Lab 3 (i)

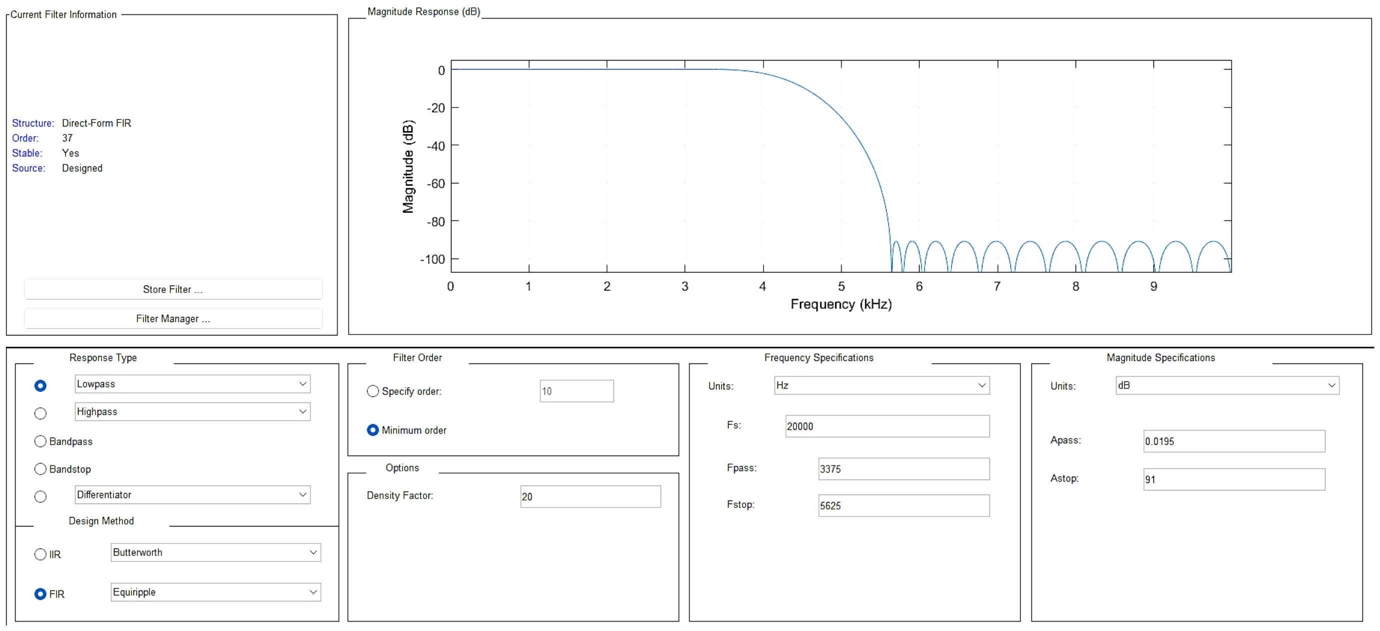
Sean Fahey *Student number: 21360313*

<u>3.1</u>

<u>I. What is the order of the filter you designed? What does this mean?</u>

Order: 37

This means the length of the filter is 38, where the length is the number of samples used in the weighted sum (the output).



Structure: Direct-Form FIR
Order: 37
Stable: Yes
Source: Designed

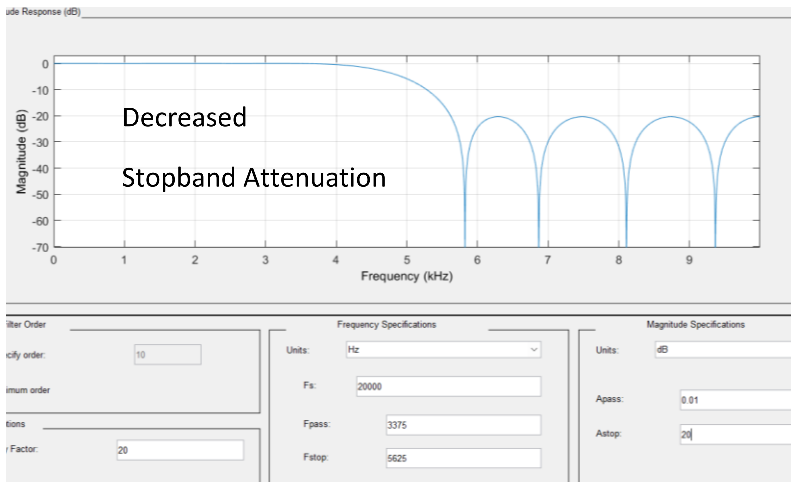
Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator
Design Method: IIR Butterworth [✓ FIR Equiripple]
Filter Order: Specify order: 10 [✓ Minimum order]
Options: Density Factor: 20
Frequency Specifications: Units: Hz; Fs: 20000; Fpass: 3375; Fstop: 5625
Magnitude Specifications: Units: dB; Apass: 0.0195; Astop: 91

<u>II. What are the effects of altering the passband and stopband attenuation?</u>

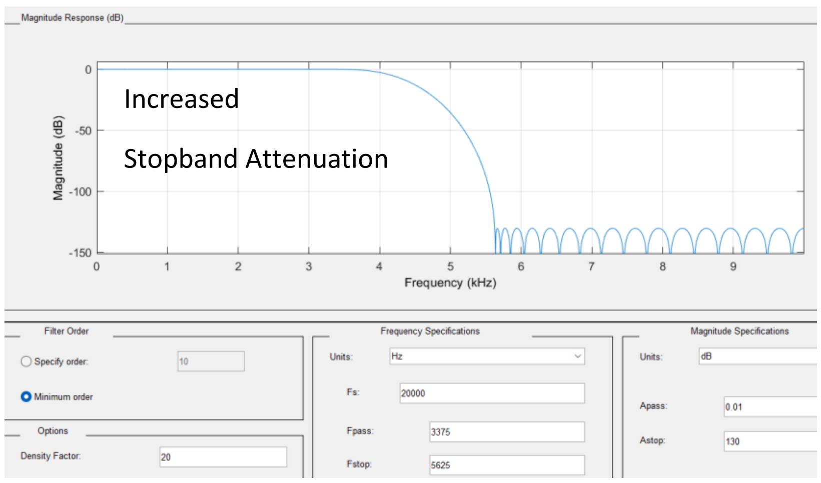
Altering the passband attenuation/ripple increases the variance in passband gain. Ideally we would like a smooth passband (low variance), as we want all of the signal frequencies preserved as much as possible.

Altering the stopband attenuation changes the difference between the top of the stopband and the bottom of the passband, which results in a reduced separation of noise and signal. A large stopband attenuation amplifies the signal much more than the noise, which is favorable.

Reducing the passband ripple or increasing the stopband attenuation *increased* the order of the filter, while increasing the passband ripple or reducing the stopband attenuation *decreased* the order of the filter.



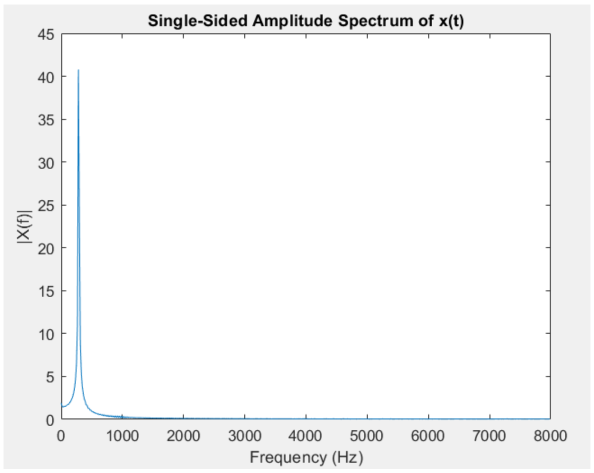
Filter Order: Specify order: 10; Minimum order
Options: Density Factor: 20
Frequency Specifications: Units: Hz; Fs: 20000; Fpass: 3375; Fstop: 5625
Magnitude Specifications: Units: dB; Apass: 0.01; Astop: 20



Filter Order: Specify order: 10 [✓ Minimum order]
Options: Density Factor: 20
Frequency Specifications: Units: Hz; Fs: 20000; Fpass: 3375; Fstop: 5625
Magnitude Specifications: Units: dB; Apass: 0.01; Astop: 130

3.2

Designing the FIR



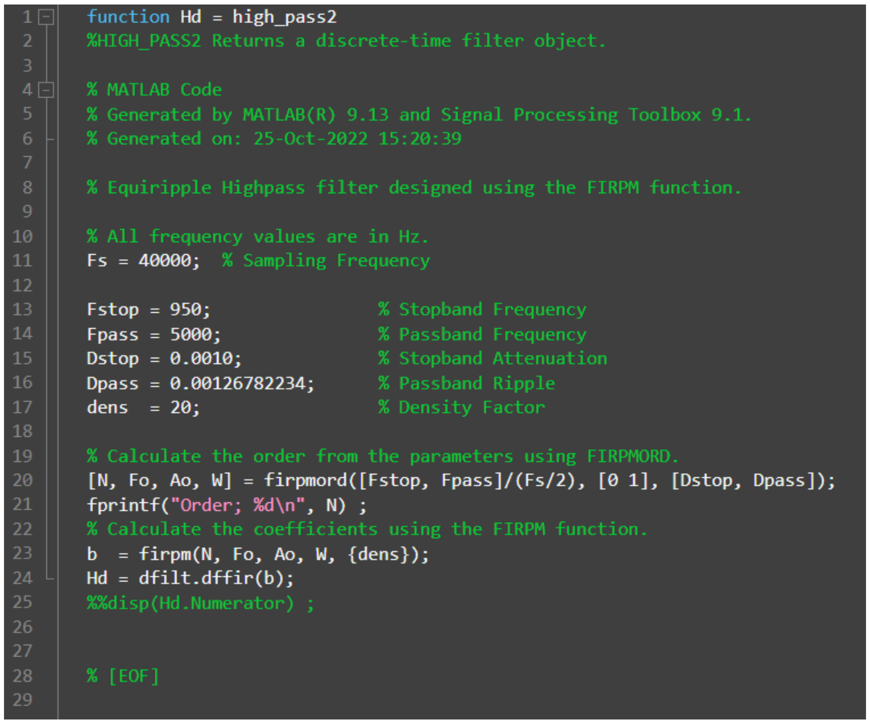
I first played the audio file through MATLAB using the `audioplayer()` and `play()` functions. I could hear the person speaking as well as a loud tone in the background. I also plotted the frequency spectrum of the audio file (shown on the left). The spike is the tone in the background, which is between about 200Hz and 500Hz.

I used the `filterdesigner` GUI in MATLAB to make an initial filter with rough estimates used for the parameters like stopband attenuation, passband ripple, Fs, etc.

I decided to use a *high pass* filter instead of a *band pass* filter, given that the noise spike is very close to 0 Hz.

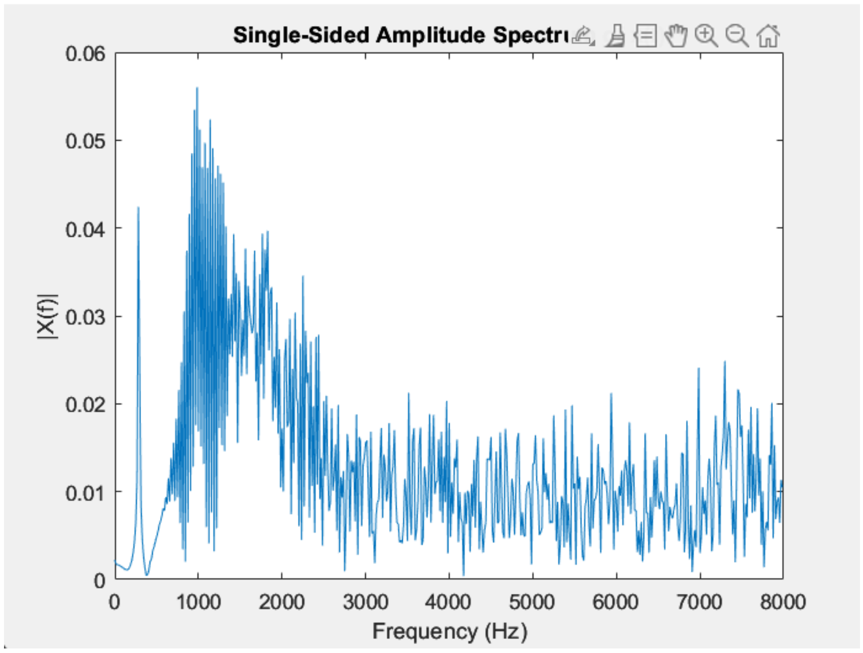
Through trial-and-error I changed all of the parameters, but mainly Fstop and Fpass, to remove the noise while keeping the clarity of the person's voice. The generated MATLAB function from the filter designing tool also produced *N*, the order of the filter. I saw that although I had removed the noise and kept the signal very accurately, the order was >200. This filter is not viable for practical use, so I began optimizing the filter to reduce the order while keeping the quality.

I ended up with the following high pass filter parameters:



```matlab
function Hd = high_pass2
%HIGH_PASS2 Returns a discrete-time filter object.

% MATLAB Code
% Generated by MATLAB(R) 9.13 and Signal Processing Toolbox 9.1.
% Generated on: 25-Oct-2022 15:20:39

% Equiripple Highpass filter designed using the FIRPM function.

% All frequency values are in Hz.
Fs = 40000;  % Sampling Frequency

Fstop = 950;                % Stopband Frequency
Fpass = 5000;               % Passband Frequency
Dstop = 0.0010;             % Stopband Attenuation
Dpass = 0.00126782234;      % Passband Ripple
dens  = 20;                 % Density Factor

% Calculate the order from the parameters using FIRPMORD.
[N, Fo, Ao, W] = firpmord([Fstop, Fpass]/(Fs/2), [0 1], [Dstop, Dpass]);
fprintf("Order; %d\n", N) ;
% Calculate the coefficients using the FIRPM function.
b  = firpm(N, Fo, Ao, W, {dens});
Hd = dfilt.dffir(b);
%%disp(Hd.Numerator) ;


% [EOF]

```

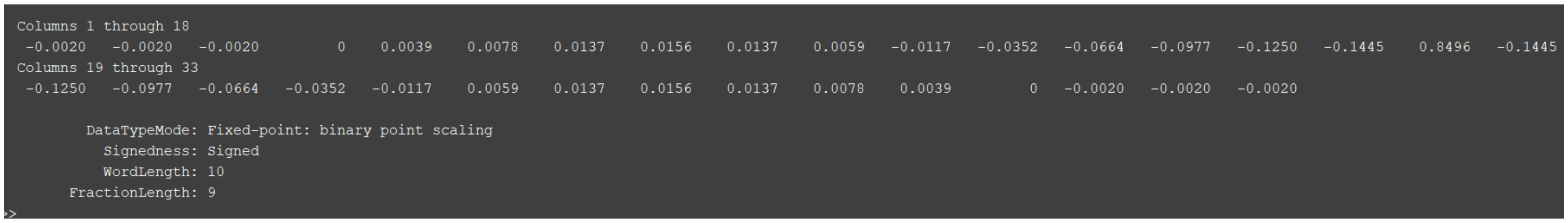
The frequency spectrum of the filtered audio



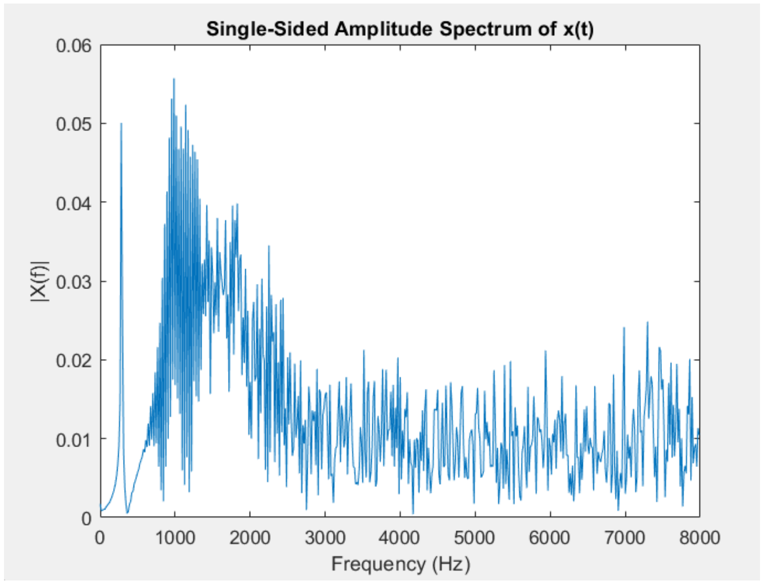
Here we can see that the noise spike still exists, but has a lesser magnitude than much of the signal frequency components. The filter eliminates the noise and keeps the signal with reasonable precision, while keeping the order realistic (**order N=32**).

Quantise the FIR Filter:

Appropriately-quantized



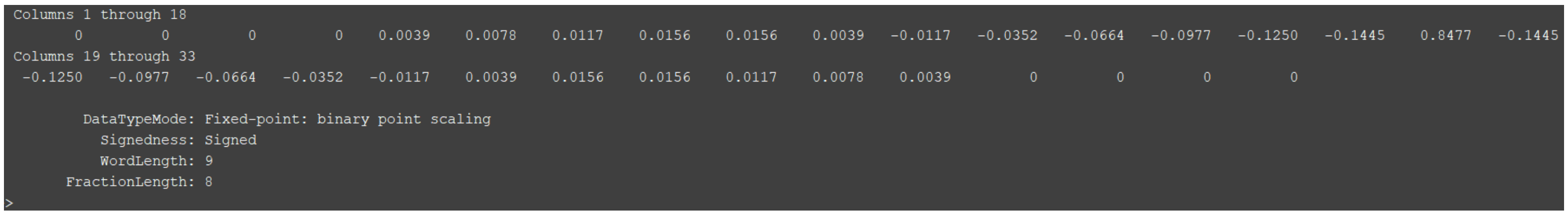
```
Columns 1 through 18
 -0.0020   -0.0020   -0.0020         0    0.0039    0.0078    0.0137    0.0156    0.0137    0.0059   -0.0117   -0.0352   -0.0664   -0.0977   -0.1250   -0.1445    0.8496   -0.1445
Columns 19 through 33
 -0.1250   -0.0977   -0.0664   -0.0352   -0.0117    0.0059    0.0137    0.0156    0.0137    0.0078    0.0039         0   -0.0020   -0.0020   -0.0020

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 10
      FractionLength: 9
>>
```



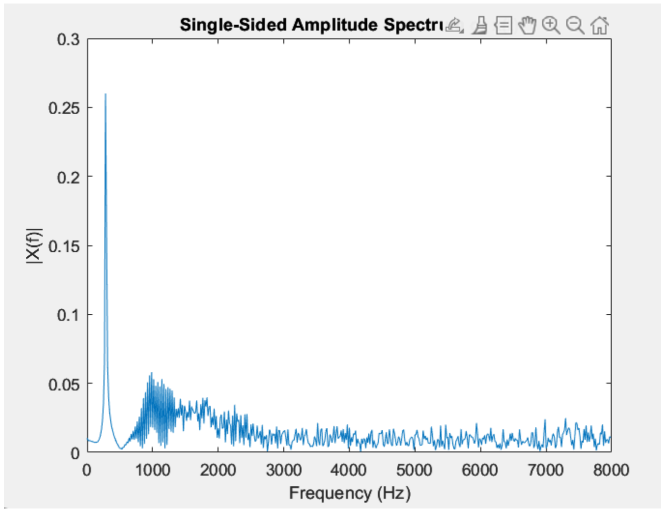
A wordlength of 10 was the minimum necessary to keep the filter characteristics as they were prior to quantization. As You can see below, reducing the wordlength to 9 made a large difference to the noise reduction.

The coefficients (above) were rounded to the nearest value available

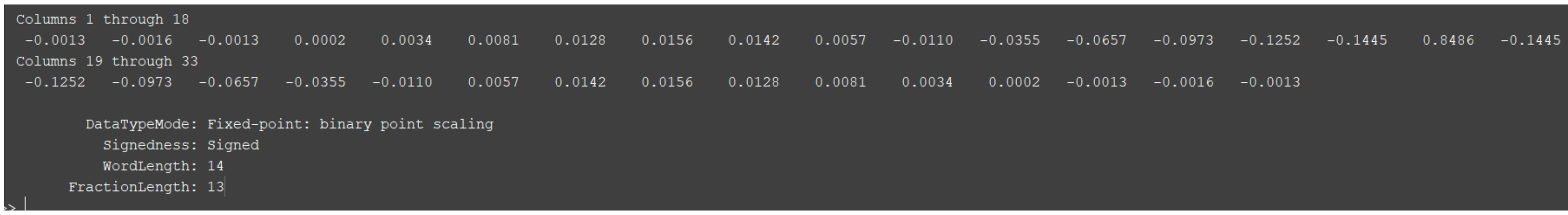
Under-quantized



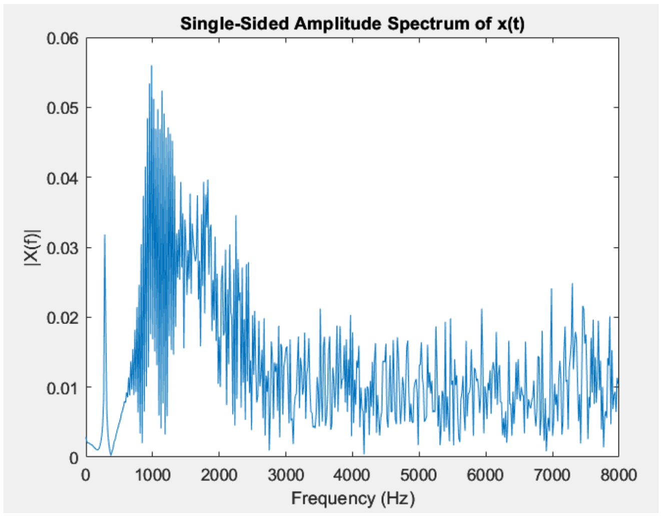
```
Columns 1 through 18
        0         0         0         0    0.0039    0.0078    0.0117    0.0156    0.0156    0.0039   -0.0117   -0.0352   -0.0664   -0.0977   -0.1250   -0.1445    0.8477   -0.1445
Columns 19 through 33
 -0.1250   -0.0977   -0.0664   -0.0352   -0.0117    0.0039    0.0156    0.0156    0.0117    0.0078    0.0039         0         0         0         0

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 9
      FractionLength: 8
>
```



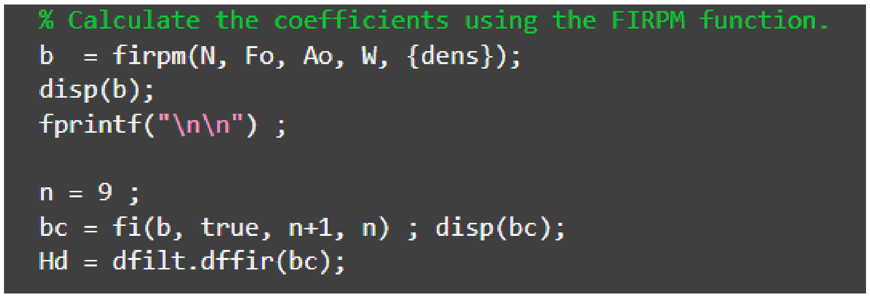
Over quantized



```
Columns 1 through 18
 -0.0013   -0.0016   -0.0013    0.0002    0.0034    0.0081    0.0128    0.0156    0.0142    0.0057   -0.0110   -0.0355   -0.0657   -0.0973   -0.1252   -0.1445    0.8486   -0.1445
Columns 19 through 33
 -0.1252   -0.0973   -0.0657   -0.0355   -0.0110    0.0057    0.0142    0.0156    0.0128    0.0081    0.0034    0.0002   -0.0013   -0.0016   -0.0013

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 14
      FractionLength: 13
>
```



Code used to quantize filter:



```
% Calculate the coefficients using the FIRPM function.
b  = firpm(N, Fo, Ao, W, {dens});
disp(b);
fprintf("\n\n") ;

n = 9 ;
bc = fi(b, true, n+1, n) ; disp(bc);
Hd = dfilt.dffir(bc);
```

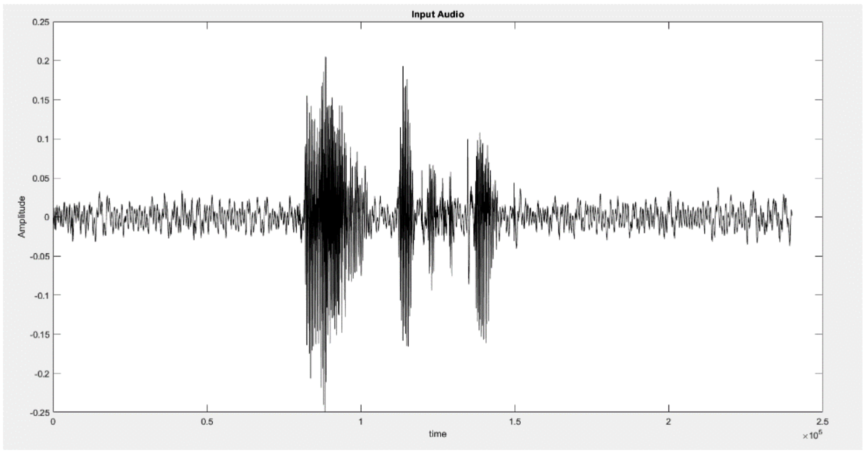
Lab3 part 2

Sean Fahey student number: 21360313

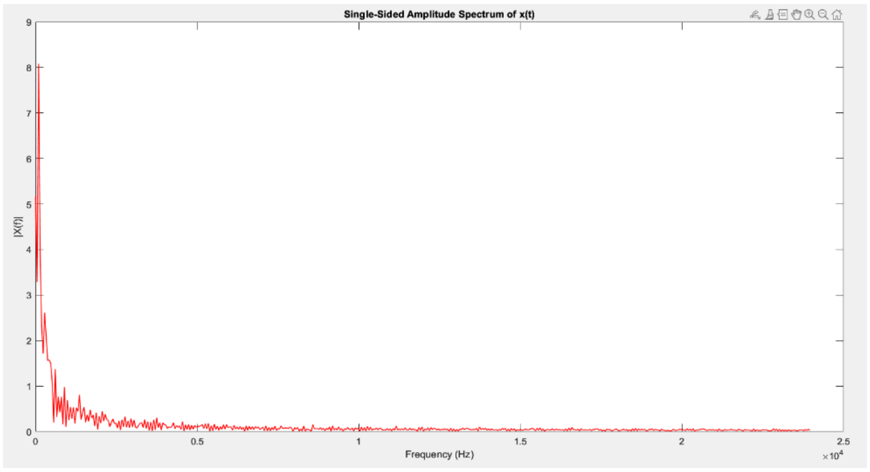
Intro

The aim of this lab was to produce a Finite-Impulse-Response filter to remove noise from an audio file, and implement it in hardware using an FPGA, via Jupyter notebooks. The filter will be designed using MATLAB and exported using a .coe file into Xilinx Vivado.

FIR Filter Design

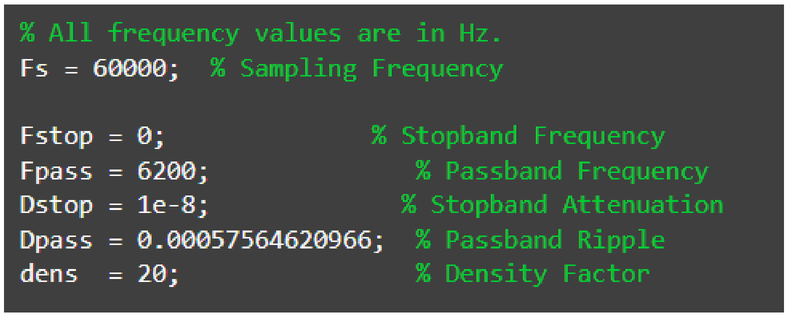


The original audio file with the noise unfiltered.

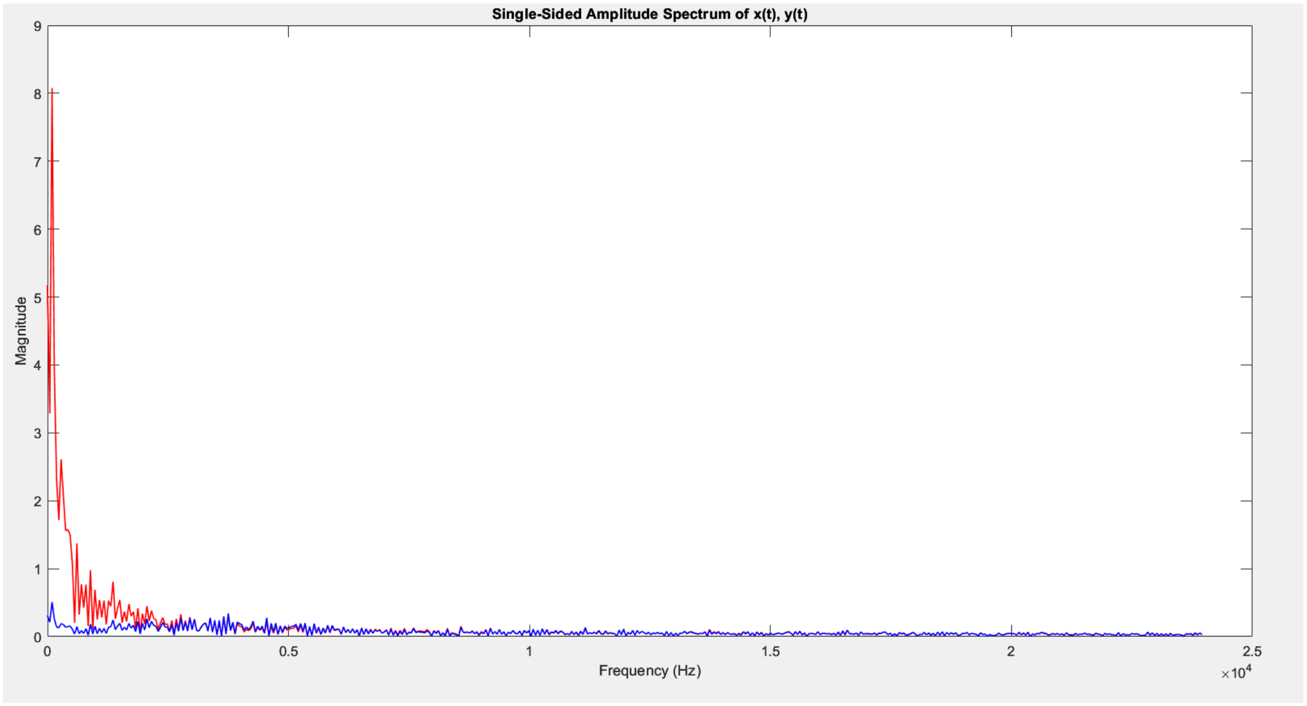


The frequency spectrum of the original audio. As we can see there is a large spike near 0Hz, which we will seek to eliminate.

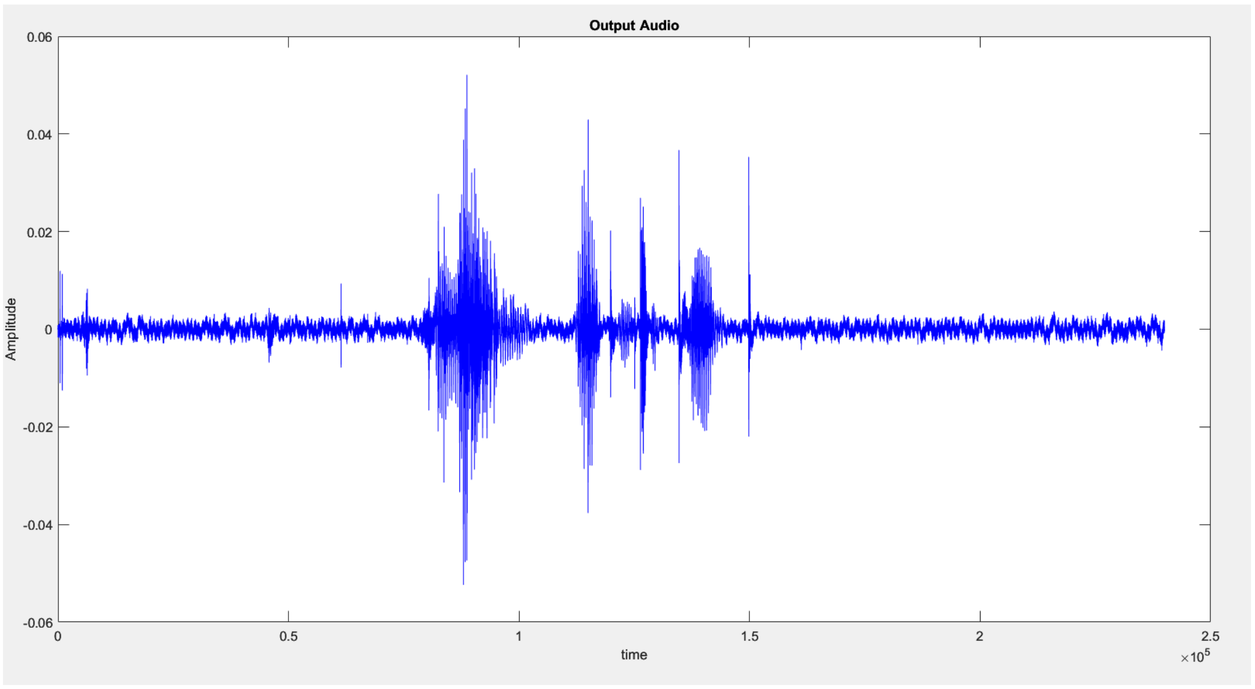
I chose a **high-pass** filter, because I want to keep all of the frequency components except for the lower end of the spectrum. I used the `filterDesigner` tool in MATLAB to create an initial high-pass filter. After analyzing the frequency spectrum graph and some trial and error, I ended up with the following filter parameters:

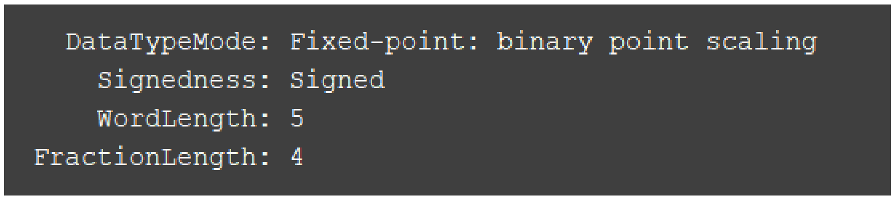


```
% All frequency values are in Hz.
Fs = 60000;  % Sampling Frequency

Fstop = 0;                 % Stopband Frequency
Fpass = 6200;                 % Passband Frequency
Dstop = 1e-8;               % Stopband Attenuation
Dpass = 0.00057564620966;  % Passband Ripple
dens  = 20;                   % Density Factor
```

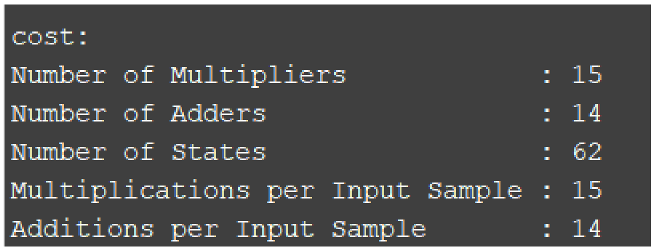


The filtered audio (blue) has been levelled out so that the spike has been reduced to the same level as the other frequency components, which means the noise has been reduced.





```
   DataTypeMode: Fixed-point: binary point scaling
     Signedness: Signed
     WordLength: 5
 FractionLength: 4
```



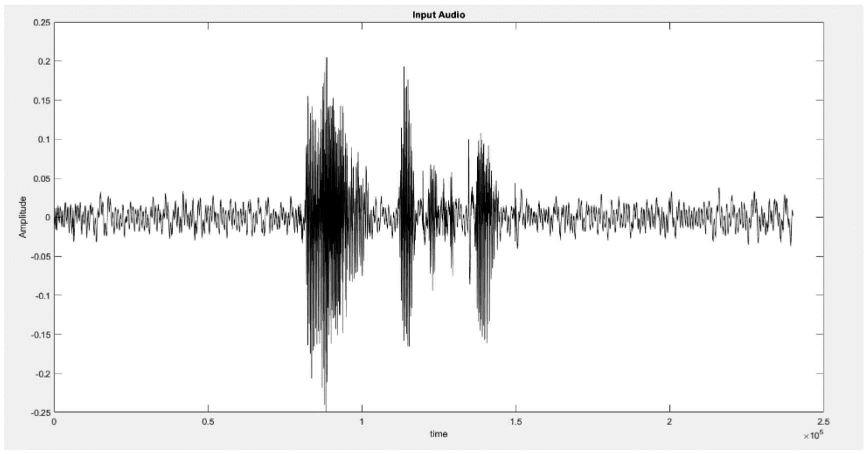
```
cost:
Number of Multipliers            : 15
Number of Adders                 : 14
Number of States                 : 62
Multiplications per Input Sample : 15
Additions per Input Sample       : 14
```

I also used quantization to reduce the complexity of the filter, as there were many bits which contributed very little to the filtering. A word length of 5 bits was enough to keep the quality high.

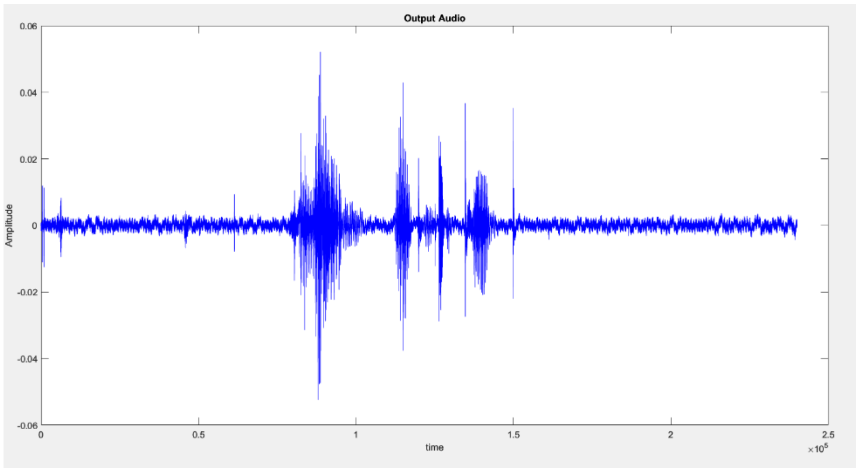
Overlay IP Components

Processor System Reset	Module for any type of reset. Options selected using pins set to high/low. Supports synchronous, asynchronous reset. An asynchronous reset can also be synchronized with the clock.
AXI4-Stream Data FIFO	AXI protocol interface with 2kb of Tx and Rx memory (stack-style /First-In-First-Out)
FIR Compiler	A filter compiler; can generate FIR filters given an array of coefficients. Supports up to 2048 taps. Up to 64 input streams. Allows reloadable coefficients.
AXI Direct Memory Access	Facilitates access to memory via AXI4 interface. Can also offload data movement tasks from a CPU.
AXI Interrupt Controller	Can map several different interrupt signals to one interrupt signal. Uses AXI4 protocol.
AXI SmartConnect	Connects one or more memory-mapped master devices to one or more memory-mapped slave devices, using the AXI protocol.
ZYNQ7 Processing System IP	Acts as a wrapper around the ZYNQ7 processing system. Facilitates integration of other IP into the processing system.
Audio_codec_controller	An IP block for controlling the reset, clock and address of codec. and A codec is a dedicated compressor/decompressor for handling audio data.

Before & After Filtering

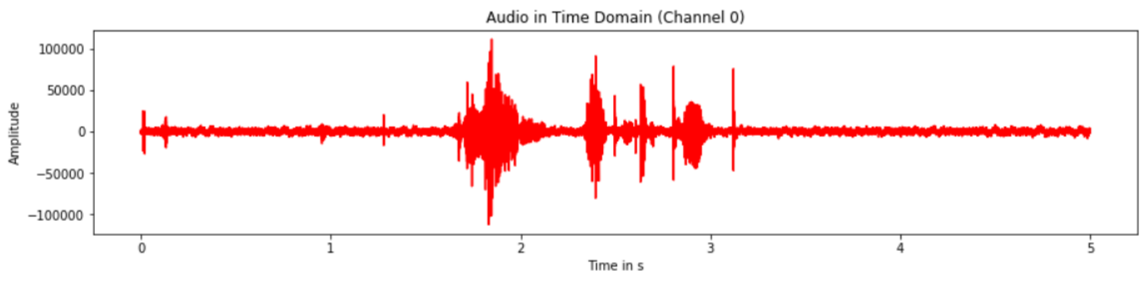


Before



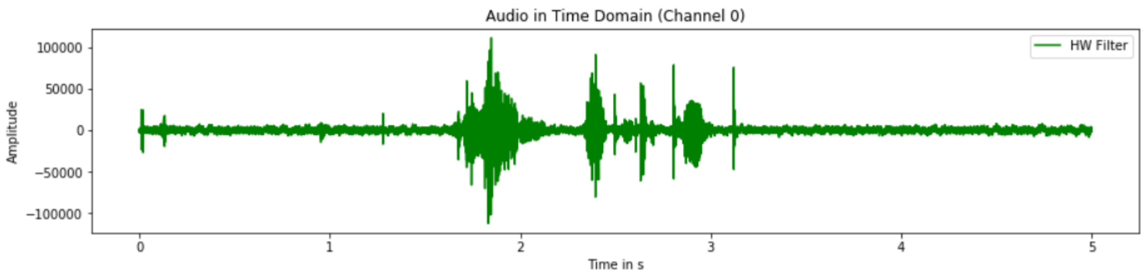
After

Hardware vs Software



Software

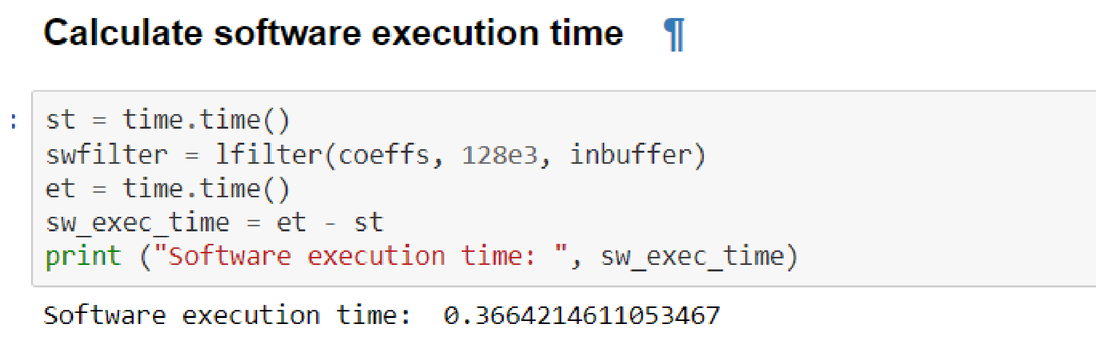
Filter



Hardware

Filter

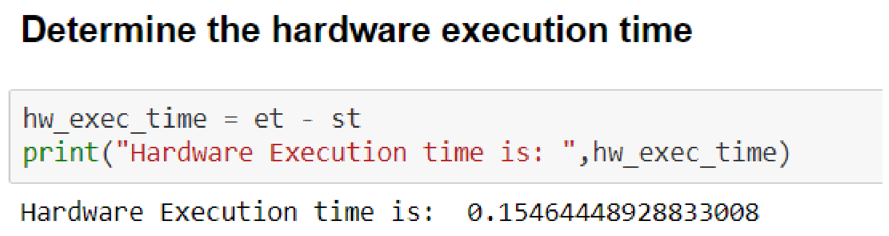
As we can see the hardware and software implementations are identical. However, there are a number of differences in cost and performance.



Calculate software execution time ¶

```
st = time.time()
swfilter = lfilter(coeffs, 128e3, inbuffer)
et = time.time()
sw_exec_time = et - st
print ("Software execution time: ", sw_exec_time)
```

```
Software execution time:  0.3664214611053467
```



Determine the hardware execution time

```
hw_exec_time = et - st
print("Hardware Execution time is: ",hw_exec_time)
```

```
Hardware Execution time is:  0.15464448928833008
```

The hardware implementation is ~2.4x faster than the software implementation. The cost of using dedicated hardware is more, given that you cannot use the IC for anything else while it is being used as a filter. Using software is cheaper as we can re-use the compute power of the computer for anything else, so there is no opportunity cost lost.

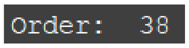
The quantized coefficients are as follows:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2048, -2048, -2048, -2048, -2048, -2048, -2048, 30720, -2048, -2048, -2048, -2048, -2048, -2048, -2048, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In the list of coefficients there are many zeros because the word length is just 5 bits, so many coefficients round down to zero. Also, the large integers are actually small real numbers, interpreted without any fractional bits.

Conclusion

In this lab I was able to demonstrate how to take a noisy audio file and implement a FIR filter using MATLAB and Vivado. I was surprised by how well a simple multiply-accumulate filter with just 5 bits could function. As I expected, the hardware outperformed the software implementation.



```
Order:  38
```

The order of the high-pass filter was 38, which was higher than I would have liked, but was minimum order necessary to keep the quality of the audio to an acceptable level.