RTDSP Lab5 Report

Calvin Chan (CID: 01048905)

Zheyuan Li (CID: 01181358)

Imperial College London

EE3-19: Real-Time Digital Signal Processing (2018-2019)

Content

1 Simple-pole IIR Filter

1.1 Design

1.1.1 Continuous-time/Analogue Filter

In this lab, we are asked to design a simple-pole digital IIR filter. The corresponding analogue filter is given in the lab instruction shown in Figure 1 below:

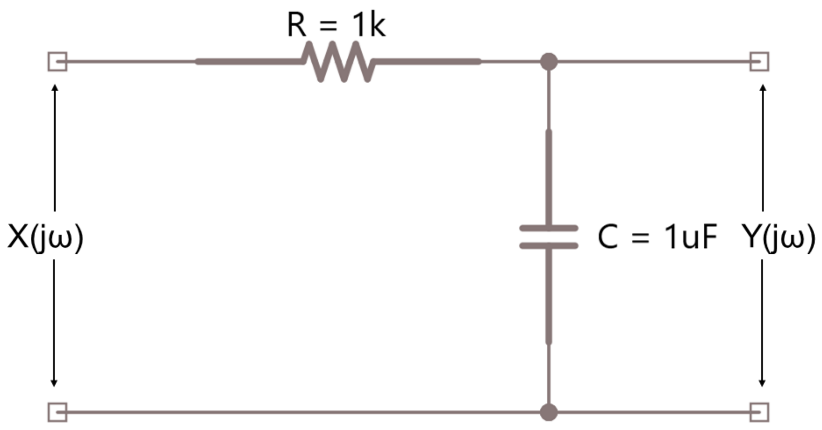

Figure 1: RC circuit diagram

Where the Y(jω) being the output signal and X(jω) being the input signal.

From analogue circuit analysis, we can easily calculate the transfer function H(jω) of this system:

$$H(j\omega) = \frac{Y(j\omega)}{X(j\omega)} = \frac{\frac{1}{j\omega C}}{\frac{1}{j\omega C} + R} = \frac{1}{1 + j\omega RC}.$$

This RC circuit is actually a low-pass analogue filter with corner frequency $\omega_c = 1/RC = 1k$ rad/s. And the time constant $\tau = RC = 1$ ms. The ω_c is the angular frequency and the linear cutoff frequency is $f_c = \omega_c/(2\pi) \approx 159.2$ Hz. The cutoff frequency here means the frequency at which the magnitude of output signal is -3dB from 0dB. The frequency response of this low-pass analogue filter is shown in Figure 2.

Then by using Laplace transform, the transfer function H(jω) is converted into H(s):

$$H(s) = \frac{1}{1 + sRC}$$

where s = jω. This system has a simple pole at -1/RC which is -1000 rad/s and the zero-pole plot is shown in Figure 3.

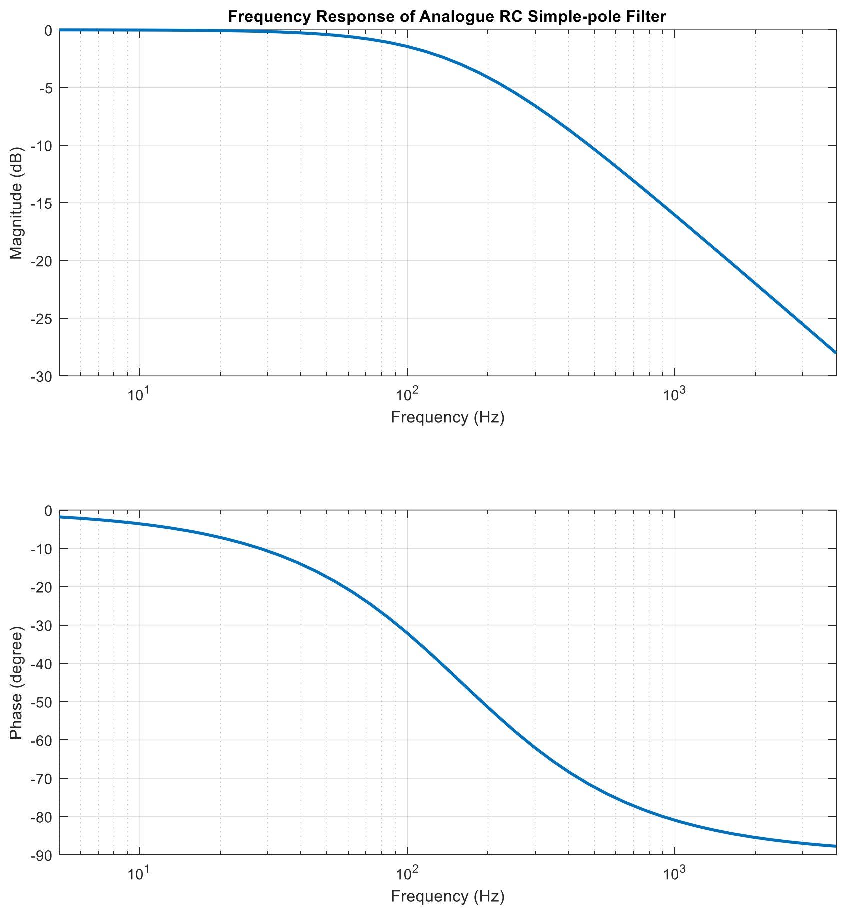

Figure 2: Frequency response of RC analogue filter

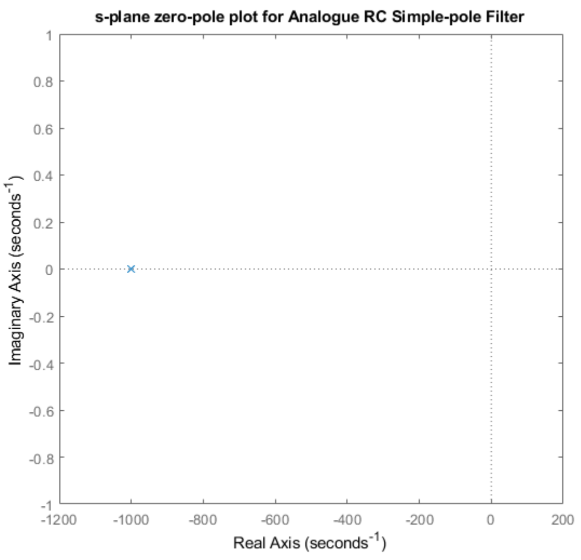

Figure 3: s-plane for the RC analogue filter

1.1.2 Discrete-time/Digital Filter

However, the system above is analogue and is in continuous-time domain. We need to somehow convert it into discrete-time domain in order to build a digital filter. We know that in z-domain, a unit of delay is represented by z^{-1}. And in s-domain, using shift in time property, a unit of delay can be implemented by multiplying e^{-sT} where T represents the sampling period, i.e. $T = 1/f_s$.

If we equate z^{-1} with e^{-sT}, we would have:

$$z = e^{sT} \, .$$

And then:

$$\begin{aligned} \ln z &= \ln e^{sT} = sT \\ s &= \frac{1}{T}\ln z \, . \qquad (*) \end{aligned}$$

We could expand ln(z) by using Tayler Series. If we let:

$$z = \frac{1+x}{1-x} \, ,$$

so that:

$$x = \frac{z-1}{z+1} .$$

And think of:

$$\begin{aligned} \ln(1+x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots \\ \ln(1-x) &= -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \cdots \, . \end{aligned}$$

Combine them:

$$\begin{aligned} \ln\frac{1+x}{1-x} &= \ln(1+x) - \ln(1-x) \\ &= \left(x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots\right) - \left(-x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \cdots\right) \\ &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots + x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \frac{x^5}{5} + \cdots \\ &= 2x + \frac{2x^3}{3} + \frac{2x^5}{5} + \frac{2x^7}{7} + \cdots \\ &= 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots\right) . \end{aligned}$$

Replacing x by z:

$$\begin{aligned}\ln z &= 2\left(\frac{z-1}{z+1} + \left(\frac{z-1}{z+1}\right)^3 + \left(\frac{z-1}{z+1}\right)^5 + \left(\frac{z-1}{z+1}\right)^7 + \cdots\right) \\ &\approx 2\frac{(z-1)}{z+1}.\end{aligned}$$

Continue with (*), replacing ln(z):

$$s \approx \frac{2}{T}\cdot\frac{z-1}{z+1}\ .$$

This is the **Tustin Transform**, can be used to convert the transfer function H(s) from s-domain:

$$H(s) = \frac{1}{1+sRC}$$

into z-domain:

$$\begin{aligned}H(z) &= \frac{1}{1+\frac{2}{T}\frac{z-1}{z+1}RC} \\ &= \frac{T(z+1)}{T(z+1)+2(z-1)RC} \\ &= \frac{Tz+T}{(T+2RC)z+(T-2RC)} \\ &= \frac{\frac{T}{T+2RC}+\frac{T}{T+2RC}z^{-1}}{1+\frac{T-2RC}{T+2RC}z^{-1}}.\end{aligned}$$

We know that T is the sampling period which is 1/8000s (8000 is the sampling frequency f_s), RC is equivalent to the time constant τ which is 0.001s.

Substituting all the symbols by their corresponding values, we get:

$$H(z) = \frac{\frac{1}{17}+\frac{1}{17}z^{-1}}{1-\frac{15}{17}z^{-1}} = \frac{Y(z)}{X(z)}.$$

Then we can derive the **difference equation** for this digital filter:

$$\begin{aligned}y(n) - \frac{15}{17}y(n-1) &= \frac{1}{17}x(n) + \frac{1}{17}x(n-1) \\ y(n) &= \frac{1}{17}x(n) + \frac{1}{17}x(n-1) - \left(-\frac{15}{17}\right)y(n-1) \qquad (**)\end{aligned}$$

From the equation (**), we can deduce that this digital filter is a causal IIR filter. It is causal because the output y(n) only depends on the current and past input or output signals. It is an IIR filter rather than an FIR filter because the output y(n) does not only depend on input signals, but also output signals.

Generally, an IIR filter would have the following difference equation:

$$\begin{aligned} y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \cdots + b_N (n-N) \\ -a_1 y(n-1) - a_2 y(n-2) - \cdots - a_N y(n-N) \end{aligned}$$

where N is the order of the IIR filter.

In this simple filter, the coefficients can be easily seen from equation (**):

$$\begin{cases} b_0 = \dfrac{1}{17} = 5.8823529411764705\text{e-}02 \ldots \\ b_1 = \dfrac{1}{17} = 5.8823529411764705\text{e-}02 \ldots \\ a_1 = -\dfrac{15}{17} = \text{-}8.8235294117647056\text{e-}01 \ldots. \end{cases}$$

We convert the coefficient to be double floating-point numbers, because later in C we cannot simply use fraction numbers in the array in the declaration as C would not recognize them.

1.2 Implementation in C

1.2.1 In the declaration

```
double b[2] = {5.8823529411764705e-02, 5.8823529411764705e-02};
double a[2] = {1, -8.8235294117647056e-01};    //the a[0] will be never used
Int16 x[2] = {0};    //input samples
double y[2] = {0};    //ouput samples
```

There are 2 sets of coefficients. “a[2]” is the array of coefficients of y (output), “b[2]” is the array of coefficients of x (input). Although only the coefficient “a[1]” will be used later, we still let the size of “a” to be 2 because array index starts from 0 in C, and we do not want the index to be misleading. In the array declaration, we use double floating-point format as it is the required precision and the format that can be recognized by C.

The x[2] is a 16bits signed integer array because x is the input sample array which would be read from the H_Codec by using function “mono_read_16Bit”. The function reads samples from the R and L channel of the input 3.5mm audio jack, averages them and returns a 16bits signed integer. The y[2] is the output sample array and contains double floating-point numbers to retain the accuracy of the calculated output samples. We initialize both y and x to be 0 in order to prevent uncertainties.

1.2.2 In the ISR

```
void ISR_AIC(void)
{
	Int16 samp_out = 0;			// declaration of local variable

	simple_IIR();				// call the function simple_IIR
	samp_out = round(y[0]);			// convert double to Int16
	mono_write_16Bit(samp_out);		// output the rounded samp_out
}
```

Firstly, we define a local variable called “samp_out” being the output sample after calculation. Then the function “simple_IIR()” is called, which is the actual implementation function of the filter design. The y[0] which is the non-delayed output sample, is a double floating-point number, but the output sample is required to be a 16bits signed integer. So the rounding value of y[0] is taken an stored in samp_out. After that, the samp_out is passed to the function mono_write_16Bit.

1.2.3 In the filter implementation function

```
void simple_IIR(void)
{
	// perform delay operation
	x[1] = x[0];
	y[1] = y[0];

	// read new samples from H_Codec
	x[0] = mono_read_16Bit();

	// difference equation
	y[0] = b[0]*x[0] + b[1]*x[1] - a[1]*y[1];
}
```

Implementing the simple-pole IIR filter in C is quite straightforward. The order of this filter is 1 and it is not necessary to implement it using a loop as it increases the clock cycles and slows the program down. We directly used the difference equation instead.

Here the x[0] and y[0] means the current or non-delayed input and output sample, the x[1] and y[1] means the input and output sample delayed by one sample or a unit of time, which is the sampling period T[1].

Actually it does not matter when the delay operation “x[1] = x[0]; y[1] = y[0];” is performed, before or after the calculation of the difference equation, as long as the x[0] gets a new sample before the difference equation. The y[0] would be non-zero only if x[0] is non-zero. The difference equation in the code is the same as we calculated in equation (**) in the very end of Section 1.1.2.

[1] $T = 1/f_s$, where f_s is the sampling frequency 8kHz.

1.3 Frequency Response

1.3.1 Amplitude Response

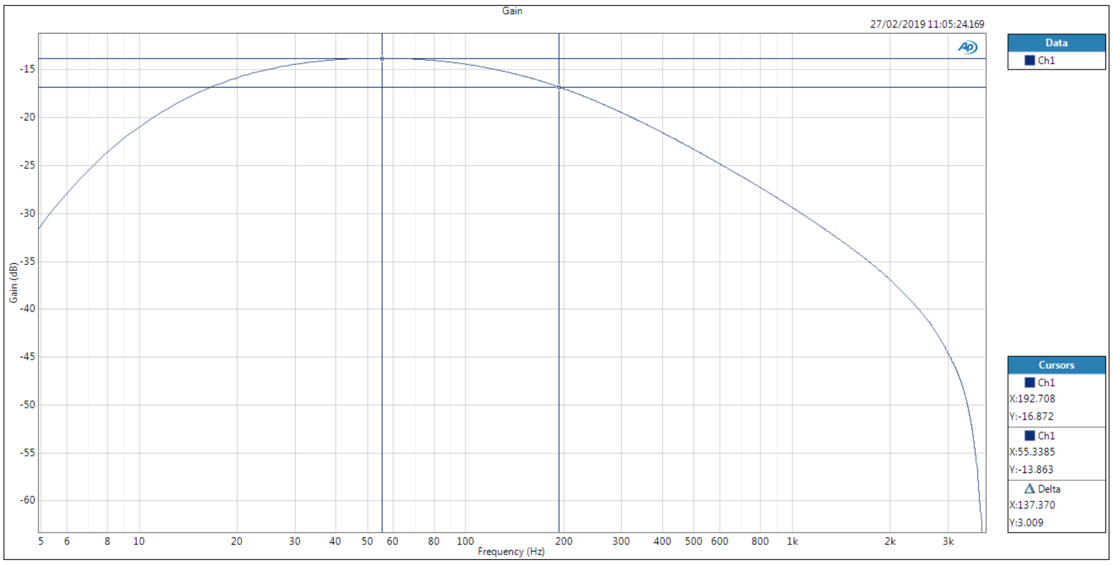

Figure 4: Unprocessed amplitude response from APx500 Audio Analyzer

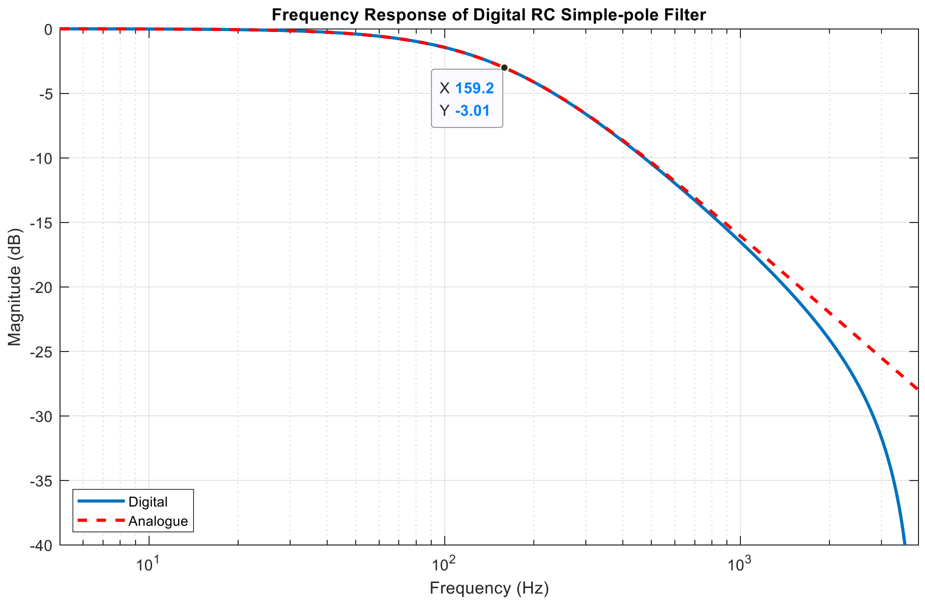

Figure 5: Amplitude Response of Theoretical Digital RC filter

Figure 4 shows the actual unprocessed plot from the APx500 Audio Analyzer. It looks quite different from the analogue RC frequency response in Figure 5. The biggest difference is that there is a clear attenuation in amplitude as the frequency gets very low (<50Hz). This is because there is a built in high-pass filter in the DSK board, illustrated in Figure 6. Also, the amplitude responses of both the Audio Analyzer and theoretical digital filter deviate from the amplitude response of the analogue filter when the frequency is above 1kHz, this is due to **frequency warping** and will be discussed later.

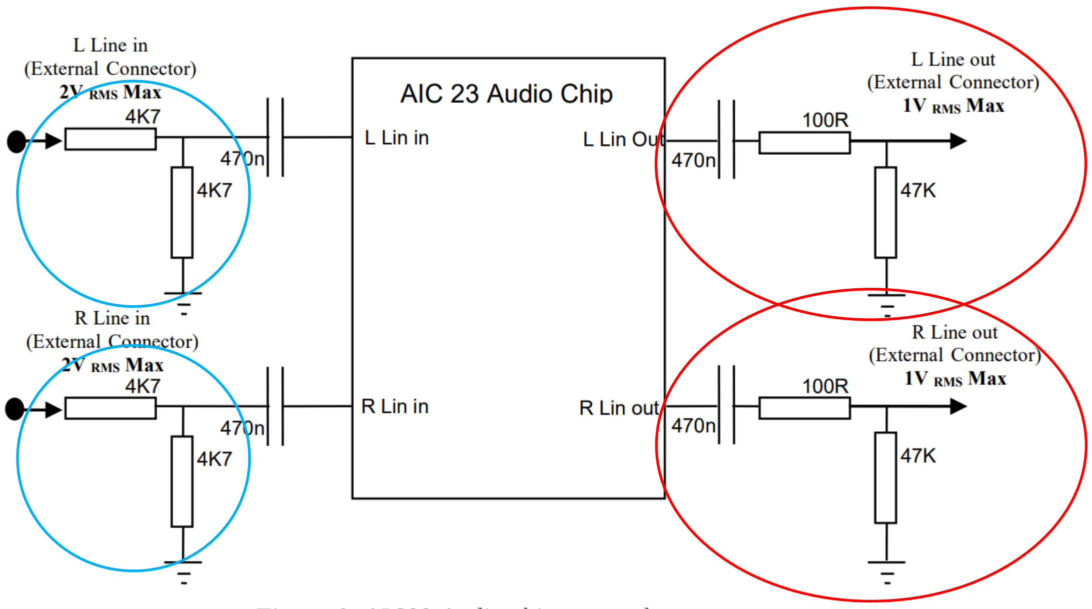

Figure 6: AIC23 Audio chip external components
(adapted from TMS320C6713 Technical ref (page A-14, 2003 revision A)

Each of the L and R "lin out" in the audio chip is followed by a capacitor, a 100Ω resistor and a 47kΩ resistor connecting to the ground labelled in red circles shown in Figure 6. This composition of analogue circuit components forms a high-pass filter. If we let X(jω) be the analogue signal directly from the AIC23 Audio Chip, and Y(jω) be the actual analogue signal sent to the external connector, and look at the transfer function:

$$H(j\omega) = \frac{47k}{\frac{1}{j\omega C} + 100 + 47K} = \frac{47kj\omega C}{47.1kj\omega C + 1} = \frac{0.02209j\omega}{0.022137j\omega + 1}.$$

Then we discover that the corner frequency of this high-pass filter whose corner frequency is around 45.2 rad/s (7.19Hz). This important factor contributes the rapid attenuation in low frequency at the output of the DSK board.

Besides, we already found that there is an overall attenuation of approximately -12dB early in Lab 4 when we analysed the frequency response of FIR filters. This is again due to the attenuations before the samples manage to get into the DSK board:

1. There are two resistors forming a potential divider at each of L and R "Line in" stage labelled in blue circles in Figure 6. This factor causes a ½ attenuation of the input sample.
2. In the ISR, we used mono_read_16Bit() function which does an average of the L and R channels, causing another ½ attenuation.

These two factors combine and get us ¼ attenuation in total. Converting ¼ into dB we can get -12.04dB of attenuation. Figure 7 gives us a better clue of the location of the attenuations happening.

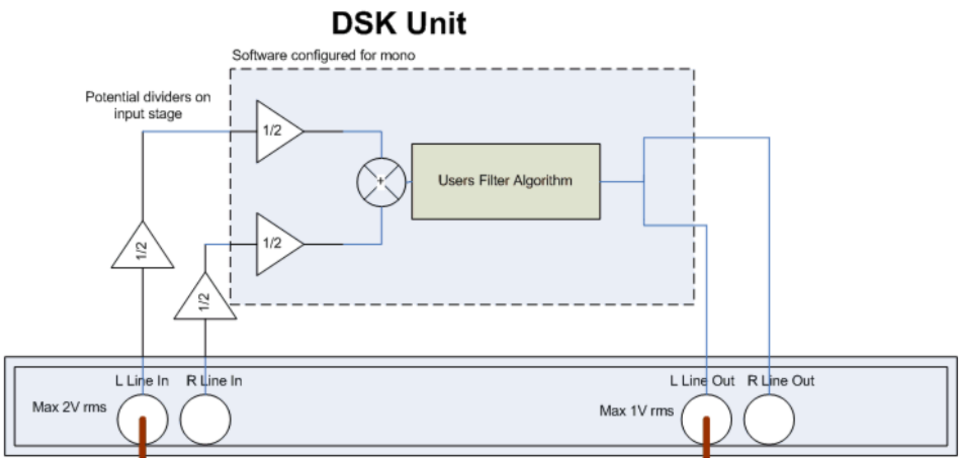

Figure 7: DSK Unit and APx500 Audio Analyzer

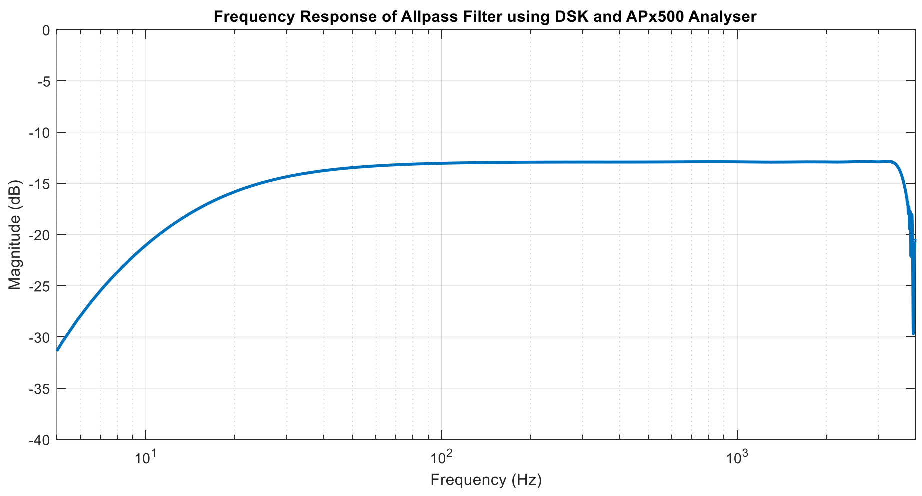

Figure 8: Amplitude response of all-pass filter in DSK

To counteract the effect of the low frequency attenuation and the overall -12.04dB attenuation, we can design an all-pass filter, which has the amplitude response in Figure 8. Then we deduct the frequency response of this all-pass filter from the frequency response of the RC digital filter that we designed, generating a corrected frequency response. Figure 10 in the next page illustrates the difference in amplitude response of filter in DSK (corrected), digital filter in Matlab and analogue RC filter.

We can see that both the DSK filter and the digital filter have a steep attenuation when the frequency is above 1kHz. This is because after Tustin Transform, there is one pole at point (15/17, 0) but also a zero at point (-1, 0). The pole contributes the 0dB gain an the zero contributes the steep attenuation in the amplitude response for frequencies larger than 1kHz. This can also be illustrated in a z-domain zero-pole plot as shown in Figure 9. The presence of zero makes the amplitude response of both digital filter and the filter in DSK behave differently from that of the analogue RC filter. Another explanation of this is frequency warping which will be discussed later.

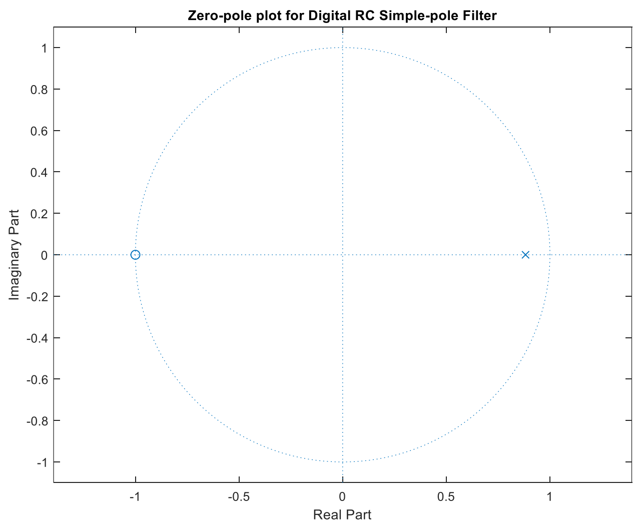

Figure 9: z-domain of the digital filter

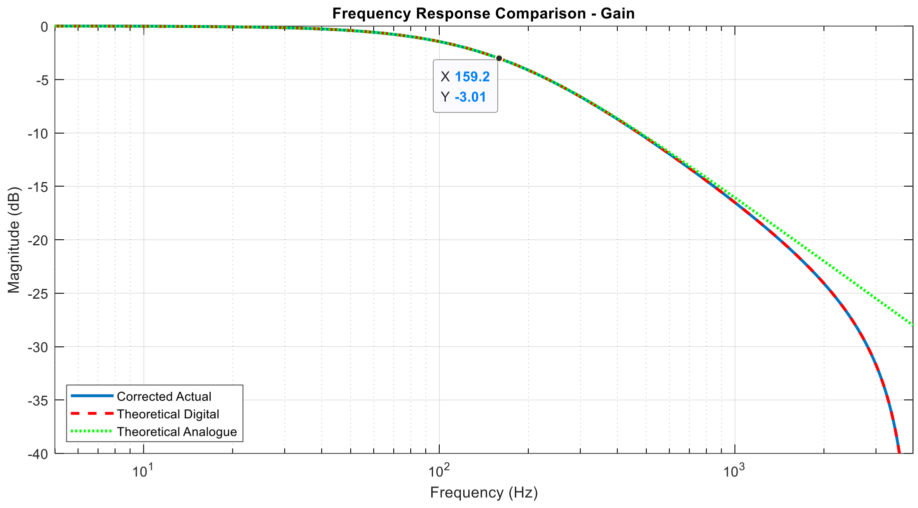

Figure 10: Amplitude response comparison

From Figure 10, we can conclude that the filter we implement in C on the DSK board works as predicted. The amplitude response of it behaves exactly the same as that of a digital simple pole low-pass filter. It also has a very accurate portray as the analogue filter when the frequency is below 1kHz.

1.3.2 Phase Response

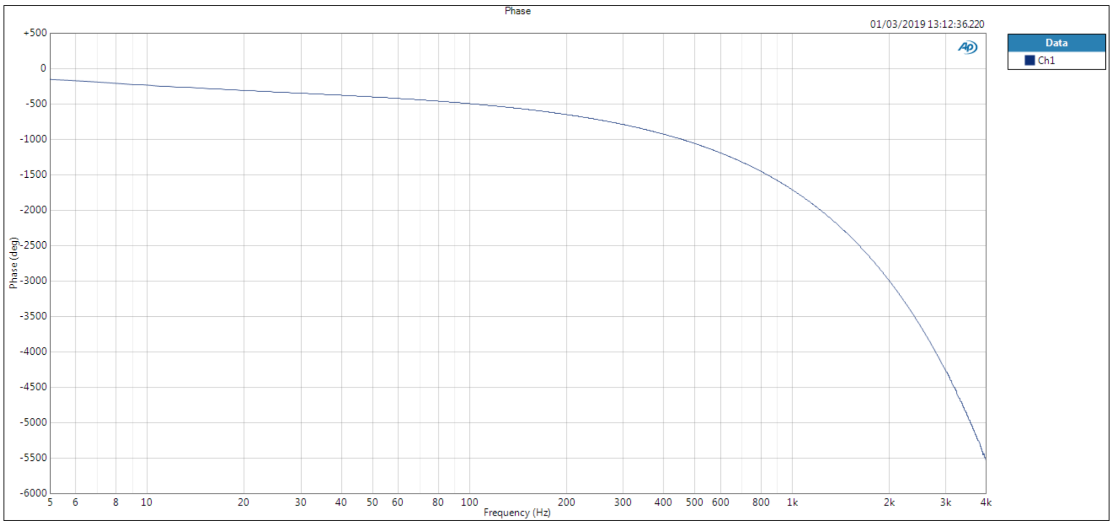

Figure 11: Unprocessed amplitude response from APx500 Audio Analyzer

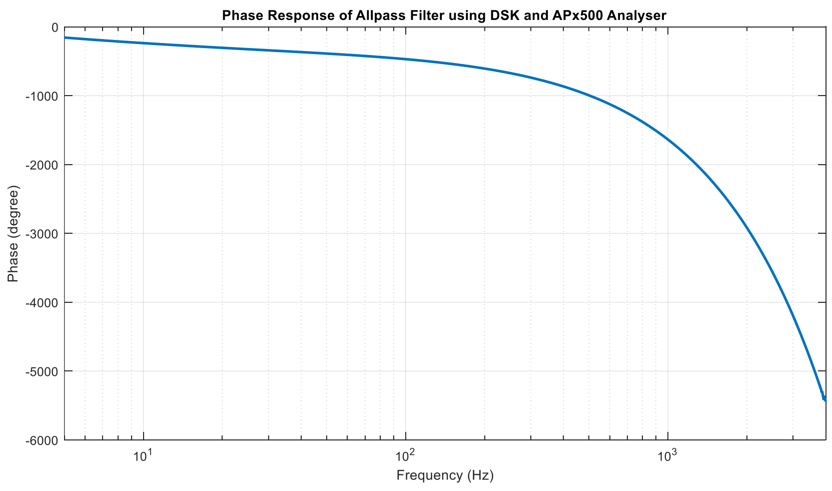

Figure 12: Phase response of all-pass filter in DSK

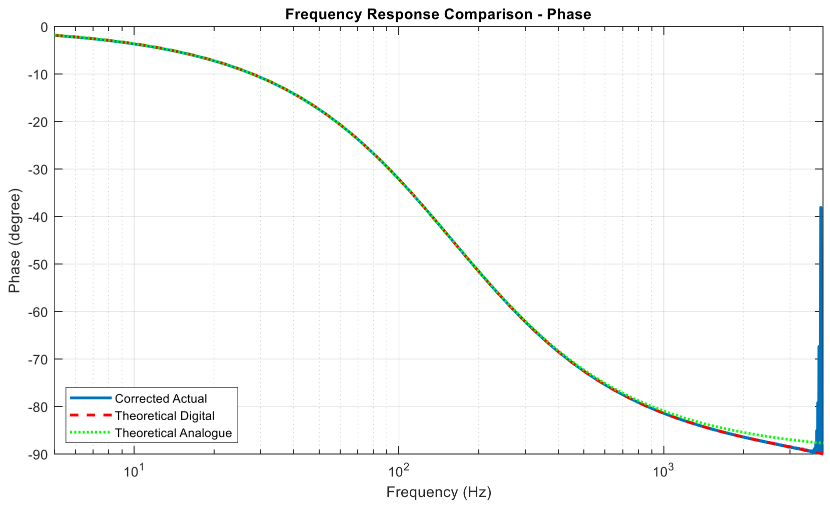

Figure 13: Phase response comparison

Figure 11 shows the actual unprocessed phase response of the filter on the DSK board using the APx500 Audio Analyzer. Using an all-pass filter, we can see the phase decreases rapidly as the frequency increases (Figure 12). This is due to the inaccuracy of the Audio Analyzer measures the phase response of a system.

When processing the actual amplitude response, we use the difference of amplitude response between the simple RC digital filter on DSK and the all-pass filter to find the corrected gain response. Similarly, we subtract the phase response of the all-pass filter from the filter we designed and implemented on the DSK board, the resulting phase response would not contain any systematic aspect of inaccuracy.

Figure 13 shows the comparison of phase response of filter on DSK (corrected), digital filter in Matlab and analogue RC filter. The phase changes from 0° to -90° because of the presence of j in the denominator of the s-domain transfer function.

Figure 13 also shows that when the frequency is close to the Nyquist frequency (4kHz), the phase response of the DSK filter gets fluctuated and unpredictable because of the non-ideal reconstruction filter, which is a low-pass filter at the reconstruction stage in the digital-to-analogue converter (DAC). It would ideally have cutoff frequency at Nyquist frequency with $-\infty$dB stopband attenuation and 0Hz transition band. The corrected actual and theoretical digital phase response at high frequencies slightly deviate from the analogue phase response due to frequency warping again.

In conclusion, the filter we implemented on the DSK works as we expected, for both amplitude and phase responses.

1.4 Frequency Warping

From continuous-time RC filter to discrete-time digital filter, we used the Tustin Transform which converts a continuous-time system into a discrete-time system. However, we used the approximation:

$$s \approx \frac{2}{T} \cdot \frac{z-1}{z+1} \ .$$

Consequently, the analogue frequency ω_a does not map exactly to the digital frequency ω_d. In another word, the relationship between ω_a and ω_d is not linear.

Actually, the analogue frequency ω_a and digital frequency ω_d have this relationship:

$$\omega_a = \frac{2}{T} \tan\left(\frac{\omega_d T}{2}\right)$$

or

$$\omega_d = \frac{2}{T} \tan^{-1}\left(\frac{\omega_a T}{2}\right)$$

where the T stands for the sampling period, which is the reciprocal of sampling frequency.

From the equation, we discovered that when the frequency is low, they almost have 1-to-1 mapping, or they are approximately equivalent. For example, the corner frequency of the analogue RC filter is 159.2Hz which is 1k rad/s. The corresponding digital frequency would be:

$$\omega_d = \frac{2}{1/8000}\tan^{-1}\left(\frac{1000 * 1/8000}{2}\right) = 998.7\, rad/s$$

The digital filter cut-off frequency f_d would be $\omega_d/2\pi$ which is 158.9Hz. It is almost the same as the analogue filter corner frequency.

However, if we take the digital frequency f_d 2kHz (ω_d = 4000π rad/s), the analogue filter corner frequency would be:

$$\omega_a = \frac{2}{1/8000}\tan\left(\frac{4000\pi * 1/8000}{2}\right) = 16000\, rad/s$$

$$f_a = \frac{\omega_a}{2\pi} = \frac{16000}{2\pi} \approx 2546\, Hz$$

The digital frequency of 2kHz is mapped into the analogue frequency of 2546Hz. As a result, when we plot the amplitude response of the digital filter, the amplitude of digital frequency 2kHz is actually the amplitude of analogue frequency 2546Hz. Because the amplitude of analogue frequency 2546Hz is smaller than that of 2kHz, we would expect much lower amplitude at digital frequency 2kHz. This can be seen from Figure 14, which is the zoomed version of frequency response comparison of digital and analogue filter:

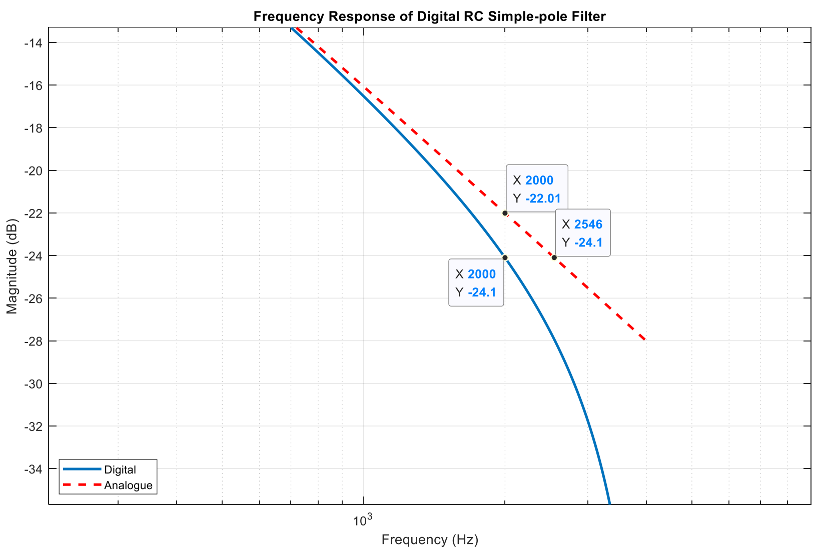

Figure 14: Zommed frequency response

We can see that the amplitude of digital frequency 2kHz is exactly equal to the amplitude of analogue frequency 2546Hz, which is -24.1dB. The frequency warping can also be used to explain the deviation of phase responses of digital and analogue filter in a similar way.

In conclusion, when the frequency is low, the amplitude response of a digital filter is almost the same as that of an analogue filter. When the frequency is high, e.g. >1kHz, the analogue frequency ω_a does not exactly map the digital frequency ω_d, causing the amplitude response to attenuate rapidly as we expected.

1.5 Time Constant

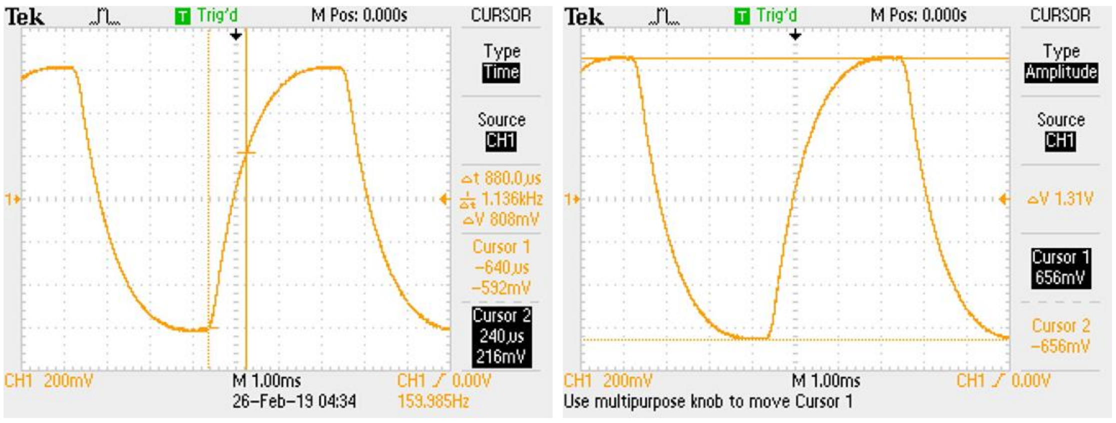

Figure 15: Scope trace

In order to measure time constant from the oscilloscope, we have driven a square wave as input at frequency 160Hz. According to the structure of the board, a high pass filter exists in the input port, so we choose 160Hz in order to prevent large effect on the shape of the output waveform; at this frequency, the outputs are easy for us to find the time constant with reference to Figure 15.

For digital filter, time constant is calculating by equations in below:

$$\tau = RC = \frac{1}{\omega_c}$$

where τ is time constant, ω_c is the corner frequency, R is the resistance and C is the capacitance.

In our case, RC time constant is

$$\tau \ = 1k\Omega \times 1\mu f \ = \ 1\, ms\, .$$

Moreover, for analogue filter, time constant is measured by the formula in below:

$$\Delta V = \ V_{pk-pk}(1 - \frac{1}{e^{t/\tau}})$$

where ΔV is the change of voltage, $V_{pk\text{-}pk}$ is the peak-to peak-voltage, t is the time in seconds and τ is the time constant.

By substituting t = τ, rearranges the function to get:

$$\frac{\Delta V}{V_{pk-pk}} = 1 - \frac{1}{e} \approx 0.632\,.$$

We use the peak to peak value times 0.632 to obtain the voltage about 828mV, then we trigger the oscilloscope to peak to peak value at 808mV (the closest we can do), which shows time constant at about 880.0μs with reference to left half of Figure 15.

Because the time constant τ is inversely proportional to the corner frequency ω_C:

$$\tau = \frac{1}{\omega_c}.$$

Increasing in ω_C would result in a decrease in τ.

Have a closer look in the amplitude response:

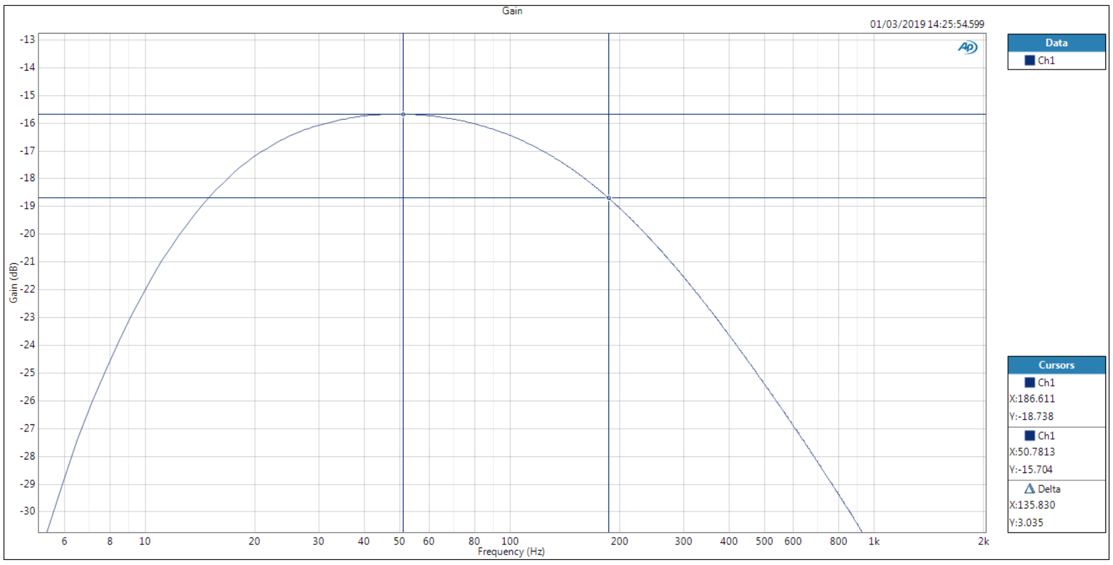

Figure 16: Zoomed unprocessed amplitude response

With reference to Figure 16, the corner frequency ω_C' is at 186.611Hz, which can be seen in Ch1. The corner frequency increases because of the effect of high-pass filter (discussed in section 1.3.1). And the corresponding time constant τ' would be:

$$\tau' = \frac{1}{\omega_c{}'} = \frac{1}{186.611 \times 2\pi} \approx 853\ \mu s\,.$$

The time constant τ' is smaller than theoretical value 1ms, which matches with our measurement 880μs, with relatively small error due to systematic inaccuracy of the oscilloscope.

2 Direct Form II Non-transposed IIR Filter

2.1 Design

2.1.1 Specification

In this exercise, we are going to design a direct form II non-transposed IIR filter.

Specification is verbally described in below:

- Filter order: 4th
- Passband frequencies: 270-450 Hz
- Passband ripple: 0.3dB
- Stopband attenuation: 20 dB

2.1.2 Coefficients calculation

By translating into Matlab code:

```
fs = 8000;      % sampling frequency

% calculate the coefficients of bandpass elliptic filter
% the cutoff frequencies is normalized frequencies
[b,a] = ellip(2, 0.3, 20, [270 450]*2/fs, 'bandpass');

% get points (f,h) from the frequency response
[h,f] = freqz(b,a,1024,fs);
```

Firstly, sampling frequency has defined at 8000Hz. Then, we have used ellip() function to output vector [b, a] with size 5, which is the filter order add one. Lastly, by using freqz(), plotting the frequency response to check whether we have reached our specification.

The ellip() function is required to used normalized frequency[2], so we do normalization by diving sampling frequency [270 450]*2/fs, where fs is the sampling frequency. Because the first argument of the function tries to return 2n order, which will give us a 4th order filter. The second and the third arguments are passband ripple and stopband attenuation respectively. The last argument tries to specify filter type, which is a bandpass filter in our case.

2.1.3 Writing coefficients into a text file

```
fileID = fopen('coefs.txt', 'w');

% write coefficient b into txt file
fprintf(fileID, 'double b[] = {');
fprintf(fileID, '%.16e', b(1));
```

[2] Normalized to Nyquist frequency, which is fs/2

```
fprintf(fileID, ', %.16e', b(2:end));
fprintf(fileID, '};\n');

% write coefficient a into txt file
fprintf(fileID, 'double a[] = {');
fprintf(fileID, '%.16e', a(1));
fprintf(fileID, ', %.16e', a(2:end));
fprintf(fileID, '};\n');

fclose(fileID);
```

In order to write the coefficients with appropriate form into a text file, Matlab code has written in below to implement it. No further explanation needed since clear comments has written.

2.1.4 Theoretical frequency response

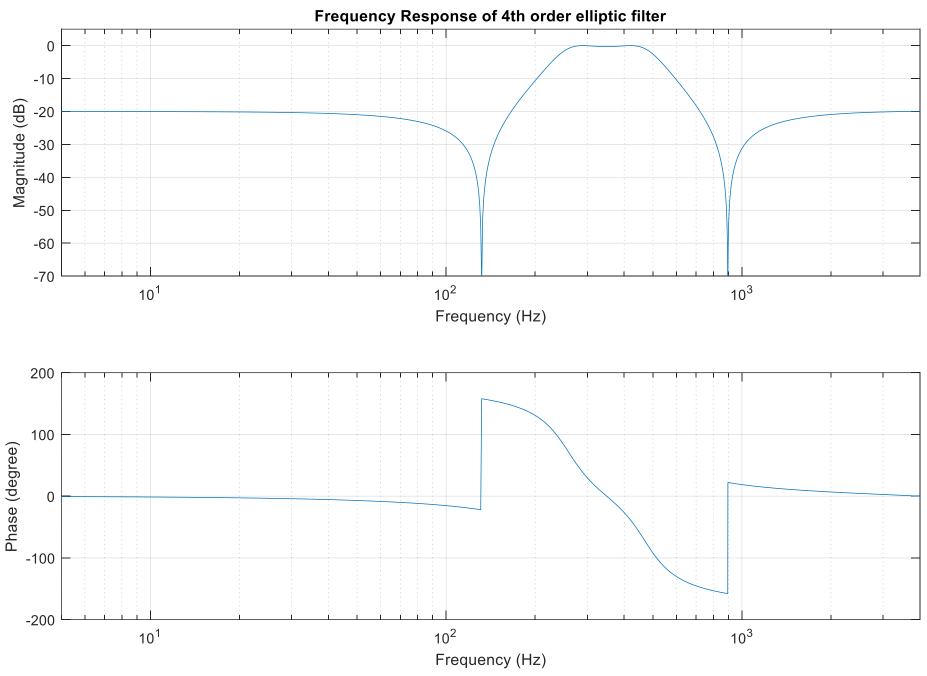

Figure 17: Frequency response of 4th order elliptic filter

The figures above have shown the output of the Matlab code, which is the design filter by meeting specification. With reference to Figure 17, the IIR filter has a very fast roll off for lower order and it is obvious to see that the phase is non-linear because of existence of poles.

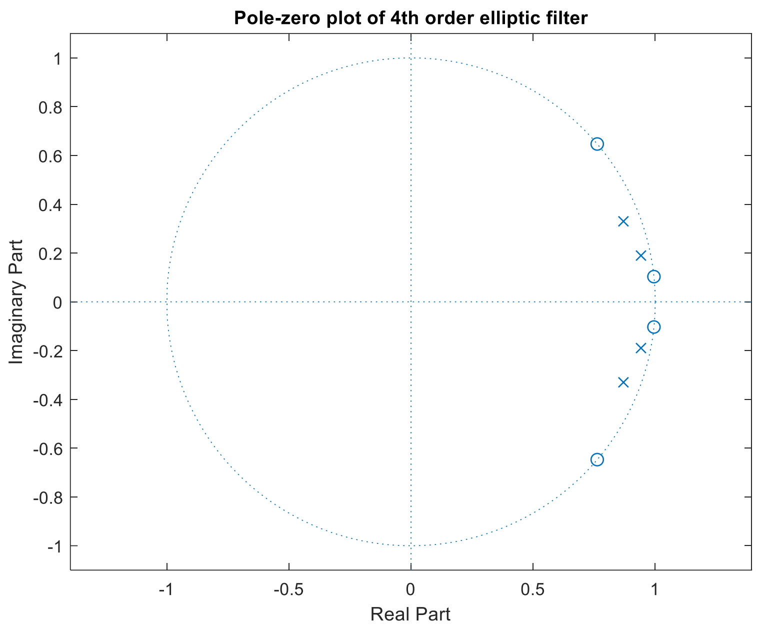

Figure 18: Pole-Zero Plot of 4th order elliptic filter

With reference to Figure 18, we can see that there are four crosses and four circles within unit circle on the right-hand side of the imaginary axis. A cross represents a pole and a circle represents a zero.

Because all poles are lying inside the unit circle, we can say this causal system (discussed in Section 1.1.2) is stable.

In summary, the theoretical plot has clearly shown above and a text file with coefficients has successfully generated.

2.2 Implementation in C

2.2.1 Global declarations

```
#include "coefs.txt"  //includes coefficient file
int N = sizeof(a)/sizeof(a[0])-1;    //Calculate order of the filter

double *w;     //define pointer for dynamic allocation later
Int16 xin = 0;        //define input
double yout = 0;      //define output
```

Firstly, we have included the coefficient file generated by Matlab. The filter order is N as shown in above, and a pointer w is defined for dynamic memory allocation. In order to prevent uncertainty, we initialize the input variable xin and output variable yout to zero.

2.2.2 In the main function

```
        //using pointer define above to perform dynamic allocation
        w = (double*)calloc(N+1, sizeof(double));
```

Then according to lab instruction, dynamic memory allocation needs to be used for creating delay buffer, so a function called calloc() is introduced. In later section, we asked to change the filter order to investigate the algorithm speed, so dynamic memory allocation has the benefits of creating different buffer spaces in run time by just simply change the order of the filter. Compiler will generate corresponding memory size to the number of coefficients. The way of using this function is similar to the C code malloc(), which uses a pointer to store values dynamically to successive memory locations. The first argument takes the number of coefficients, and the second takes the size of each element.

2.2.3 In the ISR

```
void ISR_AIC(void)
{
        //extract data from provided funtion
        xin = mono_read_16Bit();

        IIR_non_transposed();           //calling function
        mono_write_16Bit(yout);                 //output resuult
}
```

We extract 16 bits data from mono_read_16Bit() first, and then call the transposed function. In the last stage, we have used mono_write_16Bit(yout) to output result.

2.2.4 In the filter implementation function

```
void IIR_non_transposed(void)
{
        int i = 0;      //initialize counter
        w[0] = xin;             //assign values to first buffer
        yout = 0;               //initialize output variable

        //translate equation into C code
        for(i=N;i>0;i--){
                w[0] -= a[i]*w[i]; // decrement multiply sum
                yout += b[i]*w[i]; //increment multiply sum
                w[i] = w[i-1]; //delay buffer by one
        }

        //Assign the last values to output variable
        yout += b[0]*w[0];
}
```

This function implements the main algorithm based on the structure of non-transposed form, with reference to Figure 19. We assign our extracted 16 bits data to w[0], and then initialize output variable for preventing uncertainty. Within the for loop, we just implement the algorithm. In the final stage, we assign the last value to output variable.

The explanation of the algorithm is simply in the comments of the code, for more detail, and we can see the visualization in Figure 19:

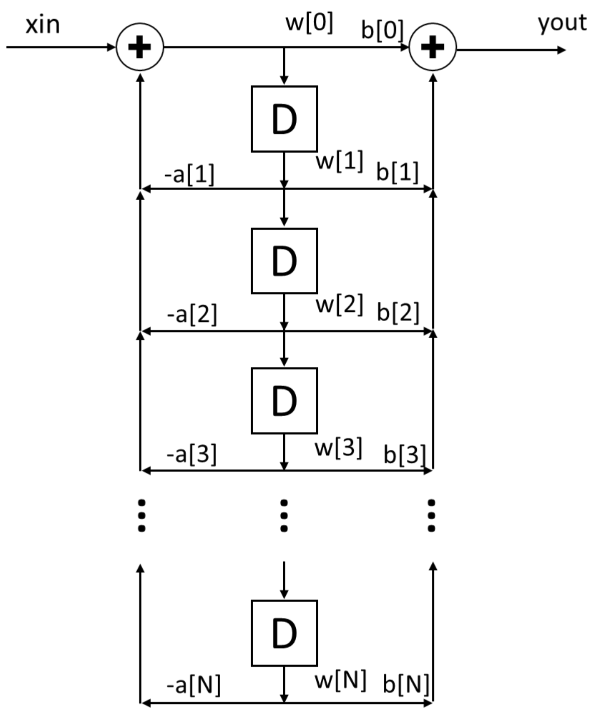

Figure 19: Visualization of Algorithm for non-transposed IIR

According to the Figure 19 with filter order N, we derive the equations in below:

$$\begin{aligned}
\boldsymbol{w[0]} &= \boldsymbol{xin - a[N]w[N] - \cdots - a[2]w[2] - a[1]w[1] - a[0]w[0]} \\
&= \boldsymbol{xin - \sum_{i=N}^{1} a[i]w[i]} \\
\boldsymbol{yout} &= \boldsymbol{b[N]w[N] + \cdots + b[2]w[2] + b[1]w[1] + b[0]w[0]} \\
&= \boldsymbol{\left(\sum_{i=N}^{1} b[i]w[i]\right) + b[0]w[0]} \\
\boldsymbol{w[i]} &= \boldsymbol{w[i-1]}\,.
\end{aligned}$$

And we just directly transform the equations into the C code.

2.3 Frequency Response

In this section, we are going to verify the frequency response of our design filter by using APX500 Audio Analyzer.

2.3.1 Amplitude response

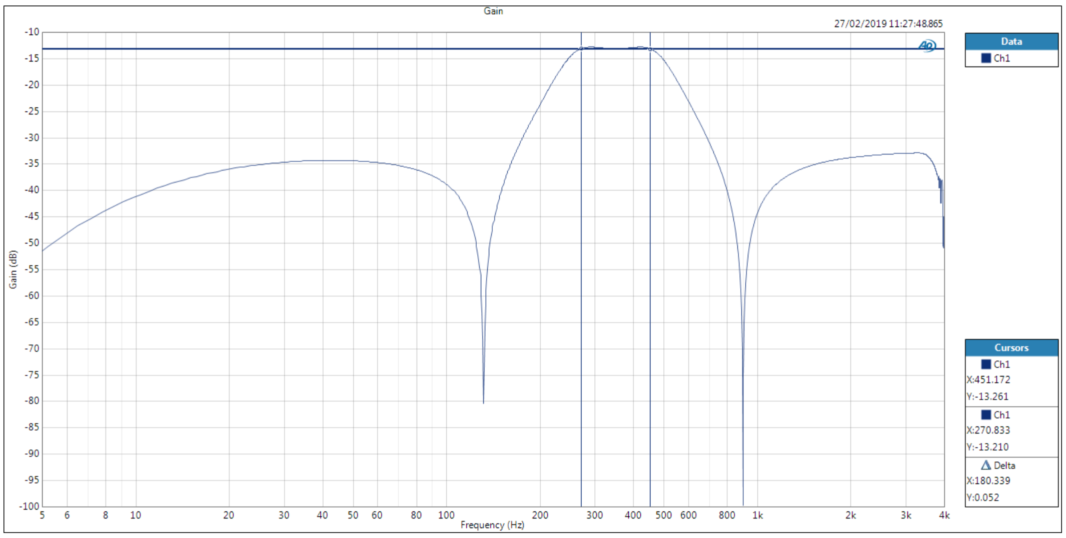

Figure 20: Unprocessed amplitude response of Direct Form II non-transposed elliptic filter

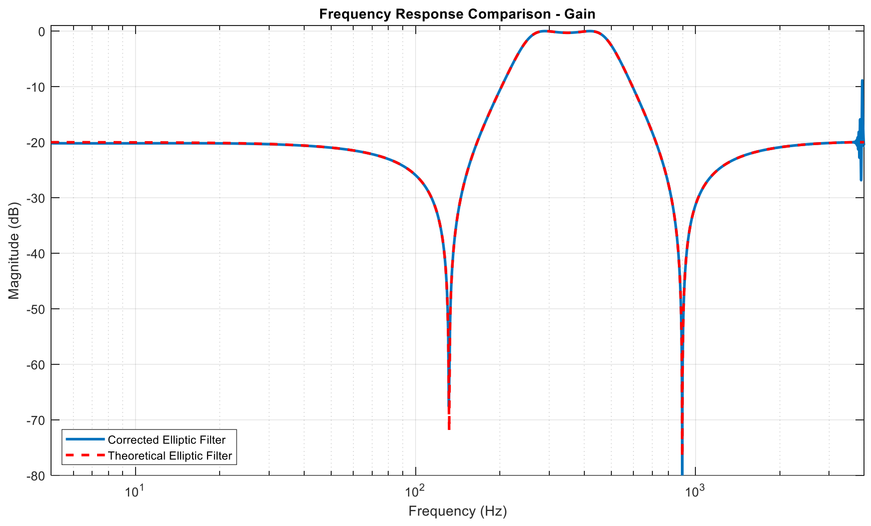

Figure 21: Corrected amplitude response of Direct Form II non-transposed elliptic filter

With reference to Figure 20, we can see the spectrum is distributed in three parts. As already discussed in previous section 1.3.1, the attenuation at low frequency is because of high pass filter effect. And the last part, which near to the Nyquist frequency, has distortion due to the non-ideal reconstruction filter of DAC in the DSK board. The spectrum has overall attenuation of -12.84 dB because of 1/4 attenuation mention in section 1.3.1.

By exporting the data in Spectrum Analyzer, we have plotted theoretically and practically gain response by Matlab in one plot, as we expect, we have found out two responses are almost the same. However, for the digital spectrum, we can see the distortion near to Nyquist frequency due to non-ideal reconstruction filter and this would not show in theoretical plot.

In conclusion, by discussing with my partner, the gain response is correct.

2.3.2 Phase response

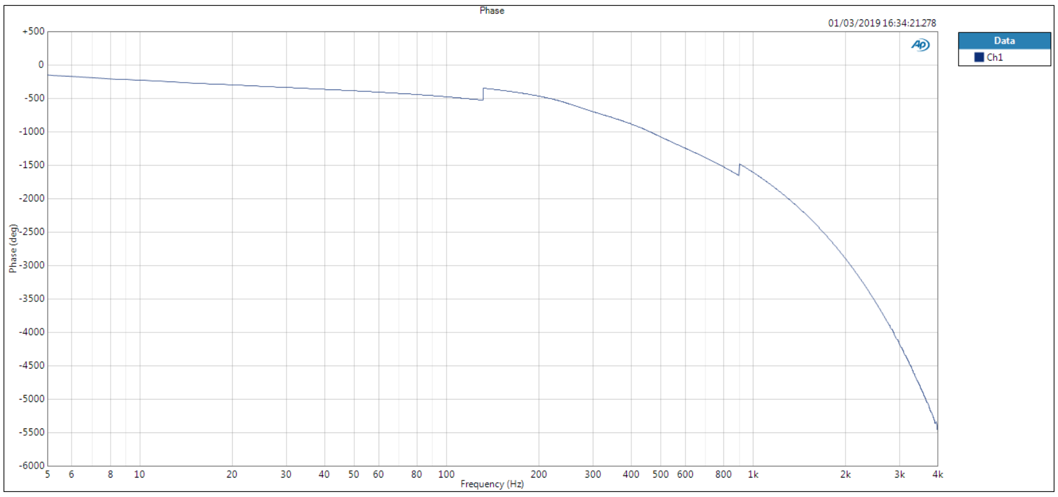

Figure 22: Unprocessed phase response of Direct Form II non-transposed elliptic filter

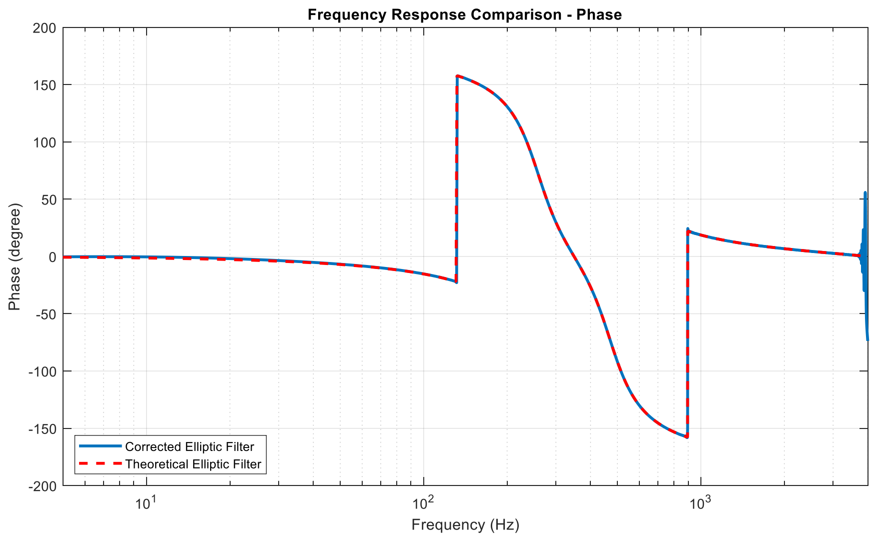

Figure 23: Corrected phase response of Direct Form II non-transposed elliptic filter

By exporting the data in Analyzer again, we have successfully putting theoretically and practically gain in one Matlab plot. We observe that the phase is non-linear within passband, and as we expect, the distortion due to non-ideal reconstruction filter has shown near Nyquist frequency.

In conclusion, with reference to Figure 23, one line is aligning with the other one, so the result of digital plot is correct.

3 Direct Form II Transposed IIR Filter

3.1 Design

The requirement has not changed compared to the previous design, however, this time we need to convert a non-transposed filter into transposed.

[3]*The network is unchanged in behavior if:*

1. *Reverse the direction of each branch*
2. *Interchange branch divisions and branch summations*
3. *Swap input for output*

The output of transposed form is just adding two results together, which are the coefficient multiplication in both input and output ports.

We expect transposed form performs faster than the non-transposed one. Because transposed form IIR filters use less memory and requires fewer computations.

3.2 Implementing in C

The transposed IIR filter shares the same global declaration and main function but in separate functions.

3.2.1 In the ISR

```
void ISR_AIC(void)
{
	//extract data from provided funtion
	xin = mono_read_16Bit();

	IIR_transposed();		//calling function
	mono_write_16Bit(yout);	//output result
}
```

We differentiate two filters by calling different functions in the ISR. The other parts of this function are exactly the same as previous.

3.2.2 In the filter implementation function

```
void IIR_transposed(void)
{
	int i = 0;		//initialize counter

	//assign values to output variable
```

[3] Real Time Digital Signal Processing Section 6 – IIR Filters and their design. Available at: https://bb.imperial.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_1328482_1&course_id=_13633_1

```
        yout = w[0] + b[0]*xin;

        //translate equation into C code
        for(i=1;i<N;i++){
                w[i-1] = w[i] + b[i]*xin - a[i]*yout;
        }

        //Last one with no delay(special case)
        w[N-1] = b[N]*xin - a[N]*yout;
}
```

This function implements the algorithm to generate design filter. Firstly, a counter has initialized to zero, and then we assign values to output variable yout. Within the for loop, we simply implement the equations, and the detail explanation can be seen in visualization below, with reference to Figure 24. The last thing needs to mention is that the last index has no more delay, so we have specified in the last line within the for loop of the C code.

The visualization of algorithm for the transposed IIR filter:

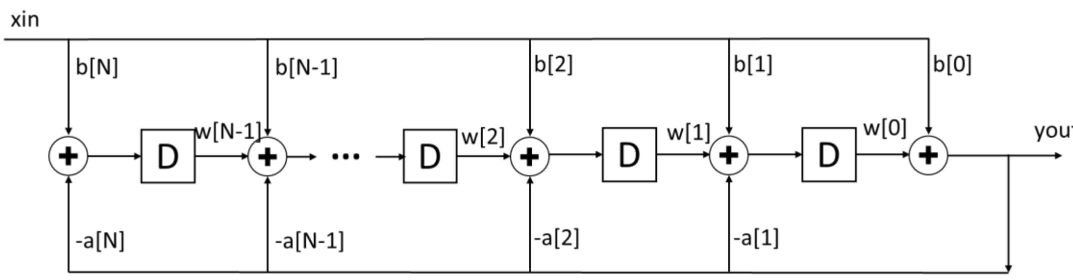

Figure 24: Visualization of algorithm for transposed IIR filter

Figure 24 has shown the structure of the transposed IIR filter. Same as the previous filter design, we need to convert the flow into equations:

$$yout = w[0] + b[0]xin$$

$$For\ loop:$$

$$- -$$

$$\begin{aligned} w[0] &= w[1] + b[1]xin - a[1]yout \\ w[1] &= w[2] + b[2]xin - a[2]yout \\ &\vdots \\ &\vdots \\ w[i-1] &= w[i] + b[i]xin - a[i]yout \\ &\vdots \\ &\vdots \\ w[N-2] &= w[N-1] + b[N-1]xin - a[N-1]yout \end{aligned}$$

$$- -$$

$$w[N-1] = b[N]xin - a[N]yout$$

The above equations are derived by the flow of the structure in Figure 24, and implement in C code in a straightforward way.

3.3 Frequency Response

3.3.1 Amplitude response

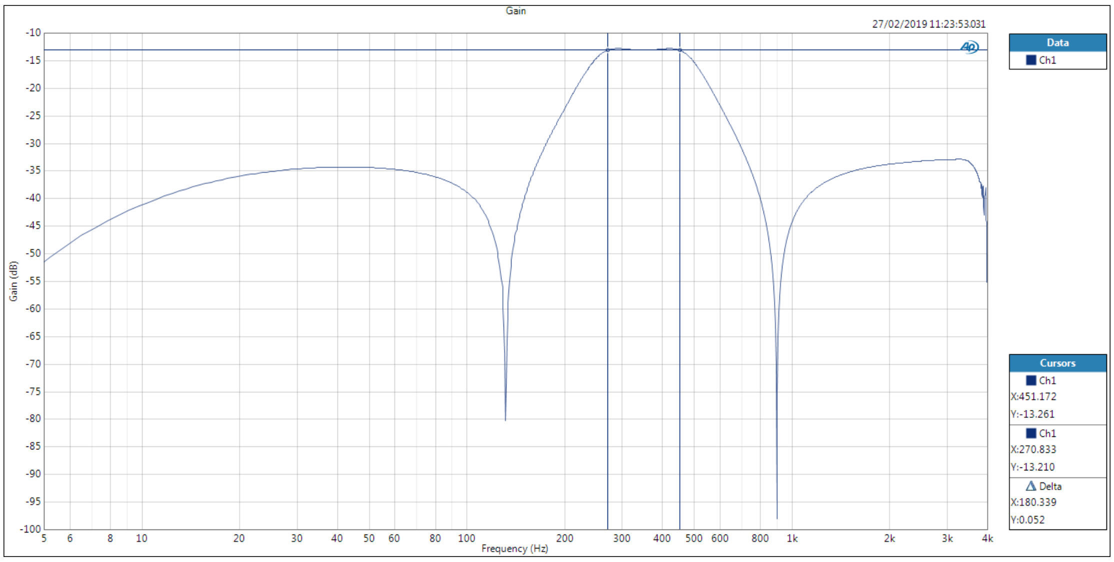

Figure 25: Unprocessed amplitude response of Direct Form II transposed elliptic filter

With reference to Figure 25, the spectrum output from Spectrum Analyzer has distributed in three parts. During first part starts from the low frequency range, the attenuation due to high pass filter has shown. In the third part near to Nyquist frequency, distortion due to non-reconstruction filter can be seen. The overall attenuation is -12.83 dB, and this is because of the 1/4 attenuation of the DAC in the DSK board.

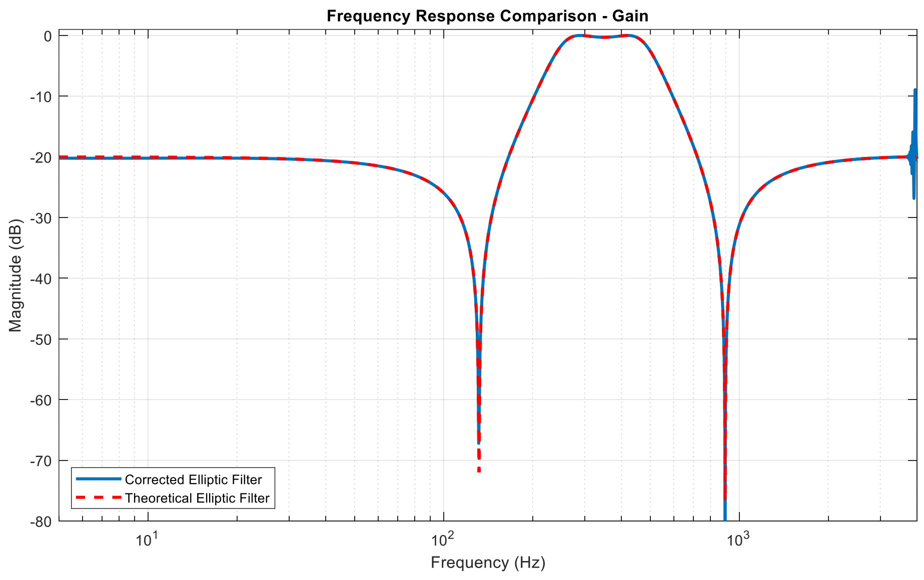

Figure 26: Corrected amplitude response of Direct Form II transposed elliptic filter

By exporting data from Spectrum Analyzer, we have successfully plotted the theoretical graph and practical graph in one plot. With reference to Figure 24, two plots are almost the same, however, the practical plot has distortion near to the Nyquist frequency due to non-ideal reconstruction filter.

In conclusion, after discussing with my partner, we agree that the amplitude response is unchanged while the algorithm is completely different.

3.3.2 Phase response

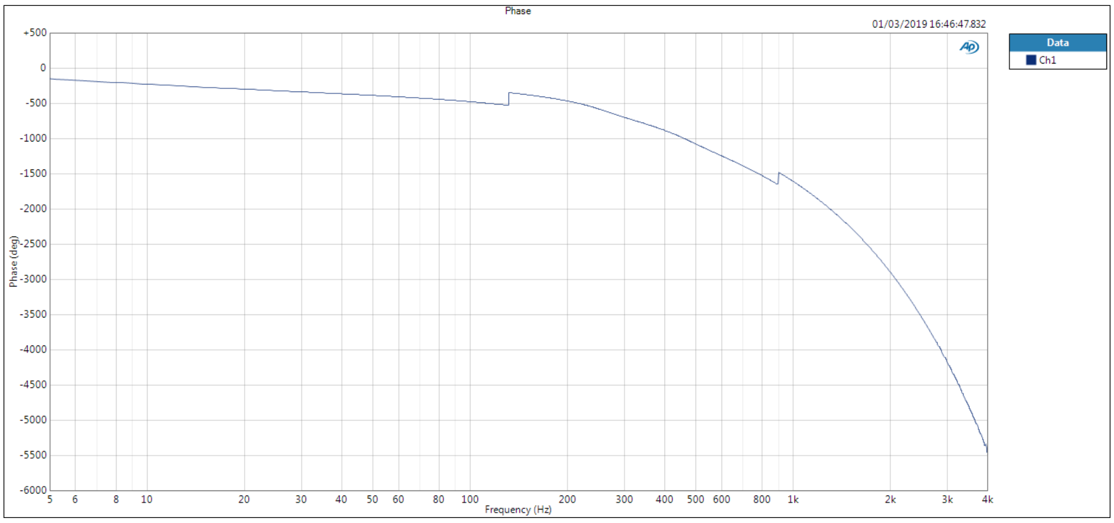

Figure 27: Unprocessed phase response of Direct Form II transposed elliptic filter

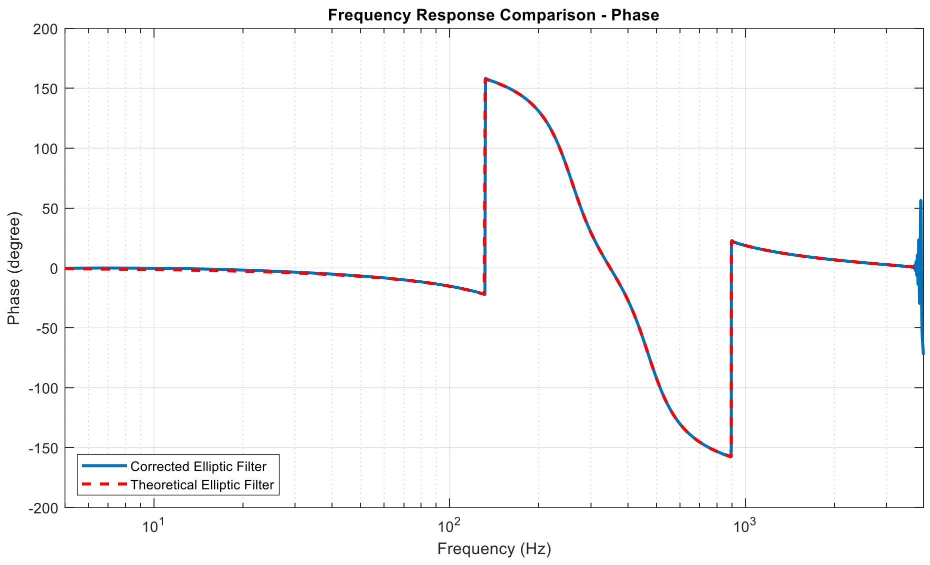

Figure 28: Corrected phase response of Direct Form II transposed elliptic filter

By exporting data from Spectrum Analyzer, we have successfully plotted the theoretical graph and practical phase response in one plot. With reference to Figure 26, two plots are almost the same, except distortion appears near to the Nyquist frequency due to non-ideal reconstruction filter again.

In conclusion, after discussing with my partner, our phase response is unchanged as well. The frequency responses are the same regardless of using Direct Form II transposed or non-transposed IIR filter.

4 Performance Comparison

4.1 Data Collected

Number of Cycles	order	Optimization Level		
		None	-o0	-o2
Non-transposed	4	412	394	257
	6	582	552	333
	8	752	710	409
	10	922	868	485
	12	1092	1026	561
	14	1262	1184	637
Transposed	4	279	247	187
	6	397	343	171
	8	515	439	172
	10	633	535	182
	12	751	631	192
	14	869	727	202

Table 1: Original data of number of cycles

By setting two breakpoints at the beginning and the end of the filter implementation functions, we can measure the clock cycles that the program would take to compare performance. Table 1 recorded the number of clock cycles for one single run of the filter implementation function in C.

In the Table 1, we cannot easily see the clear differences for the performances, so detailed analysis for the data is performed in Section 4.2.

One more important thing needs to mention is that above order 14th, the filter stop functioning correctly. Matlab plot of order 16th has shown in below, with reference to Figure 29.

It is obvious that the plots are showing discontinuity property, which means that the filter does not work above order 14th, and this is because of the limitation of ellip () function in Matlab. According to Matlab documentation, ellip() function only works properly in lower filter order, so we only test our algorithm from filter order 4th up to 14th.

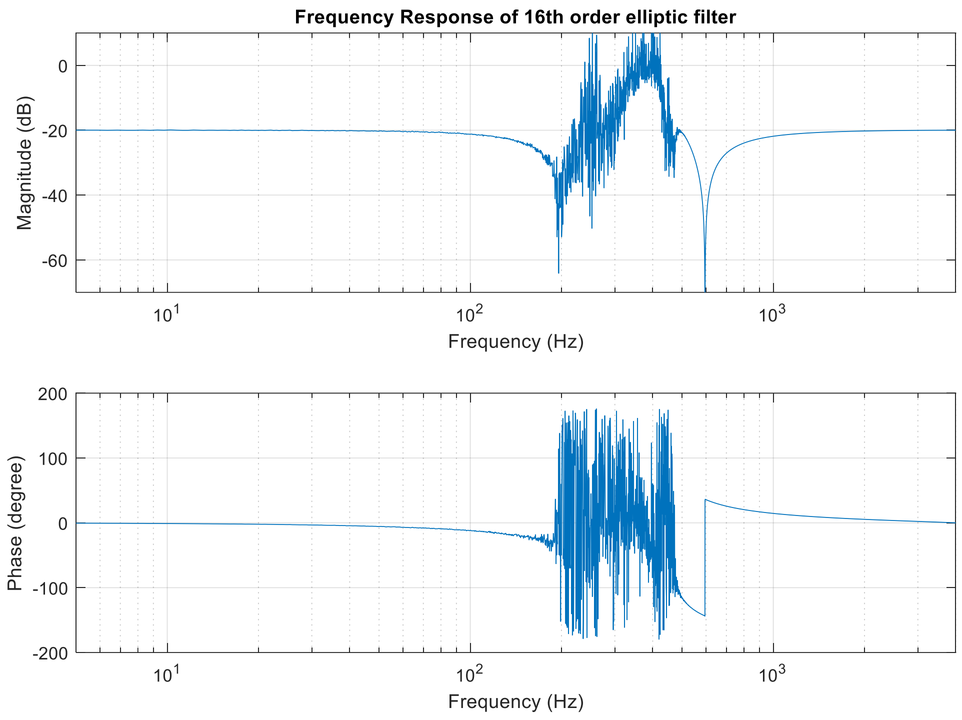

Figure 29: Frequency response of 16th order elliptic filter in Matlab

4.2 Data Analysis

4.2.1 Differences Δ

Number of Cycles	order	Optimization Level					
		None	Δ	-o0	Δ	-o2	Δ
Non-transposed	4	412		394		257	
	6	582	170	552	158	333	76
	8	752	170	710	158	409	76
	10	922	170	868	158	485	76
	12	1092	170	1026	158	561	76
	14	1262	170	1184	158	637	76
Transposed	4	279		247		187	
	6	397	118	343	96	171	-16
	8	515	118	439	96	172	1
	10	633	118	535	96	182	10
	12	751	118	631	96	192	10
	14	869	118	727	96	202	10
NB: Δ is the difference of clock cycles taken by filter order n+2 and order n							

Table 2: Delta between two data

The Table 2 clearly shows that the differences between two adjacent data of clock cycles is same. In another word, if the order of filter increases by 2, the number of clock cycle taken would increase by a constant value in the same level of optimization. Therefore, they have a linear relationship y=Ax+B, where x is the filter order and y is the number of clock cycles taken.

However, there is one exception when the transposed IIR filter order is 4 and 6 under -o2 level of optimization. The number of clock cycles decreases when the order increases from 4 to 6 and remains approximately the same when the order is 8. If the filter order continues to rise, the number of clock cycles would keep the linear relationship. The reason for this is when the filter order is low, the optimization for the code would not be as apparent as that of higher order filters.

4.2.2 Recap for calculating regression lines

To calculate the exact expression for number of cycles and the filter order, we are going to use some tools covered in the statistic module in year 2, i.e. regression lines. Basically, a regression line is similar with the best-fit straight line in a set of discrete 2-D points.

A linear relationship between x and y would be something like:

$$y = Ax + B$$

where A is the gradient and B is the y-intercept. The formula for calculating A and B is:

$$A = \frac{Var(x, y)}{Var(x)}$$
$$B = \bar{y} - B\bar{x}$$

where Var(x) is the variance of x, Var(x,y) is the covariance of x and y, $\bar{x}$ and $\bar{y}$ are the mean values. Var(x) and Var(x,y) is calculated as:

$$Var(x) = \frac{1}{n}\sum(x - \bar{x})^2 = \frac{1}{n}\sum x^2 - \bar{x}^2$$
$$Var(x, y) = \frac{1}{n}\sum(x - \bar{x})(y - \bar{y}) = \frac{1}{n}\sum xy - \bar{x}\,\bar{y}\,.$$

We can also use the correlation coefficient R to indicate the linear correlations of data:

$$R = \frac{Var(x, y)}{\sqrt{Var(x) \cdot Var(y)}}.$$

The range of R is -1 ≤ R ≤ 1, and the value of R would give us such information:

$$R = \begin{cases} 1 & (perfect\ positive\ linear\ correlation) \\ -1 & (perfect\ negative\ linear\ correlation) \\ 0 & (no\ correlation) \end{cases}$$

4.2.3 Regression lines for non-transposed IIR filter

Number of Cycles	order	Optimization Level		
		None	-o0	-o2
Non-transposed IIR Filter	4	412	394	257
	6	582	552	333
	8	752	710	409
	10	922	868	485
	12	1092	1026	561
	14	1262	1184	637
Sum	54	5022	4734	2682
Number of data	6	6	6	6
Mean	9	837	789	447
Variance	11.666667	84291.667	72811.667	16846.667
Covariance		991.66667	921.66667	443.33333
Gradient (B)		85	79	38
y-intercept (A)		72	78	105
Correlation Coefficient		1	1	1

Table 3: Regression line calculations for non-transposed IIR filter

The gradient B, y-intercept A and the correlation coefficient R are calculated in the light-yellow region in Table 3 using the method introduced in Section 4.2.2. From the table we found the gradient B decreases and y-intercept increases as we increase the optimization level, indicating that the difference in performance of different orders of filters is smaller in higher level of optimization level. The correlation coefficients R is all equal to 1 meaning that the number of clock cycles and the filter order are perfectly positively linearly correlated.

4.2.4 Regression lines for transposed IIR filter

Number of Cycles	order	Optimization Level			-o2 Corrected
		None	-o0	-o2	
Transposed IIR Filter	4	279	247	187	152
	6	397	343	171	162
	8	515	439	172	172
	10	633	535	182	182
	12	751	631	192	192
	14	869	727	202	202
Sum	54	3444	2922	1106	1062
Number of data	6	6	6	6	6
Mean	9	574	487	184.3333	177
Variance	11.66667	40611.67	26880	118.8889	291.6666667
Covariance		688.3333	560	24.66667	58.33333333
Gradient (B)		59	48	4.81982	5
y-intercept (A)		43	55	140.955	132
Correlation Coefficient		1	1	0.662318	1

Table 4: Regression line calculations for transposed IIR filter

As mentioned in Section 4.2.1, there is an exception when the transposed IIR filter order is 4 and 6 using -o2 level of optimization. Only these two points deviate from the best-fit line and need to be corrected. The corrected data is shown in the rightmost column. The corrected data is basically the lower linear extension of the data of filter order 8 to 14. This would generate a perfect positive linear correlation.

4.3 Linear Expressions

Filter Category	Info	Optimization Level			
	y=Ax+B	None	-o0	-o2	-o2 corrected
Non-transposed IIR	Gradient (B)	85	79	38	
	y-intercept (A)	72	78	105	
	Expression	y=85x+72	y=79x+78	y=38x+105	
Transposed IIR	Gradient (B)	59	48	4.82	5
	y-intercept (A)	43	55	140.95	132
	Expression	y=59x+43	y=48x+55	y=4.82x+140.95	y=5x+132
NB: y=clock cycles, x=filter order					

Table 5: Linear expressions of number of clock cycles vs filter order

Extract the values of gradient and y-intercept calculated in Section 4.2.3 and Section 4.2.4, and put them in one table. We found that the gradients and y-intercepts for transposed IIR filter is all smaller than the corresponding values of non-transposed IIR filter. More meaningful and clearer comparisons are shown in the graphs.

4.4 Graphical Comparison

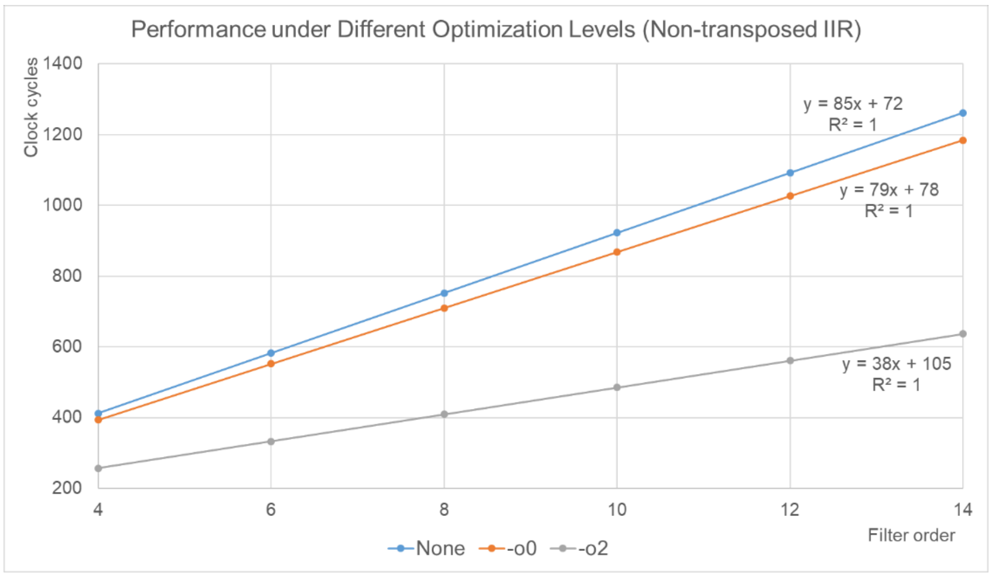

Figure 30: Performance of non-transposed IIR filter

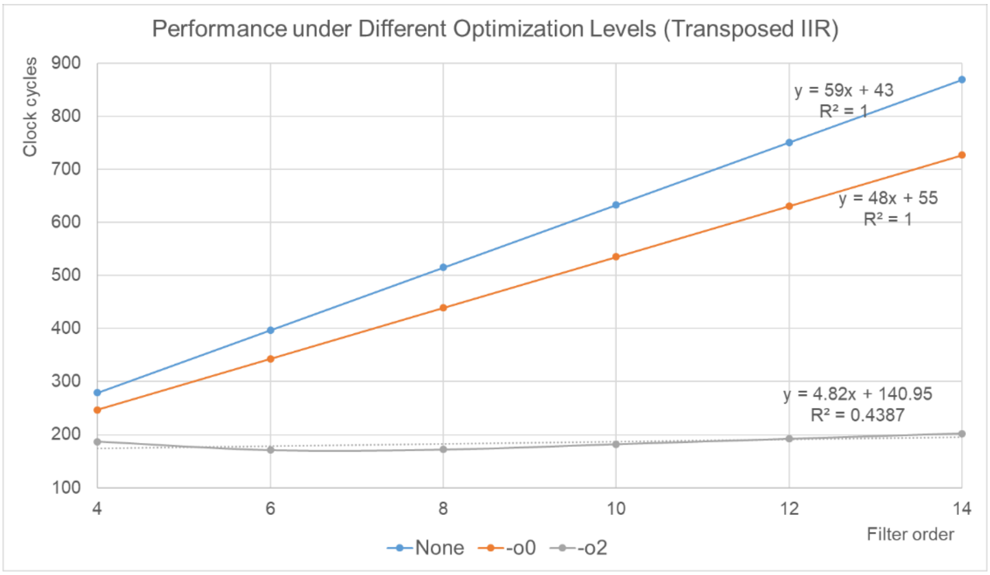

Figure 31: Performance of transposed IIR filter

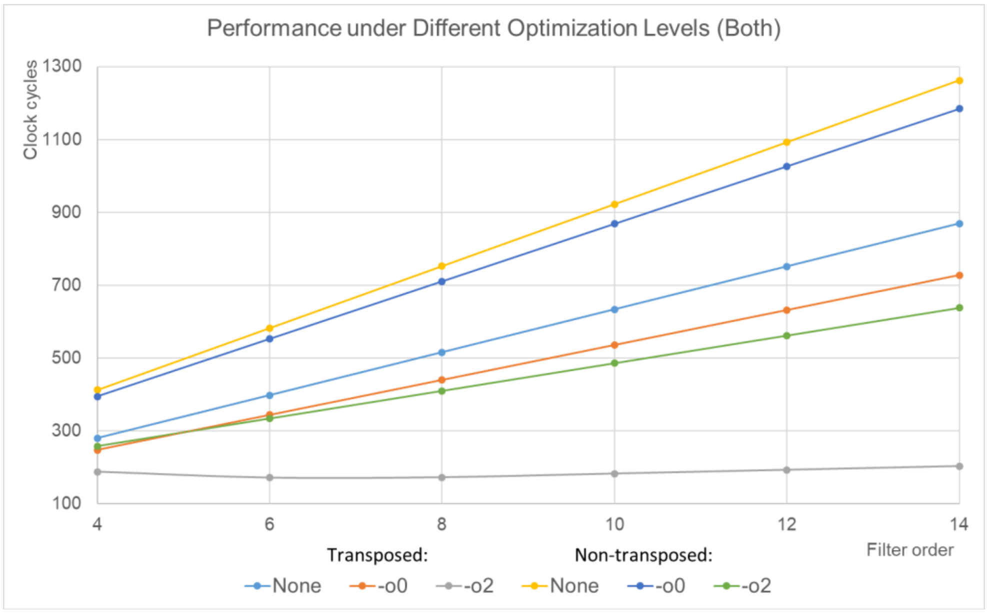

Figure 32：Overall comparison

Figure 30 and Figure 31 are the 2-D plots for the number of clock cycles versus filter order for Direct form II non-transposed and transposed IIR filter. The linear expression y=Ax+B and the square of correlation coefficient R is shown near each line.

From Figure 30 and Figure 31, we discovered that for each of non-transposed and transposed IIR filter, the gradient reduces as the optimization level increases. Except the transposed IIR filter in -o2 level of optimization, all points are fitted perfectly in lines because the correlation coefficients are all equal to 1.

Figure 32 shows the overall comparison by putting two previous graphs together. The slowest filter is the non-transposed filter using no optimization level with highest gradient. The fastest filter is the transposed filter using -o2 optimization level with significantly small gradient as we expected, because of its less memory usage and fewer computations.

5 Appendix

5.1 Code in C

5.1.1 Part of Code of Simple-Pole Filter

```
double b[2] = {5.8823529411764705e-02, 5.8823529411764705e-02};
double a[2] = {1, -8.8235294117647056e-01};  //the a[0] will be never used
Int16 x[2] = {0};
double y[2] = {0};

void ISR_AIC(void)
{
        Int16 samp_out = 0;             // declaration of local variable

        simple_IIR();                   // call the function simple_IIR
        samp_out = round(y[0]);         // convert double to Int16
        mono_write_16Bit(samp_out);     // output the rounded samp_out
}

void simple_IIR(void)
{
        // perform delay operation
        x[1] = x[0];
        y[1] = y[0];

        // read new samples from H_Codec
        x[0] = mono_read_16Bit();

        // difference equation
        y[0] = b[0]*x[0] + b[1]*x[1] - a[1]*y[1];
}
```

5.1.2 Part of Code of Elliptic Filter

```
#include "coefs.txt"                    //includes coefficient file
int N = sizeof(a)/sizeof(a[0])-1;       //Calculate order of the filter

double *w;              //define pointer for dynamic allocation later
Int16 xin = 0;          //define input
double yout = 0;        //define output

void main()
{
        //using pointer define above to perform dynamic allocation
        w = (double*)calloc(N+1, sizeof(double));

        // initialize board and the audio port
        init_hardware();

        /* initialize hardware interrupts */
        init_HWI();

        /* loop indefinitely, waiting for interrupts */
        while(1)
        {};

}

void ISR_AIC(void)
{
        //extract data from provided funtion
        xin = mono_read_16Bit();

        IIR_transposed();               //calling function
        mono_write_16Bit(yout);          //output resuult
}
```

```
void IIR_non_transposed(void)
{
        int i = 0;              //initialize counter
        w[0] = xin;             //assign values to first buffer
        yout = 0;               //initialize output variable

        //translate equation into C code
        for(i=N;i>0;i--){
                w[0] -= a[i]*w[i];
                yout += b[i]*w[i];
                w[i] = w[i-1];
        }

        //Assign the last values to output variable
        yout += b[0]*w[0];
}
```

```
void IIR_transposed(void)
{
        int i = 0;      //initialize counter

        //assign values to output variable
        yout = w[0] + b[0]*xin;

        //translate equation into C code
        for(i=1;i<N;i++){
                w[i-1] = w[i] + b[i]*xin - a[i]*yout;
        }

        //Last one with no delay(special case)
        w[N-1] = b[N]*xin - a[N]*yout;
}
```

5.2 Code in Matlab

5.2.1 Code used to plot Figure 2 and Figure 3

```
clear
close all

%%%%%%%%%%%%%%%%%%%%%%%%% Calculate data %%%%%%%%%%%%%%%%%%%%%%%%%%
H = tf(1,[0.001 1]);            % Transfer Function
[mag,phase,wout] = bode(H);     % Calculate points for plots
f = wout/(2*pi);                % Convert angular frequency to frequency

%%%%%%%%%%%%%%%%%%%%%%%%% Plot graphs %%%%%%%%%%%%%%%%%%%%%%%%%%
% Not using bodeplot() directly to have more control on the axis limit Amplitude Response
figure
subplot(2,1,1)
semilogx(f, 20*log10(squeeze(mag)),'LineWidth',2)
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-30 0];
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
grid
title('Frequency Response of Analogue RC Simple-pole Filter')

% Phase Response
subplot(2,1,2)
semilogx(f, squeeze(phase),'LineWidth',2)
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-90 0];
xlabel('Frequency (Hz)')
ylabel('Phase (degree)')
grid

% Pole-zero Plot
figure
pzplot(H)
ax = gca;
ax.XLim = [-1200 200];
ax.YLim = [-1 1];
title('s-plane zero-pole plot for Analogue RC Simple-pole Filter')
```

5.2.2 Code used to plot Figure 10 and Figure 13

```
clear
close all

%%%%%%%%%%%%%%%%%% Read data from APx500 %%%%%%%%%%%%%%%%%%%
filename1 = 'allpass.xlsx';
xfreq = xlsread(filename1,'Gain','A5:A6142');
ygain1 = xlsread(filename1,'Gain','B5:B6142');
yphase1 = xlsread(filename1,'Phase','B5:B6142');

% Uncomment to choose data of different frequency response from different filters
filename2 = 'RC.xlsx';
%filename2 = 'non_transposed.xlsx';
%filename2 = 'transposed.xlsx';

ygain2 = xlsread(filename2,'Gain','B5:B6142');
yphase2 = xlsread(filename2,'Phase','B5:B6142');

% Frequency Response Correction
ygain = ygain2 - ygain1;
yphase = yphase2 - yphase1;


%%%%%%%%%%%%%%%%%%%% Analogue filter %%%%%%%%%%%%%%%%%%%%%%%
H = tf(1,[0.001 1]);    % analogue filter transfer function
[mag,phase,wout] = bode(H);
fa = wout/(2*pi);       % frequency of analogue filter

%%%%%%%%%%%%%%%%%%%%% Digital filter %%%%%%%%%%%%%%%%%%%%%%%
fs = 8000;
a = [1 -15/17];
b = [1/17 1/17];
[h, fd] = freqz(b,a,4096,fs);

%%%%%%%%%%%%%%%%%%%%%% Plot graphs %%%%%%%%%%%%%%%%%%%%%%%%%
% Not using freqz() directly to plot graphs
% in order to have more control on the axis limit.
% Magnitude plot
figure
semilogx(xfreq, ygain,'LineWidth',2)
hold on
semilogx(fd,20*log10(abs(h)),'r--','LineWidth',2)
semilogx(fa, 20*log10(squeeze(mag)),'g:','LineWidth',2)
hold off
grid
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-40 0];
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
legend('Corrected Actual', 'Theoretical Digital',...
    'Theoretical Analogue','Location','southwest')
title('Frequency Response Comparison - Gain')

% Phase plot
figure
semilogx(xfreq, yphase,'LineWidth',2)
```

```
hold on
semilogx(fd,angle(h)/pi*180,'r--','LineWidth',2)
semilogx(fa, squeeze(phase),'g:','LineWidth',2)
hold off
grid
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-90 0];
xlabel('Frequency (Hz)')
ylabel('Phase (degree)')
legend('Corrected Actual', 'Theoretical Digital',...
    'Theoretical Analogue','Location','southwest')
title('Frequency Response Comparison - Phase')
```

5.2.3 Code used to plot Figure 17 and Figure 18

```
clear all
close all

%%%%%%%%%%%%%%%%%%%% Calculate coefficients %%%%%%%%%%%%%%%%%%%%
fs = 8000;      % sampling frequency

% calculate the coefficients of bandpass elliptic filter
% the cutoff frequencies are normalized to Nyquist frequency
[b,a] = ellip(2, 0.3, 20, [270 450]*2/fs, 'bandpass');

% get points (f,h) from the frequency response
[h,f] = freqz(b,a,8192,fs);


%%%%%%%%%%%%%%%%%%%%%%%%% Plot graphs %%%%%%%%%%%%%%%%%%%%%%%%%%
% Not using freqz() directly to plot graphs
% in order to have more control on the axis limit.
% Magnitude Response
figure
subplot(2,1,1)
semilogx(f,20*log10(abs(h)))
grid
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-70 10];
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
title('Frequency Response of 16th order elliptic filter')

% Phase Response
subplot(2,1,2)
semilogx(f,angle(h)/pi*180)
grid
ax = gca;
ax.XLim = [5 4000];
xlabel('Frequency (Hz)')
ylabel('Phase (degree)')

figure
zplane(b,a)
title('Pole-zero plot of 4th order elliptic filter')


%%%%%%%%%%%%%%% Write coefficients to a txt file %%%%%%%%%%%%%%%%
fileID = fopen('coefs.txt', 'w');       % Open file

% write coefficient b into txt file
fprintf(fileID, 'double b[] = {');
fprintf(fileID, '%.16e', b(1));
fprintf(fileID, ', %.16e', b(2:end));
fprintf(fileID, '};\n');

% write coefficient a into txt file
fprintf(fileID, 'double a[] = {');
fprintf(fileID, '%.16e', a(1));
fprintf(fileID, ', %.16e', a(2:end));
fprintf(fileID, '};\n');

fclose(fileID);     % Close file
```

5.2.4 Code used to plot Figure 21, Figure 23, Figure 26 and Figure 28

```
clear
close all

%%%%%%%%%%%%%%%%%% Read data from APx500 %%%%%%%%%%%%%%%%%%%
filename1 = 'allpass.xlsx';
xfreq = xlsread(filename1,'Gain','A5:A6142');
ygain1 = xlsread(filename1,'Gain','B5:B6142');
yphase1 = xlsread(filename1,'Phase','B5:B6142');

% Uncomment to choose data of different frequency response from different filters
%filename2 = 'transposed.xlsx';
filename2 = 'non_transposed.xlsx';

ygain2 = xlsread(filename2,'Gain','B5:B6142');
yphase2 = xlsread(filename2,'Phase','B5:B6142');

ygain = ygain2 - ygain1;
yphase = yphase2 - yphase1;


%%%%%%%%%%%%%%%%%%%% Calculate coefficients %%%%%%%%%%%%%%%%%%%%
fs = 8000;      % sampling frequency

% calculate the coefficients of bandpass elliptic filter
% the cutoff frequencies are normalized to Nyquist frequency
[b,a] = ellip(2, 0.3, 20, [270 450]*2/fs, 'bandpass');

% get points (f,h) from the frequency response
[h,f] = freqz(b,a,4096,fs);


%%%%%%%%%%%%%%%%%%%%%% Plot graphs %%%%%%%%%%%%%%%%%%%%%%%%%
% Magnitude plot
figure
semilogx(xfreq, ygain,'LineWidth',2)
hold on
semilogx(f,20*log10(abs(h)),'r--','LineWidth',2)
hold off
grid
ax = gca;
ax.XLim = [5 4000];
ax.YLim = [-80 1];
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
legend('Corrected Elliptic Filter',...
    'Theoretical Elliptic Filter','Location','southwest')
title('Frequency Response Comparison - Gain')

% Phase plot
figure
semilogx(xfreq, yphase,'LineWidth',2)
hold on
semilogx(f,angle(h)/pi*180,'r--','LineWidth',2)
hold off
grid
ax = gca;
ax.XLim = [5 4000];
xlabel('Frequency (Hz)')
ylabel('Phase (degree)')
legend('Corrected Elliptic Filter',...
    'Theoretical Elliptic Filter','Location','southwest')
title('Frequency Response Comparison - Phase')
```