RTDSP Lab 5

Yihan Qi; CID:00813873
Lizhang Lin; CID:00840705

Contents

1. Single-pole Filter Design

1.1. Filter Design Using Tustin Transforms

Our first task is to map an analogue filter into a discrete time model using Tustin transform. The filter required is shown below:

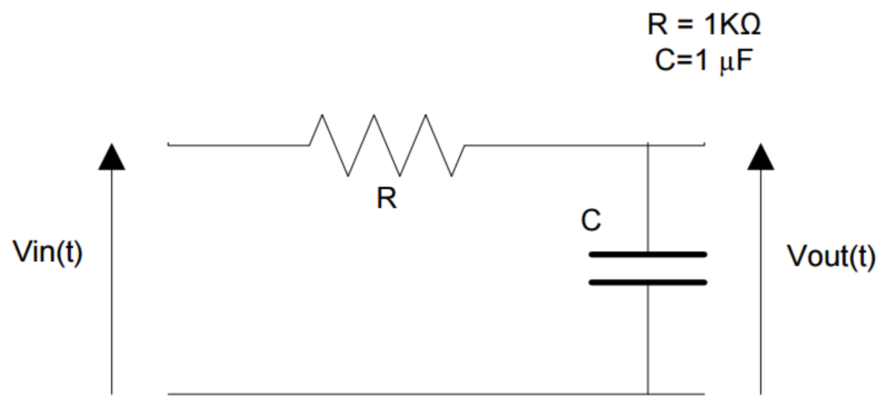


Figure 1 Low pass RC filter with $R = 1k\Omega \;\; C = 1\mu F$

From simple circuit analysis, we knew the filter in Figure 1 has the characteristic equation:

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

Then we applied Tustin's method to transform a continuous system in s-domain into a discrete one in z-domain. $z^{-1}$is reckoned as a unit delay which equals to the sampling time, T_s. In our case, sampling frequency is 8000Hz thus $T_s = \frac{1}{8000}s$. According to the characteristics of Laplace domain, we knew that multiplying by e^{-sT_s} is equivalent to introducing a delay by time T_s. Therefore we can write:

$$z = e^{sT_s}\,; \;\; s = \frac{1}{T_s}\ln z$$

An approximation of series expansion of logarithm leads us write:

$$s = \frac{2}{T_s}\frac{z-1}{z+1}$$

So we could replace all s with the equation above:

$$H\left(\frac{2}{T_s}\frac{z-1}{z+1}\right) = \frac{1}{1 + \frac{2}{T_s}\frac{z-1}{z+1} \times RC} = \frac{1 + z^{-1}}{\left(1 + \frac{2RC}{T_s}\right) + (1 - \frac{2RC}{T_s}) \times z^{-1}}$$

Substituted R, C, T_s with their values:

$$H(z) = \frac{1 + z^{-1}}{\left(1 + \frac{2 \times 1k \times 1\mu}{\frac{1}{8000}}\right) + \left(1 - \frac{2 \times 1k \times 1\mu}{\frac{1}{8000}}\right) \times z^{-1}}$$

$$H(z) = \frac{1 + z^{-1}}{(1 + 16) + (1 + 16) \times z^{-1}} = \frac{1 + z^{-1}}{17 - 15 \times z^{-1}}$$

The transfer function of an IIR filter is given by the equation:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}}$$

We manipulated our filter transfer function as shown below in order to match the coefficients:

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

Clearly, we got coefficients:

$$b_0 = \frac{1}{17}, b_1 = \frac{1}{17}, a_1 = -\frac{15}{17}$$

1.2. Implementation in C

The convolution of a direct form I IIR filter is described by the equation:

$$y[n] = \sum_{i=0}^{M} b[i]x[n-i] - \sum_{i=1}^{N} a[i]y[n-i]$$

To implement this expression into C language, the equation needs to be optimized with a single summation chain so that the multiply summations can be implemented by a single ***for*** loop. We noticed that if we regarded b[0] as an outlier of the summation chain and handled $b[0]x[n]$ individually; two summation chains could be written as one. Moreover, we were aware that both summation operations may have different lengths so the larger number between M and N were chosen as the length of the new summation chain, and the rest of a and b coefficients are set to 0.

$$y[n] = b[0]x[n] + \sum_{i=1}^{\max(M,N)} b[i]x[n-i] - a[i]y[n-i]$$

The related diagram is shown below:

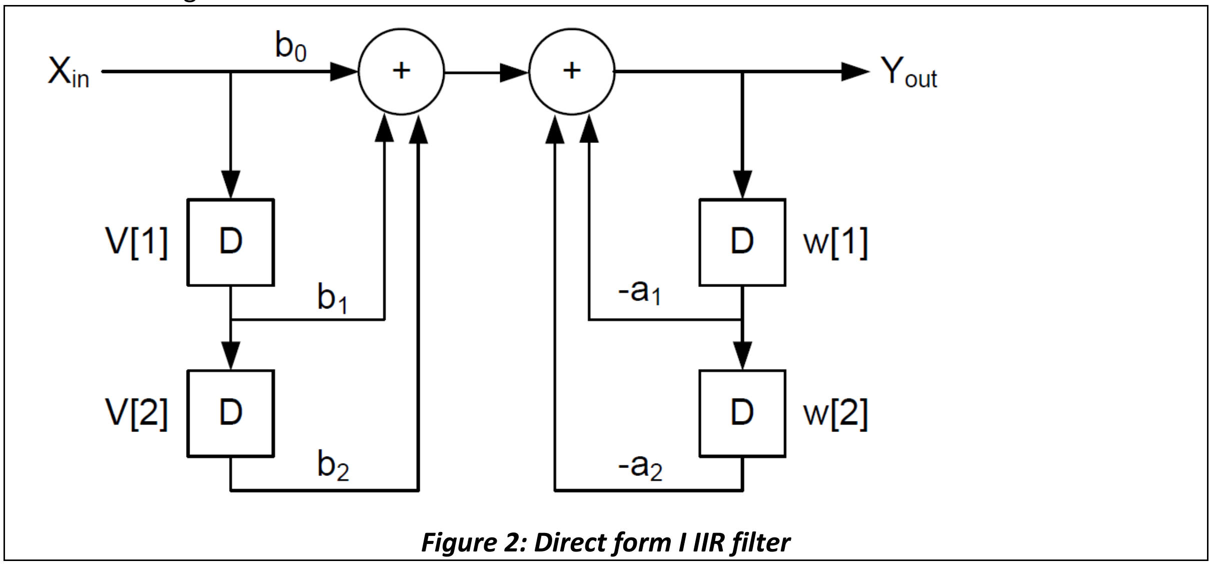


Figure 2: Direct form I IIR filter

Its implementation in C language is shown in the code below.

```
/******************** LAB 5 direct_implementation***********************************/
double direct_implementation(void){
 //this function works for any order number of IIR filter
 //this function implements direct form
 //it uses non-circular buffer, thus there is a potential enhancement
   samp = mono_read_16Bit(); //read from ADC and store the input into a 16bit int variable
   yout = b[0]*samp;//first coefficient multiplication
   for(i=order;i>0;i--){
      yout += x[i]*b[i]-y[i]*a[i]; //coefficients multiplication arithmetic
      y[i]=y[i-1]; //delay by shifting y[] buffer by one element
      x[i]=x[i-1]; //delay by shifting y[] buffer by one element
   }
   x[1] = samp; //x[1] is the buffer which stores current input
   y[1] = yout; //y[1] is the buffer which stores current output
   return yout;
}
```

This code is generic for any ORDER of direct form implementation. Firstly, we must note that y[0] and x[0] are junk values (buffers) since they are not used so that we can always multiply b[i] with x[i] and a[i] with y[i]. Although y[1] and x[1] are updated with y[0] and x[0] inside the ***for*** loop but are set back to the correct values outside the ***for*** loop. In addition, the delays are updated in reverse order.

1.3. Result Analysis

1.3.1. Frequency Response

Frequency response of the design were obtained via Audio APX500. In order to achieve comprehensive approaches, three frequency responses were compared, which are noted as ‘processed output’, ‘ideal digital filter’ and ‘ideal analogue filter’. (will be analysed and explained later)

Firstly, we should have a look at figure 3 below.

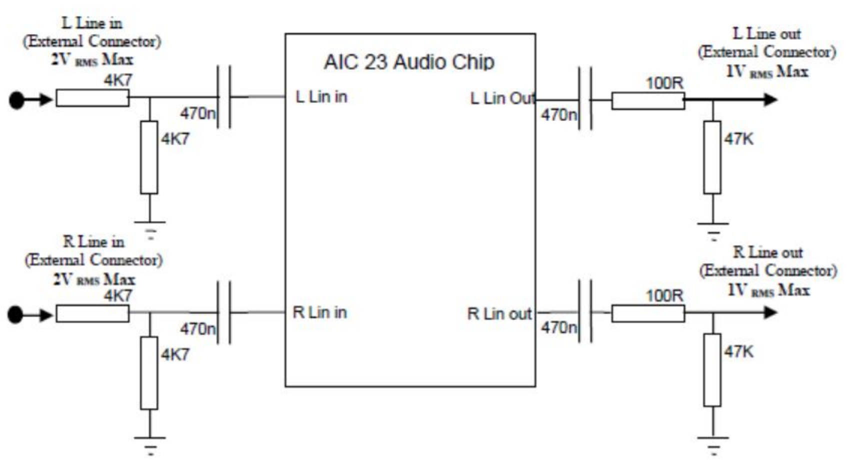


Figure 3: External components of AIC 23 Audio chip

From figure 3, we could see a RC filter (high pass filter) at the output terminal of AIC 23 Audio Chip, which results in a small gain at very low frequency for the single pole low pass filter implementation (as shown in figure 4). We noted the unprocessed frequency response of the filter as ‘raw data’.

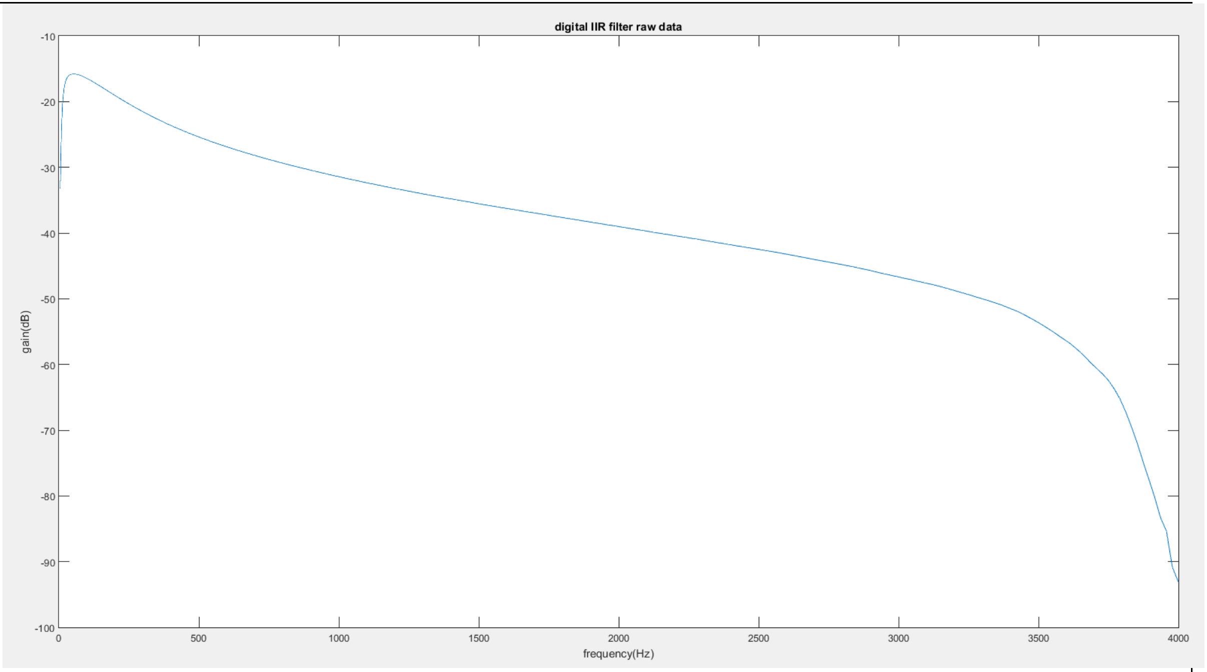


Figure 4: unprocessed frequency response of single-pole digital IIR filter. (haven't subtract the response of DSK board)

Therefore, we measured the bare board frequency response (figure 5), which behaves as we expected at low frequency (high pass filter). In addition, its gain attenuates as the frequency approaches Nyquist frequency, 4000 Hz. This is due to the existence of anti-aliasing filter whose real cut-off frequency is below 4000 Hz. Therefore, the gain starts to attenuate before reaching 4000 Hz.

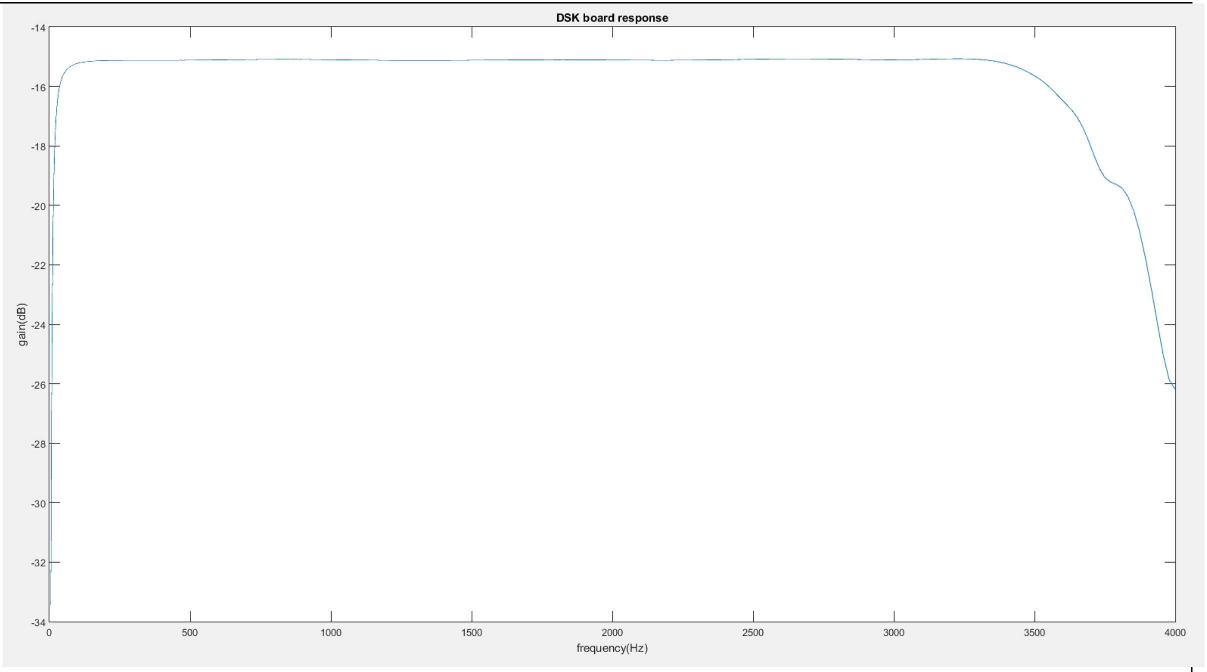


Figure 5: frequency response of DSK board

In order to verify our IIR digital filter implementation in C language, we subtracted DSK board response from the raw data. Then we plotted the resulting frequency response, denoted as 'processed output', in figure 6 along with the frequency response of the RC (analogue) and digital filters.

Moreover, to make fair comparison and accurate analysis, we always subtracted bare board response to require a 'processed' frequency response for other implementations that will be mentioned in the sections 2&3.

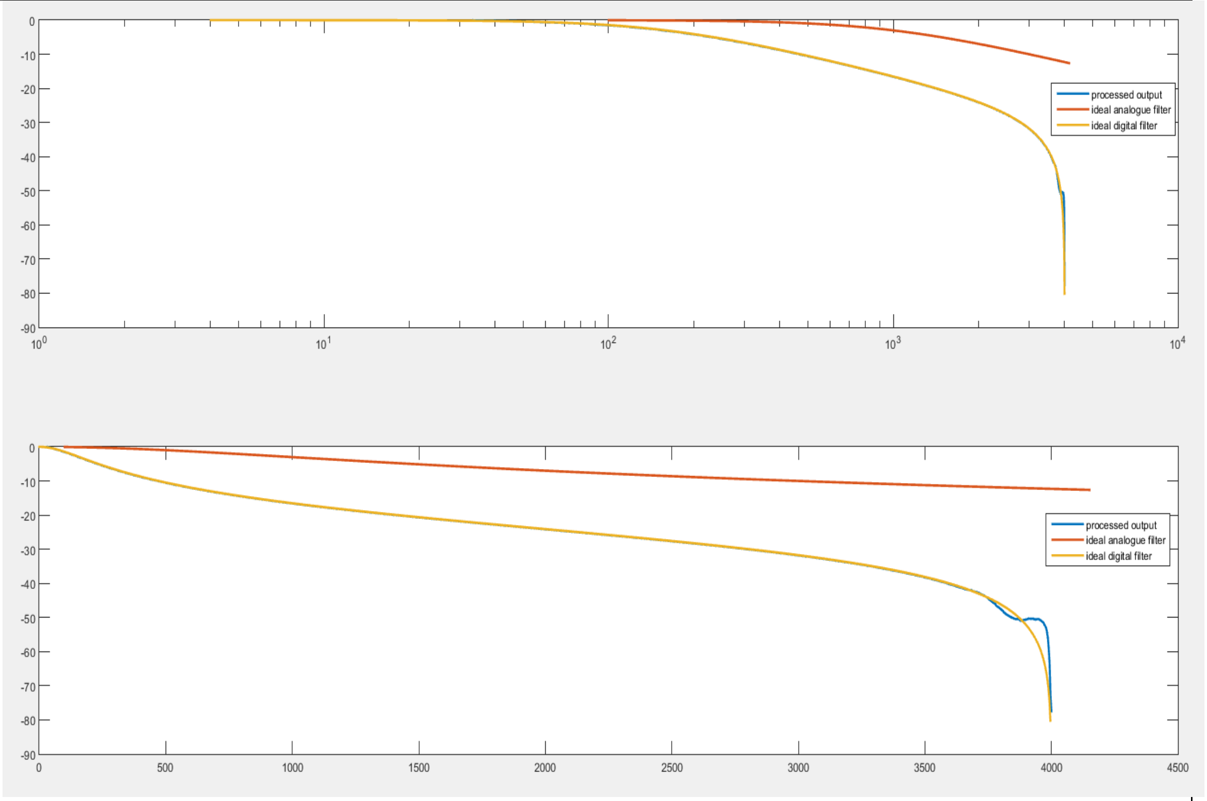


Figure 6: plots of frequency responses of processed output, ideal analogue filter and ideal digital filter with logarithm and linear x-axes.

Note that 'ideal analogue filter' is the frequency response of the analogue RC filter. Similarly, 'ideal digital filter' is the frequency response of the digital filter whose coefficients were mapped from the analogue filter by Tustin transformation.

The ideal digital filter was directly implemented by Matlab function ***freqz()***. In addition to digital filter, we determined analogue RC filter frequency response with transfer function $H(s) = \frac{1}{1+sRC}$ and implemented it using Matlab function ***freqs()***. The code can be found in section 1.3.1.4.

1.3.1.1. Analysis between Ideal IIR Digital Filter and Processed Output

From figure 6, we found that the processed output response is very closely matched to the expected ideal digital filter except in high frequency range from 3700 to 4000 Hz. In another word, the practical frequency response no longer matched to the ideal one as the frequency approaches to the Nyquist frequency (4000 Hz). This is due to the anti-aliasing filter at the input of DSK tool, which doesn't behave ideally. Its cut-off frequency is around 3700 Hz instead of 4000 Hz, and its frequency rolls off slowly instead of having a sharp decrease. This defect leads to imperfect reconstruction of sine wave at frequencies near 4000 Hz.

1.3.1.2. Analysis between Analogue Filter (RC Circuit) and converted Digital Filter

Moreover, from figure 6, we noticed that the difference between the frequency responses of analogue filter and its mapped digital IIR filter. The divergence exists especially at high frequency region.

This can be explained by approximations made in Tustin transform and Nyquist frequency effects.

Looking back to section 1.1, $\omega_{continuous} = \frac{2}{T_s}\tan\frac{\omega_{discrete}T_s}{2}$ $where\ T_s = \frac{1}{8000}s$. As $\omega_{discrete}$ approaches Nyquist frequency (4000Hz), we could see that

$$\omega_{continuous} = \lim_{\omega_{discrete}\to 2\pi\times 4000Hz}\frac{2}{T_s}\tan\frac{\omega_{discrete}T_s}{2} = \lim_{\omega_{discrete}\to 2\pi\times 4000Hz}\frac{2}{T_s}\tan\frac{2\pi\times 4000\times\frac{1}{8000}}{2} = +\infty\ (\boldsymbol{since\ \tan\frac{\pi}{2}} = +\infty).$$

Based on the above analysis, we concluded that the RC analogue filter will have a similar Nyquist effect of the digital filter only in a case where $\omega_{continuous}$ approaches infinity. This is also the reason why the gain of RC filter is decaying very slowly and diverging from digital filter response.

1.3.2. Cut-off Frequency

Now we would like to find the cut-off frequency.

Theoretically, cut off frequency of continuous filter is defined as $f_{3dB\ continuous} = \frac{1}{2\pi RC} = \frac{1}{2\pi\times 10^{-3}} \cong 159.155\ Hz$ and $\omega_{continuous} = \frac{2}{T_s}\tan\frac{\omega_{discrete}T_s}{2}$.

Thus, we could also write down theoretical cut off frequency of discrete Tustin transformed z-domain as
$\omega = 2\pi\times\ f_{3dB\ discrete} = \frac{2}{T_s}\tan^{-1}\left(\frac{2\pi\times f_{3dB\ continuous}\times T_s}{2}\right) \to f_{3dB\ discrete} = 158.943Hz.$

In practice, however, the real cut off frequency obtained from APX500 may suffer from many interferences. After measurement, we found that $f_{3dB\_real(unprocessed)} = 189.103Hz$ (observed from unprocessed data, figure 7). As we discussed previous, there is a high pass RC filter in the DSK board which increases the cut off frequency.

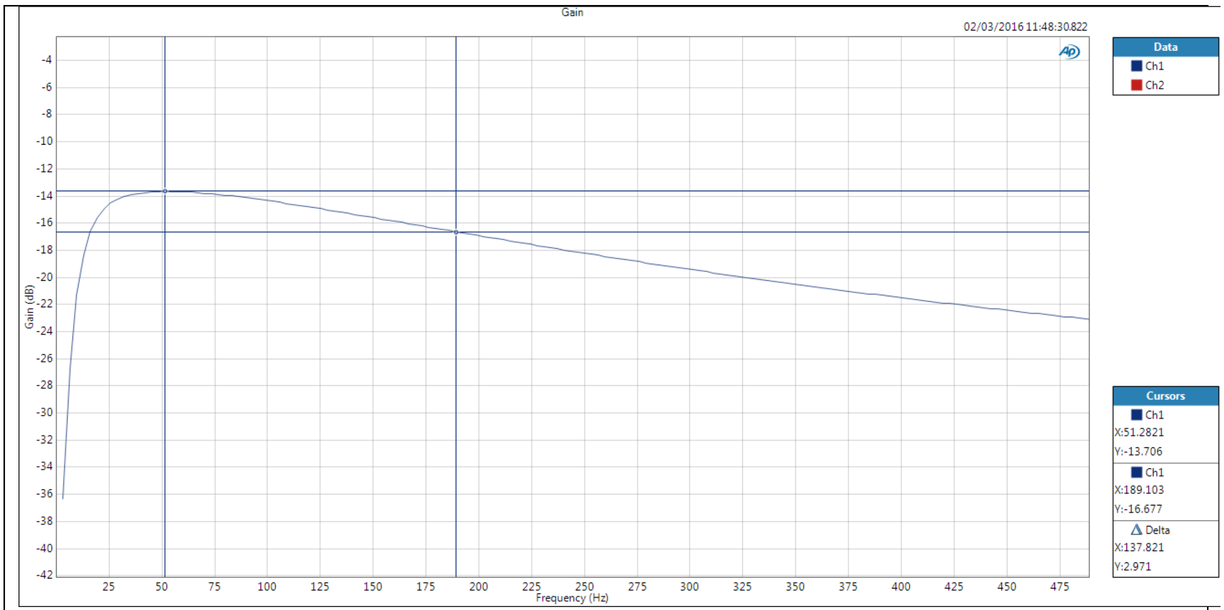


Figure 7: the unprocessed frequency response of the single-pole IIR digital filter. (cut-off frequency is also the -3dB point) In this case, cut-off frequency is 189.103 Hz.

Hence we removed the effect caused by the DSK board, hence acquired a processed response (figure 8). From figure 8, we observed that $f_{3dB_{real(processed)}} = 158.2\ Hz$

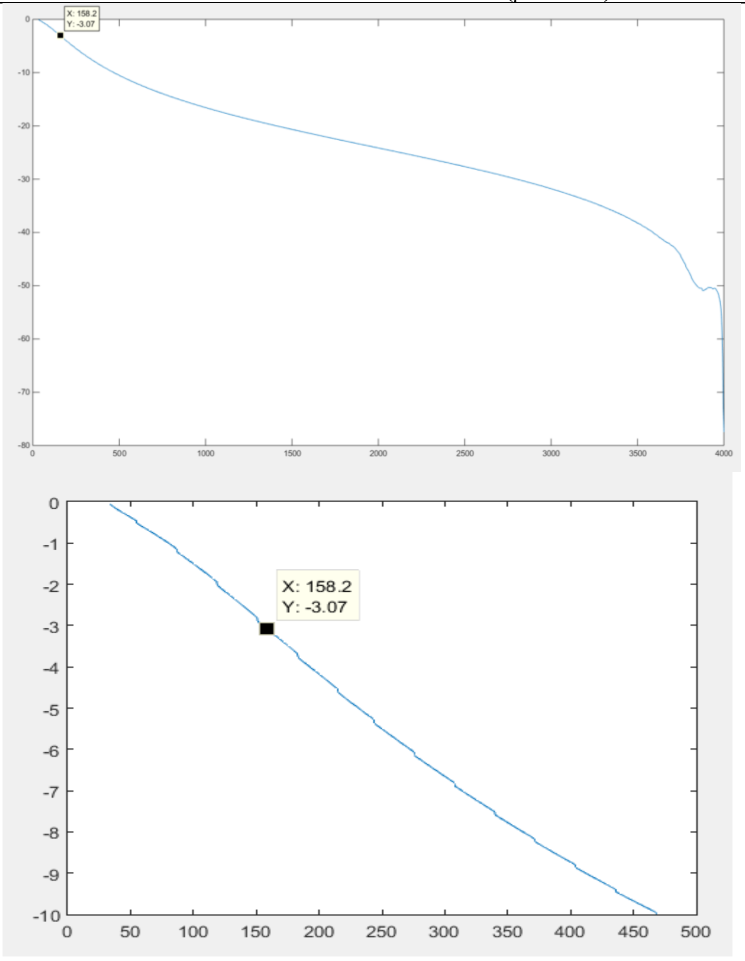


Figure 8: the top figure is the processed frequency response of our implementation; the bottom one is the amplified version of the top one such that we could see the -3dB point.

1.3.3. Time Constant

The time constant is defined by the amount of time taken to achieve a magnitude decrement by a factor of $1 - \frac{1}{e} \cong 0.6321$, which is identical to the effect of -3dB attenuation. The estimated time constant is $\tau = RC = 1k\Omega \times 1\mu F = 1ms$. Look at figure 8, we found that the -3dB point is at 158.02 Hz. Hence, we concluded that the actual time constant $\tau = \frac{1}{2\pi f_{3dB\_real(processed)}} = 1.006ms$ which is very close to the ideal value, $1ms$.

1.3.4. Plots in Matlab

A full version of Matlab code below shows how the bare board response was subtracted and how all the three frequency responses were produced in a single diagram.

```
%**********************find the processed output response**********************
%read bare board response
filename = 'bareBoard.xlsx';
sheet = 'Gain';
xlRange = 'A5:B1252';
gain_fre_board = xlsread(filename,sheet,xlRange);%get date from file
gain_board = gain_fre_board(1:1:(length(gain_fre_board)-1),2);%array of gains
%read unprocessed response of digital filter
filename = 'task1_gainLargeRange.xlsx';
sheet = 'Gain';
xlRange = 'A5:B1252';
gain_fre_filter = xlsread(filename,sheet,xlRange);%get date from file
gain_filter = gain_fre_filter(1:1:length(gain_fre_filter),2);
fre_filter = gain_fre_filter(1:1:length(gain_fre_filter),1);
%result of subtraction
gain_real = gain_filter - gain_board;
%plot result of processed output %subtract the effect from bare board
plot(fre_filter(10:1:1247),gain_real(10:1:1247),'Linewidth',2);
hold on;

%**********************find the ideal response of the analog filter**********************
[h, w]=freqs([1],[0.001 1]);
plot(w(1:1:162), db(h(1:1:162)),'Linewidth',2);
hold on;

%**********************find the ideal response of the digital filter***********************
[hd, wd] = freqz([1/17 1/17],[1 -15/17],1000,8000);
plot(wd, db(hd),'Linewidth',2);
legend('processed output', 'ideal analogue filter', 'ideal digital filter');
```

Note that in the following sections where frequency response comparison plots will be mentioned again, we don't provide Matlab codes anymore since they are similar to the code above.

2. Direct Form II

In this section, we improved our previous code by implementing direct form II. We were given certain design specifications of the 4th order band pass IIR filter, which was created by Matlab. We measured the frequency response of our design using audio analyser AXP500. It was then subtracted by bare DSK board response to make a fair comparison with the ideal filter response in Matlab. In the last part of this section, we varied the order number of the IIR filter, measured the corresponding instruction cycles at different compiler optimisation levels, and obtained the relationship between order number of filter and the number of instruction cycles.

2.1. IIR Filter design using Matlab

IIR filter design specifications:
Order: 4th
Passband range: 180-450 Hz
Passband ripple: 0.4 dB
Stopband attenuation: 23dB

Following the design specifications, we used Matlab function ***ellip*** to create a digital IIR filter and wrote the coefficients in a text file in C language format. In addition, we plotted frequency response of the filter and the distribution diagram of poles and zeros in Z-plane. In the following paragraphs, Matlab code was sectioned and explained by parts. The full version of Matlab code can be found in the appendix.

2.1.1. Matlab - Create IIR Filter

The first part of Matlab code below creates the filter and stores coefficients in ***a*** and ***b*** respectively.

```
%This creates a digital IIR filter using ellip, and stores the coefficients
%into a text file.
%design specification
order = 4; %order: 4th
fp = [180/4000 450/4000]; %passband: 180-450 Hz
rp = 0.4; %passband ripple: 0.4dB
stop_atte = 23; %Stopband attenuation: 23dB
%ellip(order/2, ripple(dB), stopband attenuation (dB), Passband frequencies)
[b, a]=ellip(order/2, rp, stop_atte, fp);%get
```

2.1.2. Matlab - Store Coefficients in Text File

Then we stored the coefficients in a text file called ‘coef.txt’.

```
%write coefficients into a text file
file = fopen('coef.txt','w');
fprintf(file,'#define order %d\n',order); %define order number
fclose(file);
%coefficient a
file = fopen('coef.txt','at');
fprintf(file,'double a[]={');
dlmwrite ('coef.txt',a,'-append');
fprintf(file,'};\n');
%coefficient b
fprintf(file,'double b[]={');
dlmwrite ('coef.txt',b,'-append');
fprintf(file,'};');
fclose(file);
```

```
copyfile('coef.txt', 'H:\RTDSPlab\lab5\RTDSP');%copy file into RTDSP project
```

Note that we not only declared the coefficients of ***a*** and ***b*** but also defined the size of order number in Matlab. Therefore, there is no need to operate dynamic memory using ***calloc()***. We believed it create a more efficient C code since the length of array is already determined, thus compiler level optimisations, such as loop unwinding, can be operated earlier.

The content of the text file 'coef.txt' is shown below.

```
#define order 4
double a[]={1,-3.5748,4.8975,-3.0531,0.73249
};
double b[]={0.075949,-0.23461,0.31747,-0.23461,0.075949
};
```

2.1.3. Matlab – Plots

The last part of our Matlab code plots frequency response (figure 9) of the IIR filter, and the distribution diagram (figure 10) of poles and zeros in Z-plane.

```
%plot frequency response of the digital filter
figure(1);
freqz(b,a,1000,8000)%1000 is the number of points which must match the no. of data of audio analyser
               %8000 is the sampling frequency

%locations of zeros and poles in z-plane
figure(2);
zplane(b,a);
```

The plots are shown below.

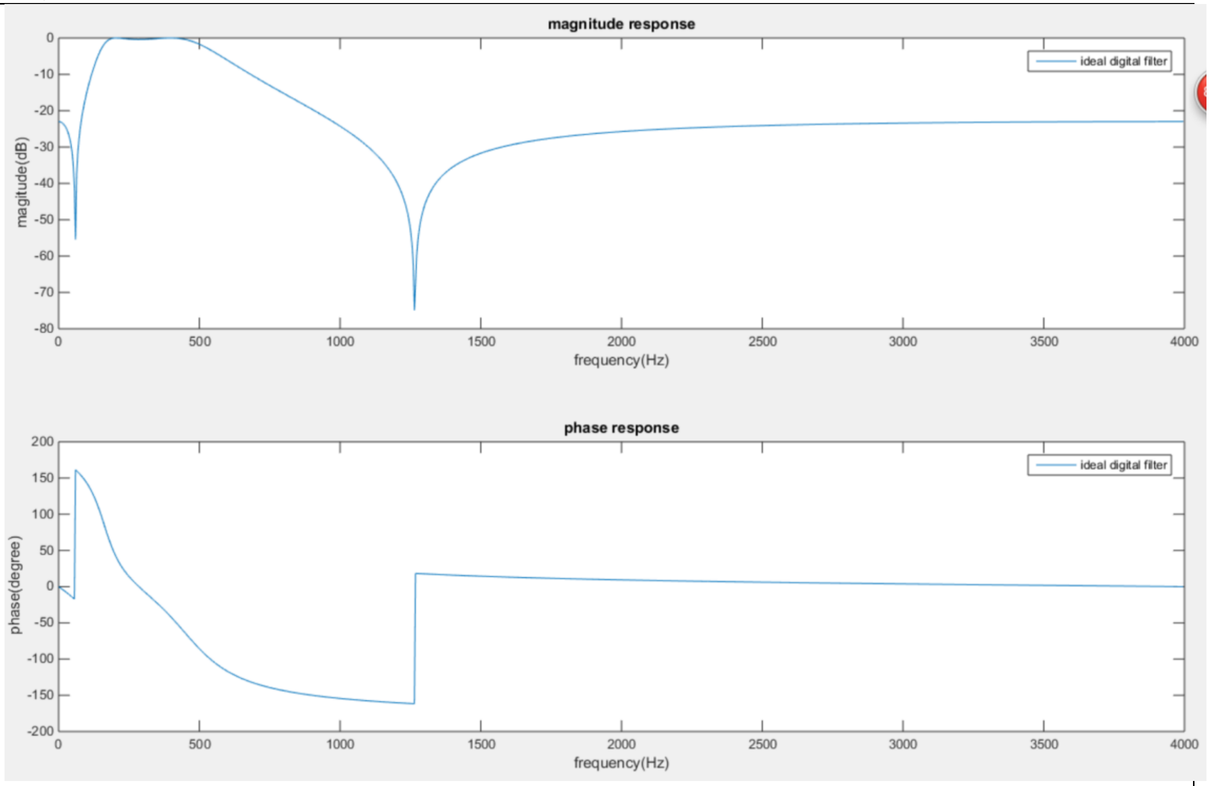


Figure 9: Frequency response of the IIR filter, labelled as 'ideal digital filter'.

As we could see that the phase response in the passband is non-linear, which is what we expected for an IIR filter.

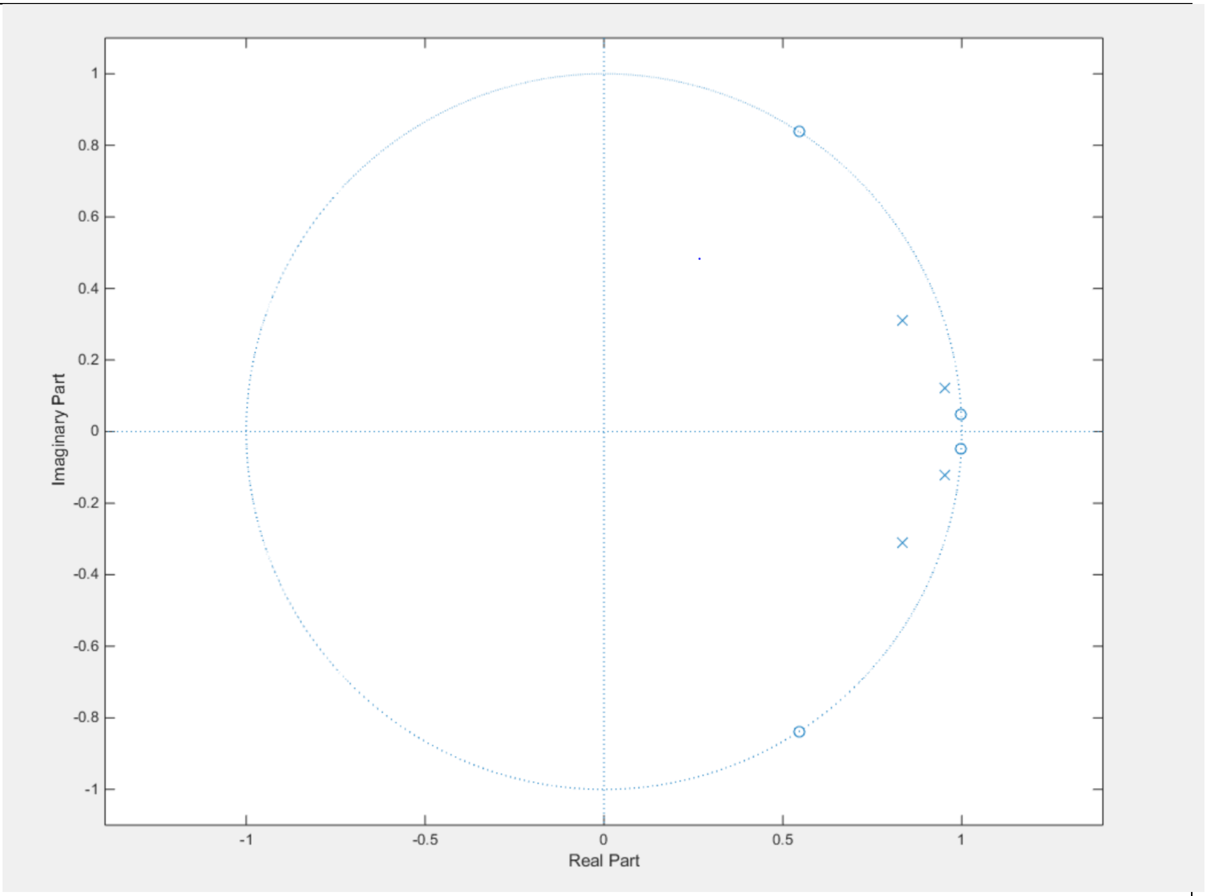


Figure 10: distribution of poles and zeros in Z-plane

From figure 10 we discovered that all the poles are inside the unit circle, which indicates a stable IIR filter design.

2.1.4. Matlab - Check the Filter with Design Specifications

Before proceeding to measure the frequency response of the filter implemented in C language, we must check whether the frequency response of the 'ideal filter' matches the design specifications. Hence, we wrote Matlab scripts to determine whether the maximum deviation in the pass band matches 0.4 dB (demonstrated in figure 11), and to check if the stop band attenuation has reached -23 dB (shown in figure 12).

2.1.4.1. Check Passband Ripple

The Matlab code is shown below. It calculates the maximum and minimum magnitudes within the pass band range (from 180 to 450 Hz), and added two horizontal lines which are labelled 'pass band upper bound' and 'pass band lower bound' respectively.

```
%design specification
order = 4; %order: 4th
fp = [180/4000 450/4000]; %passband: 180-450 Hz
rp = 0.4; %passband ripple: 0.4dB
stop_atte = 23; %Stopband attenuation: 23dB
%ellip(order/2, ripple(dB), stopband attenuation (dB), Passband frequencies)
[b,a]=ellip(order/2, rp, stop_atte, fp);%get

%magnitude plot
[h, w] = freqz(b,a,1000,8000);%1000 is the number of points which must match the no. of data of audio analyser
plot(w,db(h));%magnitude
hold on;

%add horizontal lines
flag1=0;
flag2=0;
for i=1:1:length(w)
  if(w(i)>=180 && flag1 == 0)
      index_LB = i; %finds the index of lower frequency bound (180)
      flag1 = 1;
  end
  if(w(i)>=450 && flag2==0)
      index_UB = i; %finds the index of lower frequency bound (450)
      flag2 = 1;
  end
end
UB = max(db(h(index_LB:1:index_UB))); %upper boundary
LB = min(db(h(index_LB:1:index_UB))); %lower boundary
plot ([180 450],[UB UB]);%plot upper boundary
hold on;
plot ([180 450],[LB LB]);%plot lower boundary
title('magnitude response');
xlabel('frequency(Hz)');
ylabel('magitude(dB)');
legend('ideal digital filter','pass band upper bound','pass band lower bound');
```

The graph generated by Matlab is shown below.

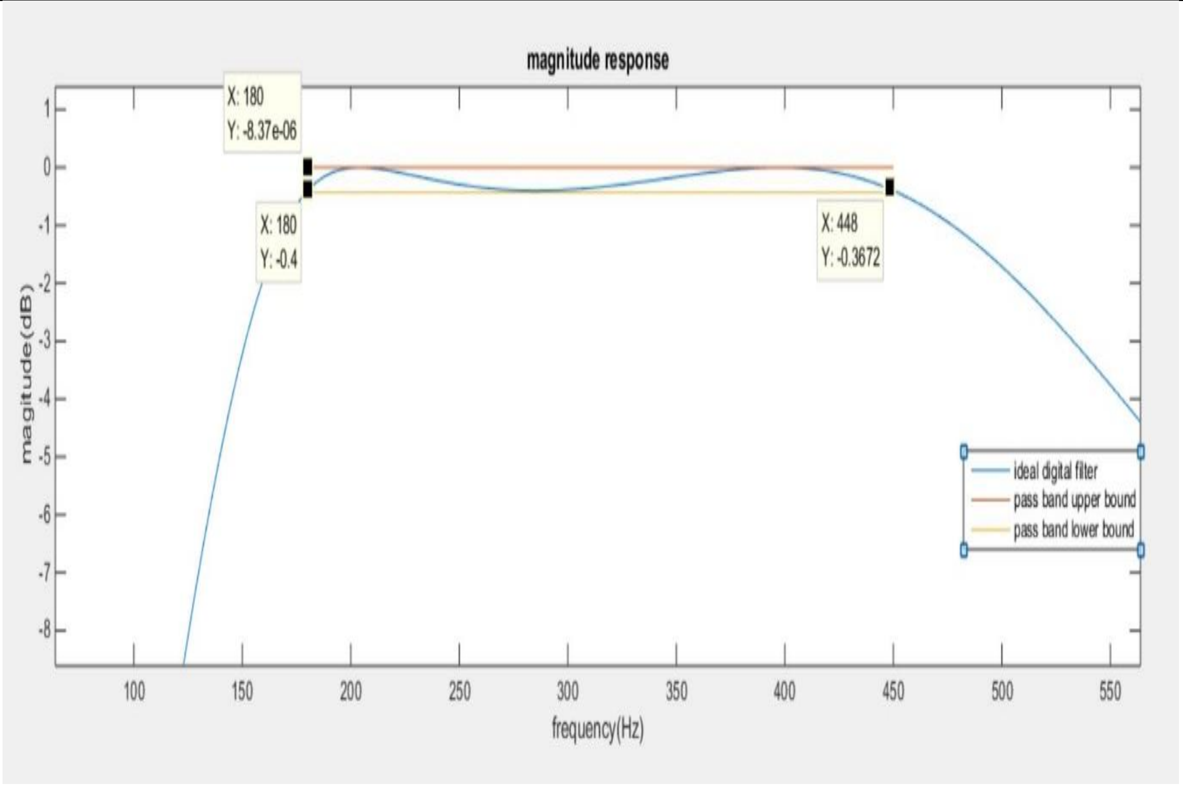


Figure 11: check passband ripple

From the result of figure 11, we determined that the pass band ripple of the ideal digital filter has achieved our expectation since the maximum deviation of pass band ripple is exactly 0.4dB.

2.1.4.2. Check Stopband Attenuation

Similarly, we wrote a Matlab script to check whether the stopband attenuation has achieved the design specification before proceeding.

```
%design specification
order = 4; %order: 4th
fp = [180/4000 450/4000]; %passband: 180-450 Hz
rp = 0.4; %passband ripple: 0.4dB
stop_atte = 23; %Stopband attenuation: 23dB
%ellip(order/2, ripple(dB), stopband attenuation (dB), Passband frequencies)
[b,a]=ellip(order/2, rp, stop_atte, fp);%get
figure(1);

%magnitude
subplot(211);
[h, w] = freqz(b,a,1000,8000);%1000 is the number of points which must match the no. of data of audio analyser
plot(w,db(h));%magnitude
hold on;
%add horizontal lines
%UB = -0.2; %upper boundary
LB = -23.2; %lower boundary
%plot ([5 4000],[UB UB]);%upper boundary
%hold on;
plot ([0 4000],[LB LB]);%lower boundary
title('magnitude response');
xlabel('frequency(Hz)');
ylabel('magitude(dB)');
legend('ideal digital filter','stop band attenuation');
```

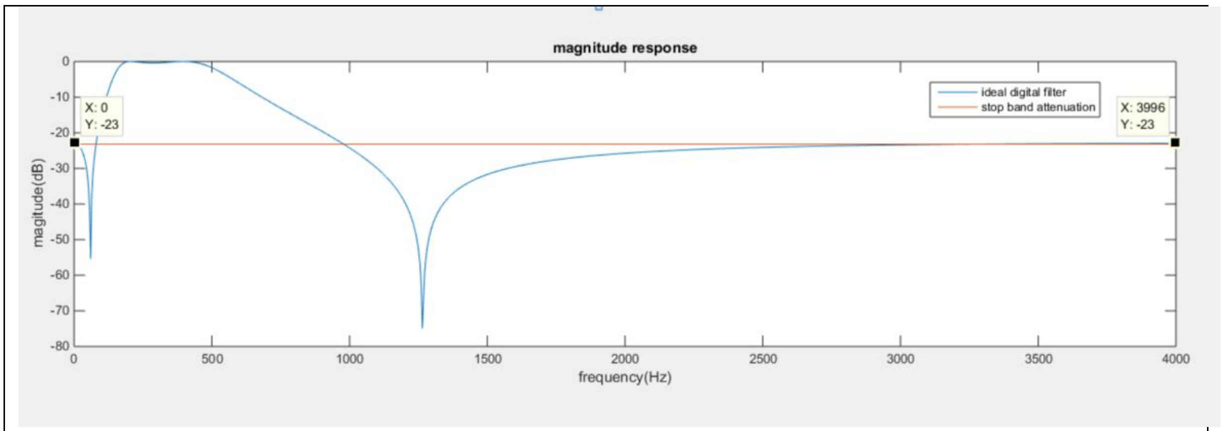


Figure 12: check stopband attenuation

From the result of figure 12, we discovered that the stop band attenuation of ideal digital filter matches our expectation since the stop band magnitude is always below -23dB and is extremely approximate to -23 dB at high frequencies.

2.2. Implementation in C

Figure 13 below illustrates the direct form II IIR filter. The output ***yout*** can be expressed by equation:

$$yout = step[0] * b[0] + step[1] * b[1] + step[2] * b[2] + \cdots + step[order] * b[order]$$

where ***order*** means the order of IIR filter.

From the equation, we can see that output can be implemented by only one coefficient ***b***, which allows us to focus on the buffers ***step[]***.

The buffers step[] can be expressed by equations:

$$step[0] = -a[1] * step[1] - a[2] * step[2] - \cdots - -a[order] * step[order]$$

$$step[1] = step[0] in\ the\ last\ sampling$$

$$step[2] = step[1] in\ the\ last\ sampling$$

$$\vdots$$

$$step[order] = step[order - 1] in\ the\ last\ sampling$$

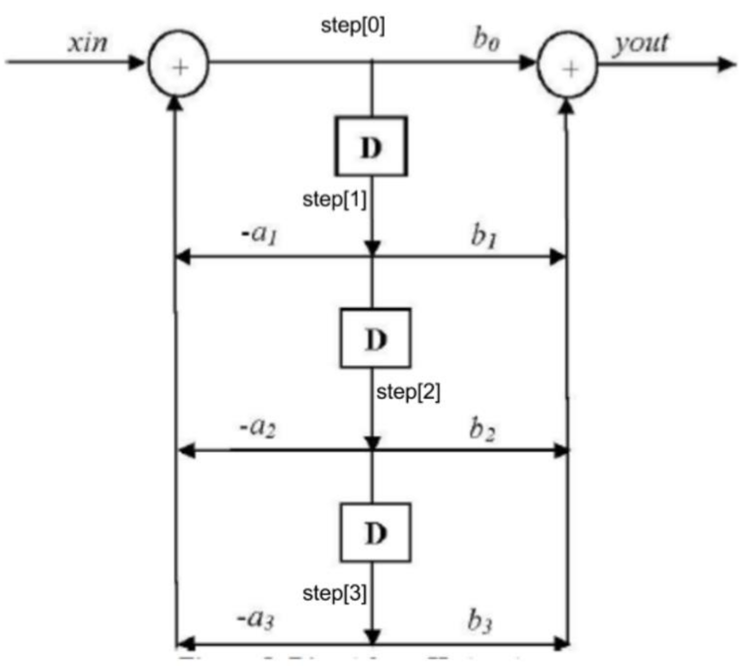


Figure 12: direct form II IIR filter

We implemented the mathematic equations of the output ***yout*** and the buffers ***step[ ]*** in the C code below. The code implements ***yout*** as shown in the equation we just derived and updates ***step[ ]*** by performing multiply accumulations for ***step[0]*** and shifting the rest of the buffers by one element in a ***for*** loop.

```
/******************* LAB 5 directTwo_implementation**********************************/
double directTwo_implementation(void){
 //this function works for any order number of IIR filter
 //It implements direct form II non-transposed
 //it uses non-circular buffer, thus there is a potential enhancement, which will be seen in the next implementation
   step[0] = mono_read_16Bit();//read from ADC and store the input into a 16bit int variable
   yout = 0;//reset output before processing
   for(i=order; i>0; i--){
      step[0]-=a[i]*step[i];//multiplication accumulation of -a[] coefficients
      yout+=b[i]*step[i];//multiplication accumulation of b[] coefficients
      step[i]=step[i-1];//performs delay by shifting the buffer by one element
   }
   yout +=b[0]*step[0];//write the last multiply accumulation to the output
   return yout;
}
```

2.3. Frequency Response Check

In this subsection, we measured the raw frequency response of the direct form II filter design as well as the bare board using audio analyser. The data of their magnitudes and phase plots were then exported into excel files, which could be read by Matlab.

As we have discussed in section 1, the DSK board has an internal anti-aliasing filter with cut-off frequency 4000 Hz as well as a high pass filter. Therefore, in order to avoid the effect caused by these additional filters, the bare board response was subtracted from the raw frequency response. Thus, we obtained the real frequency response of the direct form II filter design, noted as 'processed output' (figure 13 & 14), which is compared with the frequency response of ideal digital filter.

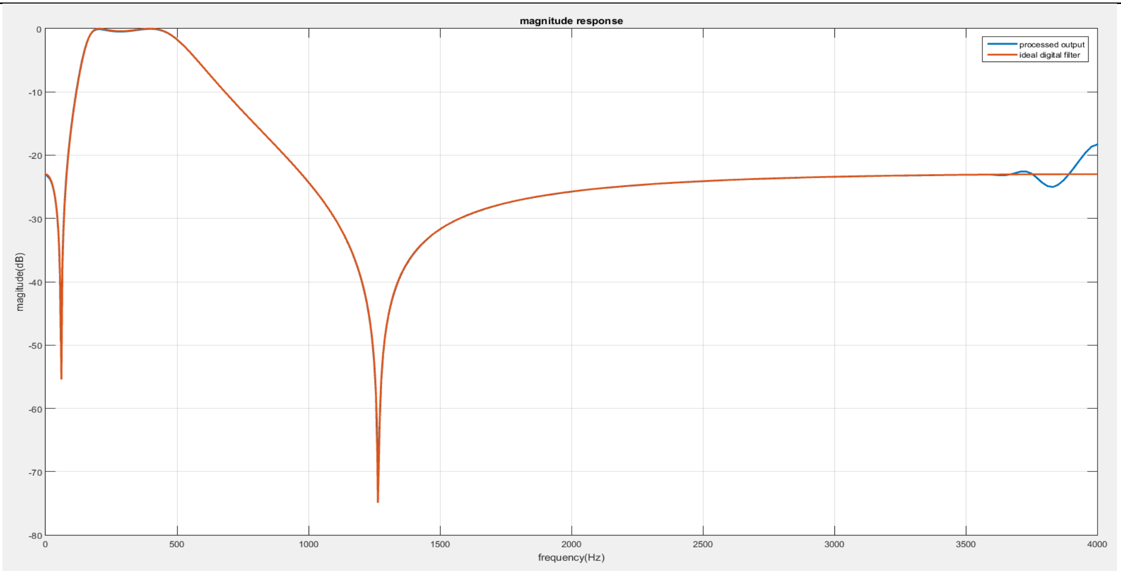


Figure 13: the magnitude plot

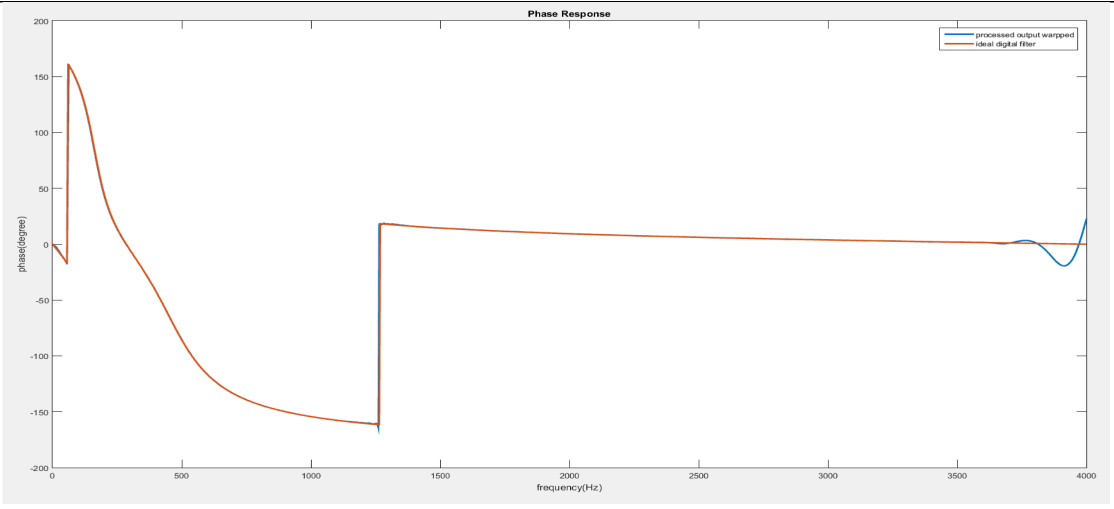


Figure 14: phase plot, wrapped into 180 degrees.

From figure 13 and 14, we found that the frequency response of the 'processed output' and the ideal filter almost matches. However, mismatch occurs as the frequency approaches to Nyquist frequency (4000 Hz). We have discussed it in section one. The main reason is that the anti-aliasing filter is non-ideal. Therefore, the reconstruction process introduces errors near 4000 Hz.

3. Direct Form II Transposed

3.1. Implementation in C

Figure 15 below illustrates the direct form II transposed IIR filter. The output ***yout*** can be expressed by equation:

$$yout = step[0] + xin * b[0]$$

From the equation, we can see that the output depends only the buffer ***step[0]***. Therefore, we could focus on the buffer ***step[0]***.

The buffers ***step[ ]*** can be expressed in the following equations:

$$step[0] = b[1] * xin - a[1] * yout + step[1]$$

$$step[1] = b[2] * xin - a[2] * yout + step[2]$$

$$step[2] = b[3] * xin - a[3] * yout + step[3]$$

$$\vdots$$

$$step[order - 2] = b[order - 1] * xin - a[order - 1] * yout + step[order - 1]$$

$$step[order - 1] = b[order1] * xin - a[order] * yout$$

Note the buffers ***step[ ]*** are updated in ascending index sequence, e.g. ***step[1]*** will not be updated until ***step[0]*** has been updated. In addition, the last express for buffer ***step[order-1]*** only depends on coefficients ***a*** and ***b***.

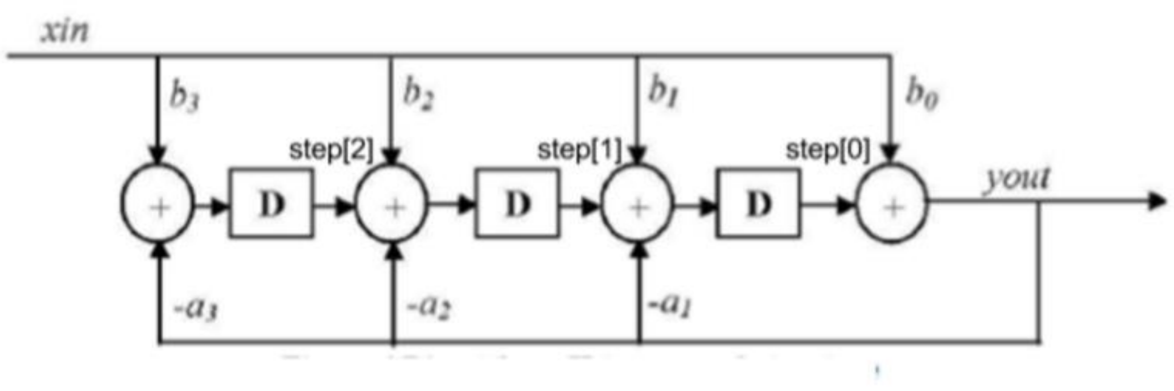


Figure 15: Direct form II transposed implementation

The above algorithm was implemented in C language as shown in the code below where the global double variable ***input*** stands for ***xin*** in figure (). As we described that the buffer step[ ] are updated in ascending index sequence, the ***for*** loop was therefore programmed with index ***i*** in ascending order from 0 to ***order-2.***

```
/******************** LAB 5 directTwo_Transposed_implementation*************************/
double directTwo_Transposed_implementation(void){
 //this function works for any order number of IIR filter
 //It implements direct form II transposed
 //it uses ncircular buffer, which significantly enhanced the code in terms of instruction cycles
   input = (double) (mono_read_16Bit()); //read from ADC, cast and store the input into a double variable
   yout = step[0] + b[0]*input;//write the output
   for(i=0; i<order-1; i++){
      step[i] = step[i+1] - a[i+1]*yout + b[i+1]*input;//sum of previous buffer and coefficients multiplications
   }
   step[order-1] = -a[order]*yout + b[order]*input;//writes the last buffer
   return yout;
}
```

3.2. Filter Response Check

Again, we must verify whether the filter we implemented in C resembles the ideal IIR filter designed in Matlab.
Like we did previously, in order to avoid additional filtering effect from anti-aliasing and high-pass filters from DSK board, the bare board response was subtracted from the raw frequency response. Thus, we obtained the real frequency response of the direct form II transposed filter design, noted as ‘processed output’ in figure 16 & 17.

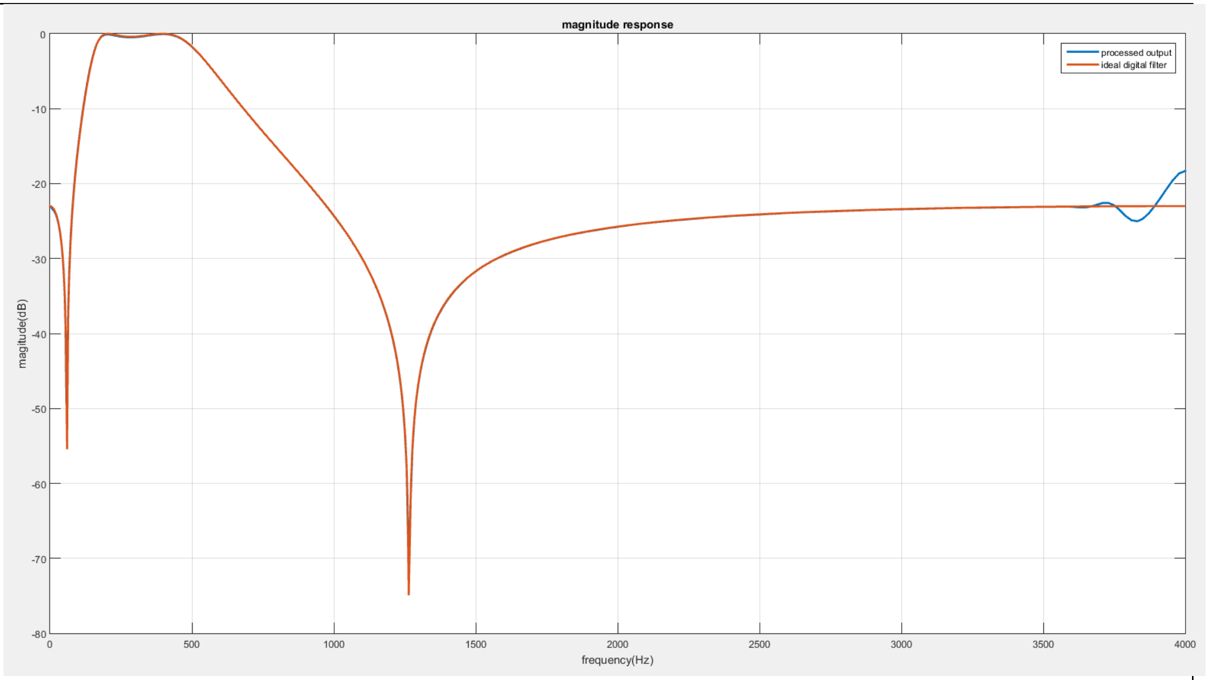


Figure 16: magnitude plot

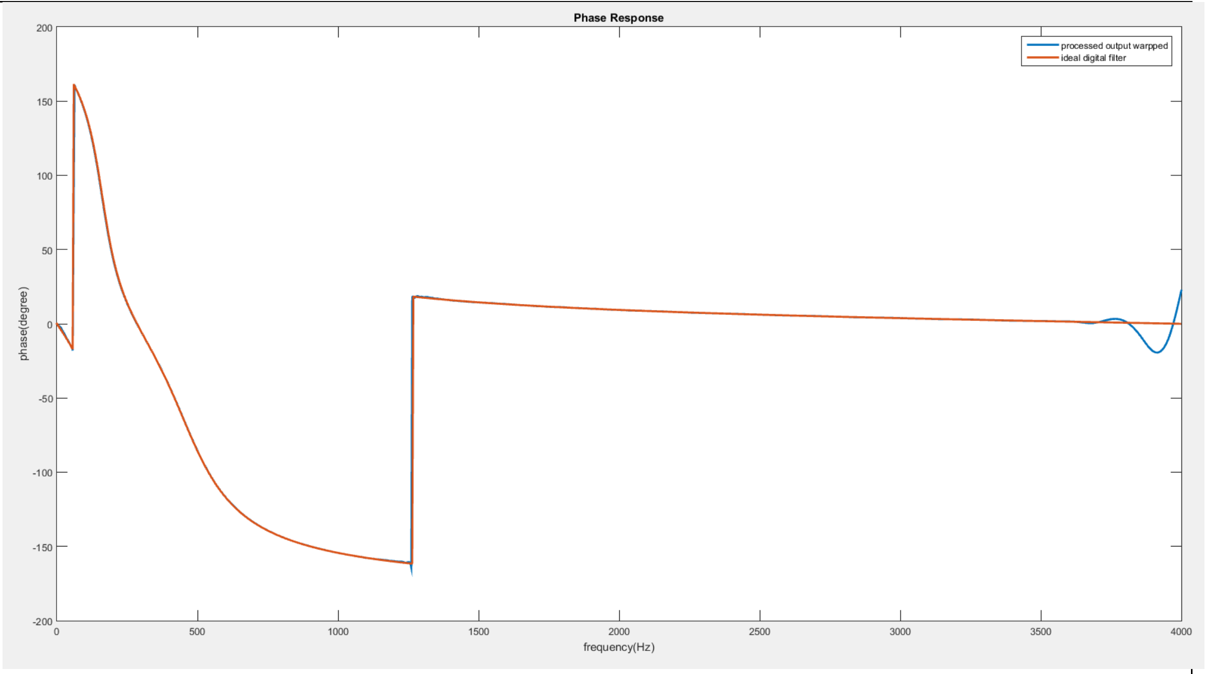


Figure 17: phase plot, wrapped into 180 degrees.

We obtained the same behaviour as direct form II frequency response (figure 14 & 15) in section 2. Therefore, we concluded that the code for direct form II transposed is correct.

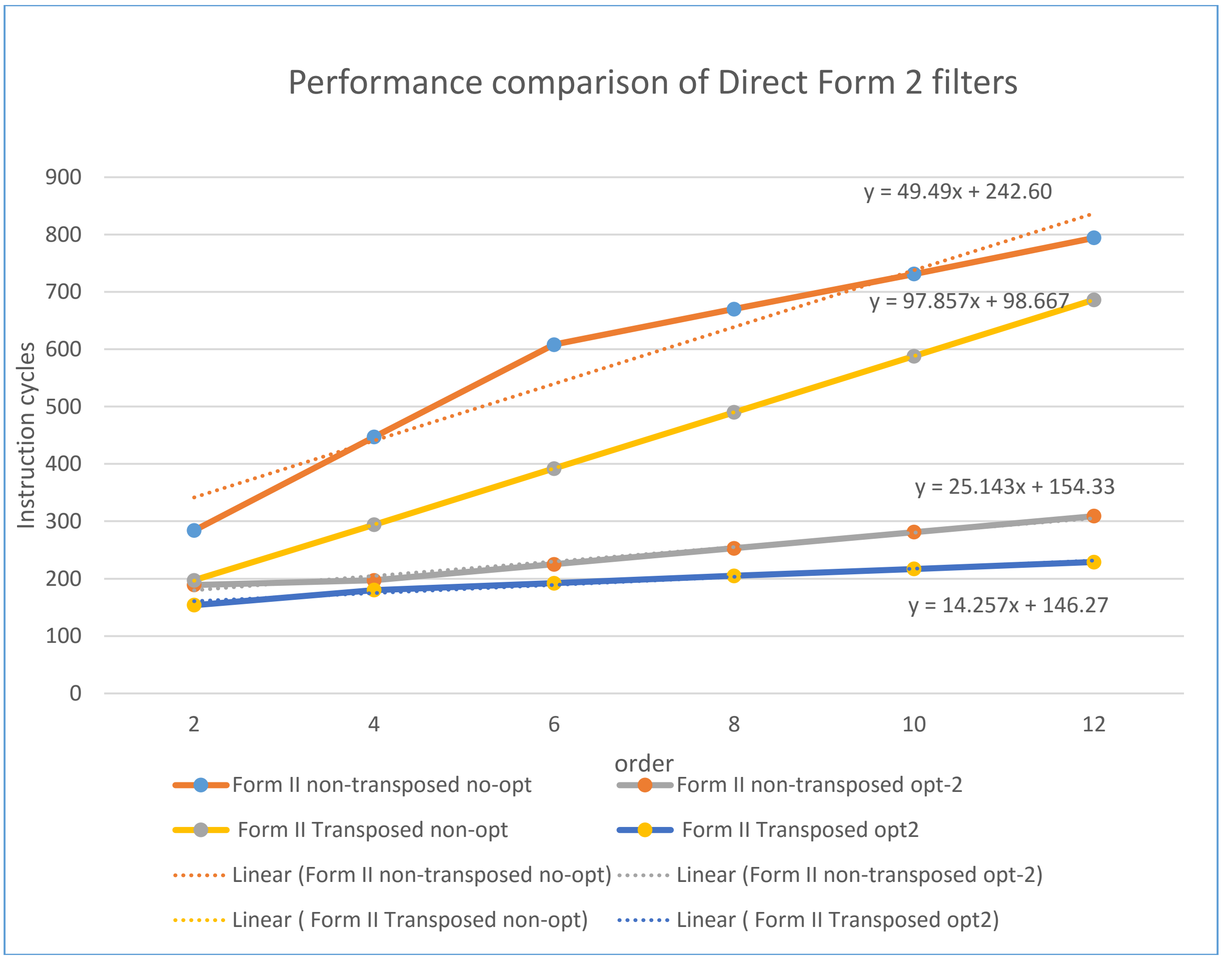
4. Performance Comparison (transposed and non-transposed)

Having implemented direct form II transposed and non-transposed version in the sections above, we were then interested in measuring the speed of our codes at various filter orders and different compiler optimisation levels (non-opt and opt-2).

The resulting measurements are shown in the table below.

order	Form II non-transposed no-opt	Form II non-transposed opt-2	Form II Transposed non-opt	Form II Transposed opt2
2	284	189	197	154
4	447	197	294	180
6	608	225	392	192
8	670	253	490	205
10	731	281	588	217
12	794	309	686	229

In order to better analyse the table, we plotted a graph (shown below). A linear relation could be discovered for each set of the data, in a form of **A+Bn**. Where **n** means order number. We calculated the relation equations with using function *Trendline*, hence illustrated in the graph below.



It shows that the transposed function works much faster than the non-transposed one at the same compiler optimisation level. This divergence further deviates as order number increases.

Filter Type and Opt Level	**Instruction Cycles per sampling (filter order *n*)**
Direct Form 2 - no opt	y = 49.49x + 242.60
Direct Form 2 - opt2	y = 25.143x + 154.33
Direct Form 2 transposed - no opt	y = 97.857x + 98.667
Direct Form 2 transposed - opt2	y = 14.257x + 146.27

The difference in speed performance between transposed and non-transposed implementations is mainly because the non-transposed code contains a ***for*** loop which has three separate instructions whereas the transposed version has only one. In addition, for the non-transposed implementation, the buffer step[ ] must be shifted by one every time in the loops, which adds more inefficient instructions.

5. Appendix

Full version of Matlab code for section 2.1:

```
%This creates a digital IIR filter using ellip, and stores the coefficients
%into a text file.
%design specification
order = 4; %order: 4th
fp = [180/4000 450/4000]; %passband: 180-450 Hz
rp = 0.4; %passband ripple: 0.4dB
stop_atte = 23; %Stopband attenuation: 23dB
%ellip(order/2, ripple(dB), stopband attenuation (dB), Passband frequencies)
[b, a]=ellip(order/2, rp, stop_atte, fp);%get



%write coefficients into a text file
file = fopen('coef.txt','w');
fprintf(file,'#define order %d\n',order); %define order number
fclose(file);
%coefficient a
file = fopen('coef.txt','at');
fprintf(file,'double a[]={');
dlmwrite ('coef.txt',a,'-append');
fprintf(file,'};\n');
%coefficient b
fprintf(file,'double b[]={');
dlmwrite ('coef.txt',b,'-append');
fprintf(file,'};');
fclose(file);
copyfile('coef.txt', 'H:\RTDSPlab\lab5\RTDSP');%copy file into RTDSP project

%plot ideal filter
figure(1);
freqz(b,a,1000,8000)%1000 is the number of points which must match the no. of data of audio analyser
                    %8000 is the sampling frequency

%locations of zeros and poles in z-plane
figure(2);
zplane(b,a);
```

Full version of C code:

```
/*************************************************************************************
              DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                         IMPERIAL COLLEGE LONDON

                 EE 3.19: Real Time Digital Signal Processing
                    Dr Paul Mitcheson and Daniel Harvey

                         LAB 3: Interrupt I/O

                     ********* I N T I O. C **********

  Demonstrates inputing and outputing data from the DSK's audio port using interrupts.

 ************************************************************************************
            Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
            Updated for CCS V4 Sept 10
 ************************************************************************************/
/*
 *  You should modify the code so that interrupts are used to service the
 *  audio port.
 */
/**************************** Pre-processor statements ******************************/
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"
/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"
// math library (trig functions)
#include <math.h>
// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>


/******************************* Global declarations ********************************/
/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
          /**********************************************************************/
          /*   REGISTER            FUNCTION               SETTINGS         */
          /**********************************************************************/\
   0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
   0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
   0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
   0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
   0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
   0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off     */\
   0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on     */\
   0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit              */\
   0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
   0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
          /**********************************************************************/
};
// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


 /******************************* Function prototypes ********************************/
```

```c
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
double direct_implementation(void);
double directTwo_implementation(void);
double directTwo_Transposed_implementation(void);


/******************************* Global Variables ***********************************/
/* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
32000, 44100 (CD standard), 48000 or 96000  */
int sampling_freq = 8000;
/* Use this variable in your code to set the frequency of your sine wave
  be carefull that you do not set it above the current nyquist frequency! */
//float sine_freq = 1000.0;
// set the sample as global variable such that its value can be observed easily in the watch table
Int16 samp;
int i;
double yout = 0.0;
double input = 0.0;
int count=0;

/*****************************lab5 direct_implementation******************************/
/*
double a[] = {1, -0.8823529411764706};
double b[] = {0.0588235294117647, 0.0588235294117647};
int order = 1; //find the order of the filter
double x[]= {0, 0};
double y[]= {0, 0};
*/

/***********************lab5 directTwo transposed and non-transposed********************/
#include "coef.txt"//read the text file generated by Matlab
               //which contains the coefficient of the FIR filter in an array b[] and
               //defines the order number by Matlab using #define order
double step[order+1];



/********************************** Main routine ************************************/
void main(){
 // initialize board and the audio port
 init_hardware();
 /* initialize hardware interrupts */
 init_HWI();
 /* loop indefinitely, waiting for interrupts */
 while(1)
 {};
}

/********************************** init_hardware() *********************************/
void init_hardware()
{
   // Initialize the board support library, must be called first
   //it resets all the hardware to the default settings
   DSK6713_init();

   // Start the AIC23 codec using the settings defined above in config
   //set the sampling frequency to 8K and the bit resolution to 16 bits
   H_Codec = DSK6713_AIC23_openCodec(0, &Config);

   /* Function below sets the number of bits in word used by MSBSP (serial port) for
   receives from AIC23 (audio port). We are using a 32 bit packet containing two
   16 bit numbers hence 32BIT is set for receive */
   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

   /* Configures interrupt to activate on each consecutive available 32 bits
   from Audio port hence an interrupt is generated for each L & R sample pair */
   MCBSP_FSETS(SPCR1, RINTM, FRM);

   /* These commands do the same thing as above but applied to data transfers to
   the audio port */
   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
   MCBSP_FSETS(SPCR1, XINTM, FRM);
}

/*********************************** init_HWI() **************************************/
void init_HWI(void)
{
   IRQ_globalDisable();          // Globally disables interrupts
   IRQ_nmiEnable();              // Enables the NMI interrupt (used by the debugger)
   IRQ_map(IRQ_EVT_RINT1,4);     // Maps an event to a physical interrupt
   IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
   IRQ_globalEnable();           // Globally enables interrupts
}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void ISR_AIC(void){
   //mono_write_16Bit(mono_read_16Bit());//test bare board response
                              //this will be measured by audio analyser and be subtracted
   //mono_write_16Bit((Int16) (direct_implementation()));//write sample back to memory
                                    //casting double into 16 bit int data type.
   mono_write_16Bit((Int16) (directTwo_implementation()));//write sample back to memory
                                    //casting double into 16 bit int data type.
   //mono_write_16Bit((Int16) (directTwo_Transposed_implementation()));//write sample back to memory
                                    //casting double into 16 bit int data type.
}

/******************** LAB 5 direct_implementation***********************************/
double direct_implementation(void){
 //this function works for any order number of IIR filter
 //this function implements direct form
 //it uses non-circular buffer, thus there is a potential enhancement
   samp = mono_read_16Bit(); //read from ADC and store the input into a 16bit int variable
   yout = b[0]*samp;//first coefficient multiplication
   for(i=order;i>0;i--){
      yout += x[i]*b[i]-y[i]*a[i]; //coefficients multiplication arithmetic
      y[i]=y[i-1]; //delay by shifting y[] buffer by one element
      x[i]=x[i-1]; //delay by shifting y[] buffer by one element
   }
   x[1] = samp; //x[1] is the buffer which stores current input
   y[1] = yout; //y[1] is the buffer which stores current output
   return yout;
}


/******************** LAB 5 directTwo_implementation**********************************/
double directTwo_implementation(void){
 //this function works for any order number of IIR filter
 //It implements direct form II non-transposed
 //it uses non-circular buffer, thus there is a potential enhancement, which will be seen in the next implementation
   step[0] = mono_read_16Bit();//read from ADC and store the input into a 16bit int variable
   yout = 0;//reset output before processing
   for(i=order; i>0; i--){
      step[0]-=a[i]*step[i];//multiplication accumulation of -a[] coefficients
      yout+=b[i]*step[i];//multiplication accumulation of b[] coefficients
```

```
        step[i]=step[i-1];//performs delay by shifting the buffer by one element
    }
    yout +=b[0]*step[0];//write the last multiply accumulation to the output
    return yout;
}


/******************** LAB 5 directTwo_Transposed_implementation*************************/
double directTwo_Transposed_implementation(void){
 //this function works for any order number of IIR filter
 //It implements direct form II transposed
 //it uses ncircular buffer, which significantly enhanced the code in terms of instruction cycles
    input = (double) (mono_read_16Bit()); //read from ADC, cast and store the input into a double variable
    yout = step[0] + b[0]*input;//write the output
    for(i=0; i<order-1; i++){
        step[i] = step[i+1] - a[i+1]*yout + b[i+1]*input;//sum of previous buffer and coefficients multiplications
    }
    step[order-1] = -a[order]*yout + b[order]*input;//writes the last buffer
    return yout;
}
```