EE3-19 Real Time Digital Signal Processing

Coursework Report 4: Lab 5

Jeremy Chan*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Table of Contents

I. Introduction 4
II. Supporting Code 4
A. Pre-processor statements 4
B. Global variables/declarations 4
C. Function prototypes 4
D. Initialisation functions 4
E. Main testbench 5
F. Interrupt Service Routine 5
III. Single Pole Filter 5
A. Design 5
B. Implementation and Code Operation 7
C. Frequency Response 7
1) MATLAB 7
2) APX Audio Analyser 9
3) Comparison between responses 11
IV. Direct Form II Filter 14
A. Design 14
B. Implementation and Code Operation 17
C. Frequency Response 18
1) MATLAB 18
2) APX Audio Analyser 20
3) Discussion on Phase Response 23
V. Direct Form II Transposed Filter 24
A. Design 24
B. Implementaiton and Code Operation 24
C. Frequency Response 25
1) MATLAB 25
2) APX 25
3) Discussion on Phase Response 28
VI. Performance Comparison between Implementations 29
VII. Conclusion 31
VIII. References 31
IX. Appendix 32
A. APX Input 32

B. MATLAB Scripts 32
1) Single Pole Filter 32
2) Direct Form II 32
3) Generated coefficients 32
C. Source Code (custom functions only) 33

I. Introduction

Modifying lab 4's FIR filter code, an IIR filter was realized on hardware given generated filter coefficients using MATLAB. This report details the process of putting such a filter on hardware, and includes discussions on results, benchmarking, and filter responses, measured using the APX audio analyser.

Multiple implementations were used to generate the IIR filter; design, implementations, and benchmarks are given for all the implementations.

II. Supporting Code

A. *Pre-processor statements*

Libraries containing helper functions such as `helper_functions_ISR.h` for read/write when using interrupts are defined at the top of the code with `#include <file>`. Other global constants are also defined in this section, e.g. N, with `#define <text> <value>`. The pre-processor will look at these statements and replace the statements with their contents into the source file to produce an expanded source code file, before handing it over to the compiler for compilation.

```
#define b0 0.0588235941
#define a1 -0.8823529
```

Other pre-processor variables defined include the IIR single pole filter coefficients, generated in section IV.A.

B. *Global variables/declarations*

Instantiations of structs defined in library files can be found below the pre-processor statements – their aim is to define a set of configurations to be called in later functions, whether this is to initialize hardware or specify new data structures. For example, the codec config is defined here, and is called in `init_hardware()`, as the configuration used by the codec `(H_Codec = DSK6713_AIC23_openCodec(0, &Config);)`.

C. *Function prototypes*

```
void init_hardware(void);
void init_HWI(void);
void init_buffer(void);
void interrupt_service_routine(void);
double Single_pole(double sample);
double Dir_form_2(double sample);
double Dir_form_2_transposed(double sample);
```

Function prototypes are declared before `int main()` for the compiler to be able to find the IIR functions as our functions are defined after the interrupt function which calls them.

D. *Initialisation functions*

`void init_hardware()` configures the hardware using defined global variables. The struct of the config for configuring the handle can be found in the included header files; the instantiation in the global declarations section. This function is duly executed as the first line of main before any testbench code is executed.

`void init_HWI()` configures hardware interrupts; it disables all interrupts, assigns events to interrupts, then enables all interrupts again.

`void init_buffer()` initializes the global buffer to 0.

E. *Main testbench*

As this code functions on interrupts, there is no testbench code to execute apart from the initialization of hardware. After initialization, the program enters a `while(1)` infinite loop, doing nothing, waiting for interrupts.

When a hardware interrupt is encountered, the program runs a pre-defined function based on the function defined in `dsp_bios_.tcf`. In our case, it `is void interrupt_service_routine(void);`

F. *Interrupt Service Routine*

The function that runs when a specific hardware interrupt is encountered is defined in the `dsp_bios_.tcf` file (Figure 1).

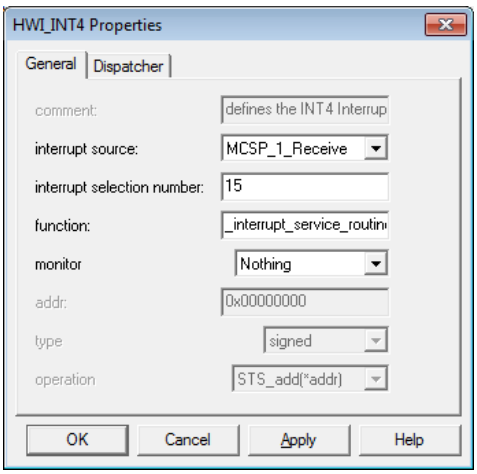


Figure 1: Configuration to launch function during interrupt

```
/**
 * This function gets executed upon a hardware interrupt. It reads in a sample,
 * then outputs the output of one the IIR functions.
 */
void interrupt_service_routine(void) {
	double sum = 0.0;                    //for IIR filter sum
	double sample = 0.0;                 //for incoming sample
	sample = (double)mono_read_16Bit();  //read incoming sample on receive interrupt
	//sum = Single_pole(sample);
	//sum = Dir_form_2(sample);
	sum = Dir_form_2_transposed(sample);
	mono_write_16Bit((Int16)(sum));      //output sum to both channels
}
```

The interrupt function merely reads in a sample and outputs the output of one of the IIR functions to both channels.

III. SINGLE POLE FILTER

A. *Design*

This design derives filter coefficients using the Tustin transform.

The transfer function of an M-order IIR filter is given by $H(z) = \frac{b_0+b_1Z^{-1}+b_2Z^{-2}+\cdots+b_MZ^{-M}}{1+a_1Z^{-1}+a_2Z^{-2}+\cdots+a_NZ^{-N}}$.

Therefore, the corresponding time domain difference equation is $y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + \cdots + b_Mx(n-M) - a_1y(n-1) - a_2y(n-2) - \cdots - a_Ny(n-N)$.

The bilinear transform allows a mapping between the analog (s) plane and the digital (z) plane. Hence, it is possible to map an analogue filter to its discrete equivalent (i.e. as an IIR filter). Figure 2 shows the analogue filter.

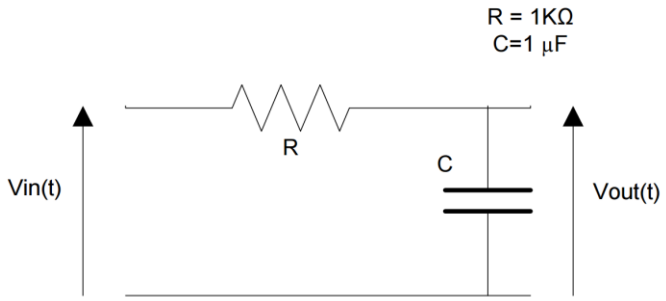


Figure 2: Given diagram of analogue filter [1]

$$T_s = RC = 10^{-3}s$$
$$Corner\ frequency = f_c = \frac{1}{2\pi RC} = 159.15Hz$$

A good approximation can be realised without applying frequency warping as the cut-off frequency in Figure 2 is low in comparison to the sampling frequency of 8000Hz [1].

The transfer function of Figure 2 is $\frac{V_{out}}{V_{in}} = H(jw) = \frac{\frac{1}{j\omega C}}{R+\frac{1}{j\omega C}} = \frac{1}{1+j\omega RC}$. In the Laplace domain, $H(s) = \frac{\frac{1}{CS}}{R+\frac{1}{CS}} = \frac{1}{1+RCS}$.

Using the Tustin transform, $s = \frac{z-1}{z+1} \cdot \frac{2}{T_s}$ [1], $where\ T_s\ is\ the\ sampling\ frequency$,

$$H(z) = \frac{1}{1 + RC\left(\frac{z-1}{z+1}\right) \cdot \frac{2}{T_S}} = \frac{T_s(z+1)}{T_s(z-1) + 2(RCz - RC)}$$
$$= \frac{T_s z + T_s}{(T_s + 2RC)z + T_s - 2RC}$$
$$= \frac{\frac{T_s z + T_s}{T + 2RC} \cdot z^{-1}}{1 + \frac{T_s - 2RC}{T + 2RC} \cdot z^{-1}}$$
$$= \frac{\frac{T_s(z+1)}{T_s - 2RC}}{1 + \frac{T_s + 2RC}{T_s - 2RC} \cdot z}$$
$$= \frac{\frac{T}{T + 2RC} z^{-1} + \frac{T}{T + 2RC}}{1 + \frac{T - 2RC}{T + 2RC} z^{-1}}$$
$$\because RC = 10^{-3}, T = 1.25 \times 10^{-4}$$
$$= \frac{0.0588 + 0.0588z^{-1}}{1 - 0.8823529z^{-1}}$$

The final coefficients are shown in Table I.

TABLE I.

Coefficient	Value
b_0	$\frac{1}{17} = 0.0588235941$
b_1	$\frac{1}{17} = 0.0588235941$
a_0	1
a_1	$-\frac{15}{17} = -0.88823529$

B. *Implementation and Code Operation*

```
double Single_pole(double sample) {
        static double out = 0.0;      //use static variable to store output value
        static double last_x = 0.0;   //use static variable to store last input value
        out = b0*sample + b0*last_x - a1*out; //difference equation
        last_x = sample;              //put current sample into last sample variable
        return out;
}
```

Since the difference equation of an IIR filter is, as previously defined, $y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + \cdots + b_Mx(n-M) - a_1y(n-1) - a_2y(n-2) - \cdots - a_Ny(n-N)$, this is easily computed in the function by multiplying the global constants of $b_0, b_1, and\ a_1$ by their respective sample values. Since $b_0 = b_1$, b_0 is used twice.

The keyword `static` is used to allow this function to use older samples and output values for future iterations of the loop. Hence, `sample` is $x(n)$, `last_x` is $x(n-1)$, and `out` is $y(n-1)$.

A low frequency square wave was required to be the input of the filter. Since a square wave is a summation of sine waves (Fourier series), the high pass filter on the input of the board will significantly attenuate high frequency sine waves. Hence, a 200Hz square wave was chosen to be the input. This square wave can be generated using http://audiocheck.net's Square Tone Generator.

C. *Frequency Response*

1) *MATLAB*

```
RC=10^-3;                                   %as defined in analogue diagram
sys=zpk([],[-1/RC],1/RC);                   %zpk converts system into zero pole form
[num,den]=zp2tf([],[-1/RC],1/RC);           %create transfer function
[zd,pd] = bilinear(num,den,8000);           %tustin transform
sys2=filt(zd,pd,1/8000);                    %implement digital filter
bode_diag=bodeplot(sys2);                   %plot frequency response
setoptions(bode_diag,'FreqUnits','Hz');     %graph options
axis([1,3800,-90 ,0]);                      %graph options
```

Using the MATLAB script above, Figure 3-5 were generated using the afore-calculated coefficients, and functions `zp2tf()` and `filt()`. A bode plot was generated for both the analogue and digital systems (sys/sys2). Figure 3 is the analogue system; Figure 4 is the Tustin transformed digital system. They have approximately the same shape.

The -3dB frequency can be seen at 159Hz for the analogue implementation (Figure 3). The -3dB frequency can be seen at approximately 158Hz for the digital implementation (Figure 4,5) (the previous data point was at -2.45dB, the 1Hz difference can be attributed to the gradient of the line joining the two data points). Figure 6 shows the pole-zero plot after the Tustin transform.

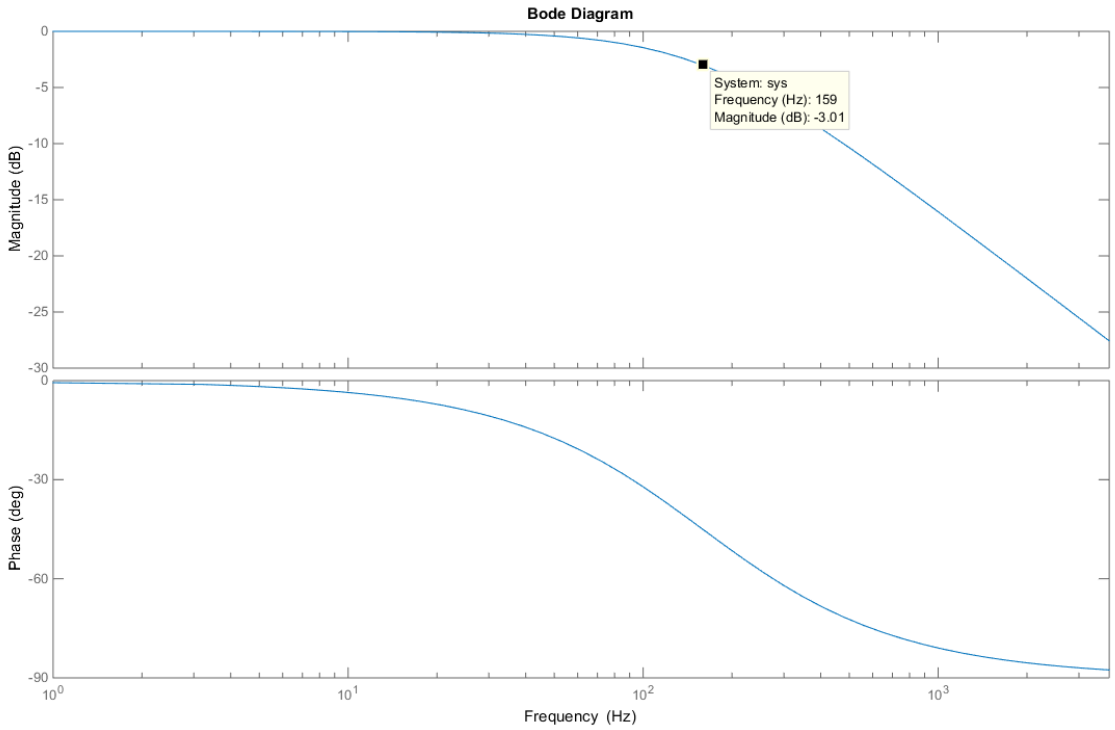


Figure 3: MATLAB Analogue Filter Bode Plot

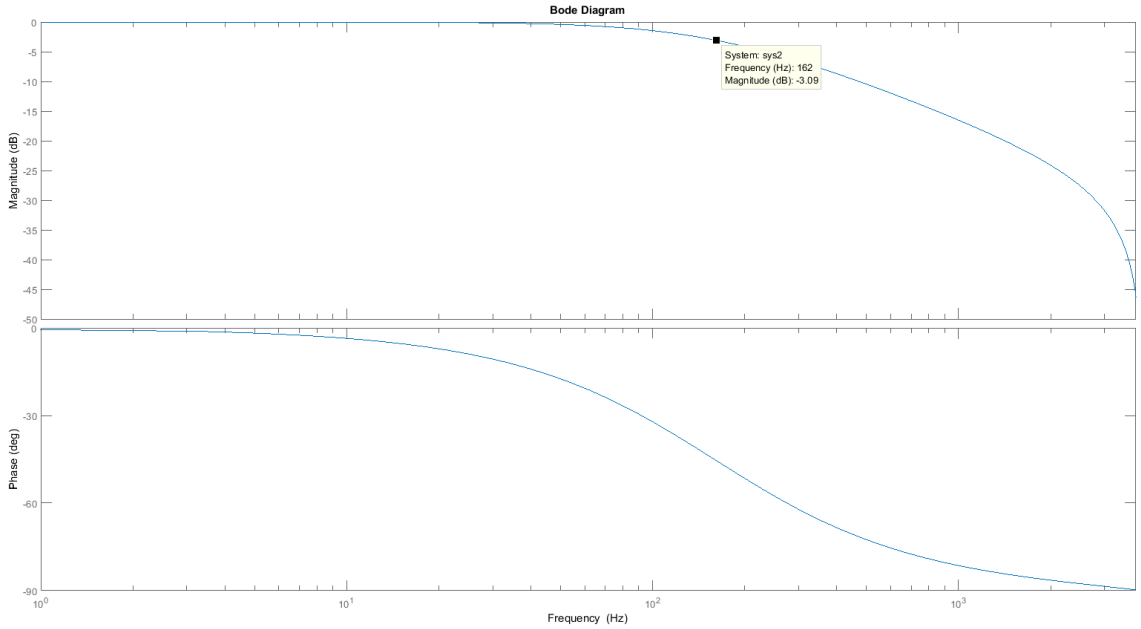


Figure 4: MATLAB Digital (after Tustin transform) Filter Bode Plot

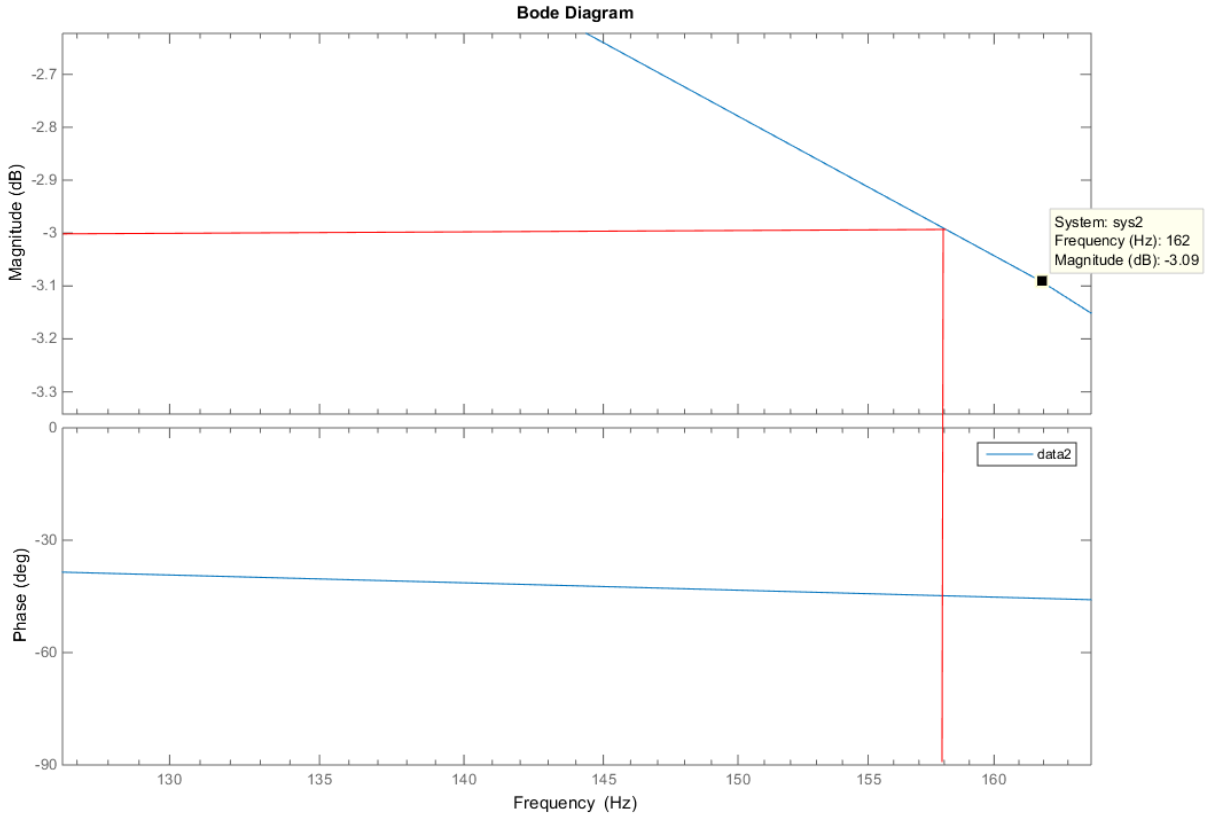


Figure 5: Zoomed in digital bode plot, showing expected corner freqency

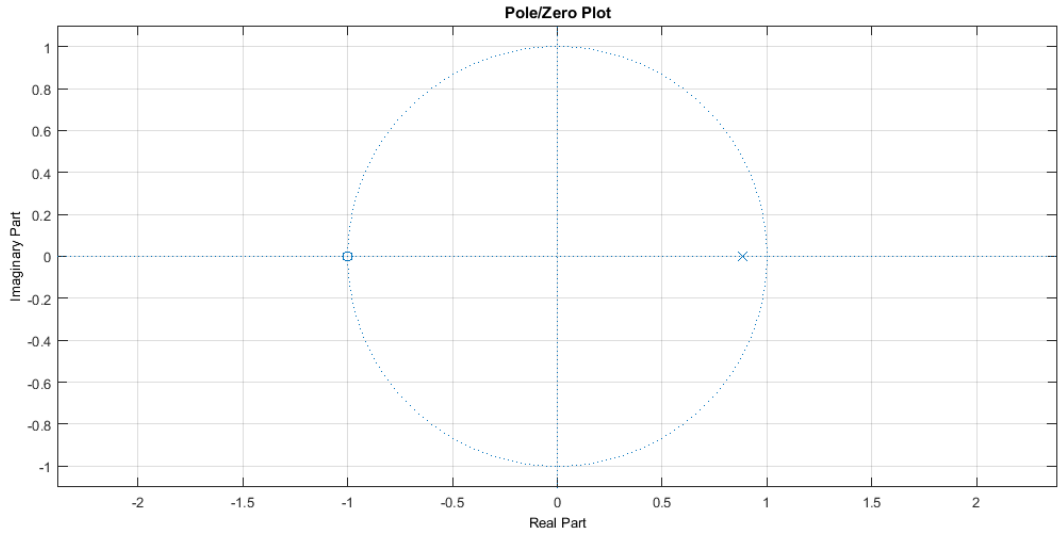


Figure 6: Pole Zero plot of digital filter

2) *APX Audio Analyser*

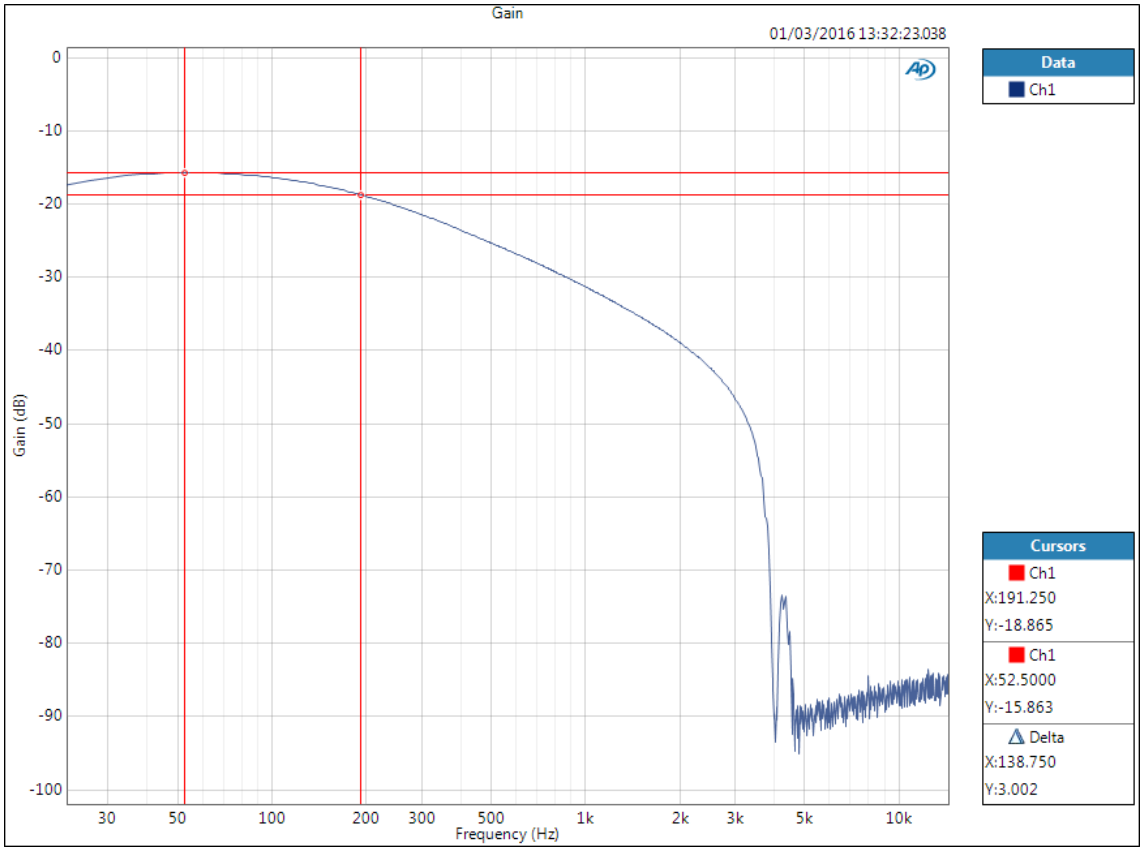


Figure 7: Magnitude Response of filter on DSK (APX)

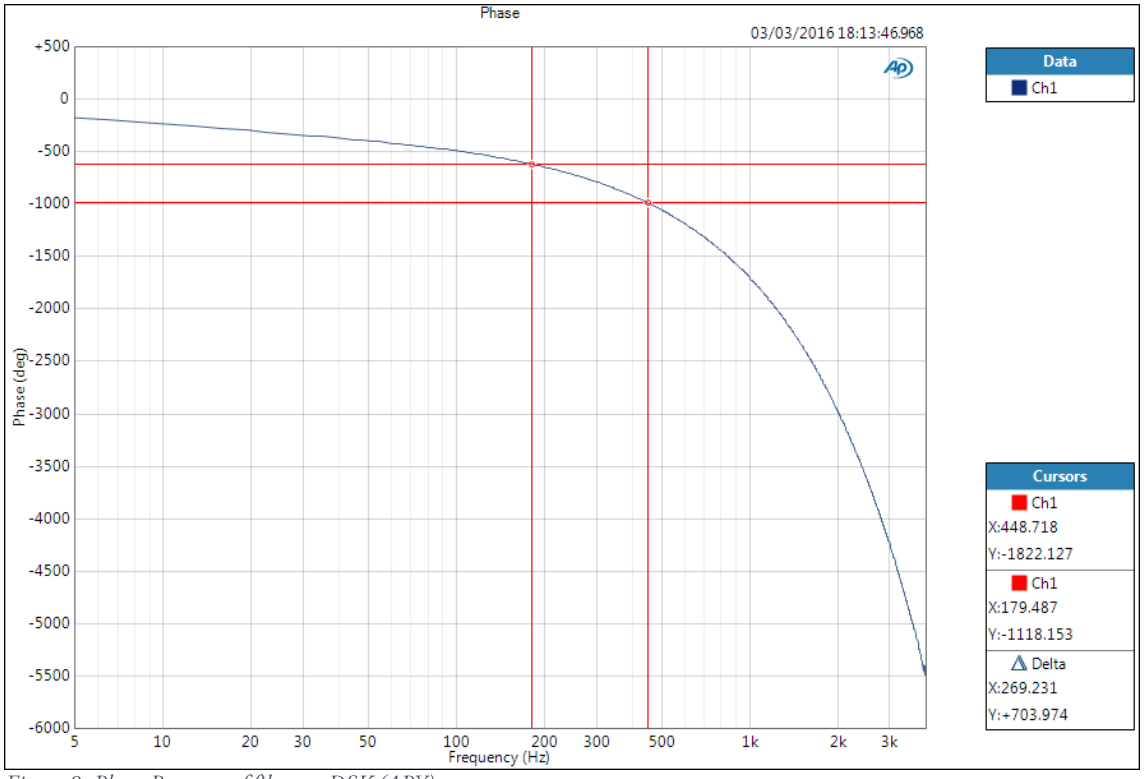


Figure 8: Phase Response of filter on DSK (APX)

Figures 7 and 8 show the frequency response of our implementation when using the 200Hz square wave input. Points to note are the significant attenuation at high frequencies – this can be attributed to the low

pass filter in the DAC to block components with a frequency higher than the Nyquist frequency (4000Hz). Additionally, the corner frequency (-3dB) was measured to be 191Hz.

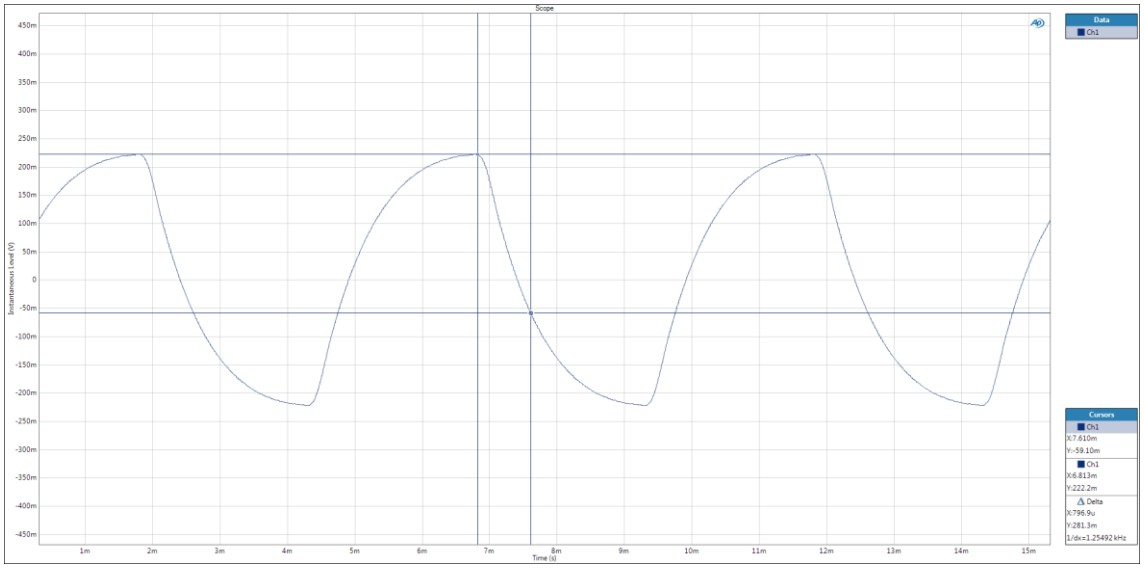


Figure 9: Time Constant measurment on APX

Figure 9 shows the time constant (time it takes to decay from the asymptotic value to $\frac{1}{e}$ (36.8%) of the asymptotic value) of our implementation to be $797\mu s$.

3) *Comparison between responses*

TABLE II.

	Analogue	Digital	Difference
Time Constant	$1ms$	$797\mu s$	20.3%
Corner Frequency	159.15Hz	191Hz	20.0%

Results are shown in Table II - there are non-negligible differences in the both digital measurements.

Using the equation for the single pole filter, $H(z) = \frac{\frac{1}{17}+\frac{1}{17}z^{-1}}{1-\frac{15}{17}z^{-1}}$, the expected digital corner frequency can be derived.

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

$$= \frac{z + 1}{17z - 15}$$

$$= \left| \frac{e^{jw} + 1}{17e^{jw} + 15} \right|$$

$$= \frac{|\cos w + 1 + j \sin w|}{|17 \cos w - 15 + 17j \sin w|}$$

$$= \sqrt{\frac{1 + 2\cos w + \cos^2 w + \sin^2 w}{(17 \cos w + 15)^2 + (17 \sin w)^2}}$$

$$= \sqrt{\frac{2(1 + \cos w)}{289 \cos^2 w - 510 \cos w + 255 + 289 \sin^2 w}}$$

$$= \sqrt{\frac{2(1+\cos w)}{289+255-510\cos w}}$$
$$= \sqrt{\frac{1+\cos w}{257-255\cos w}}$$

$$\because -3dB = \frac{1}{\sqrt{2}}$$
$$\frac{1}{\sqrt{2}} = \sqrt{\frac{1+\cos w}{257-255\cos w}}$$
$$257 - 255\cos w = 2 + 2\cos w$$
$$0.125 = w$$
$$\therefore f_{corner} = f_{norm} \times f_{samp} = \frac{\omega}{2\pi} \times 8000 = 159.15Hz$$

The leftmost cursor on Figure 10 correspond to the peak in the original measured response of our single pole filter (Figure 7); the roll off for frequencies lower than this point led to the suspicion that there was a high pass filter somewhere in the measurement flow.

Upon further investigation, our implementation is actually 4 cascaded filters (a high pass filter, followed by low pass filter in the DAC, followed by our low pass single pole IIR filter, followed by another high pass filter). By measuring the response of the high pass filters on the DSK board cascaded with DAC filter (Figure 9), the true response of our single pole filter can be found by subtracting the response in Figure 10 from the response of the single pole filter. Data points from the APX analyser were exported into Excel, then plotted (frequencies much higher than the corner frequency were not plotted).

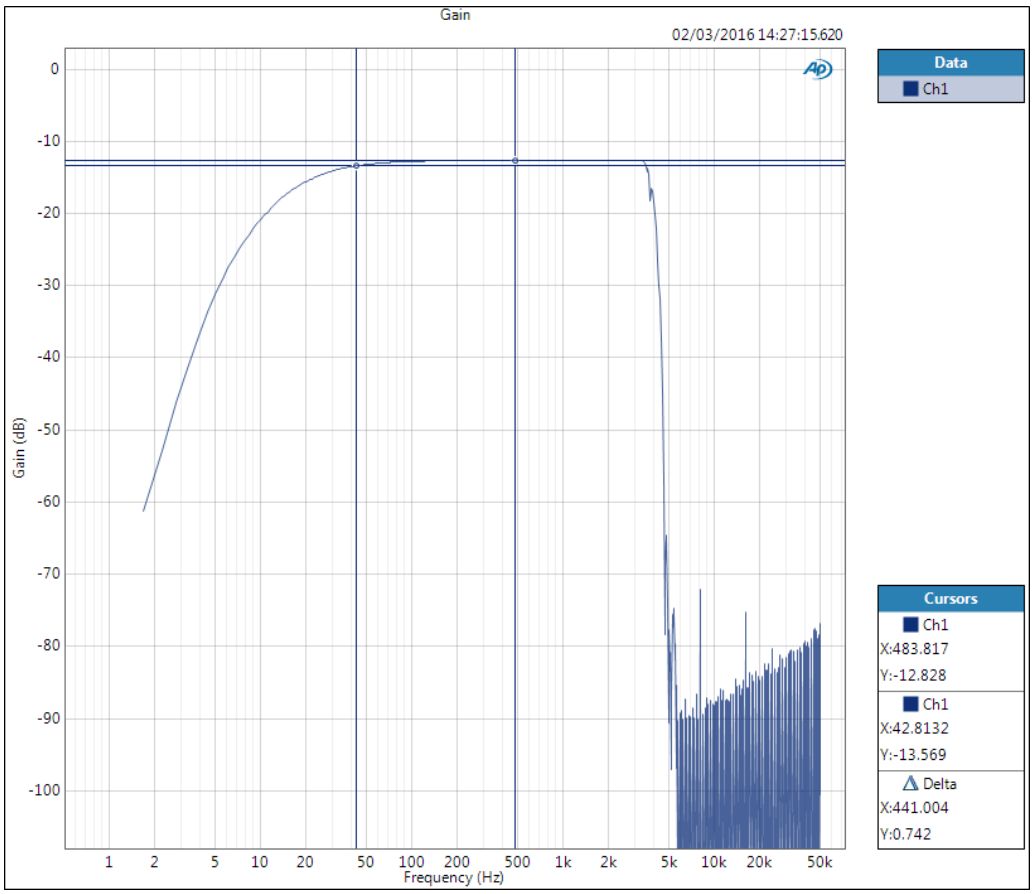


Figure 10: Magnitude Response of DSK high pass filter

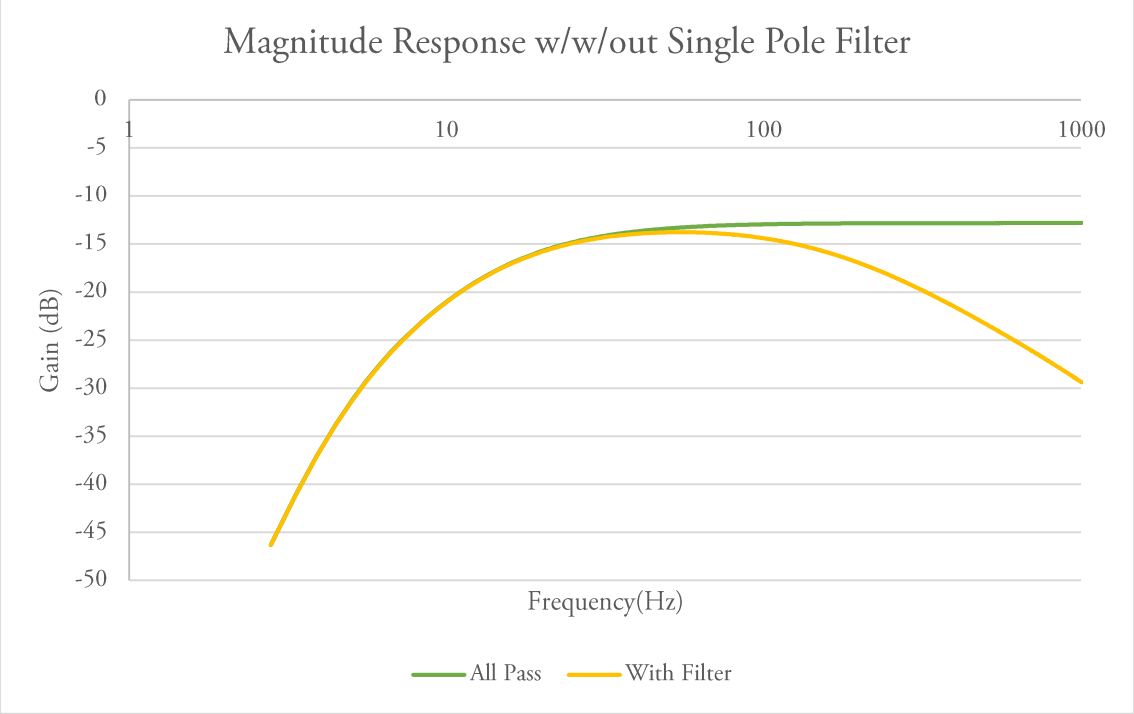


Figure 11: Comparison between magnitude responses

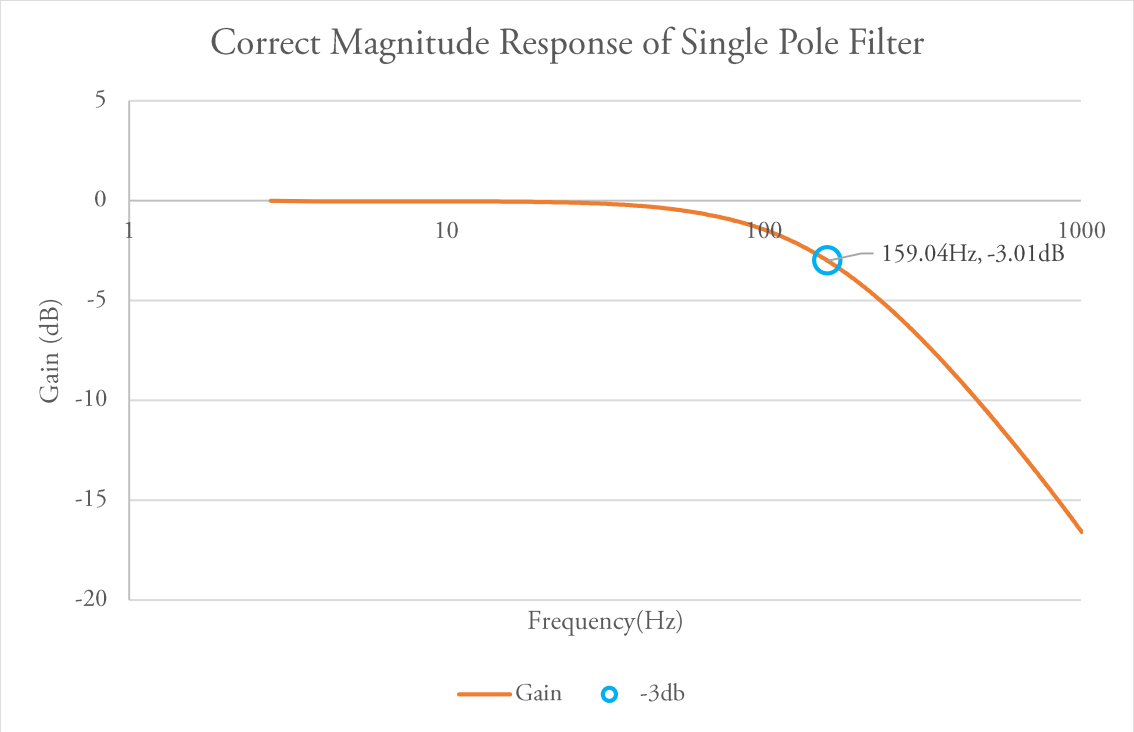


Figure 12: APX magnitude response minus DSK high pass filter magnitude response

Figure 11 shows the difference in magnitude response between an all pass filter and our implementation (with the board high pass filter). Figure 12 shows the corrected magnitude response (all pass response minus filter response). The shape corresponds to the shape plotted by MATLAB in Figure 4 - hence, the true -3dB frequency (finite measurement steps in the APX meant there was no -3.00dB data point) is actually 159Hz, which explains the difference in corner frequency.

Likewise, measurement of the time constant using Figure 9 was done on the aforementioned cascaded filters. The true time constant can be found by $t_c = \frac{1}{2\pi} \times \frac{1}{f_c}$ using the corrected corner frequency. Hence, the digital filter time constant is $\frac{1}{2\pi} \times \frac{1}{159} = 1001\mu s$, matching the analogue time constant (0.1% error).

IV. Direct Form II Filter

A. *Design*

An elliptic bandpass filter is required with specifications as specified in Table III. This filter can be designed using MATLAB to generate the coefficients for the filter using `ellip()`.

The elliptic filter places poles in an ellipse, hence its name, to produce the sharpest cut off of all commonly used filter types (Butterworth, Elliptic, Chebyshev, Inverse Chebyshev). It is also characteristic in that it has ripple in both the pass and stop bands [2].

TABLE III.

	Value
Order	4^{th}
Passband	180-450Hz
Passband Ripple	0.4dB
Stopband Attenuation	23dB

```
%[b,a] = ellip(n,Rp,Rs,Wp) returns the transfer function coefficients of an nth-order
lowpass digital elliptic filter with normalized passband edge frequency Wp. The
resulting filter has Rp decibels of peak-to-peak passband ripple and Rs decibels of
stopband attenuation down from the peak passband value. [2]
n=4;                                        %order
Rp=0.4;                                     %pass band ripple
Rs=23;                                      %stop band attenuation
fs=8000;                                    %sampling frequency
corner_f=[180 450]/(fs/2);                  %normalized frequency
[b,a]=ellip(n,Rp,Rs,corner_f);              %create elliptical filter coefficients
freqz(b,a);                                 %plot filter
fid=fopen('../RTDSPlab/lab5/iir_coef.txt','w');     %write to text file
fprintf(fid,'double b[]={');
fprintf(fid,'%.10e,\t',b);
fprintf(fid,'};\n');
fprintf(fid,'double a[]={');
fprintf(fid,'%.10e,\t',a);
fprintf(fid,'};\n');
fprintf(fid,'#define N %d',length(a)-1);     %write order to avoid messing about with
calculating order in C
fclose(fid);
figure;
zplane(b, a);                               %zero pole plot
```

Assuming the sampling frequency is 8000Hz, the above MATLAB code was used to generate a file of coefficients for use in the Direct Form IIR function. A point to note is that the corner frequencies should be normalized to the Nyquist frequency [1], hence $(\frac{f_s}{2})$. The formula to convert from Hertz to normalised frequency is $\frac{f}{f_s} \times 2\pi \times \frac{1}{\pi} = \frac{f}{4000}$.

The generated coefficients for both an order of 4 and 10 are shown below – the dynamic range of the coefficients increase with increasing order, seen in Figures 12 and 13.

a

PLOTS VARIABLE VIEW

1x9 double

	1	2	3	4	5	6	7	8	9
1	1	-7.4978	24.8256	-47.4066	57.1014	-44.4245	21.8014	-6.1710	0.7715

b

PLOTS VARIABLE VIEW

1x9 double

	1	2	3	4	5	6	7	8	9
1	0.0668	-0.4996	1.6655	-3.2348	4.0041	-3.2348	1.6655	-0.4996	0.0668
2									

Figure 13: MATLAB coefficients for order 4

a

PLOTS VARIABLE VIEW

1x21 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	-19.0826	173.6056	-1.0012e+03	4.1045e+03	-1.2716e+04	3.0889e+04	-6.0243e+04	9.5805e+04	-1.2546e+05	1.3604e+05	-1.2234e+05	9.1090e+04	-5.5850e+04	2.7923e+04

b

PLOTS VARIABLE VIEW

1x21 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0.0663	-1.2656	11.5369	-66.7452	274.8503	-856.3291	2.0945e+03	-4.1184e+03	6.6115e+03	-8.7513e+03	9.6031e+03	-8.7513e+03	6.6115e+03	-4.1184e+03	2.0945e+03
2															

Figure 14: MATLAB coefficients for order 10

TABLE IV.

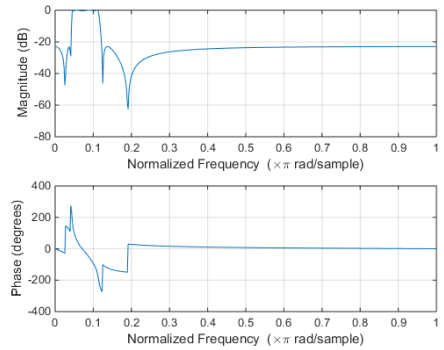
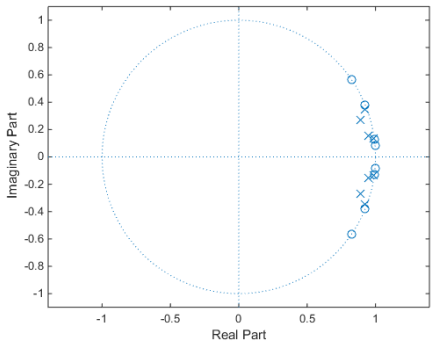
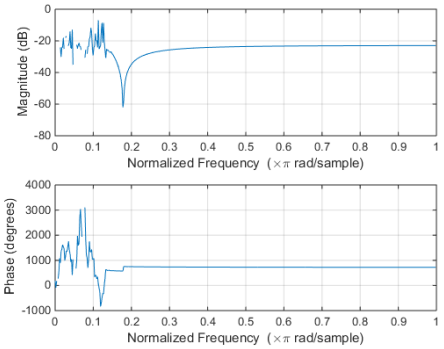
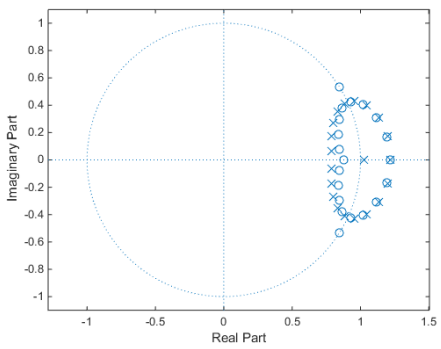
Order	Bode Plot	Pole Zero Plot
4	 *Figure 15: Bode Plot of 4th order elliptic filter*	 *Figure 16: Pole Zero plot of 4th order elliptic filter*
10	 *Figure 17: Bode Plot of 10th order elliptic filter*	 *Figure 18: Pole Zero Plot of 10th order elliptic filter*

Table IV shows the normalized frequency Bode plots as well as the respective pole zero plots for different orders as defined in `ellip()`.

The zeros appear to be directly on the unit circle for an order of 4, i.e. Figure 16, but zooming in (Figure 19), two of the zeros actually have a magnitude larger than 1 (Table V, data points from MATLAB's data point pointer), i.e. they are actually placed outside the unit circle due to limitations of precision using IEEE-754 `double`.

For an order of 10, the poles and zeros are placed very obviously outside the unit circle by MATLAB, leading to instability.

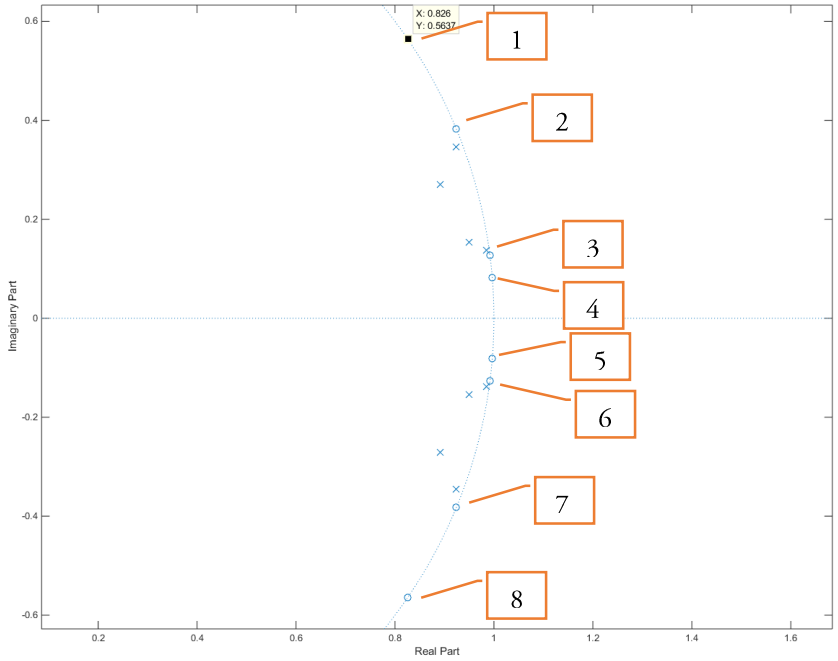


Figure 19: Zoomed in Pole Zero plot of 4th order elliptic filter

TABLE V.

Zero Data Point	X	Y	Magnitude
1	0.826	0.5637	1.000017
2	0.9242	0.3819	0.999997
3	0.9919	0.1269	0.999985
4	0.9967	0.08178	1.00004
5	0.9967	-0.08178	1.00004
6	0.9919	-0.1269	0.99998
7	0.9242	-0.3819	0.99999
8	0.826	-0.5637	1.00001

These rounding errors come from doing arithmetic on two floating point numbers which are a large order of magnitude apart [3].

Figure 20 shows the limitation of using floating point numbers, using a simple C++ test bench. The finite precision of using floating point numbers causes the shifting of the actual binary value during calculation to truncate the operand of 1.000001 to 1 in the case of multiplication. The poles and zeroes are increasingly sensitive to this round-off error in the coefficients with increasing order [4], as seen in the increasing dynamic range of the coefficients (Figure 13 and 14).

ideone.com

edit fork download

```c
#include <stdio.h>

int main(void) {
    float a = 100000000;
    float b = 1.00000001;
    double c = 100000000000000000;
    double d = 1.00000000000000001;
    printf("%f\n%f\n%f\n", (float)(a+b), (float)(a-b), (float)(a*b));
    printf("%f\n%f\n%f\n", c+d, c-d, c*d);
    return 0;
}
```

Success #stdin #stdout 0s 2156KB comments (0)

stdin

```
Standard input is empty
```

stdout

```
100000000.000000
100000000.000000
100000000.000000
100000000000000000.000000
100000000000000000.000000
100000000000000000.000000
```

Figure 20: Limitations of floating point numbers (http://ideone.com/4p6G4F)

B. *Implementation and Code Operation*

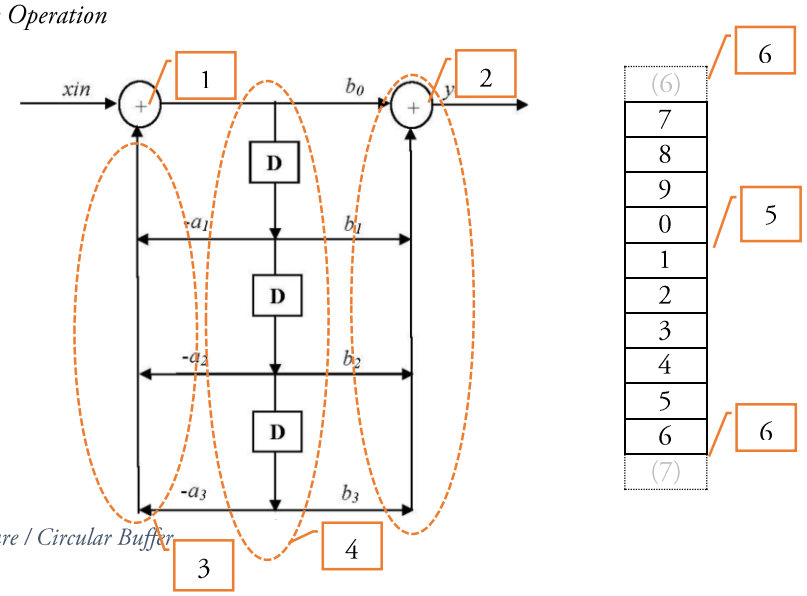


Figure 21: Direct Form II structure / Circular Buffer

The orange callouts link to corresponding callouts in the code operation section.

```c
/**
 * Elliptic IIR filter, direct form II implementation. Uses MATLAB generated
 * coefficients specified in include txt file to calculate filter. Circular
 * buffer used, initialized using calloc().
 * @param  sample sample_in, xin on diagram
 * @return        64bit output, yout on diagram
*/
```

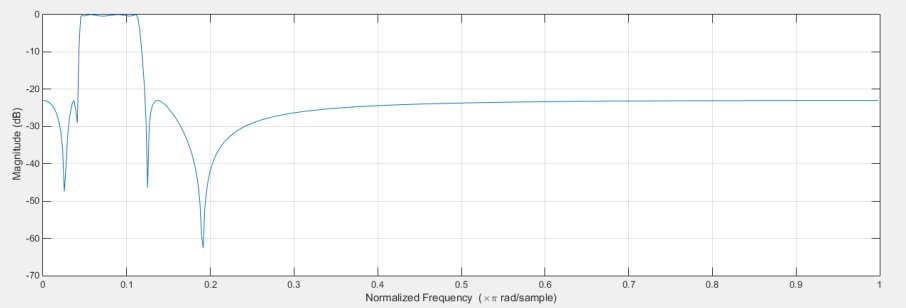
```
double Dir_form_2(double sample) {
        static int write_index = N;          //updatable write index for newest delay
line block
        double in_data = 0.0;                //for multiplying by a1, a2, a3...
        double out_data = 0.0;               //for multiplying by b1, b2, b3...
        double write_data = 0.0;             //summation of delayed coefficients,
multiplied by a1, a2, a3, with input value
        int read_index, loop_index;          //declaration of iterators
        for (loop_index = 1; loop_index <= N; loop_index++) {
            read_index = ((write_index + loop_index)>N) ? write_index + loop_index -
N - 1 : write_index + loop_index;            //circular buffer wraparound logic
            out_data += buffer[read_index] * b[loop_index];      //output acculumator
            in_data -= buffer[read_index] * a[loop_index];       //input acculumator
        }
        write_data = sample + in_data;       //summation of incoming sample and input
side of direct form
        buffer[write_index] = write_data;    //write to newest delay line block

        out_data += write_data*b[0];         //multiplication of summed write_data, aka
top line of direct form structure
        if (--write_index == -1)write_index = N; //handle circular buffer wraparound
        return out_data;
}
```

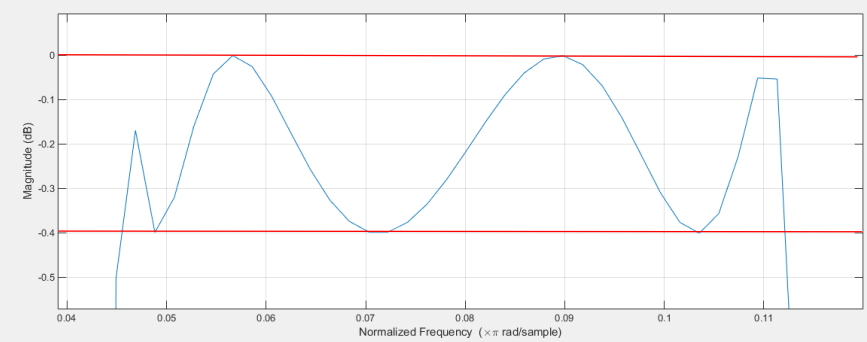
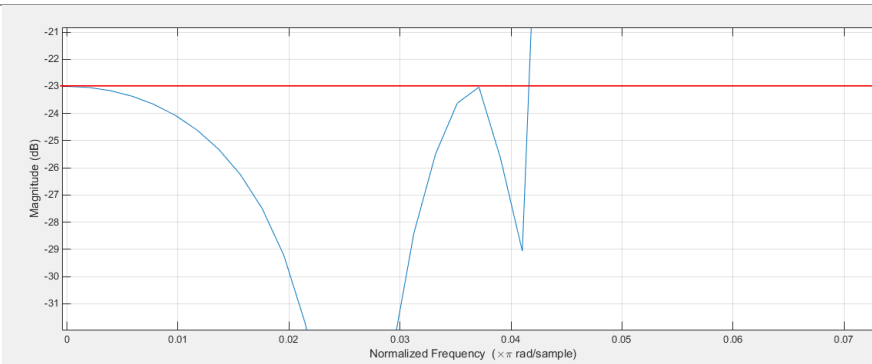
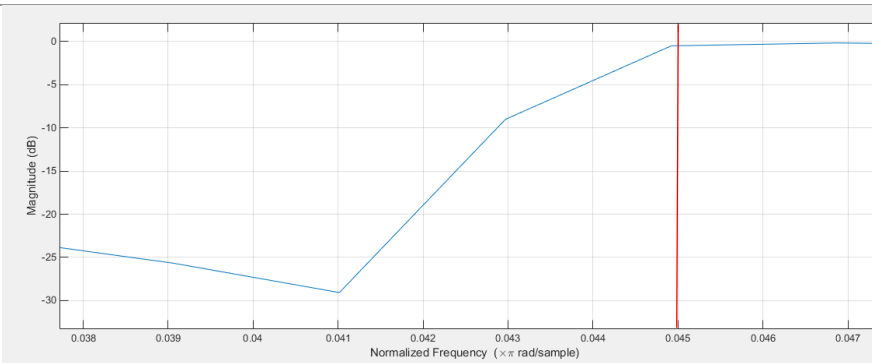
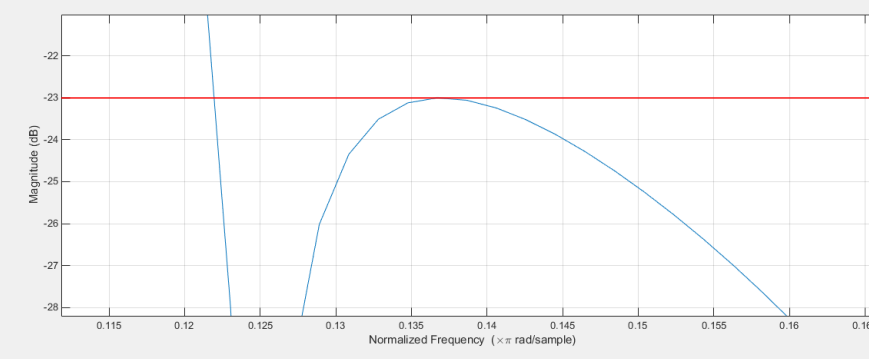
Figure 21 shows the direct form II structure, as well as a circular buffer. There are three parts to this direct form II structure: the 'a' side accumulator (callout 3), the delay line buffer (callout 4), and the 'b' side accumulator (callout 2). Samples are delayed using the delay line, a circular buffer (callout 5 and 6) in this case. The left and right side accumulator multiply the delayed samples by the corresponding coefficient in the arrays `a[]` and `b[]`, before summing them together with the current incoming sample (callout 1). The right side 'b' side accumulator also implements the summation between the sum of the 'a' side accumulator (accumulator in callout 2), multiplied by b_0, and the 'b' side accumulator.

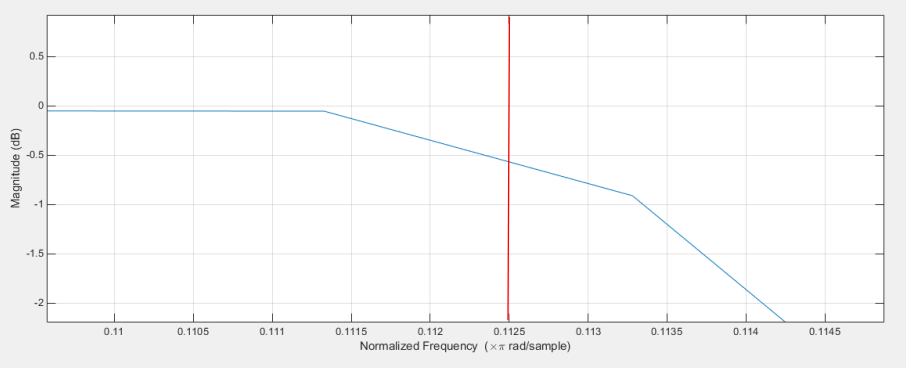
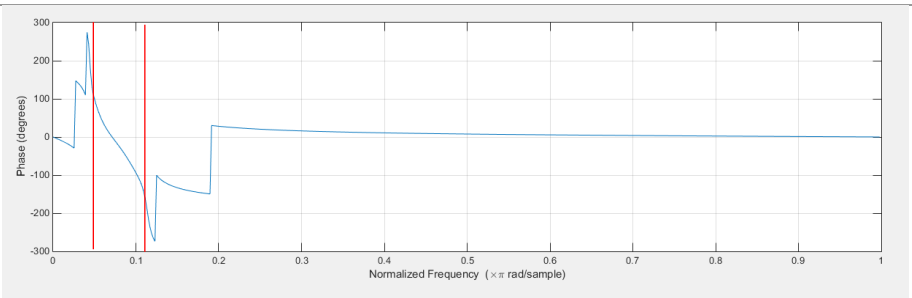
C. *Frequency Response*

1) *MATLAB*

TABLE VI.

Comment	Plot
Overall magnitude response plot	 *Figure 22*

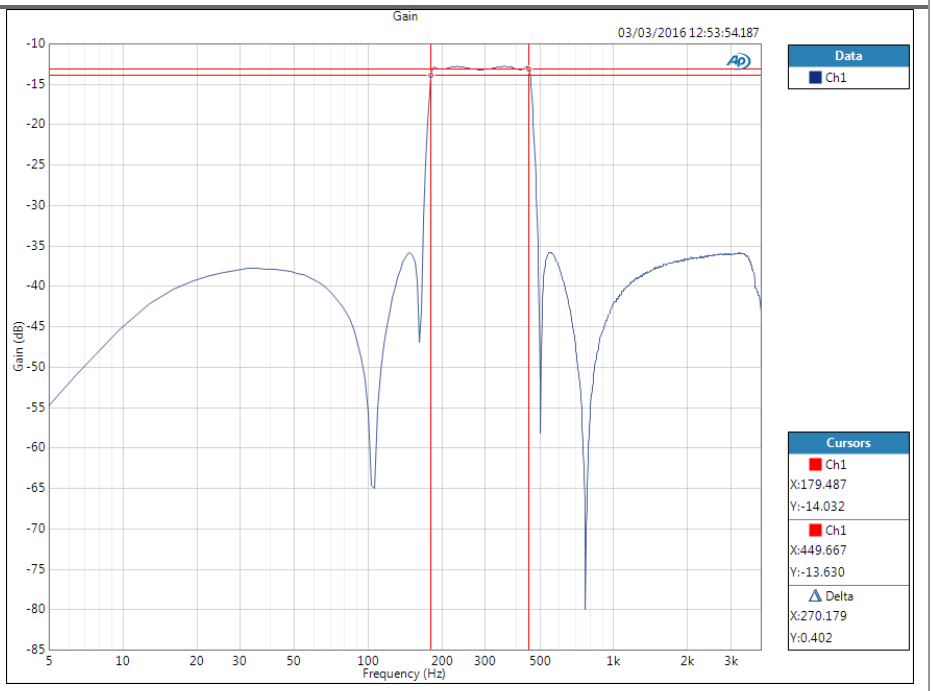
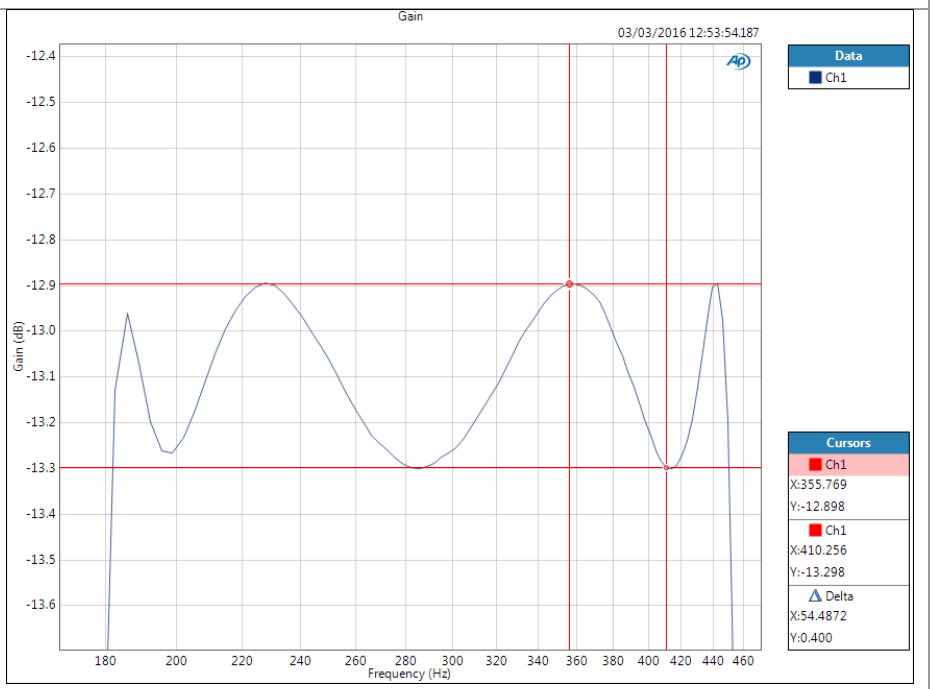
Magnitude response plot showing passband ripple within 0.4dB	 *Figure 23*
Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the lower corner frequency	 *Figure 24*
Magnitude response plot showing a rapid increase in gain from stopband to passband in the lower corner frequency (180hz, when normalized, is 0.045)	 *Figure 25*
Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the higher corner frequency	 *Figure 26*

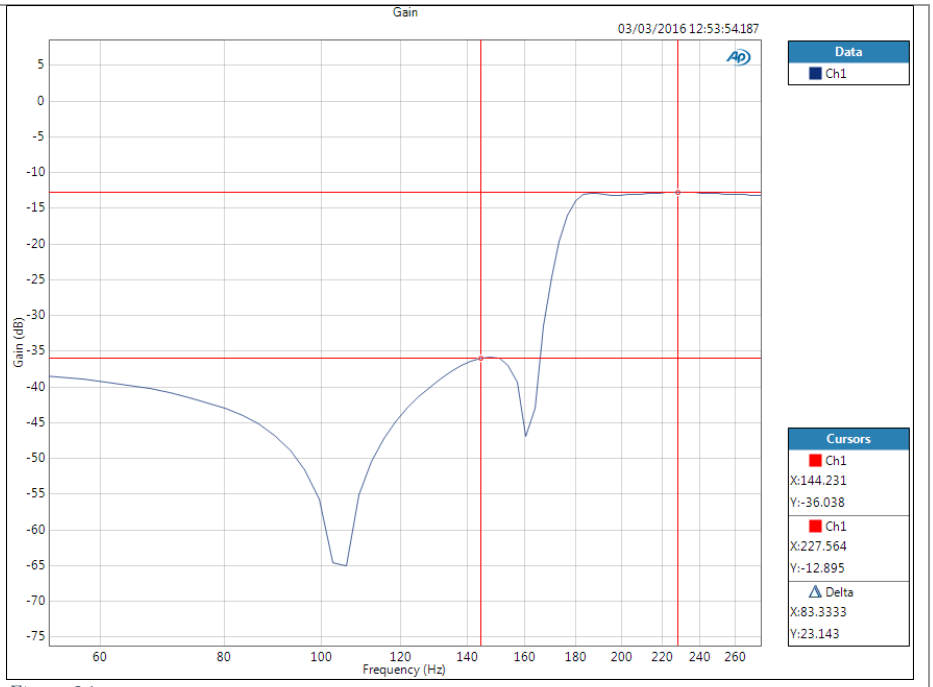
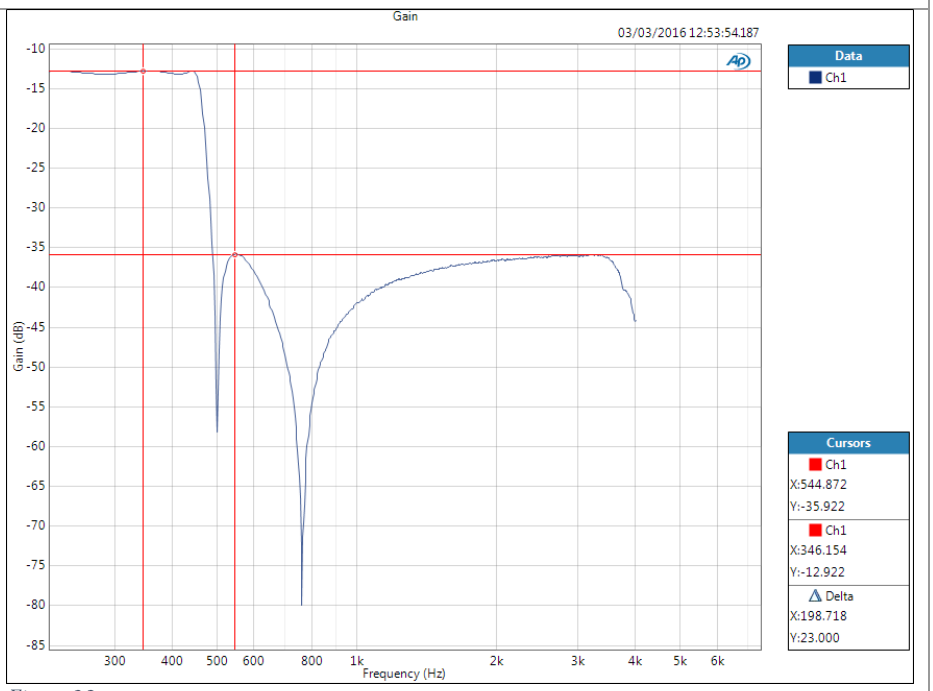
Magnitude response plot showing a rapid increase in gain from stopband to passband in the higher corner frequency (180hz, when normalized, is 0.045)	 *Figure 27*
Phase Response Plot	 *Figure 28*

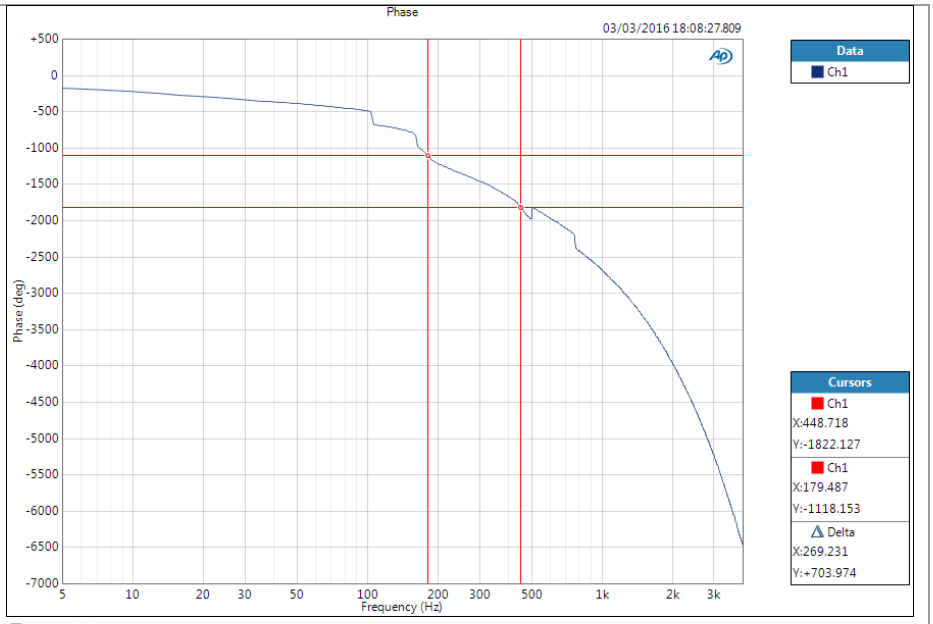
2) APX Audio Analyser

The following table shows the frequency response of our IIR filter on the DSK board, measured using the APX Audio Analyser's continuous sweep function. The waveforms are centred not on 0dB, but around -12dB, as there are 2 input potential dividers (Appendix.A), each with an attenuation factor of 2, on the input of the APX Audio Analyser. Hence, the attenuation factor of 4 corresponds to the gain being 0.25x of the original gain - -12dB.

TABLE VII.

Comment	Plot
Overall magnitude response plot of DSK board, running the Direct Form II function	 *Figure 29*
Magnitude response plot showing passband ripple within 0.4dB	 *Figure 30*

Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the lower corner frequency	 *Figure 31*
Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the higher corner frequency	 *Figure 32*

Phase response plot	 *Figure 33*

3) Discussion on Phase Response

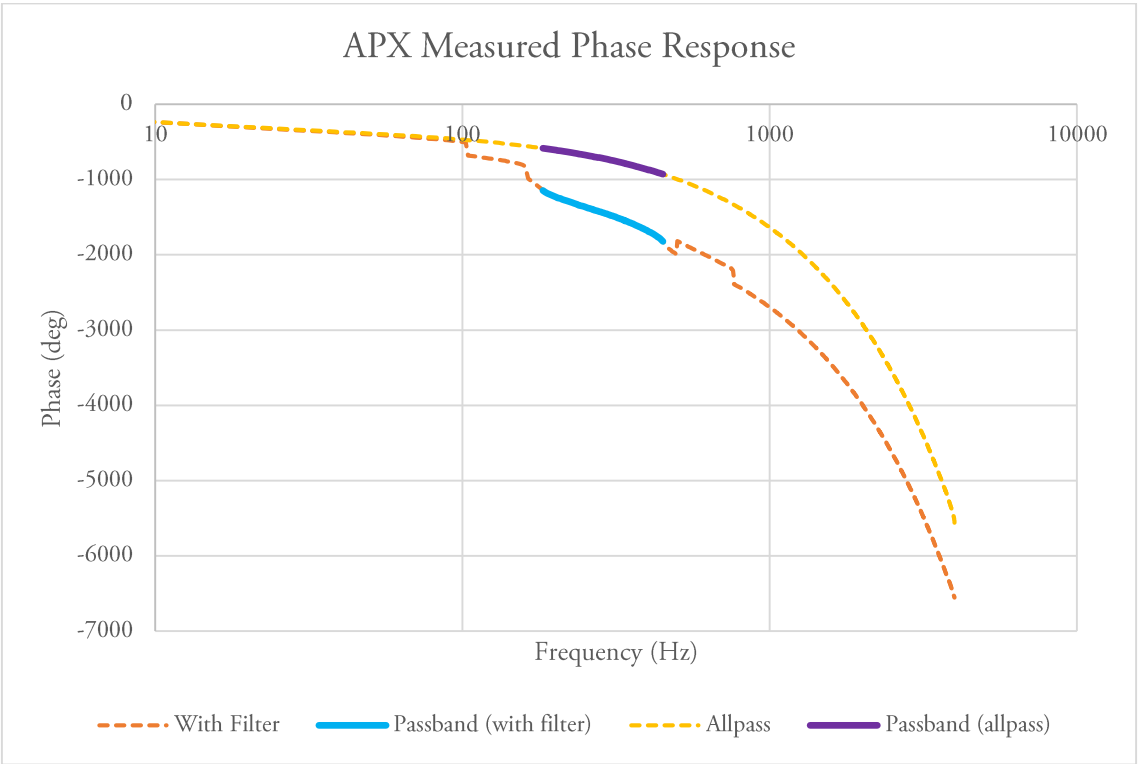


Figure 34: APX Phase Response for all pass filter and IIR filter

From Report 3, the DSK board has a hardware delay [5]. To get the true phase response, the APX measured all pass board response should be subtracted from the APX measured phase response of the double form II filter. Figure 34 plots both the all pass hardware response and the measured phase response of the filter. Figure 35 then shows the corrected phase response of the filter (filter response minus all pass filter response). The corrected phase response is in the form of the MATLAB expected phase response (Figure 28). The solid lines in both figures indicate the region of the passband; the plot is done in dashed lines otherwise.

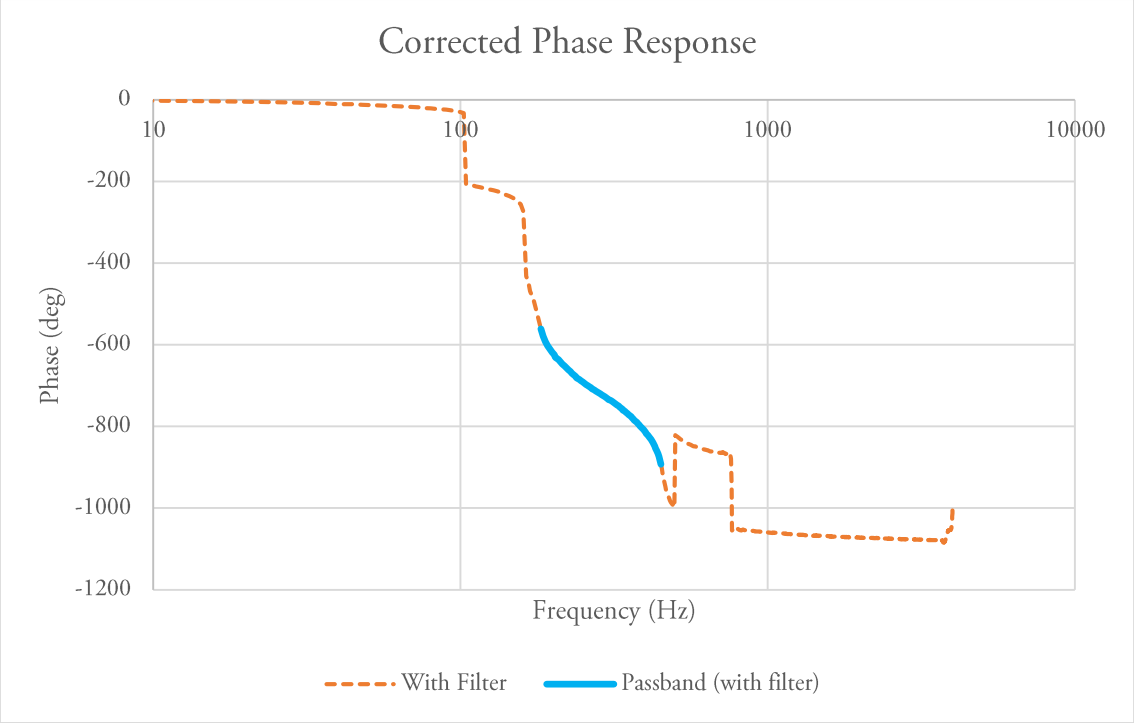


Figure 35: APX Corrected Response of IIR filter

V. DIRECT FORM II TRANSPOSED FILTER

A. *Design*

Any block diagram can be converted into an equivalent transposed form by executing the following instructions [6]:

1. Reverse direction of each branch
2. Reverse direction of multipliers
3. Change junctions to adders and vice-versa
4. Swap input and output signals

B. *Implementaiton and Code Operation*

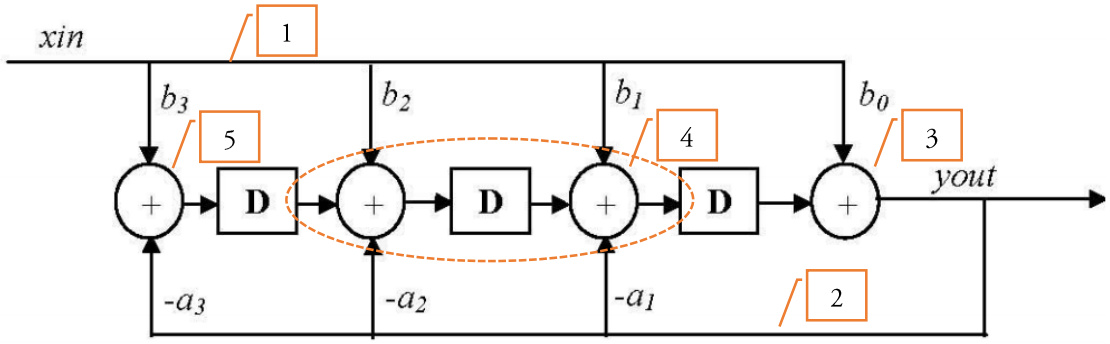


Figure 36: Direct Form II Transposed Structure

Figure 36 shows the direct form II transposed structure. The orange callouts link to corresponding callouts in the code operation section. Using this direct form II transposed structure gives a more efficient coding, as the number of delay elements is a minimum, removes the need for a circular buffer [1] (and thus logic checking for circular buffer overflow), and gives improved resiliency to rounding errors (feedforward `b[]` coefficients have a smaller range, they precede the poles in series order [7]).

```
/**
 * Direct form II transposed implementation. Feedforward b coefficients now
 * directly multiplied by sample in before delay/accumulate. a coefficients used
 * for feedback. Buffer size is N.
 * @param  sample sample_in, xin on diagram
 * @return        64bit output, yout on diagram
*/
double Dir_form_2_transposed(double sample) {
        double out_data = 0.0;        //output variable, used for feedback loop
        int index = 0;                //iterator declaration
        out_data = b[0] * sample + buffer[0]; //do edge case of b0 * sample in + newest
delay line block
        for (index = 1; index<N; index++) {
                buffer[index - 1] = buffer[index] + b[index] * sample - a[index] *
out_data;       //compute delay line, b is feedforward, a is feedback
        }
        buffer[N - 1] = sample*b[N] - a[N] * out_data; //another edge case of oldest
delay line block
        return out_data;
}
```

3 1 2 4 5

This transposed version uses the buffer to implement both the delay line and the operations on the delayed samples. There are two edge cases (callout 3 and 5), which are handled outside the for loop – they have 3 operands in the summation as opposed to 4 operands in the for loop (callout 4). Both current sample and output are multiplied by different coefficients in the accumulators (callout 1 and 2).

C. *Frequency Response*

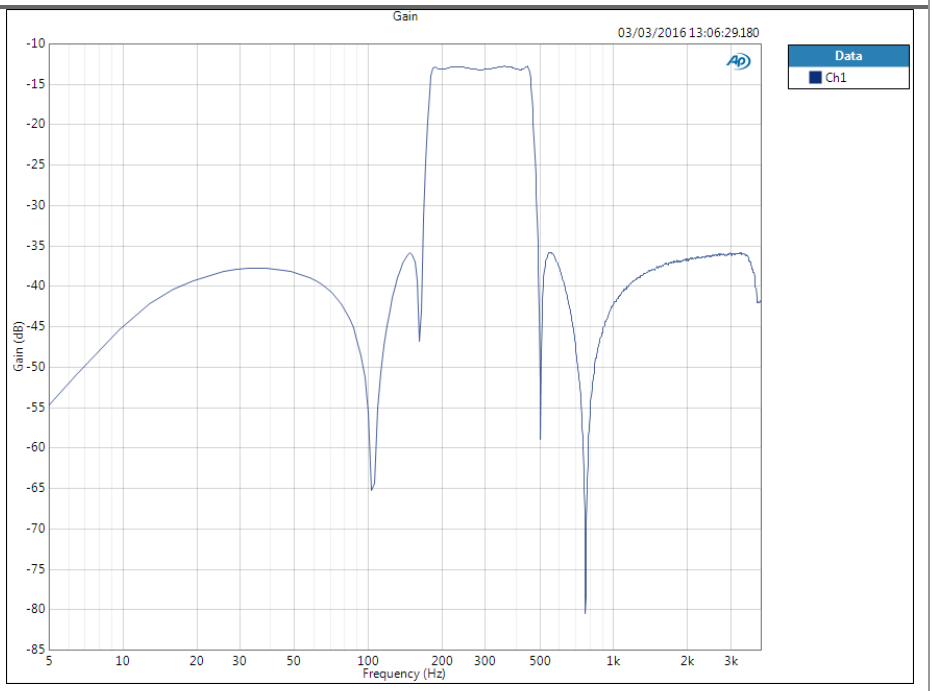
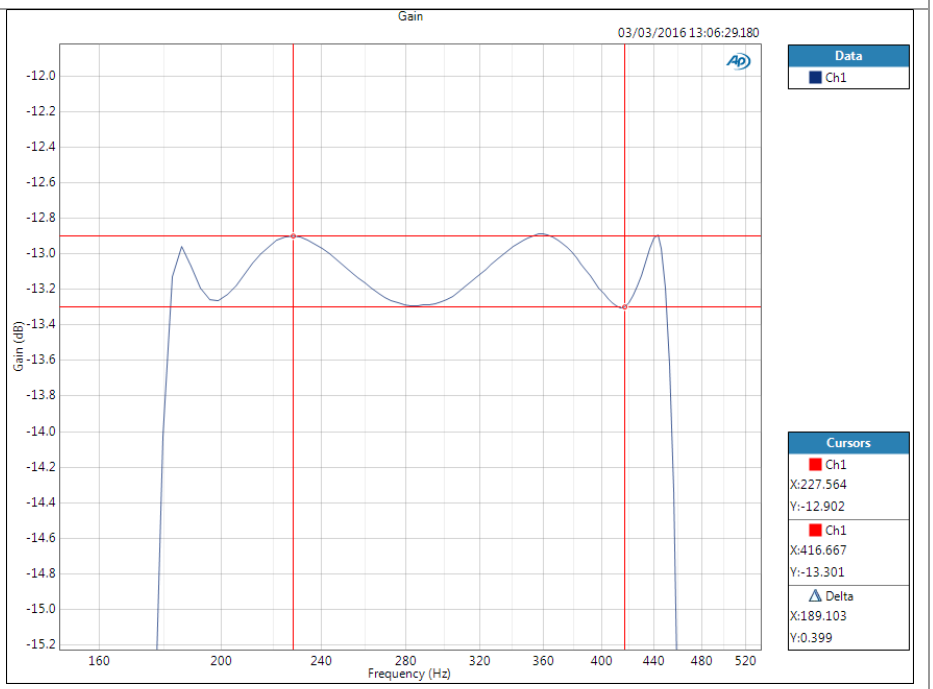
1) *MATLAB*

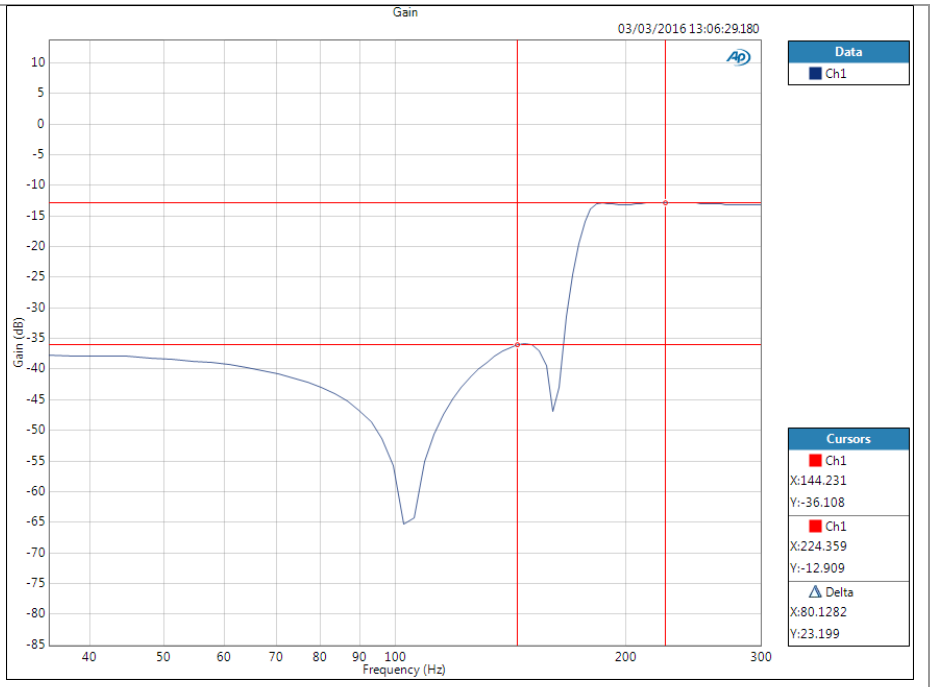
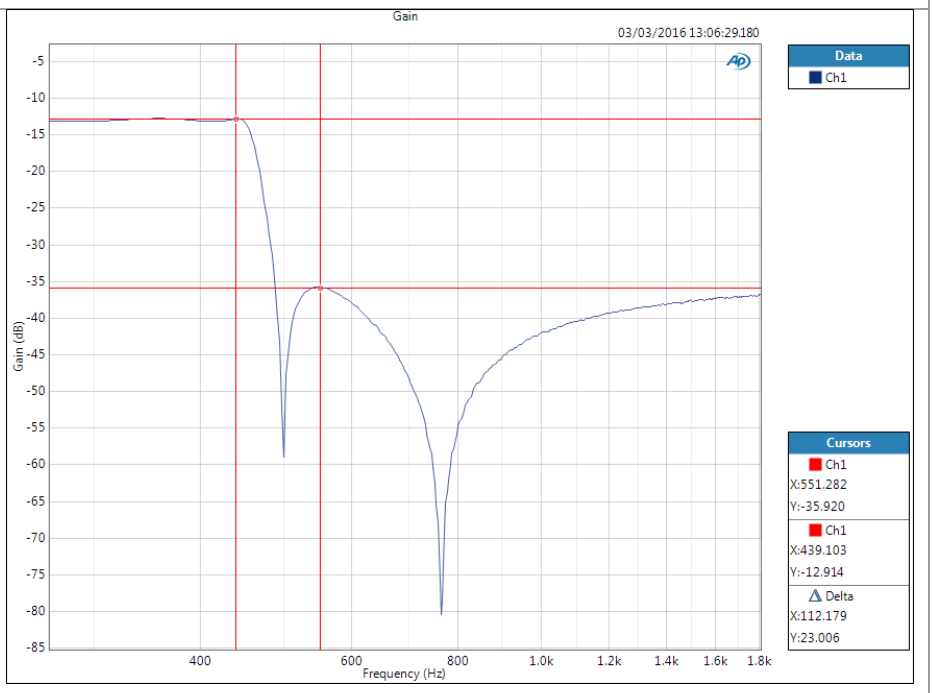
The MATLAB frequency response plots are identical to section IV.C.1, as transposing the direct form II structure does not alter the frequency response. The plots can be found in Figures 22 to 28.

2) *APX*

The following table shows the frequency response of our IIR filter on the DSK board, measured using the APX Audio Analyser's continuous sweep function. The waveforms are centred not on 0dB, but around -12dB, as there are 2 input potential dividers (Appendix.A), each with an attenuation factor of 2, on the input of the APX Audio Analyser. Hence, the attenuation factor of 4 corresponds to the gain being 0.25x of the original, hence -12dB.

TABLE VIII.

Comment	Plot
Overall magnitude response plot of DSK board, running the Direct Form II Transposed function	 *Figure 37*
Magnitude response plot showing passband ripple within 0.4dB	 *Figure 38*

Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the lower corner frequency	 *Figure 39*
Magnitude response plot showing a minimum of -23dB of attenuation between the stopband and passband at the higher corner frequency	 *Figure 40*

Phase response plot

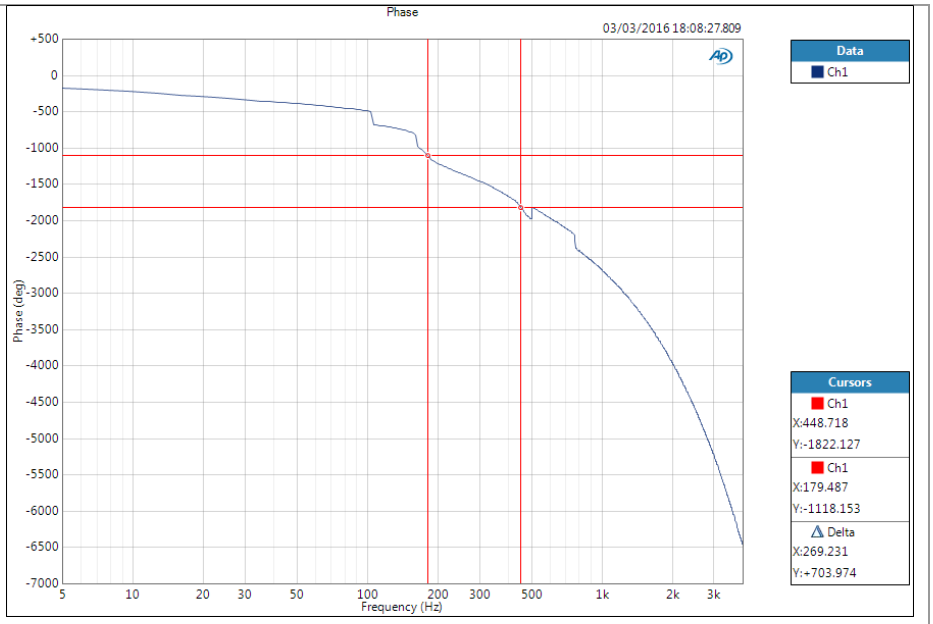


Figure 41

3) Discussion on Phase Response

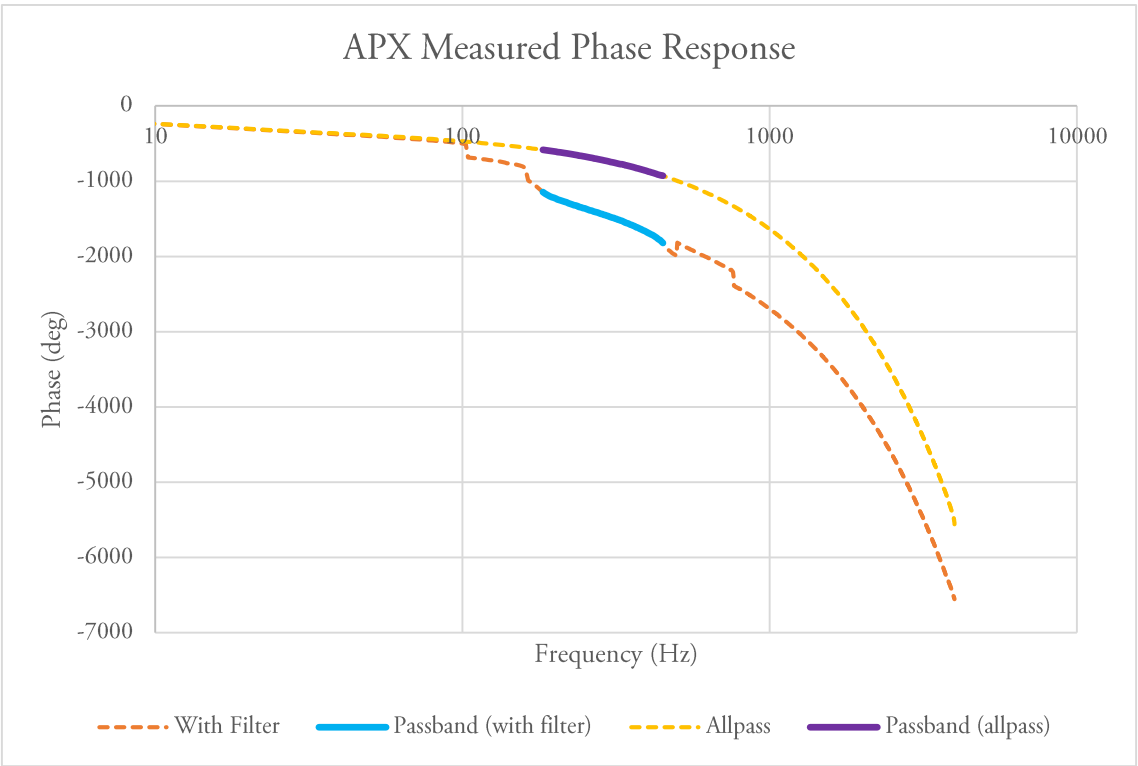


Figure 42: APX measured Phase Response of allpass and IIR filter

Direct form II and direct form II transposed have the same frequency response. Again, from Report 3, the DSK board has a hardware delay [5]. To get the true phase response, the APX measured all pass board response should be subtracted from the APX measured phase response of the double form II filter. Figure 42 plots both the all pass hardware response and the measured phase response of the filter. Figure 43 then shows the corrected phase response of the filter (filter response minus all pass filter response). The corrected phase response is in the form of the MATLAB expected phase response (Figure 28). The solid lines in both figures indicate the region of the passband; the plot is done in dashed lines otherwise.

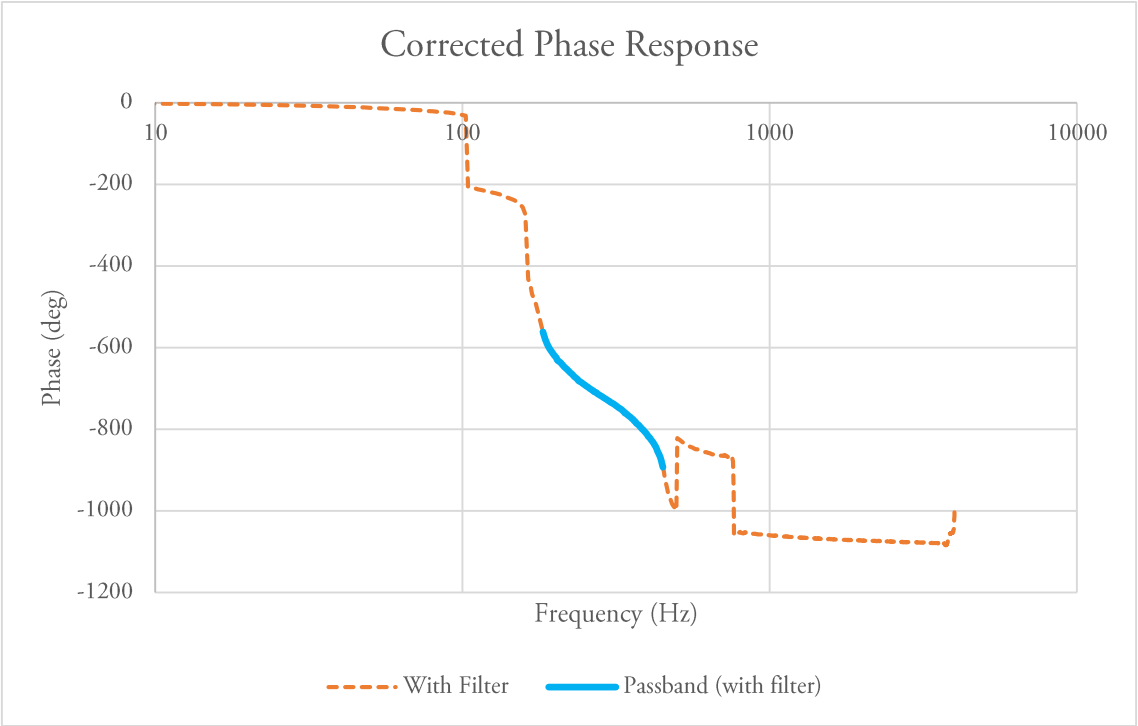


Figure 43: APX corrected phase response of IIR filter

VI. Performance Comparison between Implementations

Table IX shows the benchmarked number of instruction clock cycles needed for both no optimisations used, and –o2 optimisation used. N represents the order of the elliptical filter. DF2 and DF2T stand for direct form II structure and direct form II structure, transposed, respectively. Figure 44 and 45 plot the results.

TABLE IX.

N	None/DF2	None/DF2T	-o2/DF2	-o2/DF2T
4	838	452	169	158
5	1022	545	183	168
6	1200	643	197	183
7	1381	741	211	19
8	1579	875	731	20
9	1760	977	1113	81
10	1941	1079	1192	76

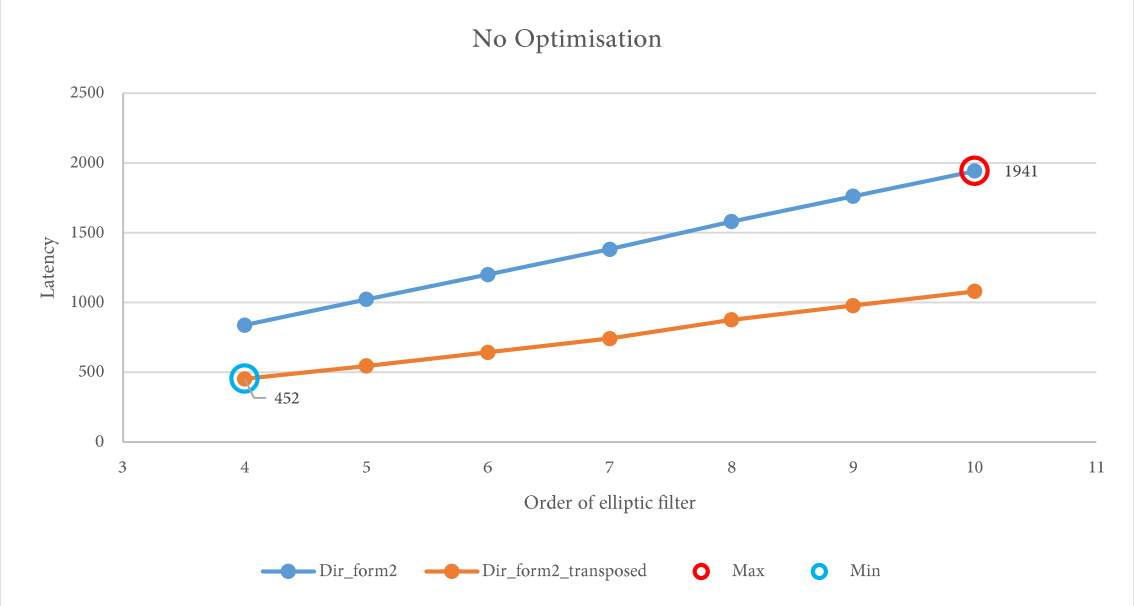


Figure 44: Latency of implementations (no optimisation)

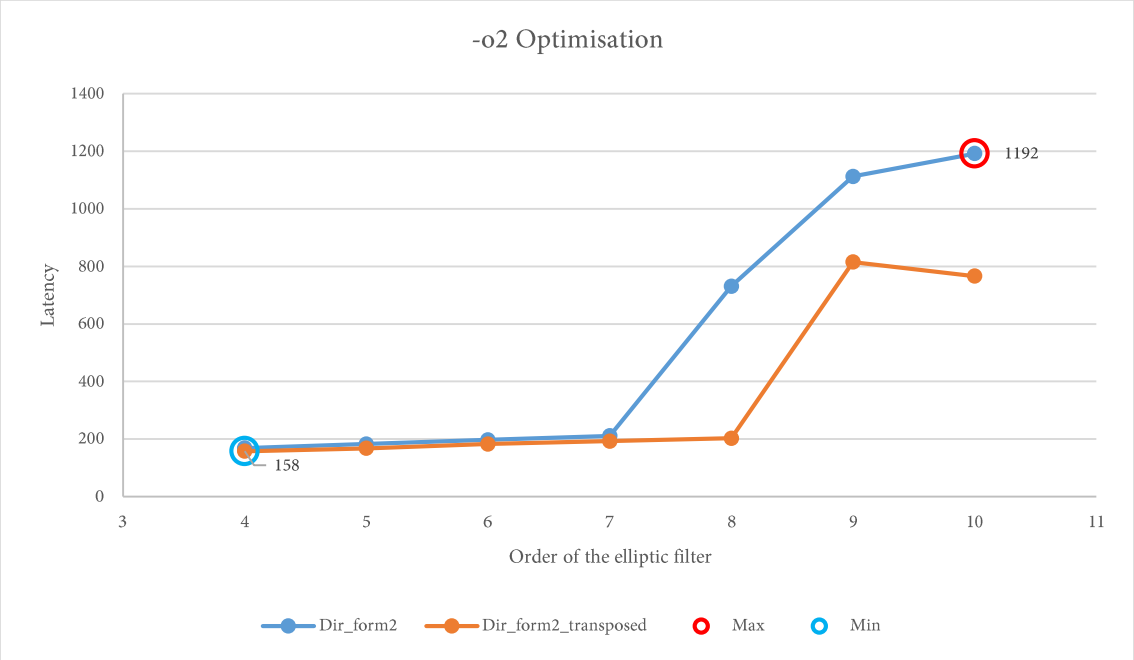


Figure 45: Latency with –o2 optimisation

From the plots in Figure 44 and 45, a linear proportional relationship can be seen. This is to be expected as a higher order means a higher number of iterations of the `for` loop. With no optimisations, the relationship between instruction cycles and order is very linear – this is again expected as the compiler generates assembly code for every line of C code written.

With –o2 optimisation, the compiler performs loop unrolling and software pipelining [8] – there is no method of seeing what optimisations were applied, and hence the relationship is not strictly linear. The jumps in latency can be attributed to, but not limited to, the ability of the compiler to pipeline the MAC operation. However, the general trend is still proportional, which is in line with the explanation that a higher order means a longer `for` loop execution.

The differences between the non-transposed and transposed forms of the direct form ii filter can be explained by looking at the operations done inside the `for` loop. In the non-transposed implementation, the usage of a circular buffer requires logic inside the for loop to check for buffer iterator overflow. This is done by the first if statement (`read_index = ((write_index + loop_index)>N) ? write_index + loop_index - N - 1 : write_index + loop_index;`). Additionally, the a and b coefficients' MAC operations are done separately as there are two adders for the delay line. Improvements to this implementation include using a double buffer to remove the need for the overflow check (memory/speed trade off). However, since speed is not assessed, this was not implemented.

In comparison, the transposed implementation uses a shuffling buffer instead of a circular buffer. Since we do not specify keywords to allow the compiler to use hardware circular buffers (when they exist), there is no need to check for overflow within the `for` loop. Additionally, both a and b's coefficients are summed together in one operation – with –o2, this can be done efficiently, provided the compiler recognises the operation and pipelines it.

VII. Conclusion

Three successful implementations of IIR filters were shown to work according to specifications, and when compared to their MATLAB frequency response plots. Performance benchmarks were as expected, with a linear trend when the number of iterations of the `for` loop increased. The Tustin transform was also used successfully, with differences in time constant/corner frequencies accounted for when looking at the response of the single pole filter.

VIII. References

[1] P. Mitcheson, "Lab 5 – Real-time Implementation of IIR Filters," 18 12 2012. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-591062-dt-content-rid-2714545_1/courses/DSS-EE3_19-15_16/lab5.pdf. [Accessed 1 3 2016].

[2] D. Chaberski, "Elliptical Filters," [Online]. Available: http://www.fizyka.umk.pl/~daras/pfc/filtr_eliptyczny.pdf. [Accessed 2 3 2016].

[3] J. Demmel, "Basic Issues in Floating Point Arithmetic and Error Analysis," University of California, Berkeley, [Online]. Available: http://www.cs.berkeley.edu/~demmel/cs267/lecture21/lecture21.html. [Accessed 1 3 2016].

[4] J. O. S. III, "Direct Form II," in *Introduction to Digital Filters*, p. https://ccrma.stanford.edu/~jos/fp/Direct_Form_II.html.

[5] J. Chan and C. Y. D. Kwok, "Report 3," [Online]. Available: https://onedrive.live.com/redir?resid=28C1FDE97C2C373D!15768&authkey=!AEMmLmZPapoFXj8&ithint=file%2cpdf. [Accessed 3 3 2016].

[6] J. O. S. III, "Transposed Direct-Forms," in *Introduction to Digital Filters*, p. https://ccrma.stanford.edu/~jos/fp/Direct_Form_II.html.

[7] J. O. S. III, "Numerical Robustness of TDF-II," in *Introduction to Digital Filters*, p. https://ccrma.stanford.edu/~jos/filters/Numerical_Robustness_TDF_II.html.

[8] Texas Instruments, "TMS320C6000 Optimizing Compiler v7.4," 7 2012. [Online]. Available: http://www.ti.com/lit/ug/spru187u/spru187u.pdf. [Accessed 23 2 2016].

[9] P. D. Mitcheson, "Lab 4 – Real-time Implementation of FIR Filters," 1 2014. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-591059-dt-content-rid-2714544_1/courses/DSS-EE3_19-15_16/lab4.pdf. [Accessed 25 2 2016].

[10] MathWorks, "ellip," [Online]. Available: http://uk.mathworks.com/help/signal/ref/ellip.html. [Accessed 1 3 2016].

IX. Appendix

A. APX Input

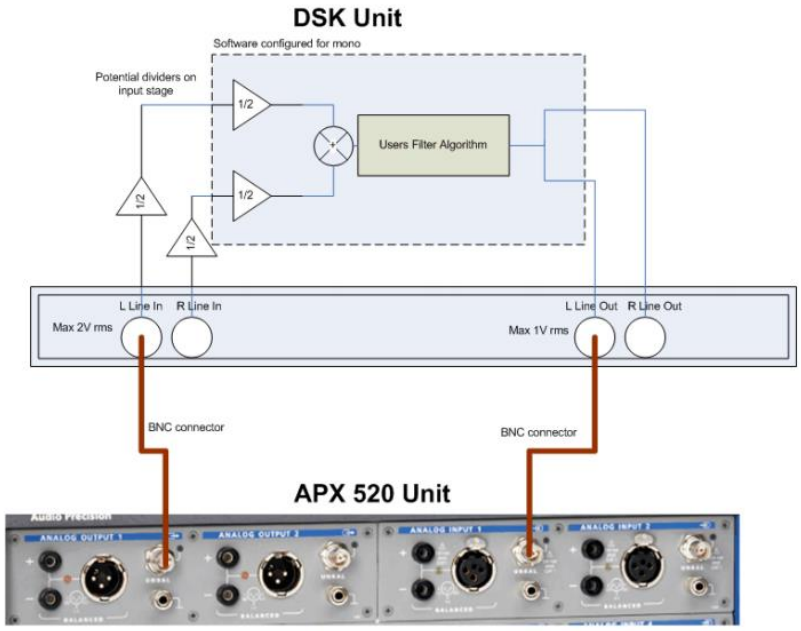


Figure 46: Attenuation factor of 4 (0.25x = -12dB) on input of DSK unit [9]

B. MATLAB Scripts

1) Single Pole Filter

```
RC=10^-3;                                   %as defined in analogue diagram
sys=zpk([],[-1/RC],1/RC);                   %zpk converts system into zero pole form
[num,den]=zp2tf([],[-1/RC],1/RC);           %create transfer function
[zd,pd] = bilinear(num,den,8000);           %tustin transform
sys2=filt(zd,pd,1/8000);                    %implement digital filter
bode_diag=bodeplot(sys2);                   %plot frequency response
setoptions(bode_diag,'FreqUnits','Hz');     %graph options
axis([1,3800,-90 ,0]);                      %graph options
```

2) Direct Form II

```
%[b,a] = ellip(n,Rp,Rs,Wp) returns the transfer function coefficients of an nth-order
lowpass digital elliptic filter with normalized passband edge frequency Wp. The
resulting filter has Rp decibels of peak-to-peak passband ripple and Rs decibels of
stopband attenuation down from the peak passband value. [2]
n=4;                                        %order
Rp=0.4;                                     %pass band ripple
Rs=23;                                      %stop band attenuation
fs=8000;                                    %sampling frequency
corner_f=[180 450]/(fs/2);                  %normalized frequency
[b,a]=ellip(n,Rp,Rs,corner_f);              %create elliptical filter coefficients
freqz(b,a);                                 %plot filter
fid=fopen('../RTDSPlab/lab5/iir_coef.txt','w');     %write to text file
fprintf(fid,'double b[]={');
fprintf(fid,'%.10e,\t',b);
fprintf(fid,'};\n');
fprintf(fid,'double a[]={');
fprintf(fid,'%.10e,\t',a);
fprintf(fid,'};\n');
fprintf(fid,'#define N %d',length(a)-1);     %write order to avoid messing about with
calculating order in C
fclose(fid);
figure;
zplane(b, a);                               %zero pole plot
```

3) Generated coefficients

```
double b[]={6.6809554678e-02,   -4.9957024702e-01,  1.6655257806e+00,   -3.2348395587e+00,
4.0041493226e+00,       -3.2348395587e+00,     1.6655257806e+00,       -4.9957024702e-01,
6.6809554678e-02,   };
double a[]={1.0000000000e+00,   -7.4978022855e+00,  2.4825584715e+01,   -4.7406557909e+01,
5.7101423045e+01,       -4.4424508907e+01,     2.1801396783e+01,       -6.1710120493e+00,
7.7148200123e-01,   };
#define N 8
```

C. *Source Code (custom functions only)*

```
/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE
HERE***********************/
/**
 * This function gets executed upon a hardware interrupt. It reads in a
sample,
 * then outputs the output of one the IIR functions.
 */
void interrupt_service_routine(void) {
       double sum = 0.0;                //for IIR filter sum
       double sample = 0.0;             //for incoming sample
       sample = (double)mono_read_16Bit();//read incoming sample on receive
interrupt
       //sum = Single_pole(sample);
       //sum = Dir_form_2(sample);
       sum = Dir_form_2_transposed(sample);
       mono_write_16Bit((Int16)(sum));  //output sum to both channels
}

/**
* Initialises the buffer to all 0's.
*/
void init_buffer(void) {
       int index = 0;
       for (index = 0; index<N + 1; index++) {
              buffer[index] = 0;
       }
}

/**
* Difference equation implementation of the IIR filter.
* @param  sample
* @return        [description]
*/
double Single_pole(double sample) {
       static double out = 0.0;               //use static variable to store
output value
       static double last_x = 0.0;            //use static variable to store
last input value
       out = b0*sample + b0*last_x - a1*out;  //difference equation
       last_x = sample;                       //put current sample into last
sample variable
       return out;
}

/**
 * Elliptic IIR filter, direct form II implementation. Uses MATLAB generated
 * coefficients specified in include txt file to calculate filter. Circular
 * buffer used, initialized using calloc().
 * @param  sample sample_in, xin on diagram
 * @return        64bit output, yout on diagram
*/
double Dir_form_2(double sample) {
```

```
	static int write_index = N;			//updatable write index for
newest delay line block
	double in_data = 0.0;				//for multiplying by a1, a2,
a3...
	double out_data = 0.0;				//for multiplying by b1, b2,
b3...
	double write_data = 0.0;			//summation of delayed
coefficients, multiplied by a1, a2, a3, with input value
	int read_index, loop_index;			//declaration of iterators
	for (loop_index = 1; loop_index <= N; loop_index++) {
		read_index = ((write_index + loop_index)>N) ? write_index +
loop_index - N - 1 : write_index + loop_index; //circular buffer wraparound
logic
		out_data += buffer[read_index] * b[loop_index];	//output
acculumator
		in_data -= buffer[read_index] * a[loop_index]; //input
acculumator
	}
	write_data = sample + in_data;			//summation of incoming sample
and input side of direct form
	buffer[write_index] = write_data;		//write to newest delay line
block
	out_data += write_data*b[0];			//multiplication of summed
write_data, aka top line of direct form structure
	if (--write_index == -1)write_index = N;	//handle circular buffer
wraparound
	return out_data;
}

/**
 * Direct form II transposed implementation. Feedforward b coefficients now
 * directly multiplied by sample in before delay/accumulate. a coefficients
used
 * for feedback. Buffer size is N.
 * @param  sample sample_in, xin on diagram
 * @return        64bit output, yout on diagram
*/
double Dir_form_2_transposed(double sample) {
	double out_data = 0.0;				//output variable, used for
feedback loop
	int index = 0;					//iterator declaration
	out_data = b[0] * sample + buffer[0];	//do edge case of b0 * sample in
+ newest delay line block
	for (index = 1; index<N; index++) {
		buffer[index - 1] = buffer[index] + b[index] * sample - a[index]
* out_data;  //compute delay line, b is feedforward, a is feedback
	}
	buffer[N - 1] = sample*b[N] - a[N] * out_data; //another edge case of
oldest delay line block
	return out_data;
}
```