Jurgen Shiqerukaj - CID 00938154 Jason Yuan – CID 00954645 Due 3rd march 2017

Lab 5 – RT Implementation of IIR Filters

Contents

Overview

In this lab session we will learn how to design and implement IIR filters. We will use the Tustin transform and MATLAB to find the corresponding coefficient for a given filter specification. Additionally we will compare the performance of different direct form of the IIR filters.

Design of Simple Filter using the Tustin Transform

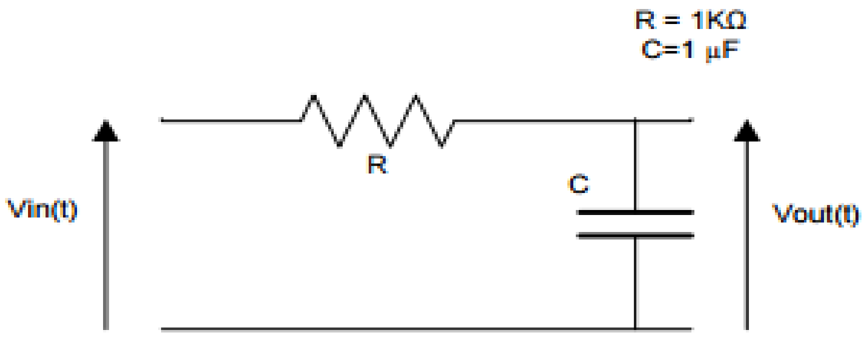


Figure 1: Image shows a simple filter RC in which we'll use the Tustin transform to find corresponding coefficients that create this filter in the digital domain [1]

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

To find the coefficients for a given filter the transfer function must be re-arranged in this given form which involves converting the s-domain to the discrete domain by the uses of the Tustin transform stated below

$$using\ s = \frac{In(z)}{T_s}\ where\ T_s\ time\ interval\ between\ every\ sample$$

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

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

$$H(s) = \frac{1}{\frac{2}{T_s}\frac{z-1}{z+1}RC + 1}$$

$$H(s) = \frac{T_s z + T_s}{2RC(z-1) + T_s z + T_s}$$

$$RC = \frac{1}{1000}\ \&\ T_s = \frac{1}{8000}$$

$$H(s) = \frac{\frac{1}{8000}z + \frac{1}{8000}}{\left(\frac{2}{1000} + \frac{1}{8000}\right)z + \frac{1}{8000} - \frac{2}{1000}}$$

$$H(s) = \frac{\frac{1}{8000}z + \frac{1}{8000}}{\frac{17}{8000}z - \frac{15}{8000}}$$

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

$$From\ the\ form\ above\ we\ obtain\ a_0 = 1\ , a_1 = \frac{-15}{17}\ , b_0 = \frac{1}{17}\ , b_1 = \frac{1}{17}$$

There are also a few other properties about that can be examined for this simple RC lowpass filter, more notably the cutoff frequency and time constant.

We can find out the time constant of the circuit which is the time taken for the capacitor to discharge by 63 %. This is a quick and simple calculation.

$$\tau = \mathrm{RC} = 1\ \mathrm{ms}$$

To find the cutoff frequency we use equation which makes use of the time constant:

$$\frac{1}{2 * \pi * RC}$$

By substituting for R and C we get:

$$\frac{1}{2*\pi*(1000)(0.000001)} = 159.15\ \mathrm{Hz}$$

Implementation on the DSP

$$y(n) = \sum_{k=0}^{M} b(k)x(n-k) - \sum_{k=1}^{N} a(k)y(n-k)$$

Using the convolution sum of a typical IIR filter we can use the coefficients found from the Tustin transform exercise to implement this simple filter on the DSP.

$$Y(n) = b(0)x(0) + b(1)x(n-1) - a(1)y(n-1)$$

Since we treated the 0th element of the array to be the most recent element x(n-1) will be treated as the 1st element of the array since we shift each input sample to the right.

```
/********Variables used for Simple IIR Filter***********/
double x[];
short output;
short read_sample;
const double simplea[] = {1,0.88235294117};
const double simpleb[] = {0.05882352941,0.05882352941};


short simple_IIR()
{
      x[1] = x[0]; // Shift previous input sample in the inputbuffer.
      x[0] = read_sample; // Stores input sample in the input buffer [0].

      output = x[0]*simpleb[0] + x[1]*simpleb[1] - output*simplea[1]; //
IIR Convolution Sum.

      return output ; // Returns output back to mono_write so it can be
outputted to DSP.
}
```

The simple IIR filter works by shifting the previous input sample to the next space in the buffer then replaces the 1st element of the input buffer with the current input sample then the output is calculated by a convolution sum. The output is then returned back to the mono_write function so the output can be outputted to audio port of the DSP.

Analysis between digital and analogue

To compare how well we have implemented our analogue model on the DSP we will be comparing our design to the theoretical properties of the RC circuit with what we actually get using the DSP.

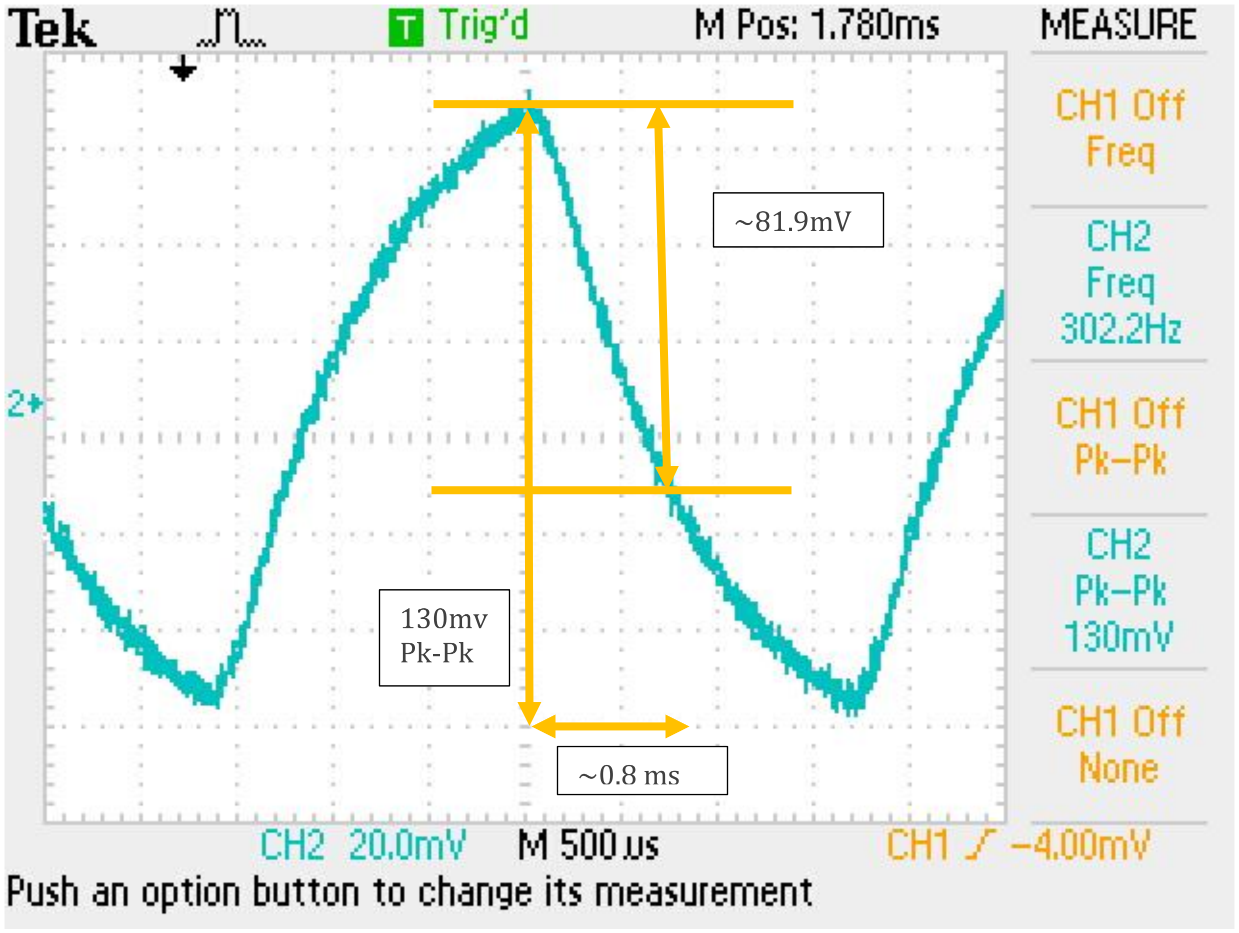


Figure 2: Image shows our digital filter output from a 300 Hz square wave input. Output shows a capacitor discharge which we used to find the Time constant and compare it to our theoretical analogue circuit model.

We can see from the figure above we do indeed get the desired output waveform with an input square wave. This is because in a square wave voltage changes almost instantaneously whilst we know that the voltage across a capacitor can't change instantly due to equation $C = \frac{dQ}{dV}$ hence it takes time for a capacitor to charge and discharge.

From this charging and discharging waveform we can retrieve the time constant. We notice that the output waveform has a Pk-Pk of 130mV hence to find the time constant we have to measure the time it takes for the capacitor to discharge by 63% of Pk-Pk voltage which would be ~81.9mV. This value can be easy found out as it's simply 20mV per division. The time from the point the capacitor discharges to the 63% mark is around 0.8 ms. This is very close to the 1 ms time constant we theoretically calculated, however we have to account for human error in reading the oscilloscope (even with our best efforts making it readable as possible it incredibly hard to get pin point accuracy).

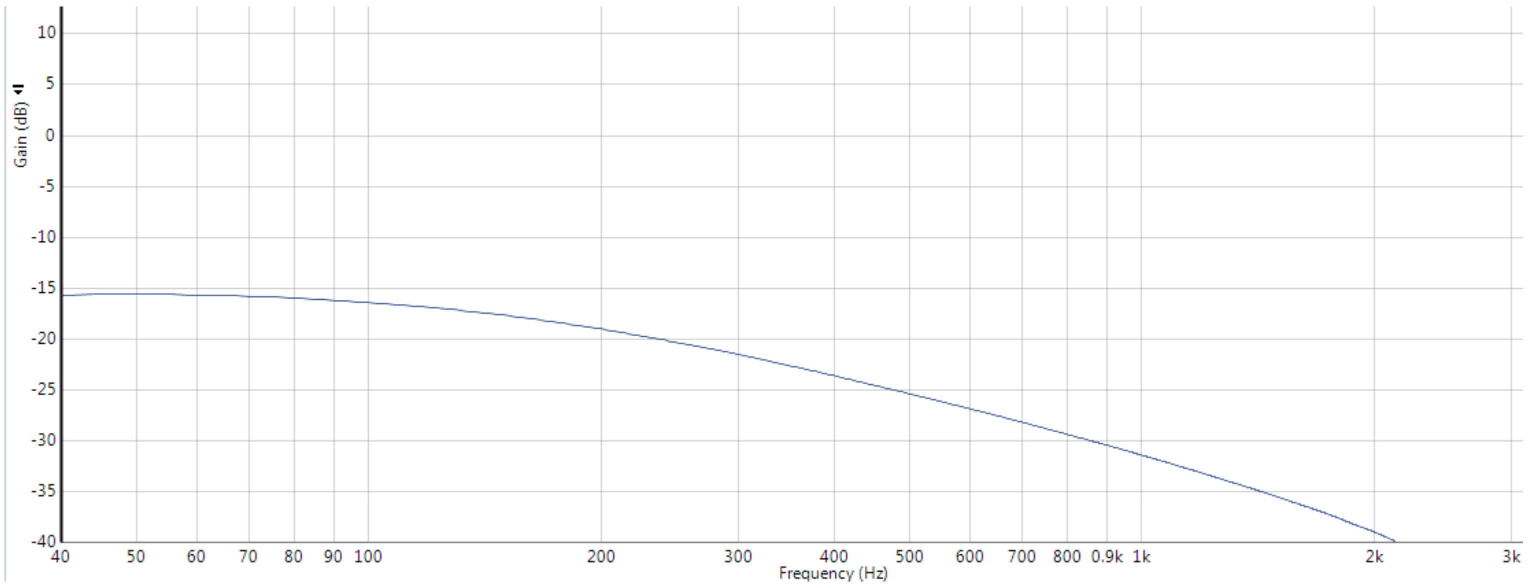


Figure 3: Image shows our digital filter through a network analyser, shows we get a low pass filter.

To compare how our cut off frequency matches the theoretical we will use the time constant we found our beforehand. Using

$$\frac{1}{2 * \pi * RC}$$

with time constant value of 0.8 ms we get the cut off frequency equal to:

$$\frac{1}{2 * \pi * 0.0008} = 199\ Hz$$

We can clearly see from running this within a network analyser that we get a low pass shape with strong attenuation roll off around after the 199Hz mark. This is slightly off the theoretical cut off being 159 Hz a possible reason for this could the high pass filter at the input and human error in finding the time constant with oscilloscope. Even though we have very carefully chosen our input square wave, after attempting several input square waves, we have found that using a input square wave of 300 Hz provides the most **readable** result with the high pass filter not effecting the result by much.

Ellipic Filters

There are four common types of IIR filters being Butterworth, Chebyshev,Inverse Chebyshev and Elliptic. Each of which have advantages and disadvantages . In this lab we will be using a Elliptic filter which has the sharpest cut off of all common filter types but this comes with a slight trade-off resulting in ripples in the pass and stop band.

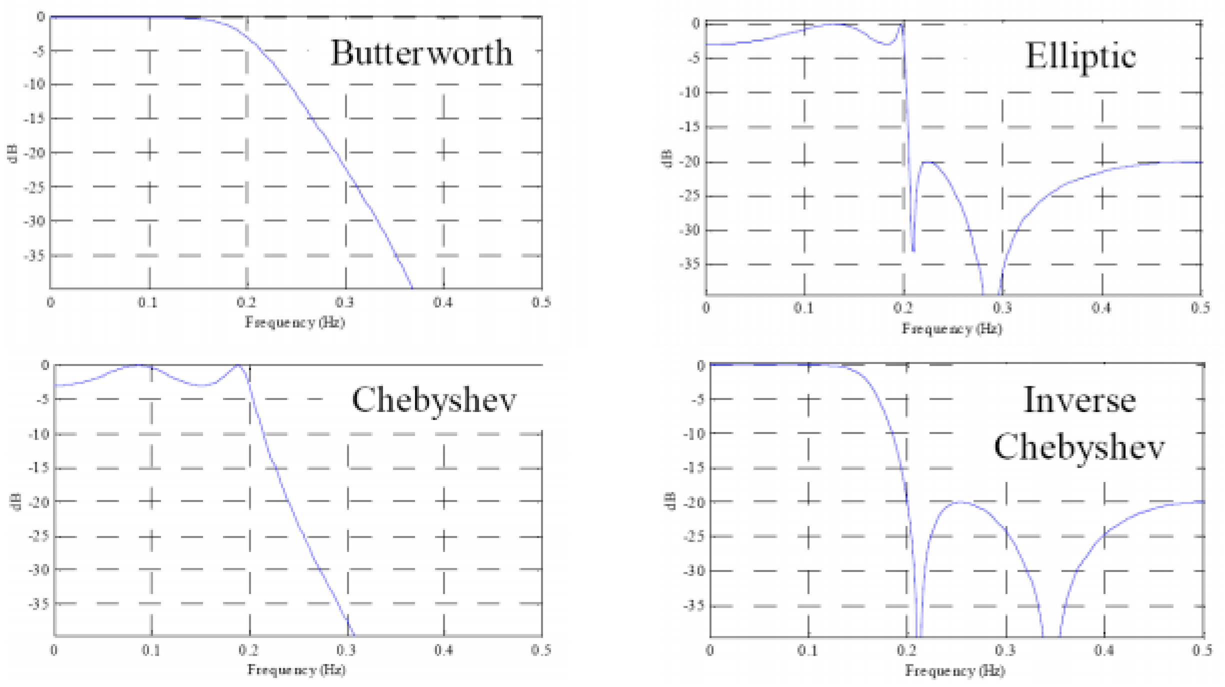


Figure 4: Image shows the traits of the 4 most common IIR filters, we can see that the elliptic filter has by far the steepest cutoff attenuation. [1]

Ellipic band pass filter design with MATLAB

In this exercise we will create an elliptic band pass filter with following specifications using MATLAB:

Order: 4th
Passband: 200-450Hz
Passband ripple: 0.3dB
Stopband attenuation: 20dB

Below is the MATLAB code used to implement this filter design:

```
%% Bandstop Elliptic Filter

Order= 4 % sets the order of filter we would like.
[b,a] = ellip(order/2,0.3,20,[200 450]/4000,'bandpass');
%Matlab function that takes in order of filter, ripple, stopband attenuation,
cut off frequencies and type of filter and returns two sets of co-efficients
later used to implement filter.

freqz(b,a); %Plot and see filter.

bb=['double b[]={']; % create a string starting with 'double b[]={'
aa=['double a[]={'];

for i = 1:(order*2+1)  % For loop used to cycle through all co-efficients.

    bb = [bb  num2str(b(i)) ', ' ]  %Add co-efficients to the string
                                    %followed by a comma.
end

for i = 1:(order*2+1)  % For loop used to cycle through all co-efficients.

    aa = [aa  num2str(a(i)) ', ' ]  %Add co-efficients to the string
                                    %followed by a comma.
end

aa = [aa ' };'];   %end string with curly brackets and semi-colon.
bb = [ bb ' };'];  %end string with curly brackets and semi-colon.
```

The function ellip takes in 5 variables as follows: order, passband ripple, passband frequency (frequency normalized to the nyquist frequency), filter type.

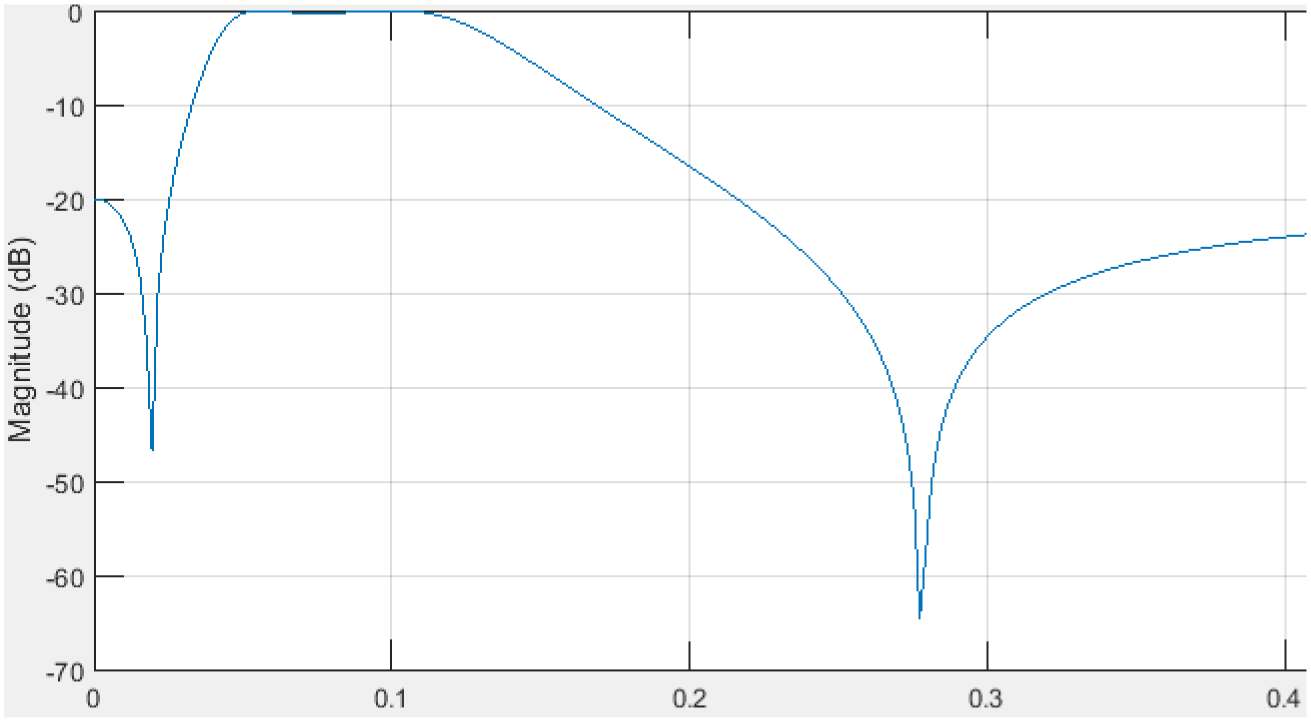


Figure 5: Images shows the amplitude response that MATLAB created with ellip function. We can see we get a band pass filter shape. The x-axis is also normalized but frequency can easily found by multiplying by 4000.

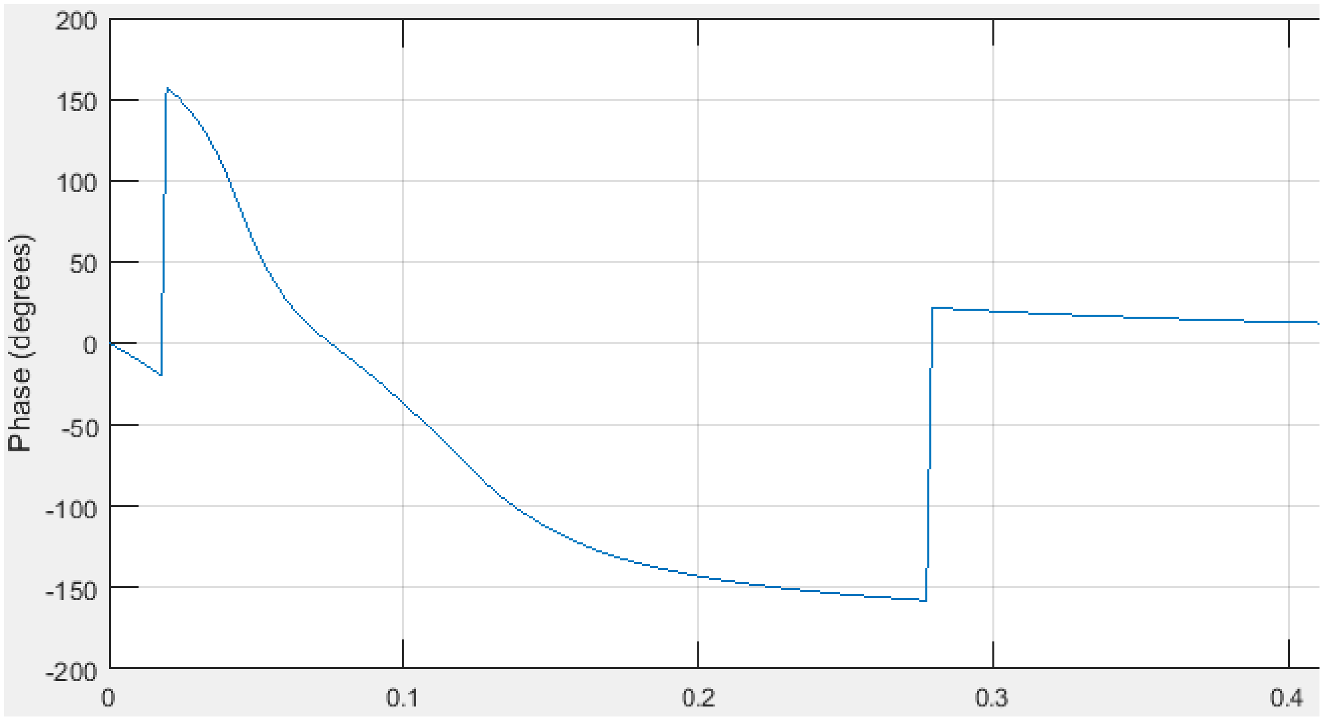


Figure 6: Images shows the phase response that MATLAB created with ellip function. We can see it is NOT linear phase. The condition for linear phase is that coefficients should be symmetric. However even though b is indeed symmetric, a is not.

Bandpass filter: Direct form II Implementation

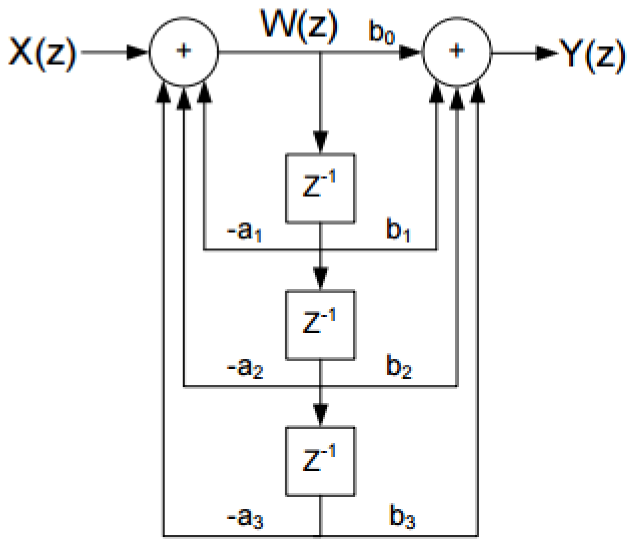


Figure 7: Image shows the direct II form implementation of an IIR Filter. Where Z^{-1} blocks represent 1 unit of delay. .[1]

Using the diagram above we can formulate 2 equations which would help in the writing this code.

$$W(z) = X(z) - a(1)W(z-1) - a(2)W(z-2) - a(3)W(z-3)$$

$$Y(z) = b(0)W(z) + b(1)W(z-1) + b(2)W(z-2) + b(3)W(z-3)$$

The direct form II implementation works by first taking in an input sample and storing this in the 0th element of the w-buffer then the output is re-setted to zero to ensure the accumulation of the elements are correct. The purpose of the first line of the for loop is used to calculate a new value of w[0] based on the previous values inside the w-buffer essentially performing the left hand side of the implementation similarly the second line of the for loop performs the right hand side. Lastly, the final line in the loop performs the unit time shift e.g. w[3] = w[2], w[2] = w[1]. After the for loop has been completed the new of w[0] is multiplied with b[0] and added to output which then gets returned back to mono_write function.

```
/**********Variables Used ***************/
#define order  (sizeof(a)/sizeof(a[0])-1)
int i; // Used as loop counter
double w[]; //Middle Buffer for directformII;
double output; // holds the output
short read_sample; // holds the input sample

short directformII()
{
      w[0] = read_sample; // Places input sample into the 0th element of
the w buffer
      output = 0; // resets the out to 0 at every function call

      for( i = order ; i > 0 ; i--)
      {
            w[0] -= a[i]*w[i]; // performs the left hand side of the branch
            output += b[i]*w[i]; // performs the right hand side of the
branch addition
            w[i] = w[i-1]; // performs the time shift required
      }

      output += w[0]*b[0]; //updates the new output and return this back
the DSP

      return output;

}
```

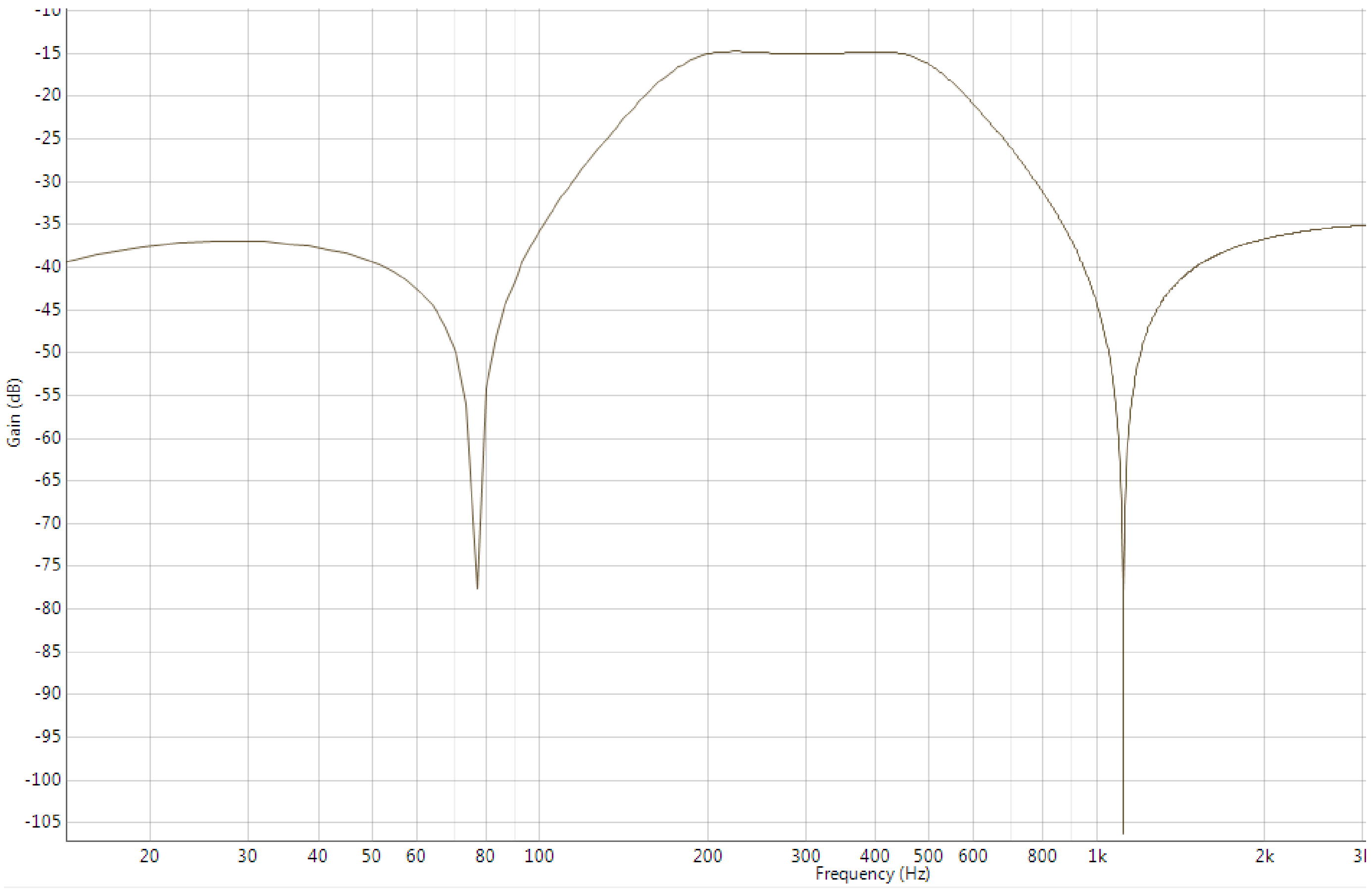
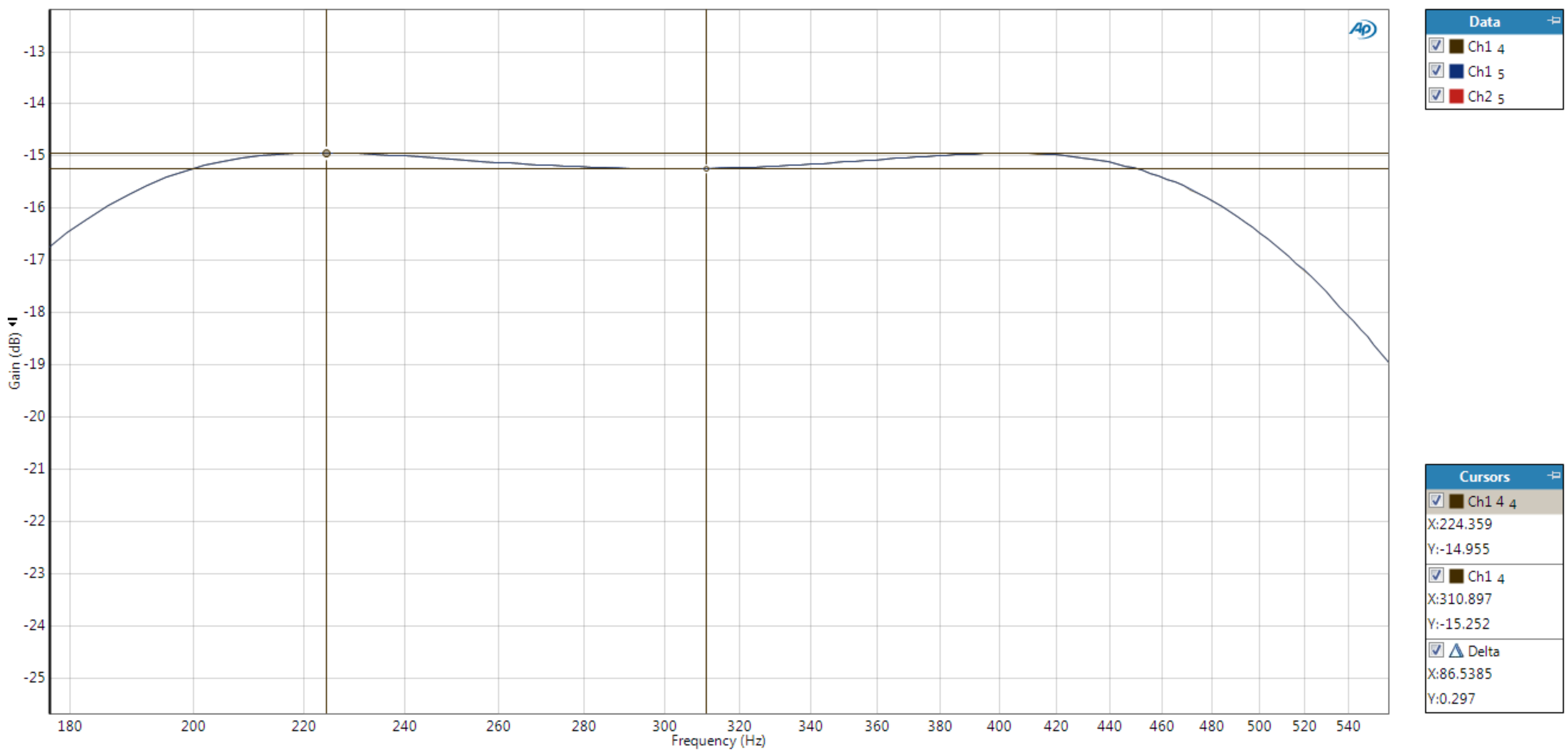


Figure 8: Image shows the filter response using the Spectrum analyser APx 520. We notice that it has a identical shape to that found within MATLAB as expected. As the output is a quarter due to DSk unit [APPENDIX] the bandpass gain starts at -15 dB



Ch1 4 4
X:224.359
Y:-14.955

Ch1 4
X:310.897
Y:-15.252

Delta
X:86.5385
Y:0.297

Figure 9: Image shows the filter response using the Spectrum analyser APx 520. We have zoomed in onto the pass band ripple. We notice that ripple is indeed just within 0.3 dB range , within specifications and near identical to result found in MATLAB.

By comparing our frequency results to MATLAB we find they are near enough identical which is to be expected as we are indeed using coefficients provided directly from MATLAB.

Bandpass filter: Direct form II Transpose Implementation

With a similar strategy for direct filter, we use a feedback structure as in figure

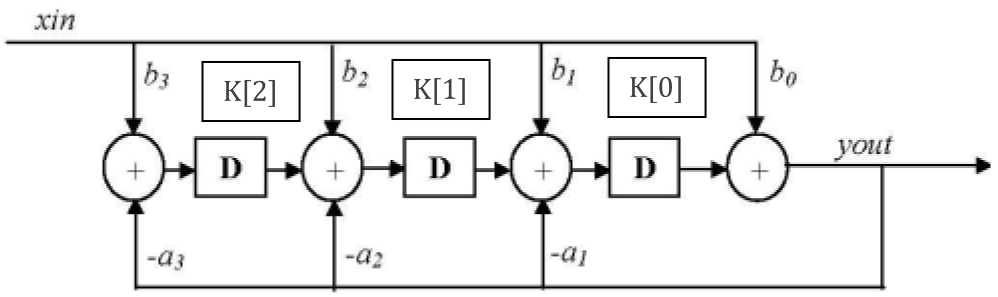


Figure 6 Direct form II transposed structure

Figure 10: Image shows the logical steps to implement the direct form II Transposed bandpass filter. [1]

This structure is an example that works with 4 coefficients of b and a. However we found that to implement our bandpass filter we needed to use five b and a coefficients. The structure works by firstly taking the convolution sum for the last coefficient of b with the input sample and a with the output. This is then used in the next step where it is summed with the convolution of the next set of coefficients and input/outputs. This delay process will continue right up to last coefficients where we will obtain the output. a_0 will have no effect as it is always 1. To better way to understand this is through a visual representation with actual simple numbers.

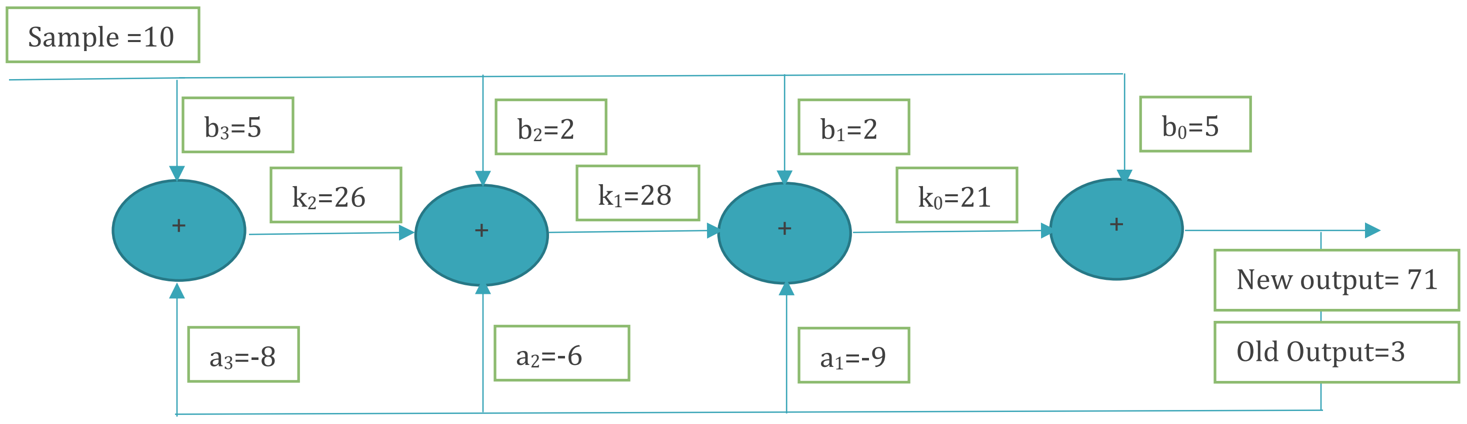


Figure 11: Image shows the logical steps to implement the direct form II Transposed bandpass filter using numbers. Old output is 3 and from this we determine our k values which is turn will be used to find our new output. The new output of 71 will then become the old output in the next iteration and processes repeats. [1]

To implement such a structure we did the following steps. We firstly calculate the initial output of the buffer using the line `output = k[0] + b[0] * read_sample;` It performs and finds the initial output which is later be used for the feedback system. We then enter `for (i = 0; i < order-1; i++)` which cycles elements in arrays from index 0 to order-1 (N-2) and does not include the last element. Within the `for` this loop we perform `k[i] = k[i + 1] + b[i + 1] * read_sample - a[i + 1] * output;` which finds the previous delayed values like in out example above. Finally we deal with the last elements in the arrays which so happens to be where the delays start, the form `k[order-1] = b[order] * read_sample - a[order] * output`; is different hence it cannot be used within the for loop. The output is then returned as a `short` and later goes through a `mono_write_16Bit()` to provide us the filtered waveform.

```
/**********Variables Used ***************/
#define order  (sizeof(a)/sizeof(a[0])-1)
int i; // Used as loop counter
double k[]; //Middle Buffer for directformII;
short output; // holds the output
short read_sample; // holds the input sample


short direct_form_transposed()
{
    output = k[0] + b[0] * read_sample; //Calculates the first element of
                                        //buffer.


    for (i = 0; i < order-1; i++)         //cycle through elements elements 0
                                        // to N-2 aka just before buffer starts.
    {
        // Forumula to find previous delays values.
        k[i] = k[i + 1] + b[i + 1] * read_sample - a[i + 1] * output;
    }

    k[order-1] = b[order] * read_sample - a[order] * output; // calulate the
                                                        //the first value of the buffer.

    return output; // Function returns the filtered output.
 }
```

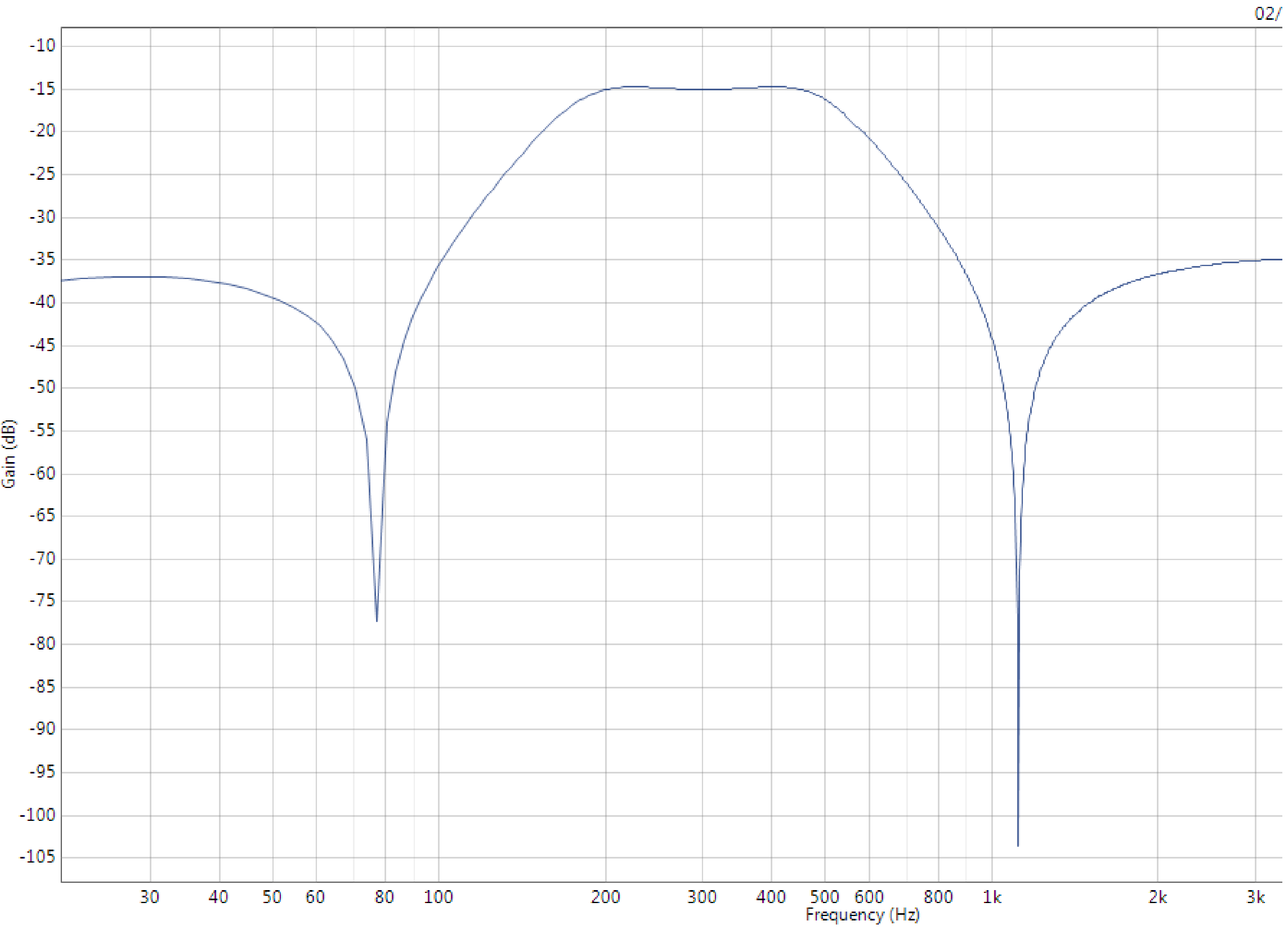


Figure 12: Image shows the filter response using the Spectrum analyser APx 520. We notice that it has a identical shape to that found within MATLAB as expected. As the output is a quarter due to DSk unit [APPENDIX] the bandpass gain starts at -15 dB

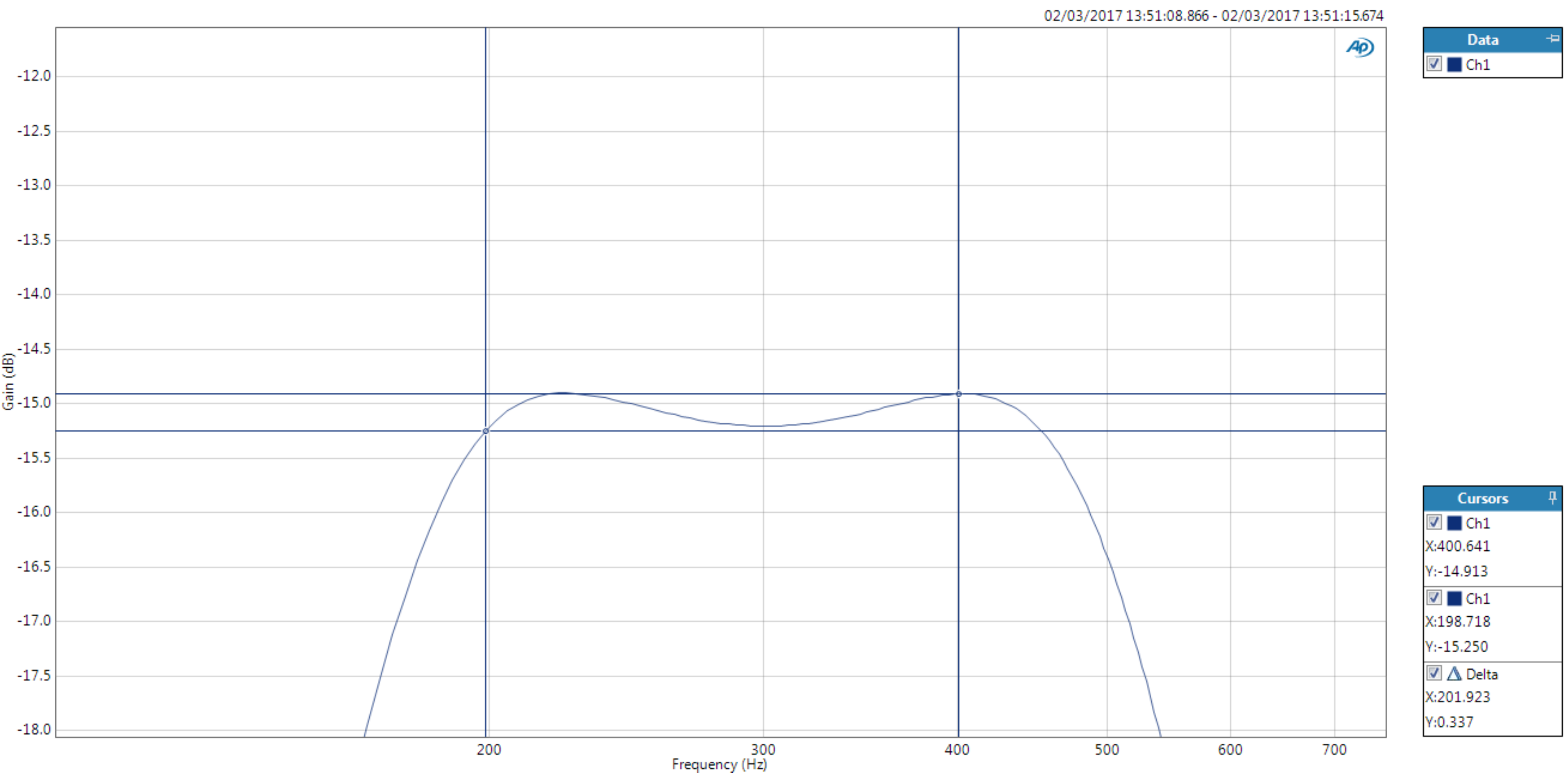


Figure 13: Image shows the filter response using the Spectrum analyser APx 520. We have zoomed in onto the pass band ripple. We notice that ripple is indeed just within 0.3 dB range,within specifications and near identical to result found in MATLAB.

Performance comparison between transposed and non-transposed

Here we compare how much of a performance difference there is for implementing the bandpass filter between transposed and non-transposed. To do this we will be solely looking at number of cycles.

	Non-transposed no opt	Non-transposed opt2	Transposed no opt	Transposed opt2
Order				
2	217	79	133	74
4	379	129	241	110
6	541	157	349	117
8	703	185	457	132
10	918	218	565	141
12	1027	246	673	221

Table 1: Tables shows table taken for different order filters on both transposed and non-transposed and how many cycles it has taken for both non-optimised and optimised level 2

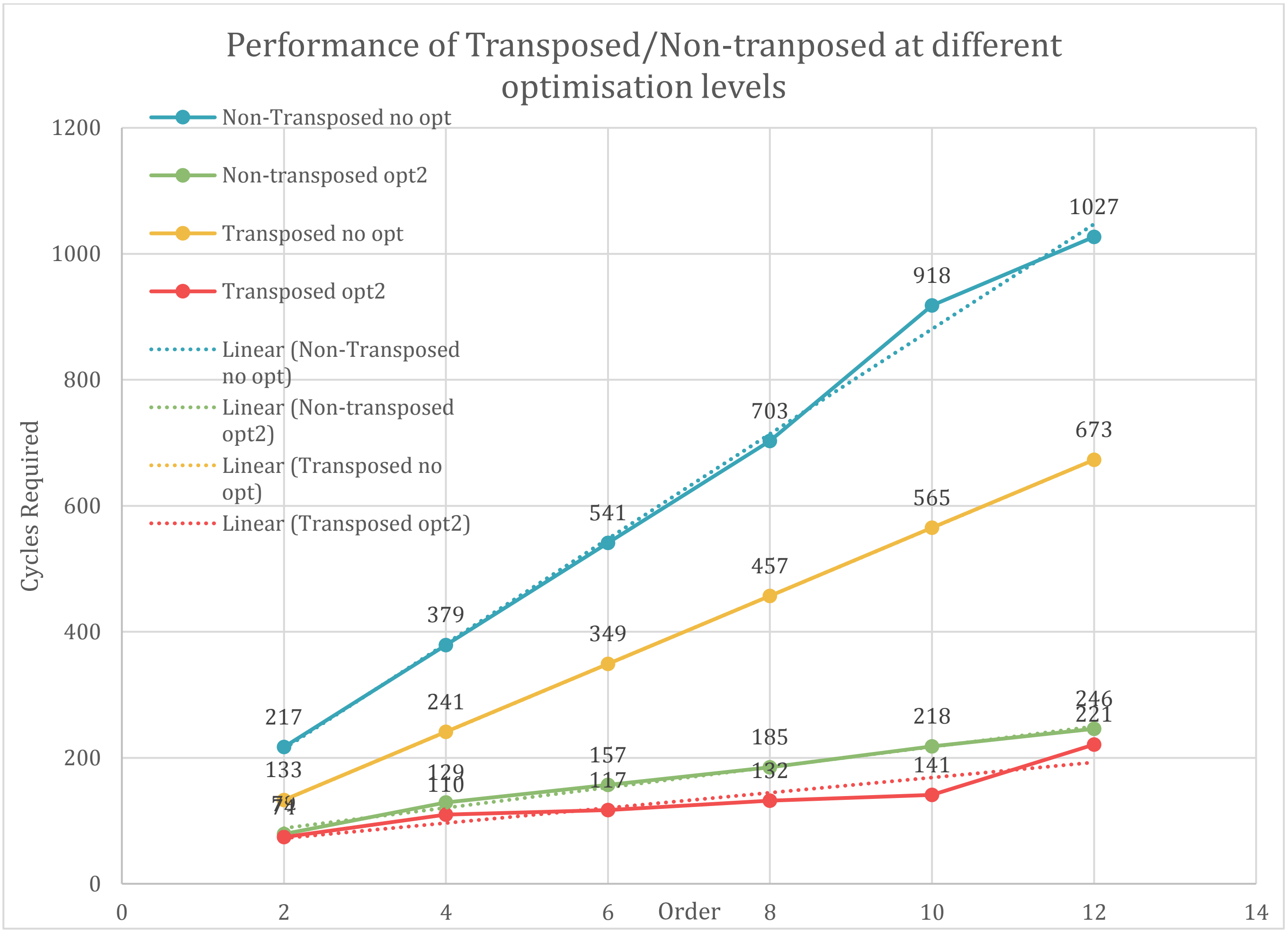
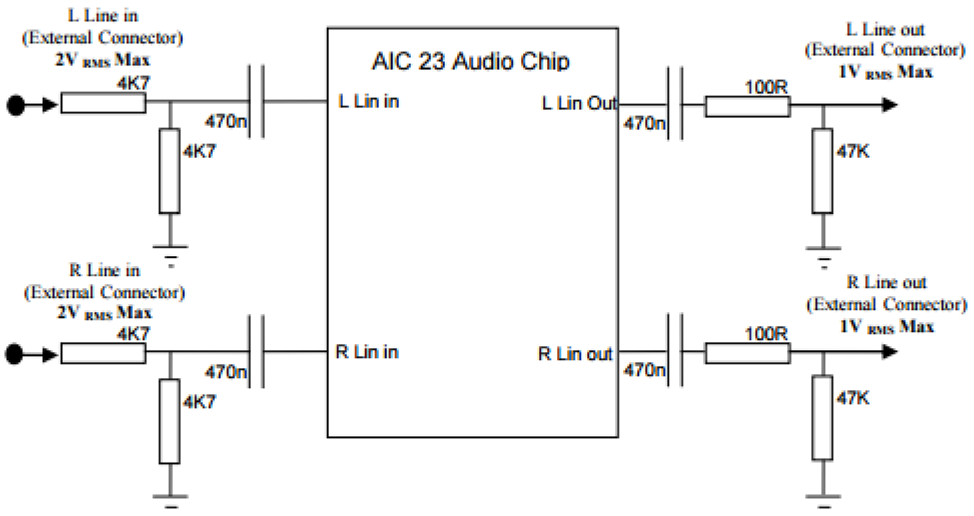


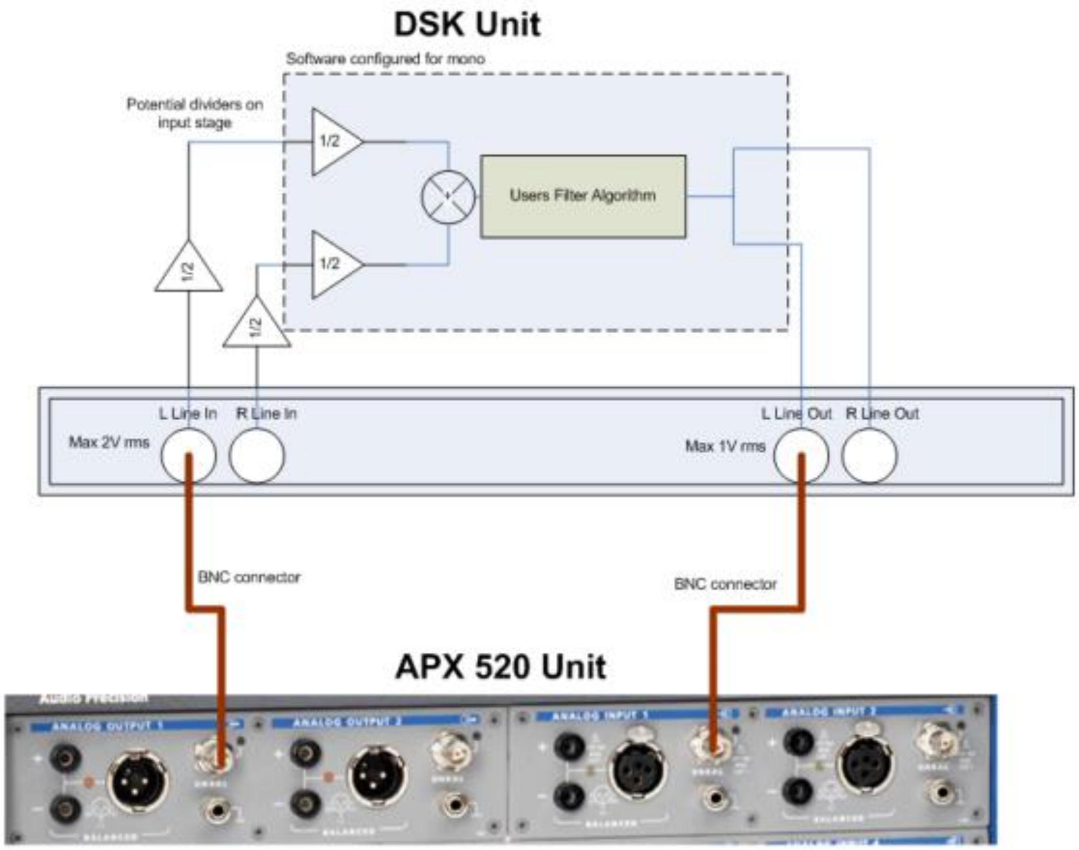
Figure 14: Image shows graph representation of number of cycles taken for transposed and non-transposed. We can clearly see transposed performs better.

Type of optimisation	Line equation (An+B form)
Non-Transposed no opt	81n+55
Non-Transposed opt2	16.7n+45.6
Transposed no opt	54n+25
Transposed opt2	14.7n+44.6

From the graph we can see that the transposed form of direct form II runs faster than the non-transposed version. The primary reason for this is due to non-transposed direct form having more instruction than its counterpart also in the transposed form there were no uses of coping and shifting sample, which resulted in significantly less cycles in no opt. However in opt2 we can see the performance disparity not as large this is due to the order of our filter but if we were to increase the filter order it would be more obvious that the transposed version works significant better than its counterpart.

Appendix


L Line in
(External Connector)
2V RMS Max
4K7
470n
4K7
AIC 23 Audio Chip
L Lin in
L Lin Out
470n
100R
L Line out
(External Connector)
1V RMS Max
47K
R Line in
(External Connector)
2V RMS Max
4K7
470n
4K7
R Lin in
R Lin out
470n
100R
R Line out
(External Connector)
1V RMS Max
47K


DSK Unit
Software configured for mono
Potential dividers on input stage
1/2
1/2
1/2
1/2
Users Filter Algorithm
L Line In
R Line In
Max 2V rms
L Line Out
R Line Out
Max 1V rms
BNC connector
BNC connector
APX 520 Unit

Simple IIR Filter:

```
/********Variables used for Simple IIR Filter***********/
double x[];
short output;
short read_sample;
const double simplea[] = {1,0.88235294117};
const double simpleb[] = {0.05882352941,0.05882352941};


short simple_IIR()
{
      x[1] = x[0]; // Shift previous input sample in the inputbuffer.
      x[0] = read_sample; // Stores input sample in the input buffer [0].

      output = x[0]*simpleb[0] + x[1]*simpleb[1] - output*simplea[1]; //
IIR Convolution Sum.

      return output ; // Returns output back to mono_write so it can be
outputted to DSP.
}
```

Bandpass filter: Direct form (Non-Transposed):

```
/**********Variables Used ***************/
#define order  (sizeof(a)/sizeof(a[0])-1)
int i; // Used as loop counter
double w[]; //Middle Buffer for directformII;
double output; // holds the output
short read_sample; // holds the input sample

short directformII()
{
      w[0] = read_sample; // Places input sample into the 0th element of
the w buffer
      output = 0; // resets the out to 0 at every function call

      for( i = order ; i > 0 ; i--)
      {
            w[0] -= a[i]*w[i]; // performs the left hand side of the branch
            output += b[i]*w[i]; // performs the right hand side of the
branch addition
            w[i] = w[i-1]; // performs the time shift required
      }

      output += w[0]*b[0]; //updates the new output and return this back
the DSP

      return output;

}
```

Bandpass filter: Direct form Transposed):

```
/**********Variables Used ***************/
#define order  (sizeof(a)/sizeof(a[0])-1)
int i; // Used as loop counter
double k[]; //Middle Buffer for directformII;
short output; // holds the output
short read_sample; // holds the input sample


short direct_form_transposed()
{
    output = k[0] + b[0] * read_sample; //Calculates the first element of
                                        //buffer.


    for (i = 0; i < order-1; i++)         //cycle through elements elements 0
                                        // to N-2 aka just before buffer starts.
    {
        // Forumula to find previous delays values.
        k[i] = k[i + 1] + b[i + 1] * read_sample - a[i + 1] * output;
    }

    k[order-1] = b[order] * read_sample - a[order] * output; // calulate the
                                              //the first value of the buffer.

    return output; // Function returns the filtered output.
 }
```

MATLAB Implementation:

```
%% Bandstop Elliptic Filter

Order= 4 % sets the order of filter we would like.
[b,a] = ellip(order/2,0.3,20,[200 450]/4000,'bandpass');
%Matlab function that takes in order of filter, ripple, stopband attenuation,
cut off frequencies and type of filter and returns two sets of co-efficients
later used to implement filter.

freqz(b,a); %Plot and see filter.

bb=['double b[]={']; % create a string starting with 'double b[]={'
aa=['double a[]={'];

for i = 1:(order*2+1)   % For loop used to cycle through all co-efficients.

    bb = [bb  num2str(b(i)) ', ' ]  %Add co-efficients to the string
                                    %followed by a comma.
end

for i = 1:(order*2+1)   % For loop used to cycle through all co-efficients.

    aa = [aa  num2str(a(i)) ', ' ]  %Add co-efficients to the string
                                    %followed by a comma.
end

aa = [aa ' };'];   %end string with curly brackets and semi-colon.
bb = [ bb ' };'];  %end string with curly brackets and semi-colon.
```

References

1. D. Mitcheson, P.P. (2014) *EE3-19 Real Time Digital Signal Processing*. Available at: https://bb.imperial.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_885311_1&course_id=_9509_1 (Accessed: 20 February 2017).