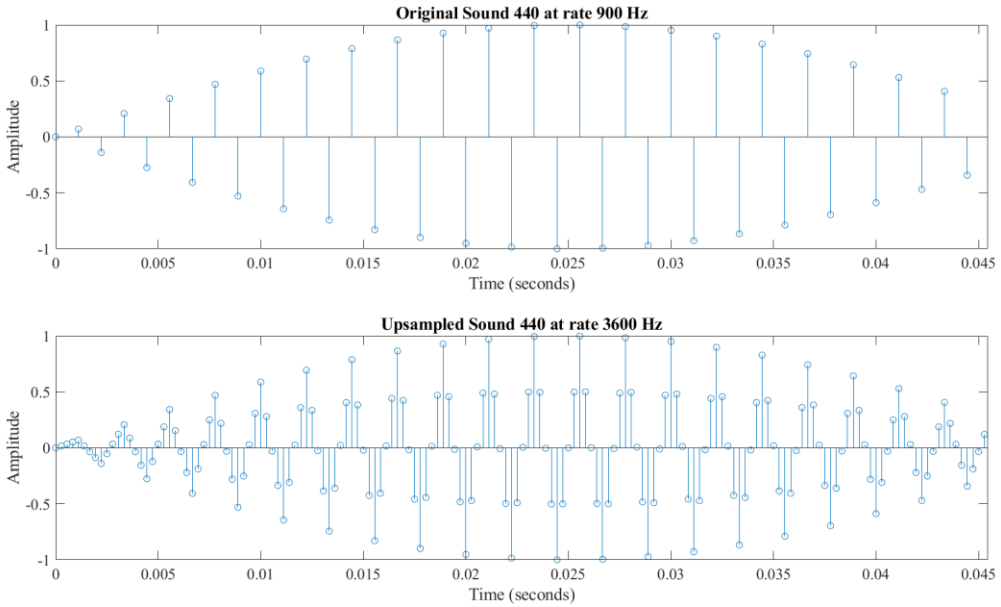
This exercise required us to create a function that would upsample a given series. Upsampling begins by first expanding the sequence by adding "$L-1$" zeros between each sample, effectively increasing the sampling rate by "L". Then we need to perform some type of interpolation between the samples to give us our final upsampled signal. While there are many interpolation schemes, for this lab I chose to implement a linear interpolation filter which is characterised by the impulse response below:

$$h[n] = \begin{cases} 1 - \frac{|n|}{L}, |n| \leq L \\ 0, otherwise \end{cases}$$

We can then convolve this impulse response with our system to obtain our final upsampled signal.

This interpolation scheme is far from perfect, it assumes that signals follow a linear relationship between samples, which for most signals isn't the case. However, we know if the time difference between subsequent samples is small enough (corresponding to a high original sampling rate) we can safely make this assumption and still achieve near-perfect upsampling. To demonstrate this, I chose the note 69 corresponding to A at 440Hz. I started with a sampling rate of 900Hz and a fixed upsampling factor of 4. I then upsampled the signal and kept changing the original sample rate until there was no perceivable distortion in the output.

At low sample rates (close to the Nyquist rate) there is noticeable distortion in the upsampled signal as can be seen in the graphs below. These distortions present as high frequency ringing in the signal. Also notice the additional frequency components in the upsampled signal.



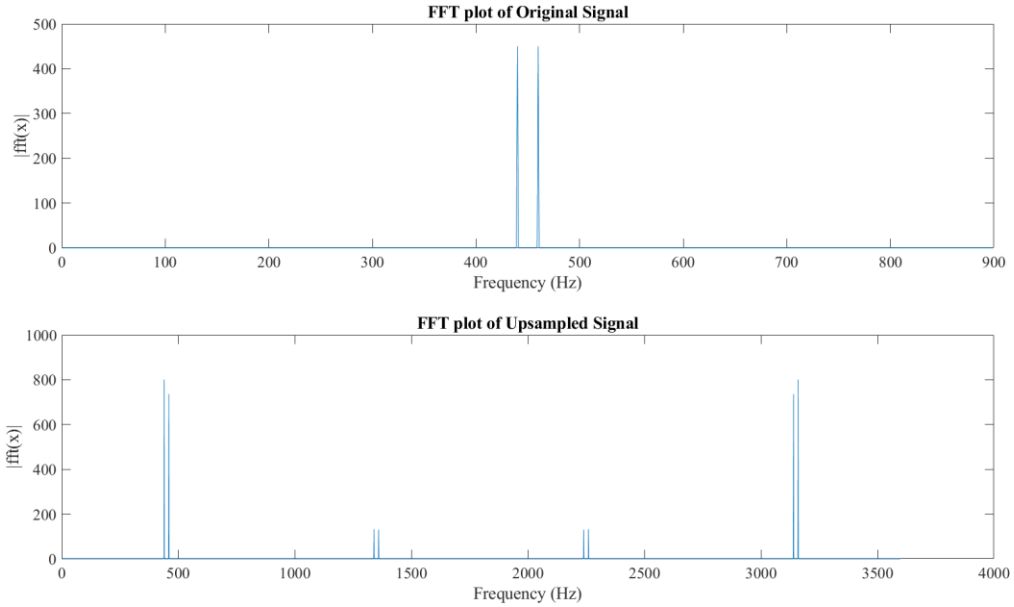
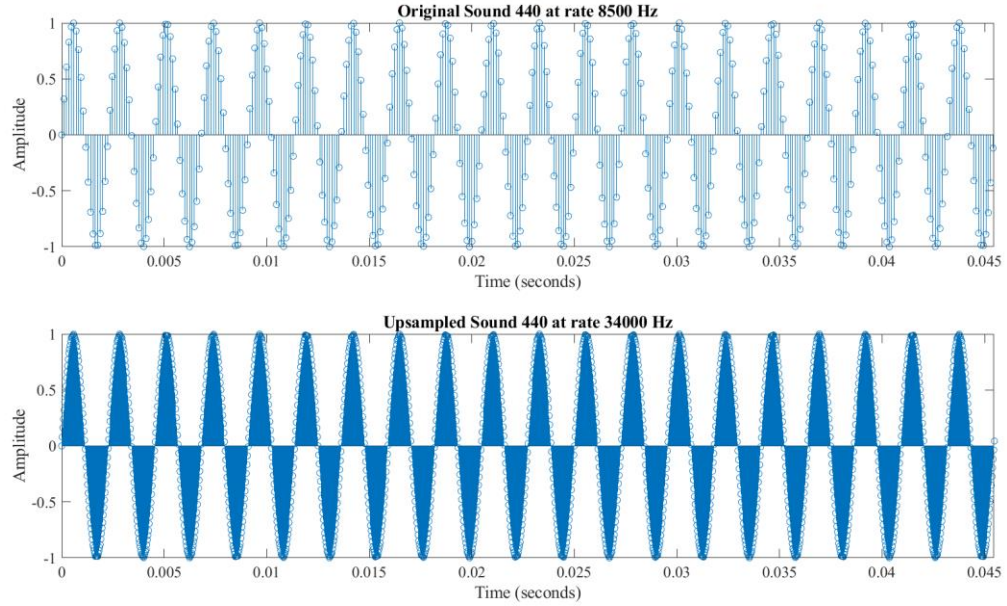


Figure 6 - Time series and FFT graph of original and upsampled audio

However, when we bump up the sampling rate to 8000Hz we perceive much less distortion in the upsampled signal as can be seen below:



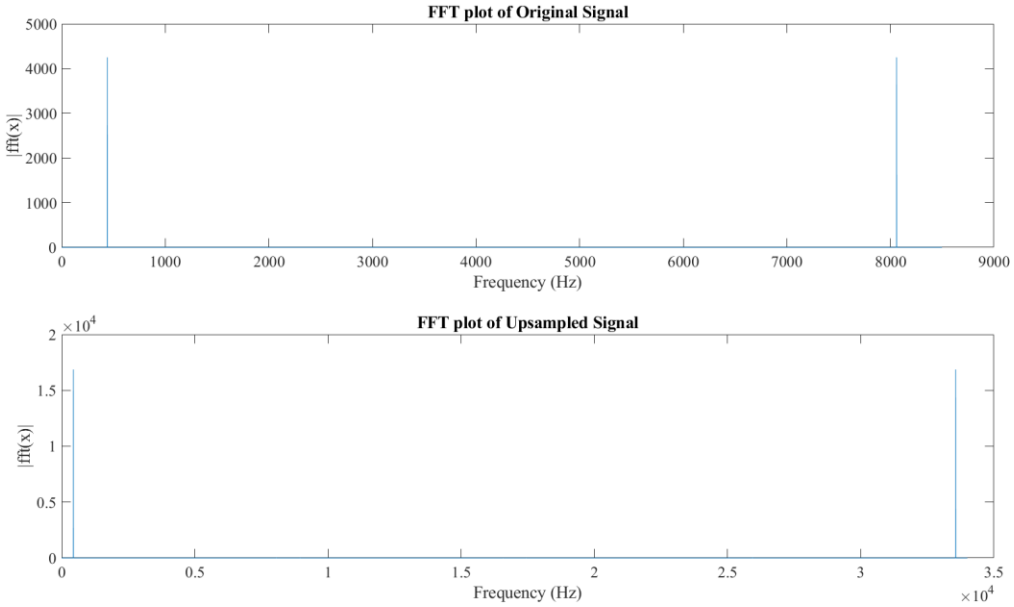


Figure 7 - Time Series and FFT graph of original and upsampled audio

This shows the limitations of linear interpolation. The linearity assumption only holds when the difference between sample points is small. So, in order to upsample by a large integer you need to have a high original sampling rate. Other interpolation schemes exist, with the most widely used one being Whittaker–Shannon interpolation. The interpolation scheme is as follows:

$$\mathrm{x}(t) = \sum_{n=-\infty}^{\infty} x[n] sinc\left(\frac{t - nT}{T}\right)$$

Where "sinc" is the normalised sinc function defined as:

$$sinc(x) = \frac{sin(\pi x)}{\pi x}$$

For $x \neq 0$.

Ex 1.5.2

Here I used my upsampling function to upsample the Handel.mat audio. I started with an upsampling factor of 2 and then progressed to factors of 4, 6 and 8. When upscaling by a factor of 2, there was a significant perceived improvement in audio quality. Upsampling had the effect of shifting the distribution of frequency towards the higher end of the spectrum, leading to a more natural sounding audio. However, past a factor of 2, there were diminishing returns. An upscaling factor of 4 provided little quality improvements and introduced a high-pitched ringing. This continued for factors 6-10 with the ringing gradually increasing in pitch. A plot of Handel.mat and a 4x upsampled version of Handel.mat can be seen below.

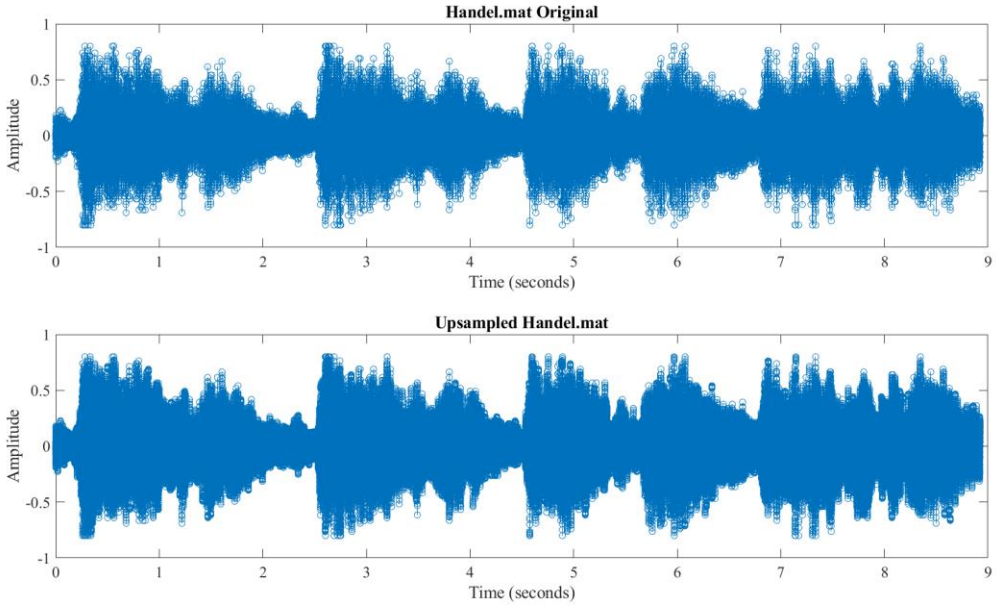


Figure 8 - Original and 4x upsampled Handel.mat