DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING IMPERIAL COLLEGE LONDON

EE3-19
REAL-TIME DIGITAL SIGNAL PROCESSING
LAB 4 REPORT

YASMIN BABA - 01196634
ZHI WEN SI - 01243716

23rd of February 2019

TABLE OF CONTENTS

1. Filter Design

1.1 Filter Specification

The purpose of Lab 4 is to design and implement a FIR filter in addition to investigating methods that will allow it to run as fast as possible. The filter must satisfy the following specifications shown in *Figure 1*.

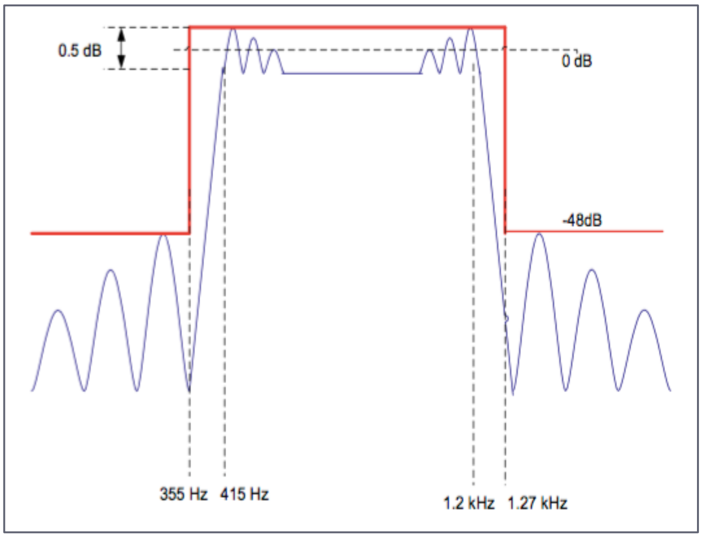

Figure 1: FIR Filter Specifications

In words, this is a bandpass filter with the following parameters:

High Pass Transition Band	$\mathbf{355\ Hz - 415\ Hz}$
Low Pass Transition Band	$1200\ Hz - 1270\ Hz$
Passband Ripple	$\leq\ 0.5\ dB$
Stopband Gain	$\leq -48\ dB$

Table 1: FIR Filter Specifications

1.2 Filter Design Using Matlab

Using Matlab, the FIR filter coefficients were designed via the Parks-McClellan algorithm using the following code shown in *Listing 1*.

```
%Filter Coefficient Calculator Via Parks McClellan Algorithm

rp = 0.5;                    %the passband ripple
sa = 48;                     %minimum stop band attenuation
fc = [355 415 1200 1270];    %cut off frequencies
fs = 8000;                   %sampling frequency
a = [0 1 0];                 %band gains
%max deviations/ripples for each band
dev = [10^(-sa/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-sa/20)];

%calculates parameters used by the firpmord function
[N, fo, Ao, W] = firpmord(fc, a, dev, fs);
%calculates FIR filter coefficients
coefs = firpm(N, fo, Ao, W);

fvtool(coefs)
save fir_coefs3.txt coefs -ascii -double -tabs
```

Listing 1: Matlab Code for FIR Filter Design

A FIR filter with 249 real coefficients was produced, creating a 248th order FIR filter. Before proceeding to the frequency response of the filter, a sufficient understanding of a FIR filter is required. Note that all following plots have been acquired using the **fvtool** function with respect to the 249 coefficients designed in Matlab.

A FIR filter is a digitally designed filter, whose impulse response eventually falls to zero after a finite amount of time, as shown below in *Figure 2*. It therefore has limited memory represented by the order of the filter, M. A very important property of the FIR filter is that its impulse response is symmetrical. The filter coefficients are the values in the impulse response, hence are also symmetrical.

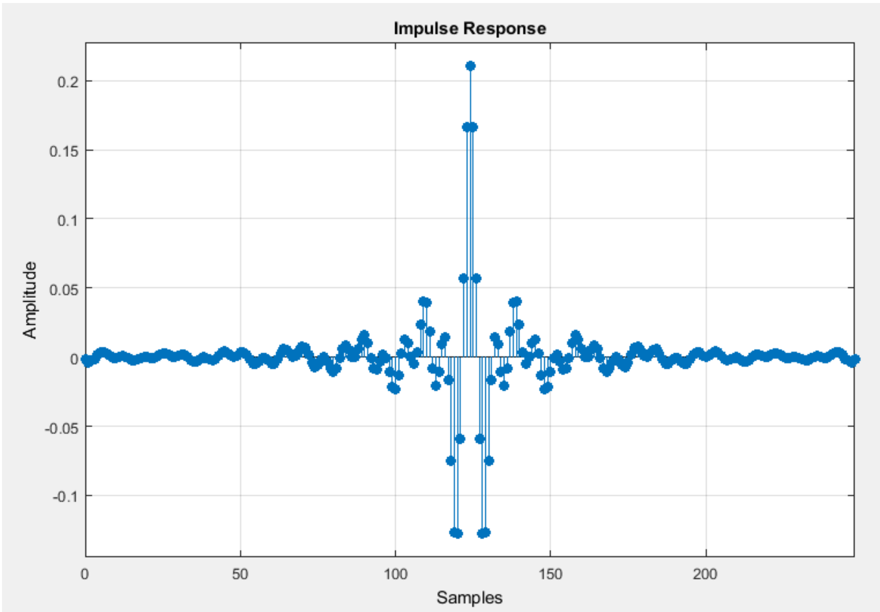

Figure 2: Impulse Response

The filter can be represented by the following time-domain difference equations, where $M + 1$ represents the number of coefficients.

$$y(n) = b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M)$$

Taking Z-transforms, the transfer function of the M^{th} order FIR filter is:

$$H(z) = b_0 + \frac{b_1}{z} + \cdots + \frac{b_M}{z^M} = \sum_{k=0}^{M} b(k) z^{-k}$$

Given this transfer function, it is clear that the filter can have zeros anywhere in the z plane, in whose positions depend on the filter's coefficients. The filter, however, will only have its poles at the origin, where $z = 0$, as shown below in *Figure 3*. This is the difference compared to an IIR filter, which can have poles anywhere on the z-plane.

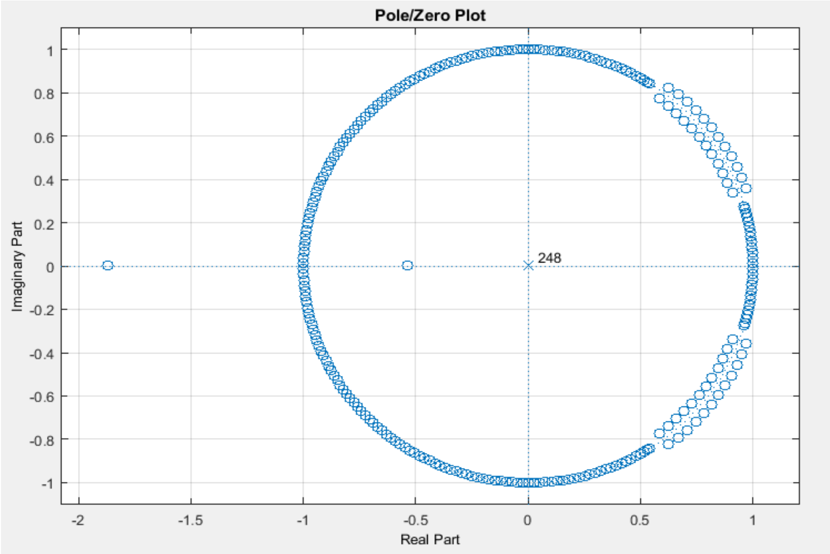

Figure 3: Z-Plane Plot

Only poles are considered when analysing the stability of the system. This is because if a pole exists outside the unit circle on the z-plane, the system will have an exponentially growing impulse response and thus will be unstable. It can be concluded that since the poles of a FIR filter are at the origin, the system is naturally stable.

Zeros, on the other hand, affect the magnitude response. The closer the zero to the unit circle, the larger the attenuation will be – hence zeros play no role in stability. Notice that there is a concentrated number of zeros around the stopband of the designed 248th ordered filter – this is desirable as it will attenuate the gain in the stopband, blocking unwanted signal frequencies. However, since there are only a finite number of zeros, the gain will never be able to attenuate to negative infinity, in which an ideal box shaped filter would. Having zeros so close together, and so close to the unit circle, produces the ripple in the stop-band observed in *Figure 4*. The gain can be viewed to have peaks and troughs, where each trough is the gain attenuation of one zero.

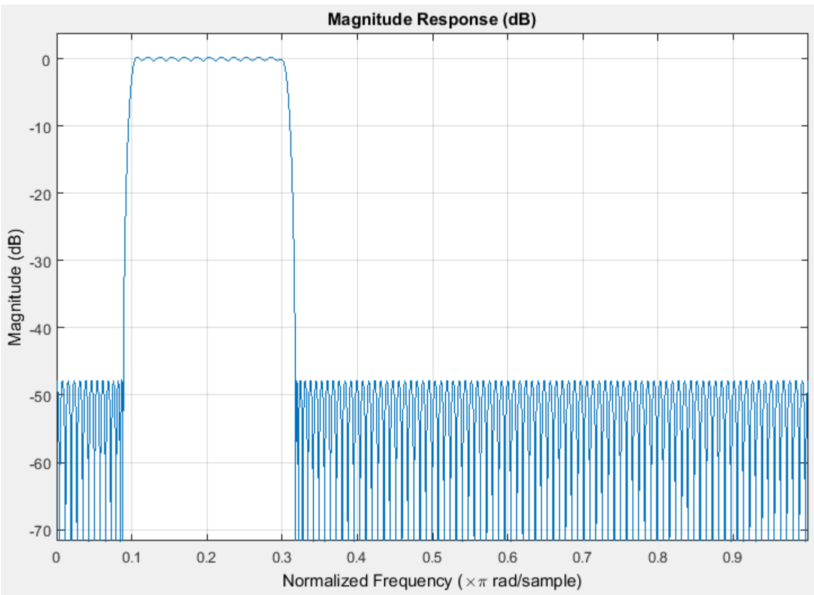

Figure 4: Magnitude Response

The phase response of the filter is shown in *Figure 5*.

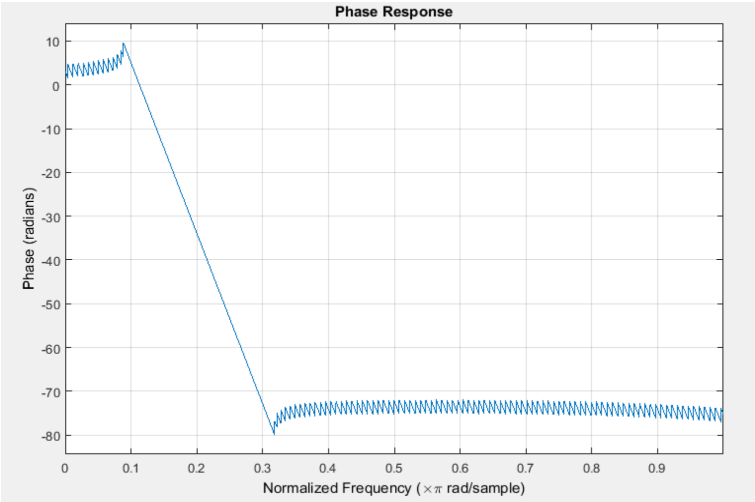

Figure 5: Phase Response

The FIR filter has an advantage of a linear phase in the passband. This is only possible when the filters coefficients are symmetrical. Since the group delay is calculated as,

$$\tau_g = -\frac{d\emptyset}{d\omega} = \frac{M-1}{2}$$

the delay through the filter is constant, preserving the shape of the waveform when it passes through the filter. This is especially important in audio signal processing where phase distortion is easily recognisable by the human ear.

In this case, the group delay has been measured to be 124, corresponding to the above equation.

$$\tau_g = \frac{M-1}{2} = \frac{249-1}{2} = 124$$

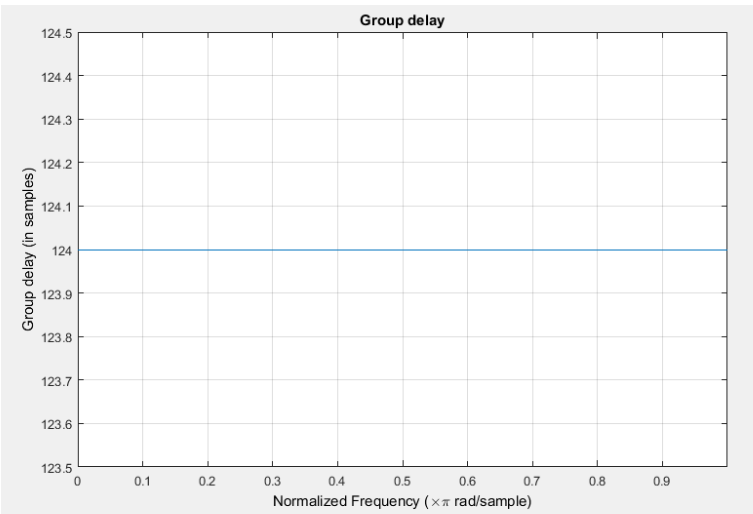

Figure 6: Group Delay

The 248th order filter does not meet the specification of remaining below a stopband gain of $-48\ dB$. *Figures 7* and *8* illustrate the gain at the edges of the transition band whilst *Figure 9* compares the stopband gain to the desired threshold.

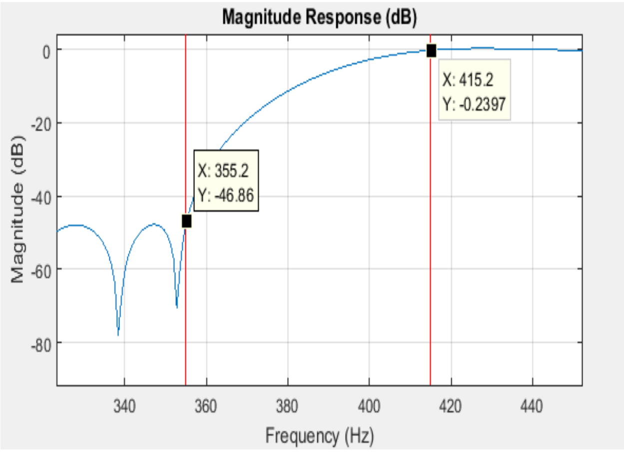

Figure 7: High Pass Transition Band

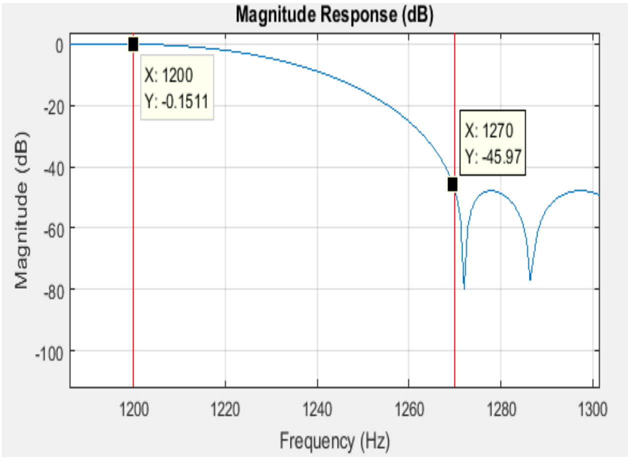

Figure 8: Low Pass Transition Band

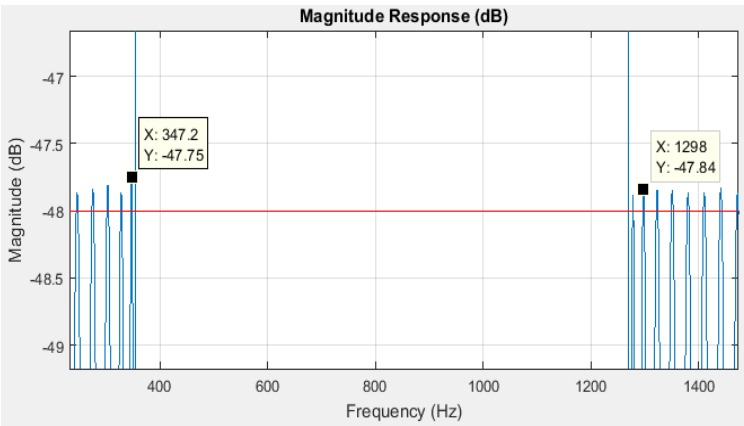

Figure 9: Minimum Stopband Gain Level

Furthermore, the passband ripple exceeds the maximum 0.5 dB deviation, as measured in *Figure 10*.

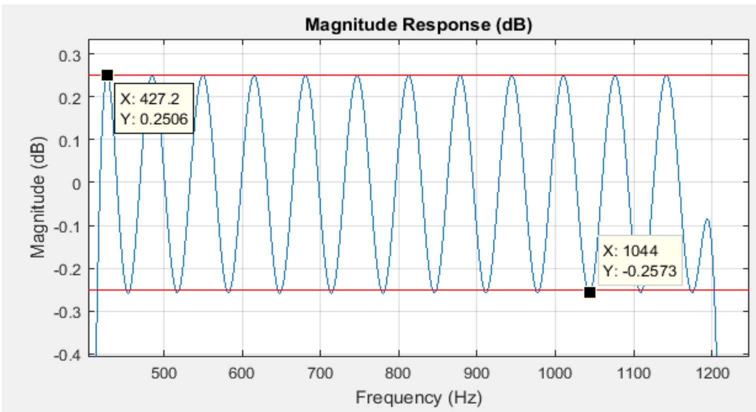

Figure 10: Gain Deviation in the Passband

In order to meet the specifications, the filter coefficients have to be increased. As the Parks-McClellan algorithm only produces filters of even orders, an even number of coefficients must be added before the **firmp** function. After some trial and error, an addition of 10 coefficients was decided to be sufficient.

The transition band has been measured as follows, illustrating the gain lies below $-48\ dB$ at the band edges.

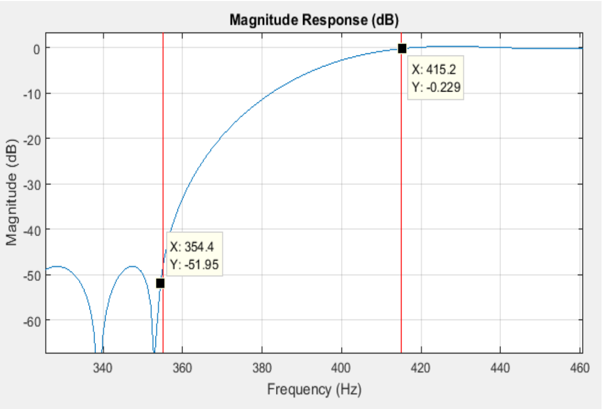

Figure 11: High Pass Filter Transition Band

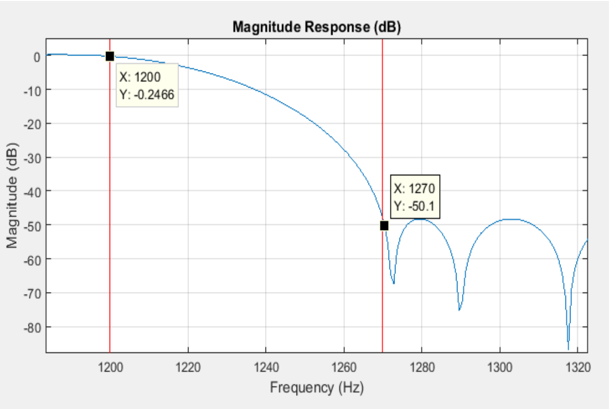

Figure 12: Low Pass Filter Transition Band

The stopband ripple is also lies below $-48\ dB$, as desired.

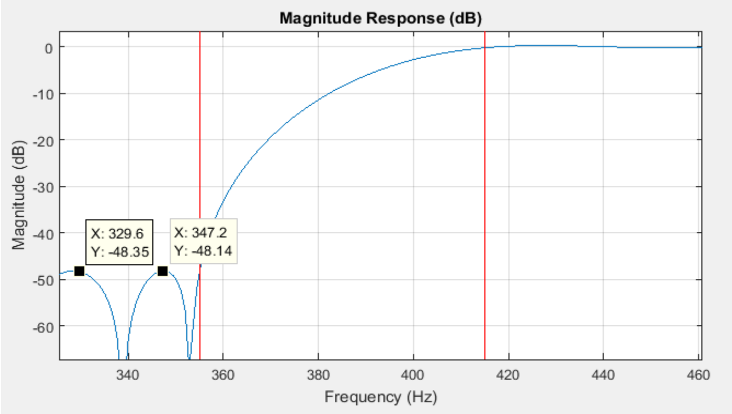

Figure 13: Stopband Gain

Finally, the passband ripple lies within the desired 0.5 dB deviation.

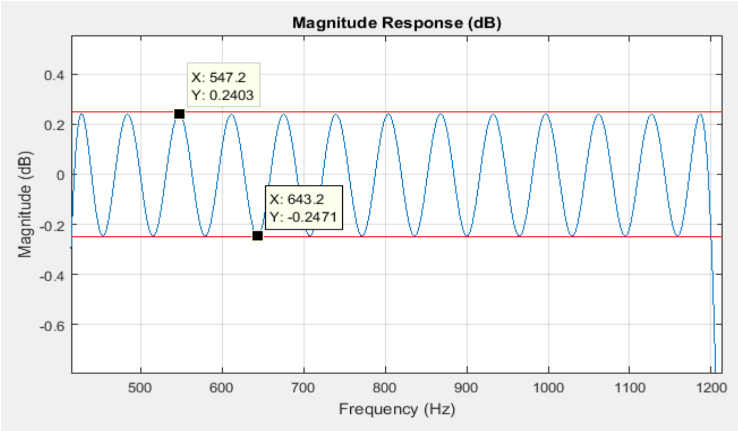

Figure 14: Passband Gain Deviation

The following plots have been repeated for the new 258th order filter for completion.

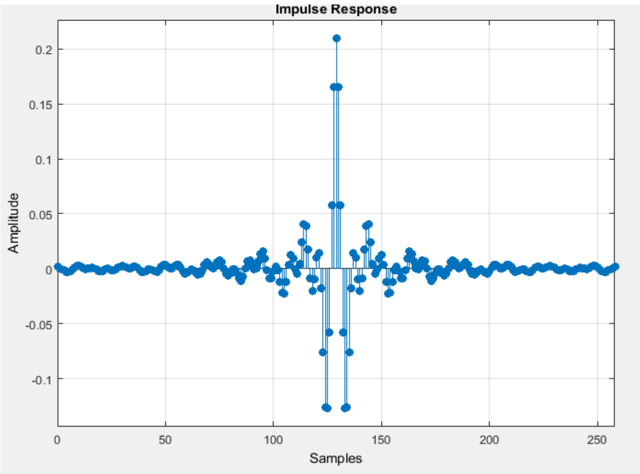

Figure 15: Impulse Response

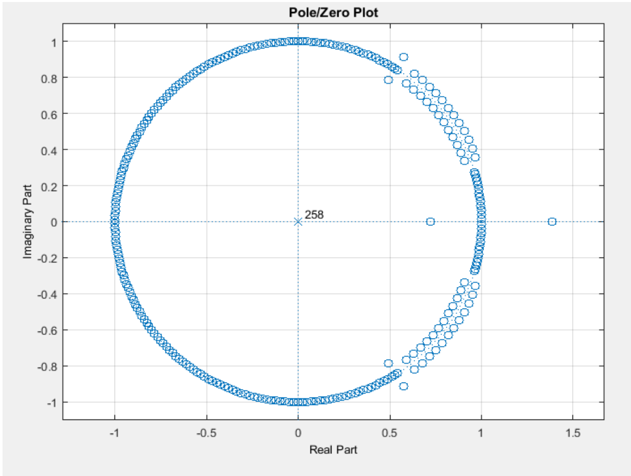

Figure 16: Pole-Zero Plot

The new group delay is,

$$\frac{M-1}{2} = \frac{259-1}{2} = 129$$

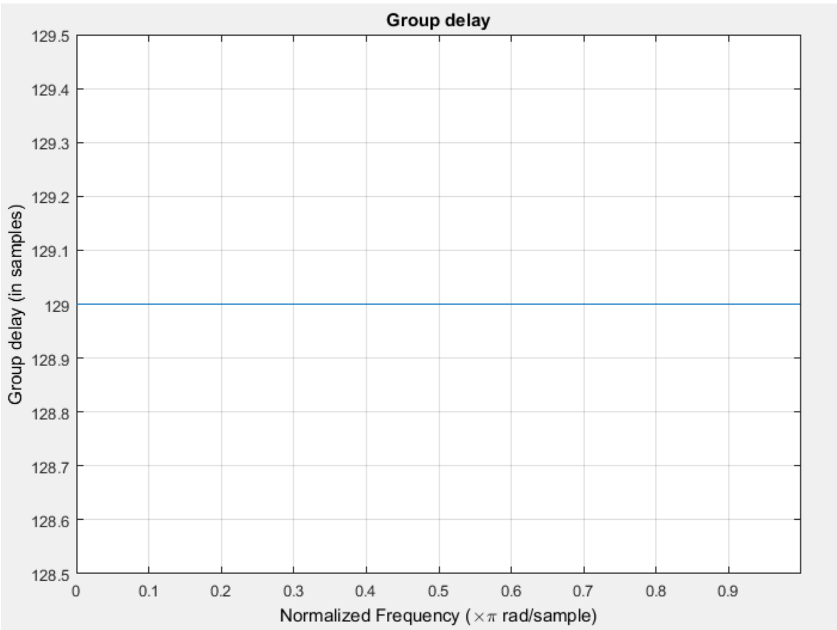

Figure 17: Group Delay

2. Filter Implementation

Using the new Matlab designed 259 filter coefficients, the filter has been implemented via non-circular and circular buffer methods. Five different functions have been created and will be discussed in terms of their execution times. Before explaining how each function is implemented, the code outside the filter functions will be discussed briefly.

The global variables have been initialised and the filter coefficients from Matlab included as follows.

```
// define a 259 element delay buffer
#define N 259
// global variables
short x[N] = {0};
short X[2*N] = {0};
int index = N-1;
//include filter coefficients
#include "fir_coefs259.txt"
```

Listing 2

Each time the interrupt service routine is called, a sample is read from the ADC of the audio chip via **mono_read_16Bit**. This sample is then filtered using either non-circular or circular buffer methods and is written to the DAC via **mono_write_16Bits**. As these samples must be 16 bits long, all buffer arrays used throughout the code have been declared as ***short***. Moreover, the arrays are declared global such that they do not re-initialise each time the ISR function is called.

```
/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void ISR_AIC(void)
{
    short sample_in;
    double output;

    sample_in = mono_read_16Bit();

    output = non_circ_FIR(sample_in);

    mono_write_16Bit((Int16)output); // converts the output to 16 bits long

}
```

Listing 3

Finally, the above breakpoints have been added in order to measure the number of clock cycles each filter implementation function takes.

2.1 A Non-Circular Buffer

The first filter design was realised via a non-circular buffer. The filter is perceived as a 'delay line', where all samples are stored in an array and are shuffled down each time a new sample arrives. In other words, the array is buffered via a FIFO method (first element in, first element out). The following **non_circ_FIR** function implements the buffer system in addition to convoluting the samples with the appropriate filter coefficients.

```
/******************** Non-Circ Convolution Function ***********************/
double non_circ_FIR(short sample)
{
    int i;
    int j;
    double sum = 0;

    for (i = N-1; i>0; i--)
    {
        //shift elements down the buffer
        x[i] = x[i-1];
    }
    //store new sample at the beginning
    x[0] = sample;

    for (j = 0; j<N; j++)
    {
        //convolution
        sum += h[j]*x[j];
    }

    return sum;
}
```

Listing 4

Each time a new sample arrives, the elements in the array ***x*** are shifted to the right using a ***for*** loop. Once shifted, the new sample is then stored in the first element of the array, $\boldsymbol{x}[\mathbf{0}]$. Convolution occurs in the second ***for*** loop, in which convolution is defined as the following MAC (multiply accumulate operation) equation.

$$\sum_{k=0}^{N-1} h[k]x[n-k]$$

The result is stored in the variable ***sum***, which is then returned to the **ISR_AIC** function to be written to the DAC.

2.2 Improving The Performance Using The Compiler

In order to evaluate the performance of the **non_circ_FIR** function, the number of clock cycles were measured, firstly with no optimisation and then with optimisation levels –o0 and –o2.

Before looking at the results of the different optimisation levels, a better understanding of each level is conducted [1].

- ***-o0:***
    - Performs control-flow-graph simplification
    - Allocates variables to registers
    - Performs loop rotation
    - Eliminates unused code
    - Simplifies expressions and statements
    - Expands calls to functions declared inline

- ***-o2:***
    - Performs software pipelining
    - Performs loop optimizations
    - Eliminates global common subexpressions
    - Eliminates global unused assignments
    - Converts array references in loops to incremented pointer form
    - Performs loop unrolling

At optimisation level -o0, the expressions and statements in the loop are simplified to reduce the complexity. Whereas at optimisation level -o2, loop unrolling and software pipelining further reduces the execution time by decreasing the number of times the loop condition is tested and carrying out multiple iterations in parallel.

Based on the results measured, optimisation level –o2 had significantly increased the speed of the function.

Optimisation Level	**Number of Clock Cycles**
None	16,644
-o0	13,511
-o2	4,220

Table 2:Execution Time using Non Circular Buffering

2.3 Scope Traces

In order to verify that the FIR filter has been correctly generated, output signals were observed and compared their subsequent input signals using the oscilloscope.

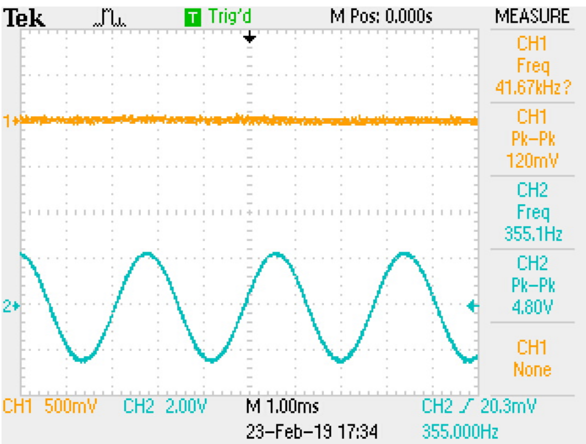

Figure 18: 355 Hz Input Frequency

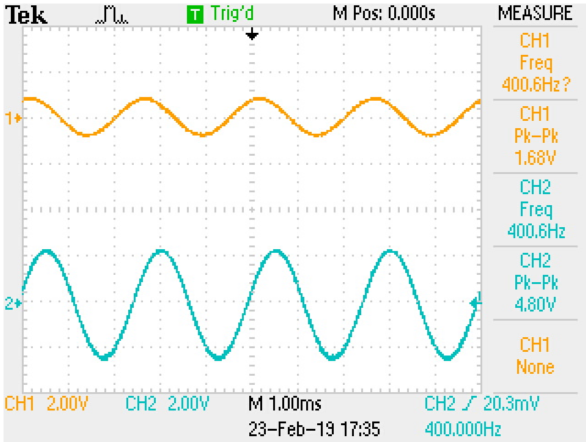

Figure 19: 400 Hz Input Frequency

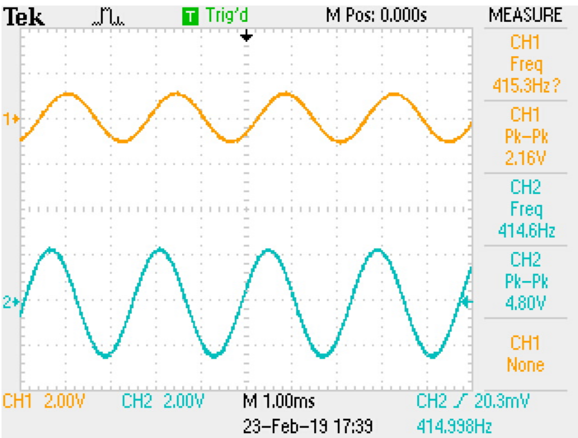

Figure 20: 415 Hz Input Frequency

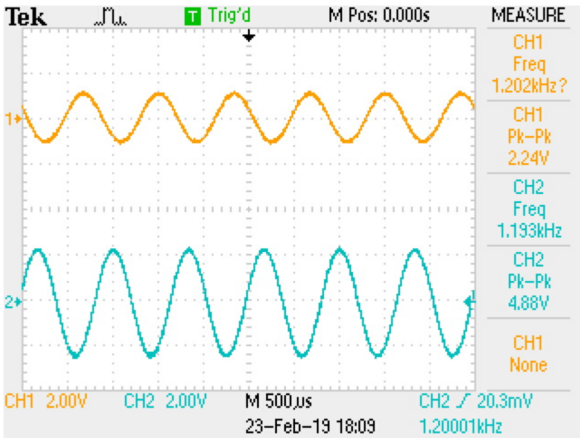

Figure 21: 1200 Hz Input Frequency

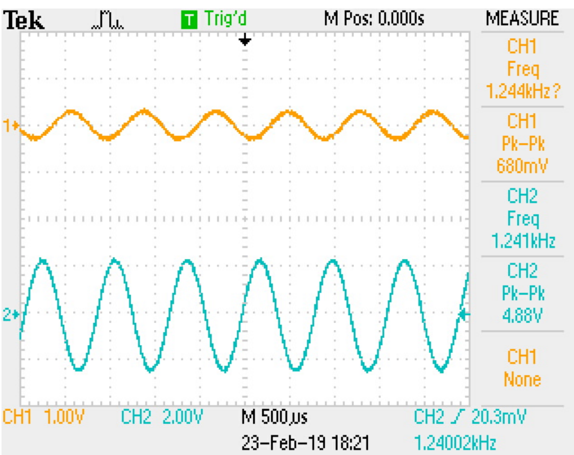

Figure 22: 1245 Hz Input Frequency

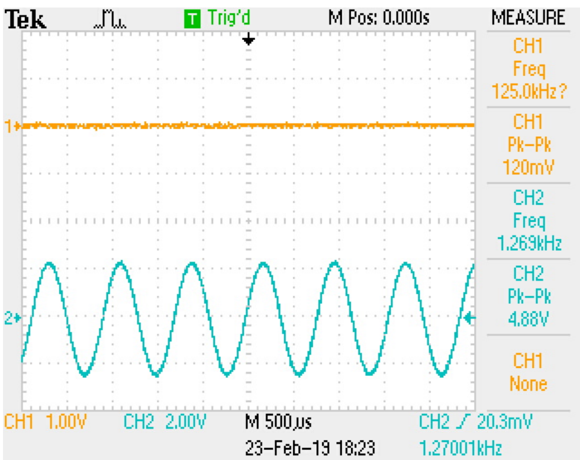

Figure 23: 1270 Hz Input Frequency

As desired, no signals were observed below $355\ Hz$ and above $1270\ Hz$. Furthermore, signals in the middle of the transition band were measured with the correct frequencies, however were attenuated. One unexpected finding was that signals in the passband were attenuated by roughly $0.5\ dB$, as illustrated in *Figures 20* and *21*.

This unexpected attenuation is due to the potential divider found at the input of the audio chip. It halves the amplitude of the input signal before reaching the ADC.

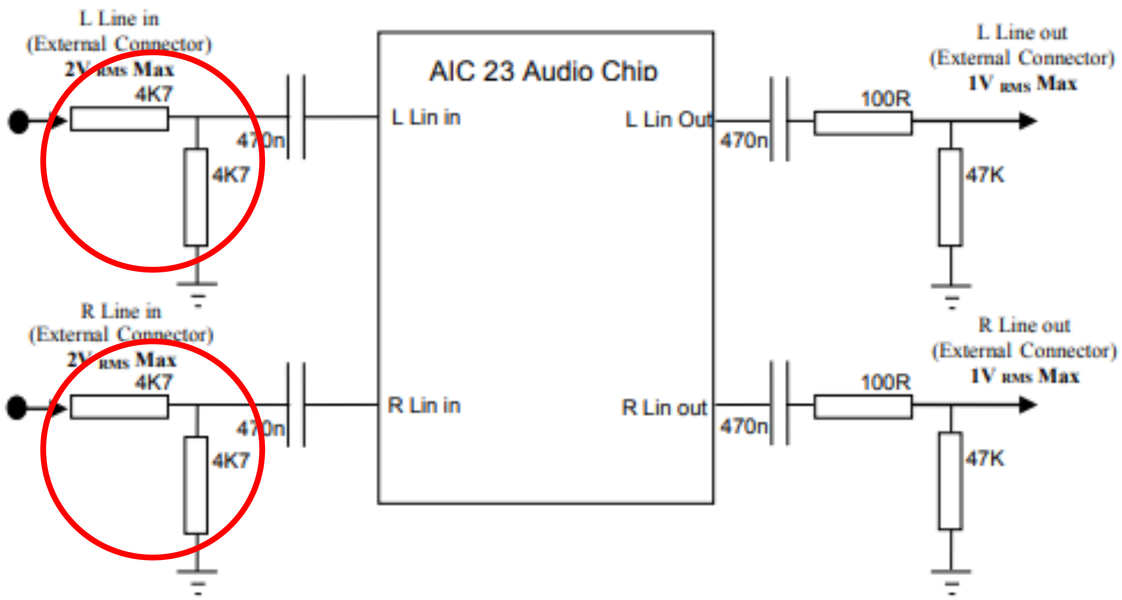

Figure 24: External Connections to the Audio Chip

Another interesting finding observed was there is a phase shift between the input and output signals, i.e. the peaks of the two signals do not coincide. This is expected due to the group delay of the implemented filter. This observation is further discussed in Section 4 of the report.

3. Designing a Faster Filter

3.1 Method 1 – Circular Buffering

As convolution is a critical operation in digital signal processing, the execution time is desired to be as low as possible, which can be achieved using efficient code. The previously used non-circular buffer is not the most efficient method due to the need to copy data from one element location to the next each time a new sample arrives.

Circular buffers can be used to eliminate this shifting process by storing the samples in an array with a size large enough to fit the new sample and all previous samples. In other words, when a new sample is inputted, it will be stored in a new location compared to the previous samples. Once the end of the array has been reached, a wrap-around effect occurs. The oldest sample will now be replaced with the newest sample as it will no longer be needed for following convolution calculations.

The first circular buffer method has been implemented via the following code.

```
/******************** Circ Convolution Function 1***********************/
double circ_FIR_1(short sample)
{
    int i;
    double sum = 0;

    //store new sample in array
    x[index] = sample;

    for(i = 0; i < N; i++){
        //convolution
        //incrementing the position in array after each iteration
        //perform wrap around via modulus function
        sum += h[i]*x[(index+i)%N];
    }

    //decrement index to prepare for next sample
    //ie to store it in a new location in array
    index--;
    //if index is negative
    if(index < 0){
        //wrap around the array
        index = N - 1;
    }
    return sum;
}
```

Listing 5

The global variable ***index*** can be perceived as a 'pointer', mapping the new sample to a new location in the array ***x***. Once the new sample has been stored, a ***for*** loop implements convolution with the according FIR coefficients. Within the loop, the sample locations are based on ***index***, which is incremented by the iterator ***i*** in order to access older samples.

However, as the first sample no longer begins at $\boldsymbol{x[0]}$, there is a risk that the programme may access undefined locations in the buffer. To avoid this hazard, a modulus operation has been applied. This function returns the remainder of the division and thus the correct location to be accessed.

For example, consider the end case when $\boldsymbol{index + i}$ is equal to 259:

$$(index + i)\%N = 259\%259 = 0$$

Therefore 0 is returned, remaining within the bounds of the buffer array.

Once convolution has ended, ***index*** is decremented in order to prepare for the new sample. When ***index*** becomes a negative number, it is set back to $N - 1$, avoiding any accessing of undefined locations in $\boldsymbol{x}$.

The function has the following clock cycle measurements, which have been found to be longer than the non-circular buffer implementation. This is believed to be due to the modulus operation, which comprises of three arithmetic operations: division, multiplication and subtraction.

$$A \,\%\, B = A - floor(A/B) * B$$

The amount of time needed to compute the result from a modulo is therefore longer than loading/storing data during shifting.

Optimization Level	Number of Cycles
None	18,232
-o0	16,133
-o2	13,533

Table 3: Execution Time using a Circular Buffer and a Modulus Function

3.2 Method 2 – Replacing Arithmetic Operation with Branching

In order to speed up the function's execution time, the modulus operation has been removed and replaced with an ***if*** statement.

```
/******************** Circ Convolution Function 2 ***********************/
double circ_FIR_2(short sample)
{
    int i;
    double sum = 0;
    //store new sample in array
    x[index] = sample;

    for(i = 0; i < N; i++){
        //convolution
        sum += h[i]*x[index];
        index++;    //increment index
        //if index reaches the end of the array
        if(index == N){
            index = 0; //wrap around
        }
    }

    index--;
    if(index < 0){
        index = N - 1; //wrap around
    }
    return sum;
```

Listing 6

Again, the pointer ***index*** is incremented each time the loop repeats. However, instead of adding ***index*** and ***i*** together, ***index*** itself is now incremented. There is no need to worry about losing the original value ***index*** holds as it is returned after exiting the ***for*** loop. An ***if*** condition has been placed to avoid ***index*** from overflowing.

This approach cuts down the number of clock cycles as comparisons can be done in parallel whereas previously taking the modulus with a number that is not a power of two can be a fairly expensive operation.

Optimization Level	Number of Cycles
None	15,116
-o0	13,784
-o2	1,026

Table 4: Execution Time using Circular Buffering and If Statements

3.3 Method 3 – Exploiting Symmetry

As discussed in Section 1.1, the impulse response of the FIR filter is symmetric, resulting in symmetrical coefficients. Considering this property, the ***for*** loop can be reduced by half, reducing the function's number of clock cycles.

In order to find a pattern between the array locations of the symmetric coefficients, convolution of an array of size 5 has been considered.

index	**h[0]**	**h[1]**	**h[2]**	**h[3]**	**h[4]**
4	x[4]	x[0]	x[1]	x[2]	x[3]
3	x[3]	x[4]	x[0]	x[1]	x[2]
2	x[2]	x[3]	x[4]	x[0]	x[1]
1	x[1]	x[2]	x[3]	x[4]	x[0]
0	x[0]	x[1]	x[2]	x[3]	x[4]

Due to the symmetry property of the coefficients, $h[3] = h[1]$ and $h[4] = h[0]$. The common terms can be grouped together to produce the following table. As the filter has an odd number of coefficients, there will always be an odd one out which lies in the middle of the array.

index	**h[0]**	**h[1]**	**h[2]**
4	x[4] + x[3]	x[0] + x[2]	x[1]
3	x[3] + x[2]	x[4] + x[1]	x[0]
2	x[2] + x[1]	x[3] + x[0]	x[4]
1	x[1] + x[0]	x[2] + x[4]	x[3]
0	x[0] + x[4]	x[1] + x[3]	x[2]

The same method of convolution in a ***for*** loop will be implemented, however, since there are now two coefficients of equal values, two pointers will be used to access them simultaneously. The pointers motions is imagined as one travelling up the array, whilst the other travels down, retrieving the symmetrical coefficients.

As an example, refer to the values in the first column and row in the table: ***ind_1*** will point to $\boldsymbol{x}[\mathbf{4}]$ whilst ***ind_2*** will point to $\boldsymbol{x}[\mathbf{3}]$**.** Throughout the convolution ***for*** loop, ***ind_1*** will be incremented and ***ind_2*** will be decremented.

Referring to the pattern found in the table, the code in *Listing 7* has been split into three different ***if else*** statements:

1. **Index lies in the upper half of the array:**
   Overflowing will only occur via ***x[ind_1]*** hence only one ***if*** statement exists within the ***for*** loop.

2. **Index lies in the lower half of the array:**
   Overflowing will only occur via ***x[ind_2]*** hence only one ***if*** statement exists within the ***for*** loop.

3. **Index lies in the middle of the array:**
   No overflow will occur hence no ***if*** statement exists.

If they had not been split in such a manner, both overflow scenarios would need to be checked in every loop of the ***for*** loop. Therefore, splitting it in such a way, and reducing the for loops by a half, has led to lower clock cycle measurements.

Optimization Level	**Number of Cycles**
None	9,944
-o0	6,415
-o2	669

Table 5: Execution Time Using Circular Buffering and Symmetry of Coefficients

```
/******************** Circ Convolution Function 3 ***********************/
double circ_FIR_3(short sample){
    int ind_1 = index; //incrementing pointer
    int ind_2 = index-1; //decrementing pointer
    int i;
    double sum = 0;

    x[index] = sample; //store sample

    if(index > (N-1)/2){//if index belongs in upper half of array
        for(i = 0; i < (N-1)/2; i++){
            if(ind_1 > N-1){//check for overflow
                ind_1 = 0;
            }
            //perform convolution using symmetry property
            sum += h[i]*(x[ind_1] + x[ind_2]);
            ind_1++;
            ind_2--;
        }
        //middle case
        sum += h[(N-1)/2]*x[ind_1];
    }
    else if(index < (N-1)/2){//if index belongs in lower half of array
        for(i = 0; i < (N-1)/2; i++){
            if(ind_2 < 0){//check for overflow
                ind_2 = N-1;
            }
            sum += h[i]*(x[ind_1] + x[ind_2]);
            ind_1++;
            ind_2--;
        }
        //middle case
        sum += h[(N-1)/2]*x[ind_1];
    }
    else{//if index belongs in center of array
        for(i = 0; i < (N-1)/2; i++){
            sum += h[i]*(x[ind_1] + x[ind_2]); //no overflow checks
            ind_1++;
            ind_2--;
        }
        //middle case
        sum += h[(N-1)/2]*x[ind_1];
    }

    index--;
    if(index < 0){
        index = N - 1; //overflow check
    }
    return sum;
}
```

Listing 7

3.4 METHOD 4 – DOUBLING THE BUFFER

The execution time of the function can be further reduced by doubling the size of the array in which the samples are stored.

```
/******************** Circ Convolution Function 4 ***********************/
double circ_FIR_4(short sample){
    int ind_1 = index;
    int ind_2 = index-1+N;
    int i;
    double sum = 0;

    X[index] = sample;  //store sample in 'first' array
    X[index+N] = sample;//store sample in 'second' array

    for(i = 0; i < (N-1)/2; i++){
        //convolution using symmetry
        sum += h[i]*(X[ind_1] + X[ind_2]);
        ind_1++;
        ind_2--;
    }
    //middle case
    sum += h[(N-1)/2]*X[ind_1];

    index--;
    if(index < 0){
        index = N - 1; //overflow prevention
    }
    return sum;
}
```

Listing 8

Two sets of samples will now be present in the array, allowing the pointer ***ind_1*** to access locations previously described as 'undefined', i.e. in the case of a 4^{h} order filter: $\boldsymbol{x}[5]$, $\boldsymbol{x}[6]$ etc. Moreover, by initialising ***ind_2*** to begin at the end of the array, no overflow will occur as it decrements within the ***for*** loop.

In other words, all overflow scenarios have been avoided and hence has reduced the number of ***if*** statements and thus the number of branching calculations and clock cycles.

Optimization Level	Number of Cycles
None	8,504
-o0	5.375
-o2	585

Table 6: Execution Times using a Double Array

4. Measurement of Filter Response

The frequency response of the latter, fastest filter implementation has been found using the spectrum analyser.

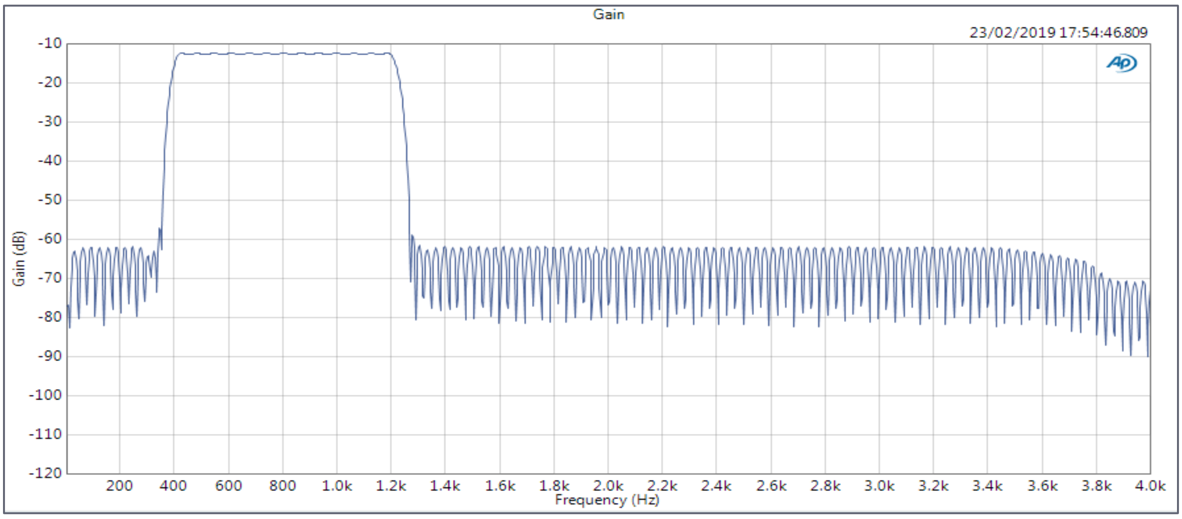

Figure 25: Magnitude Response

The gain in the passband is not the expected $0\ dB$ but lies around $-12\ dB$. This is due to a gain factor of $\frac{1}{4}$ found between the spectrum analyser and the input of the codec, as shown below in *Figure 26*. This has been implemented as a safety precaution, preventing signals of significant amplitudes damaging the DSK unit.

$$20\log\left(\frac{1}{4}\right) \approx -12.04$$

The magnitude response can thus be perceived to have been shifted down by $12\ dB$. If this 'offset' didn't exist, the gain at the passband would be $0\ dB$ and the stopband would have a maximum gain at $-50\ dB$.

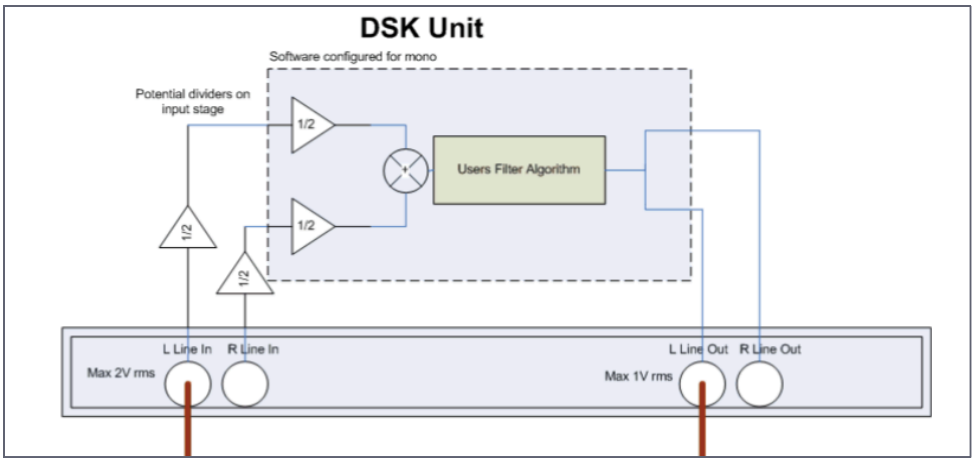

Figure 26: Connections Between the Spectrum Analyser and DSK

Furthermore, notice how the gain begins to roll as $4\ kHz$ is approached. This is due to the low pass filter at the output of the audio chip that has a cut off frequency of $4\ kHz$.

To check if the filter has met the specifications and the predictions on Matlab, the gain difference between the transition band was measured, as shown below. It was found to be $-43.456\ dB$, which does not match the specification of $-48\ dB$.

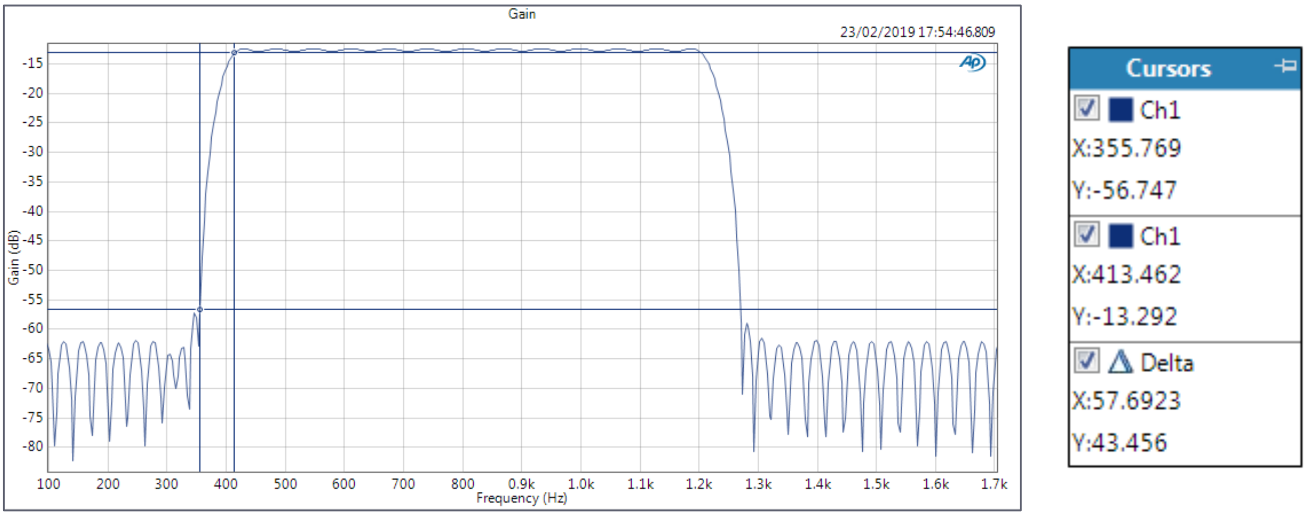

Figure 27: High Pass Transition Band

On the other hand, the passband ripple does satisfy the $0.5\ dB$ deviation limit. It was measured as $0.468\ dB$.

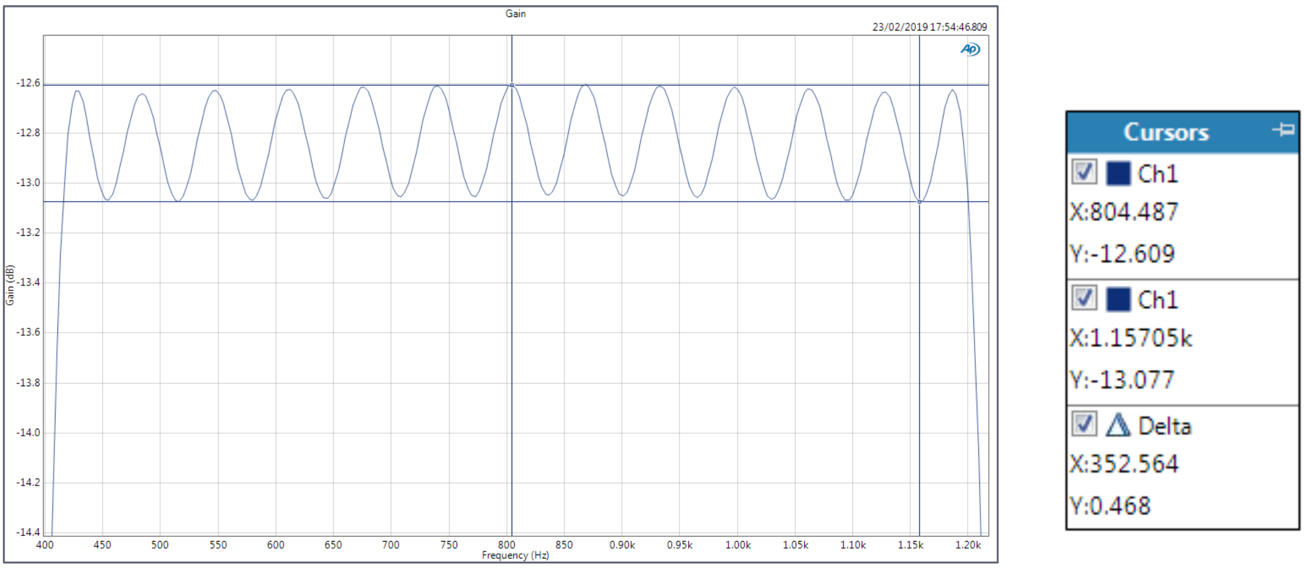

Figure 28: Passband Deviation

Figure 29 illustrates the measured phase response of the system. A linear phase response is found within the passband and is more clearly shown in *Figure 30*. However, unlike the Matlab plots, the phase does not flatten out in the stopband.

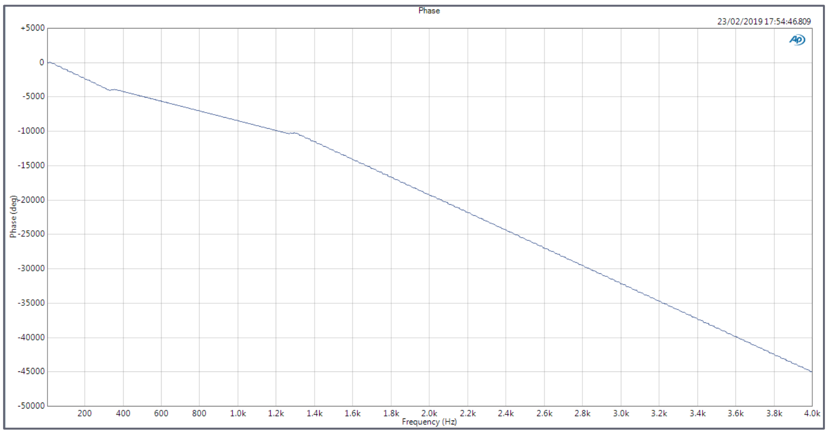

Figure 29: Phase Response

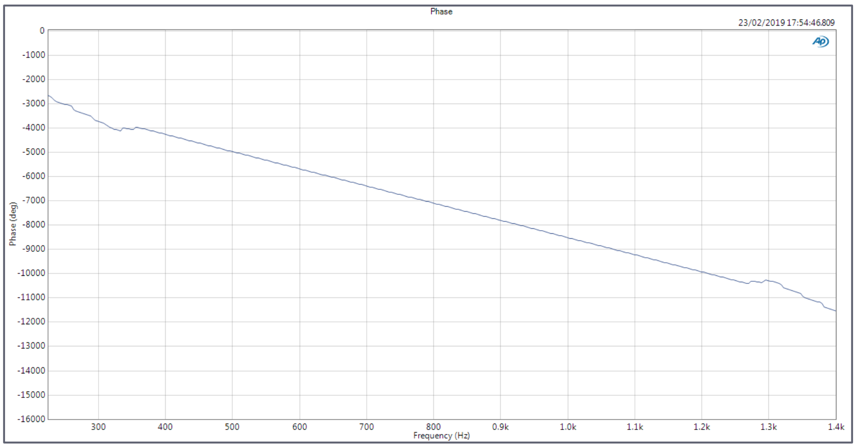

Figure 30: Zoomed in Phase Response

In order to understand the decreasing phase response in the stopband, the phase response excluding the FIR filtering has been additional measured and is shown below in *Figure 31*.

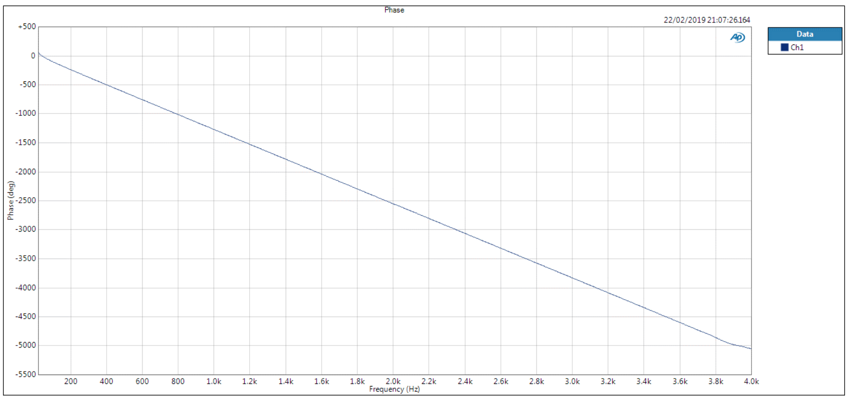

Figure 31: Unfiltered Phase Response

The group delay of the system is the reason for this linearly decreasing phase response. As discussed previously, the group delay is calculated as the derivative of the phase, hence a constant group delay results in a linear phase response. *Figure 32* shows the group delay of the unfiltered system.

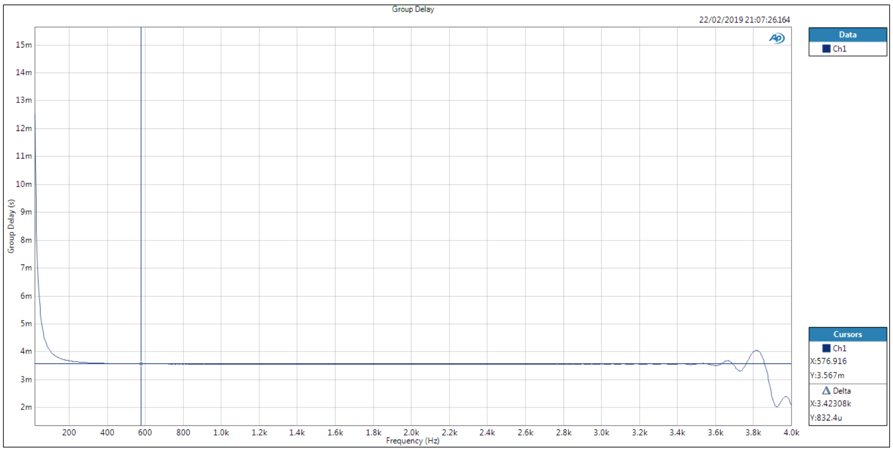

Figure 32: Group Delay of Unfiltered System

The delay was measured to be around 3.567×10^{-3} and is believed to be caused by the circuitry within overall system. Matlab does not take these parasitic delays into account when measuring the phase response and hence assumed there was 0 delay in the stopband and thus a constant phase.

The group delay of the filtered system was then measured for comparison (*Figure 33*).

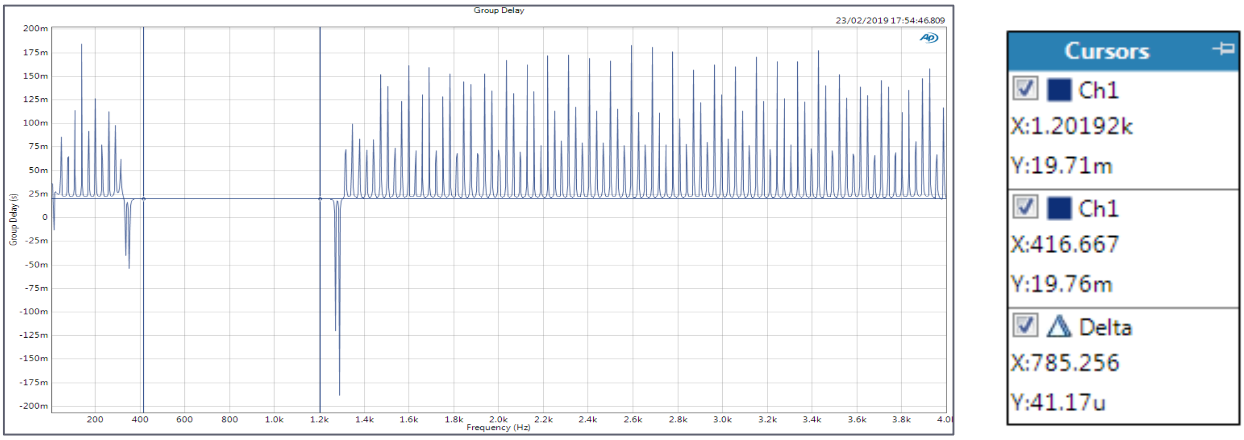

Figure 33: Group Delay of Filtered System

The expected group delay of the filter was previously measured in Section 1.2 to be 129. In the time domain, this corresponds to a value of,

$$\frac{129}{f_{samlping}} = \frac{129}{8000} = 16.125 \times 10^{-3}$$

However, due to the extra delay previous discussed, it does not correspond to the predicted value and has been measured to be 19.76×10^{-3}.

It can be concluded that even more filter coefficients would be needed to meet the desired specifications.

5. REFERENCES

[1] http://www.ti.com/lit/ug/spru187u/spru187u.pdf