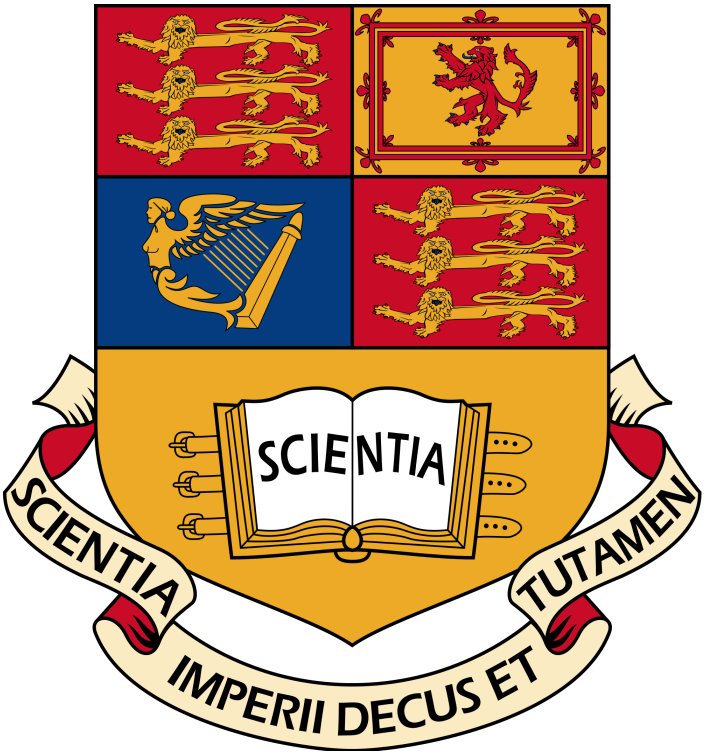
Imperial College London



Department of Electrical and Electronic Engineering

Real Time Digital Signal Processing

Lab 5 - Implementation of IIR Filters

Authors:
Eusebius Ngemera CID:00825003
Prahnav Sharma CID:00820722

Course Lecturer:
Dr. Paul Mitcheson

Declaration: We confirm that this submission is our own work. In it we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.

March 4th, 2016

Contents

1 Introduction

1.1 Infinite Impulse Response Filters

The aim of the 5^{th} laboratory experiment is to design an Infinite-Impulse-Response (IIR) filter using MATLAB, and implement it on the DSP Starter Kit (DSK) system. In particular, we explore different implementations of the IIR filter configuration, and compare and contrast their performance.

The output of an IIR filter is the weighted sum of the current and previous input signals, along with the previous output signals. This is different to Finite Impulse Response (FIR) filters, which do not have feedback from the output signal. The convolution sum for the IIR filter is given as:

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

By taking the z-Transform of this convolution sum, the transfer function of the generic IIR filter can be acquired as:

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

The numerator and denominator polynomials are both polynomials in z, indicating the presence of both poles and zeros in the z domain plot of the IIR filter. This is a key difference between FIR and IIR filters, where IIR filters can have zeros and poles placed anywhere in the z domain, while FIR filters have all poles situated at the origin.

Another distinction is that IIR filters have the potential to be unstable filters. Due to the weighted feedback signals acquired from the output, the IIR filter can enter a positive feedback loop and become unstable. Thus, when designing IIR filters, it is important to consider the location of the poles, in the z domain, and ensure that they are all situated within the unit circle. Common IIR filters include Butterworth, Chebyshev, elliptic and inverse Chebyshev filters. In this lab, we will design and implement elliptic IIR filters.

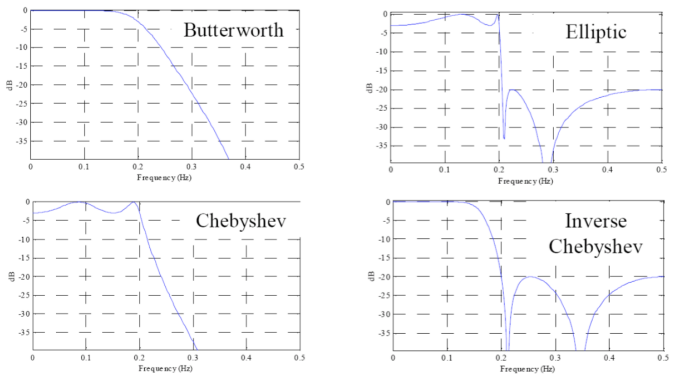


Figure 1: Common IIR Filter Implementations

2 Single Pole Filters

2.1 RC Lowpass Filter

A simple analogue, low pass filter can be achieved by using a capacitor and a resistor, in the following configuration shown below.

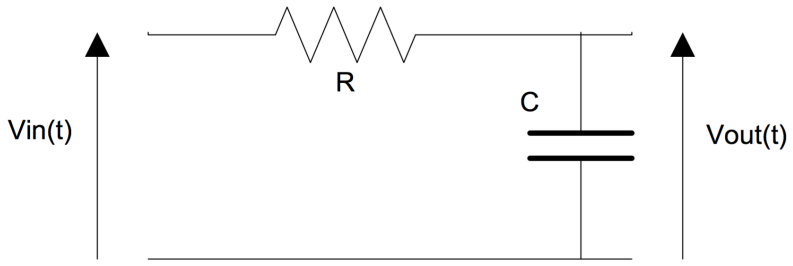


Figure 2: A simple low pass filter, achieved using two passive components

For low frequencies, the impedance of the capacitor is effectively infinite, and all of the input voltage is dropped across the capacitor. For high frequencies, the impedance of the capacitor is effectively zero, providing a short to ground for the input voltage. This low pass nature of this RC filter is evident from its transfer function:

$$\frac{V_{Out}}{V_{In}} = \frac{\frac{1}{j\omega C}}{R + \frac{1}{j\omega C}} = \frac{1}{j\omega CR + 1} \tag{3}$$

To acquire the Laplace Transform of this transfer function, we note that $s = j\omega$, and we substitute variables, yielding:

$$\frac{V_{Out}}{V_{In}} = \frac{\frac{1}{sC}}{R + \frac{1}{sC}} = \frac{1}{sCR + 1} \tag{4}$$

2.2 Tustin Transform

The Laplace transform is used to analyze continuous time systems in the s domain. To be able to analyze this signal in the discrete z domain, we need to use a bilinear transformation. To achieve this, we note that the Laplace Transform essentially introduces a time delay of T_s, as every input waveform is multiplied by the complex exponential e^{-sT_s}. Therefore, we make the substitution $z = e^{sT_s}$

$$z = e^{sT_s} \quad \Rightarrow \quad ln(z) = sT_s \quad \Rightarrow \quad s = \frac{1}{T_s} ln(z) \tag{5}$$

The series expansion of the logarithm yields the Tustin Bilinear Transform:

$$s = \frac{2}{T_s} \frac{z-1}{z+1} \tag{6}$$

The following derivation is a mapping from the Laplace domain to the z domain, using the Tustin Transform. We first make the substitution $s = \frac{2}{T_s}\frac{z-1}{z+1}$

$$\frac{V_{Out}}{V_{In}} = \frac{1}{sCR+1} = \frac{1}{\frac{2}{T_s}(\frac{z-1}{z+1})CR+1} \tag{7}$$

Multiplying the numerator and denominator by $(z+1)$

$$= \frac{z+1}{z+1+\frac{2RC}{T_s}(z-1)} \tag{8}$$

Multiplying the numerator and denominator by z^{-1}

$$= \frac{1+z^{-1}}{1+z^{-1}+\frac{2RC}{T_s}(1-z^{-1})} \tag{9}$$

Rearranging terms

$$= \frac{1+z^{-1}}{1+\frac{2RC}{T_s}+z^{-1}(1-\frac{2RC}{T_s})} \tag{10}$$

To shape the equation to match the IIR transfer function form,

$$H(z) = \frac{b_0+b_1Z^{-1}+b_2Z^{-2}+...+b_MZ^{-M}}{1+a_1Z^{-1}+a_2Z^{-2}+...+a_NZ^{-N}} \tag{11}$$

where we want the denominator polynomial to commence with 1, we multiply numerator and denominator with $\frac{T_s}{T_s+2RC}$

$$= \frac{\frac{T_s}{T_s+2RC}+\frac{T_s}{T_s+2RC}z^{-1}}{1+\frac{T_s-2RC}{T_s+2RC}z^{-1}} \tag{12}$$

Comparing this last equation with the IIR transfer function form, we can match terms to acquire

$$b_0 = \frac{T_s}{T_s+2RC} \quad \Rightarrow \quad b_1 = \frac{T_s}{T_s+2RC} \quad \Rightarrow \quad a_0 = 1 \quad \Rightarrow \quad a_1 = \frac{T_s-2RC}{T_s+2RC} \tag{13}$$

For our system, the sampling frequency is 8000 Hz, therefore $T_s = \frac{1}{8000}$. The resistor value is 1000Ω and the capacitor value is $1\mu F$. Substituting these values into the above equation, we acquire our low pass filter coefficients:

$$b_0 = \frac{1}{17} \quad \Rightarrow \quad b_1 = \frac{1}{17} \quad \Rightarrow \quad a_0 = 1 \quad \Rightarrow \quad a_1 = -\frac{15}{17} \tag{14}$$

We know that $z = e^{sT_s} = e^{j\omega T_s}$. To determine the D.C Gain of this system, we must evaluate the gain when $\omega = 0 \therefore s = 0, z = 1$. This yields a D.C Gain of 1.

3 Implementation of Low Pass Filter on DSK

3.1 MATLAB Design

After acquiring the transfer function for the RC Low-pass filter, we simulated the ideal analogue filter in MATLAB. The following figure shows the Bode plot:

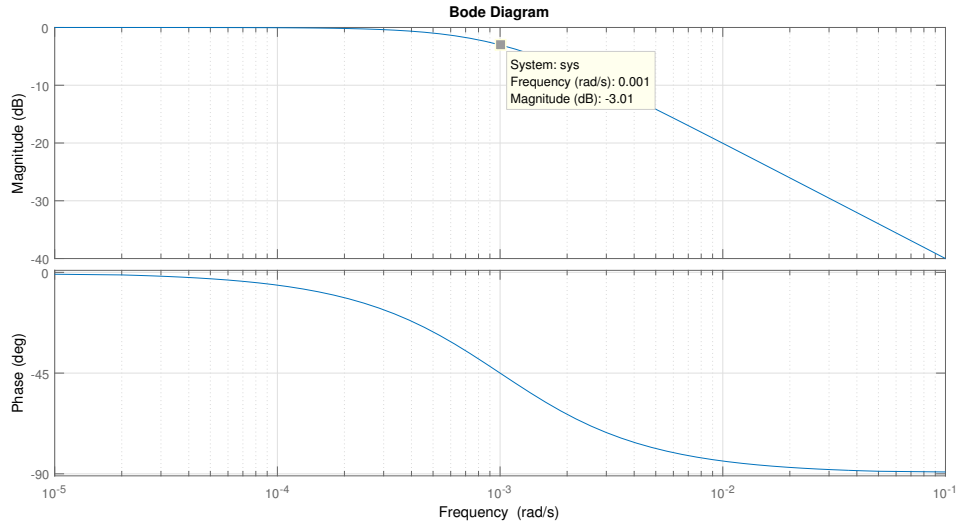


Figure 3: Bode plot for RC Low-pass filter

We can also confirm that this filter is indeed a single pole filter, from the pole zero plot shown below.

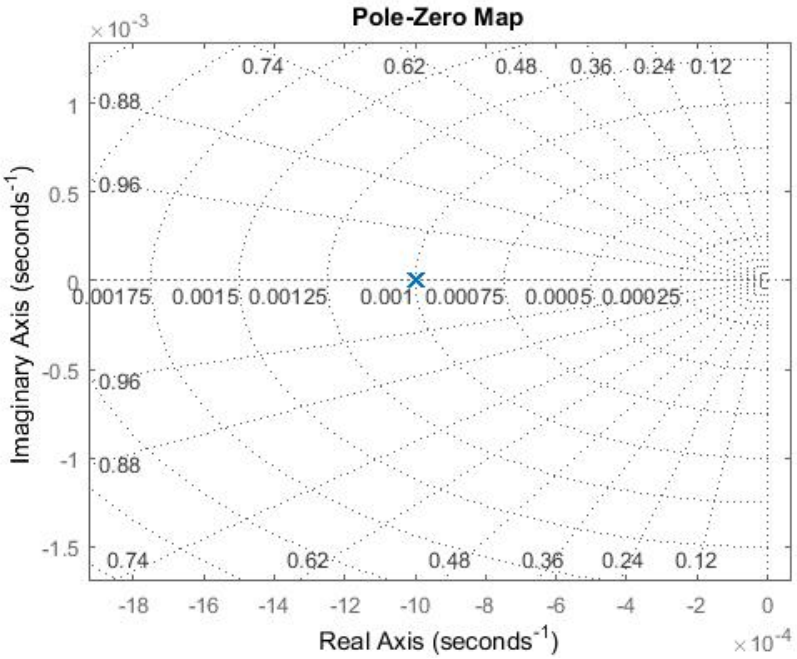


Figure 4: Pole-Zero plot for RC Low-pass filter

The corner frequency of this RC filter is given by:

$$F_{corner} = \frac{1}{2\pi RC} = 159.15Hz \tag{15}$$

This is the theoretical frequency where the gain of the filter drops by $3dB$. This is indeed the case for the MATLAB bode plot above.

Next, we implemented the z-Transformed digital IIR filter in MATLAB, using the following MATLAB Code:

```
%% Eusebius & Prahnav Lab 5 RTDSP - RC Low Pass Filter
%--------------------------------------------------

Fs = 8000;  % Sampling Frequency
Ts = 1/Fs;  % Sampling Period

% Component Specifications
R = 1e3;
C = 1e-6;

% Denominator Polynomial
denom = Ts + 2*R*C;

% IIR Filter Coefficients
a = [1, (Ts-2*R*C)/denom]
b = [Ts/denom, Ts/denom]

% Plotting the z transform IIR function
z = tf([Ts, Ts], [(Ts + 2*R*C), (Ts - 2*R*C)] , Ts);
freqz(b, a, 1024, 'half', Fs);
```

We generated the Bode plot for this system and observed the following result:

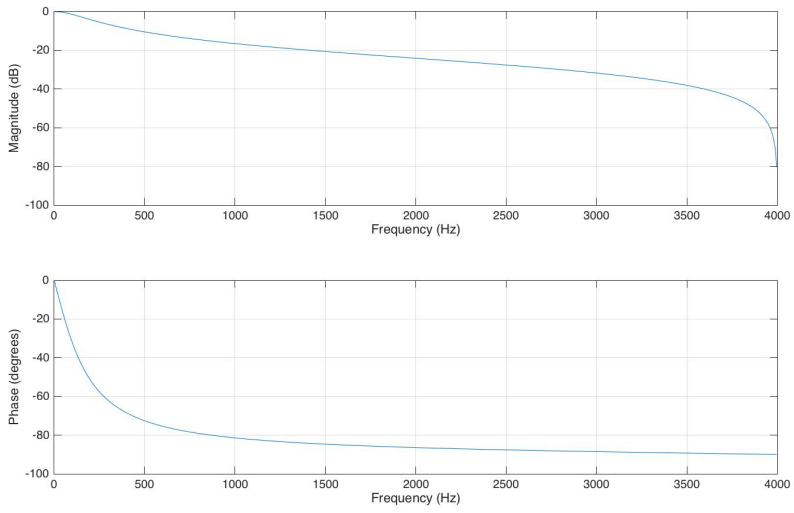


Figure 5: Theoretical Digital Low Pass Filter Response (Linear Frequency axis)

3.2 C Implementation

Figure 6 below, shows the most straightforward implementation of an IIR filter - Direct Form I. There are two delay lines, one for each coefficient stream, thus requiring two delay buffers.

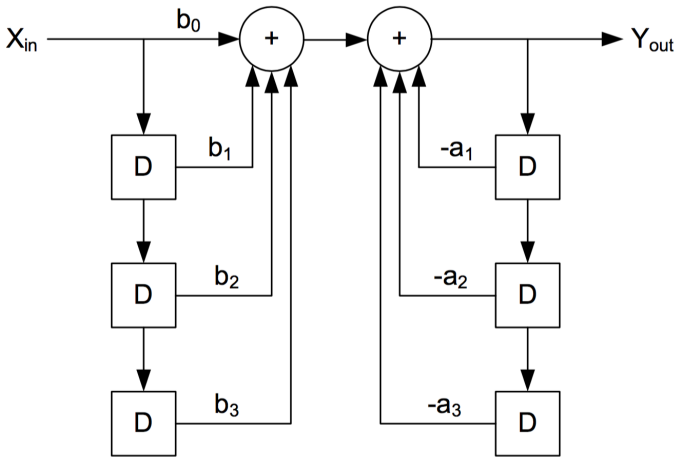


Figure 6: Direct Form I Implementation of IIR Filter

The C implementation below shows 4 for-loops. There are 2 for-loops, to shift `x` and `y` buffers, as well as 2 more for convolution with their respective filter coefficient arrays.

```
// IIR assuming a and b are of different lengths, a[M] and b[N]
double simple_IIR(short new_sample)
{
    int i;                    // for-loop iteration index
    double sum = 0.0;         // data type matches IIR coefficients

    // shifting x, the past inputs
    for (i = N-1; i > 0; i--)
    {
        // move data one index up, discarding last (oldest) input
        x[i] = x[i-1];
    }
    // prepend new sample, to be the first element
    x[0] = new_sample;

    // Convolution between b coefficients and past inputs
    for (i = 0; i < N; i++)
    {
        sum += b[i] * x[i];
    }

    // Convolution between a coefficients and past outputs
    // a[0] = 1, convolution starts from a[1]
    for (i = 1; i < M; i++)
    {
        sum -= a[i] * y[i];
    }

```

```
29      // shifting y, the past outputs
30      for (i = M-1; i > 0; i--)
31      {
32          // move data one index up, discarding last (oldest) output
33          y[i] = y[i-1];
34      }
35
36      // prepend final output value to y buffer
37      // y index matches a coefficient array
38      y[1] = sum;
39
40      return sum;
41  }
```

The code is written to model the Direct Form I figure shown above. We did not look to optimise the convolution computation of this filter, as this was the exercise carried out in the previous lab. The focus of this lab is to compare and contrast the different IIR filter implementations.

3.3 Scope Traces

We proceeded by testing our filter with a square wave input of varying frequencies. The DSK board has an input high pass filter, in addition to the low-pass RC filter with a cutoff frequency of approximately 160Hz, thus, we expect our passband to be approximately from 15Hz to 150Hz.

The following scope traces serve as evidence, that this indeed is the case:

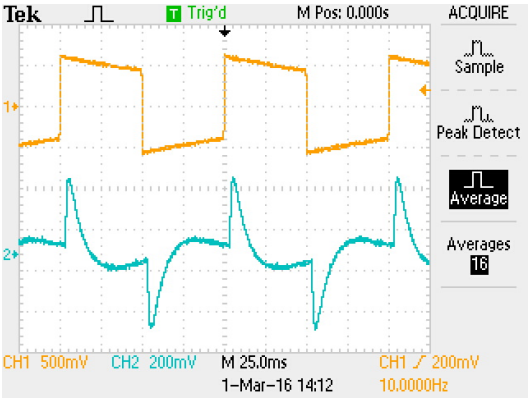


Figure 7: IIR Filter Response (10Hz)

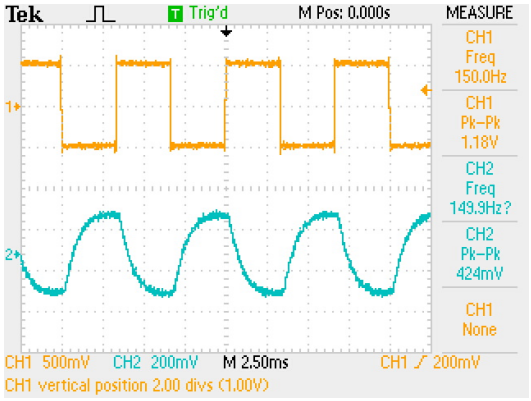


Figure 8: IIR Filter Response (150Hz)

In the above left figure, the low frequency square wave input, and the respective output waveform, are affected by the input high-pass filter. At 150Hz, we observe a more familiar trace of a capacitor charge and discharge cycle.

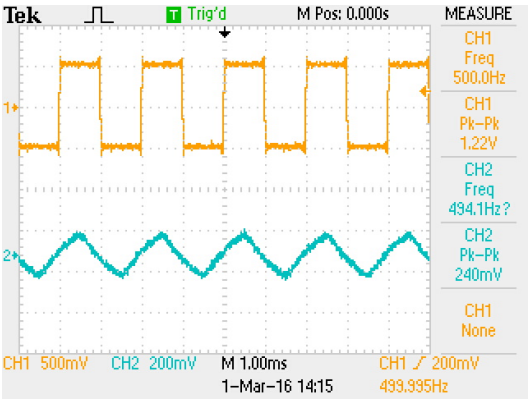


Figure 9: IIR RC Filter Response (500Hz)

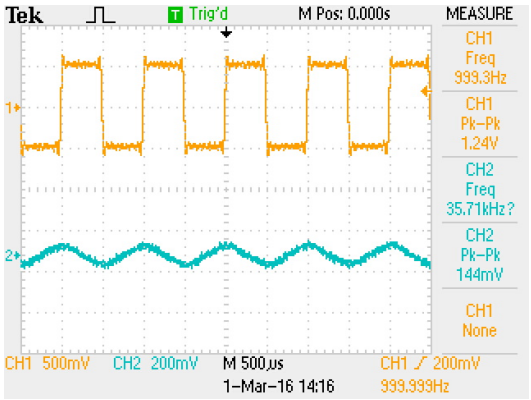


Figure 10: IIR RC Filter Response (1000Hz)

As we increase the frequency of the square wave past the cutoff frequency, we notice that 1) the output waveform tends to increasingly resemble a triangle wave and 2) the output amplitude decreases in gain. These traces are as expected.

3.4 Filter Time Constant

When a potential difference is applied to a capacitor, the change of voltage is not instantaneous, rather it requires some time and follows an exponential rise to reach its theoretical value. This steady state is reached in approximately 5 time constants of the circuit.

The time constant of the RC circuit, is the time required to charge the capacitor to approximately 63% of its final steady-state value, from the moment the potential difference is applied to the capacitor. When $t = \tau$, the time constant, a rise of 63% is reached.

$$1 - e^{-(\frac{t}{\tau})} = 1 - e^{-1} = 0.632 \tag{16}$$

Equivalently, the time constant can also be determined by the falling trace, by noting the time taken to discharge to 37% of the steady-state value.

$$e^{-1} = 0.368 \tag{17}$$

The time constant (τ), of our RC filter, is determined by the product of R and C :

$$\tau = RC = 10^3 \cdot 10^{-6} = 10^{-3} = 1ms \tag{18}$$

The following two scope traces were used to determine the time constant of this RC low pass filter. We used the aid of cursors, to determine the 63% and 37% mark of the trace.

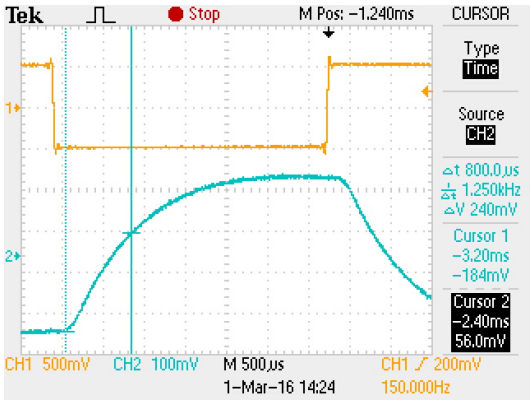


Figure 11: Time Constant determined from Rise Time at (150Hz)

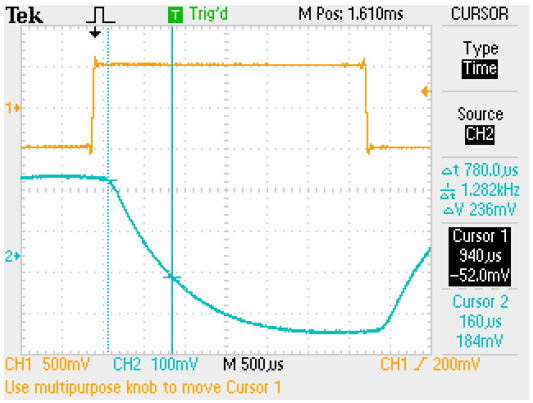


Figure 12: Time Constant determined from Fall Time at (150Hz)

For the rise time approach, we acquired a time constant of $0.8ms$, while for the fall time approach, we acquired a time constant of $0.78ms$. Respectively, these readings yield an error of 20% and 22%. When this acquired time constant is translated to the corner frequency using the following formula:

$$F_{corner} = \frac{1}{2\pi\tau} = \frac{1}{2\pi(0.8 \cdot 10^{-3})} = 198.94 \tag{19}$$

we find that the corner frequency is 198.94 Hz, 20% greater than the expected analogue frequency of 159.15 Hz. The cause for this discrepancy is discussed in the following section.

3.5 Comparison of Analogue Filter with Digital Implementation

We connected our DSK board with the APX500 Spectrum Analyzer, to verify our filter response. Below is the graph we acquired for the magnitude response:

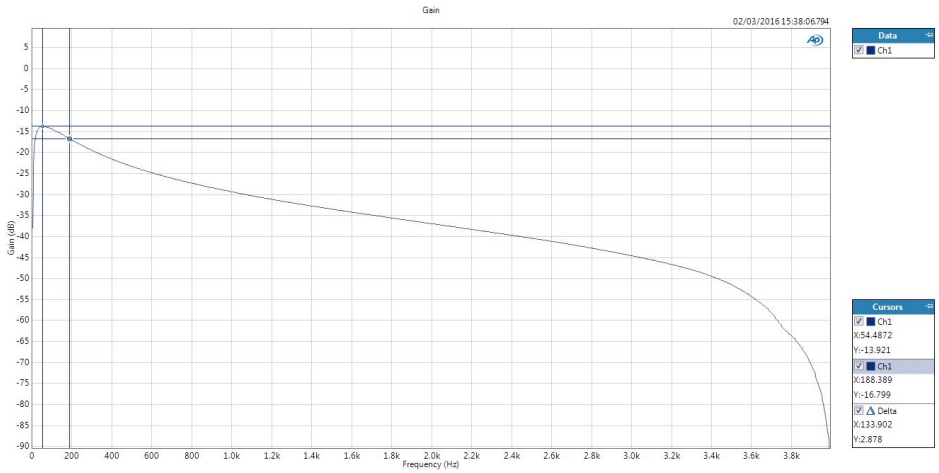


Figure 13: Overall Digital Low Pass Filter Magnitude Response on Spectrum Analyzer

From figure 13 above, we notice the effects of the DSK, most noticeably its high pass filter. At frequencies lower than the peak frequency of 55 Hz, the gain attenuates instead of being constant. Our previously-thought corner frequency of 199 Hz is replaced by 188 Hz from the frequency response above. This shows that using the RC time constant is not enough to calculate corner frequency, especially with intrinsic imperfections.

To observe the actual response of our digital filter, without effects by the DSK, we obtain the DSK's frequency response. This is done by writing the input directly to output, without any intermediate processing. The 2 plots below show the magnitude and phase response, illustrating the DSK's high pass nature.

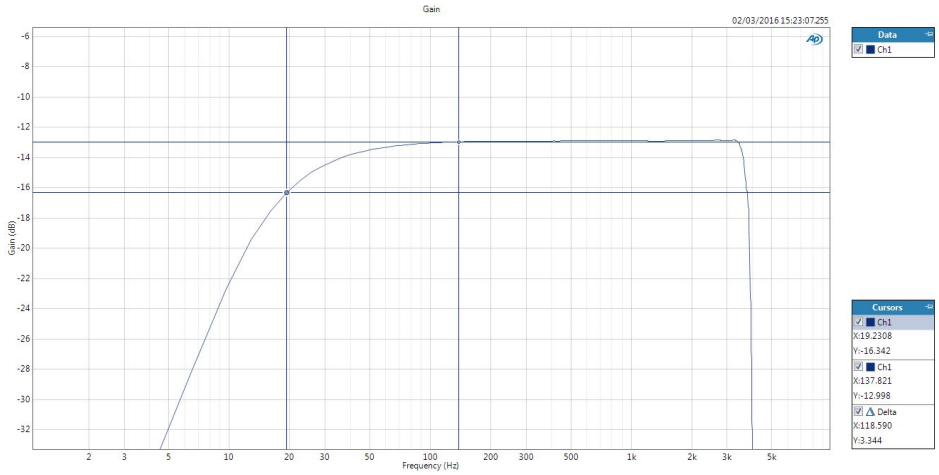


Figure 14: The DSK's Magnitude Response on Spectrum Analyzer

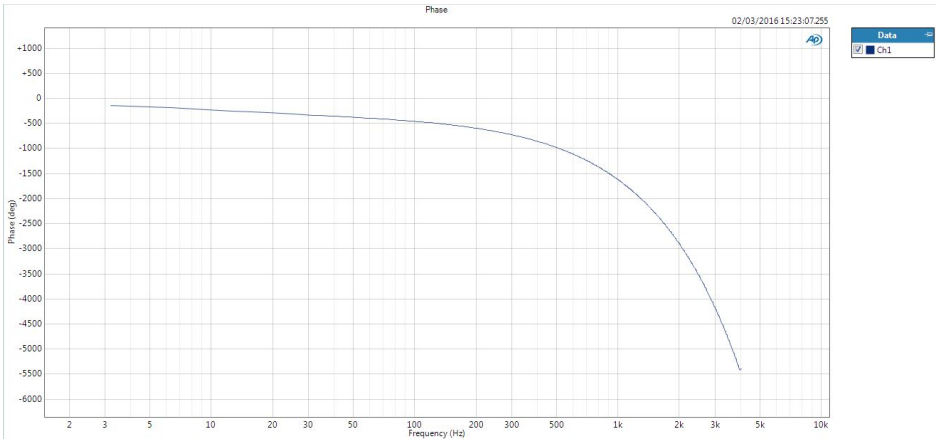


Figure 15: The DSK's Phase Response on Spectrum Analyzer

Given the above intrinsic response, we were able to obtain our filter's real response by subtraction. This produced the graph shown below in figure 16.

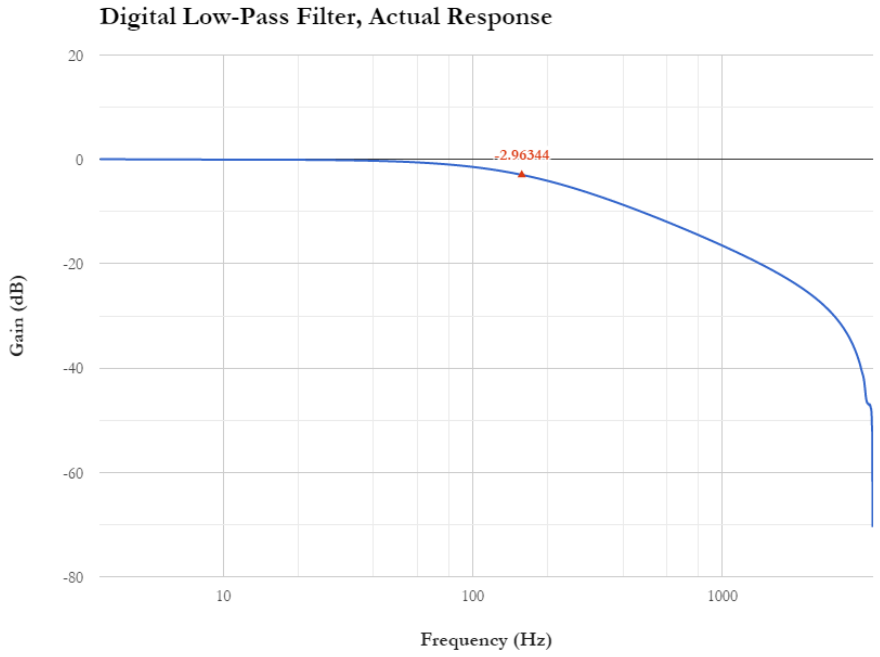


Figure 16: Processed Digital Low Pass Filter Magnitude Response

The labelled data point above in red, shows the data point at 157 Hz with gain -2.96 dB. The next data point is (160 Hz, -3.05 dB) which shows the filter corner frequency is in range]157, 160[Hz. A linear interpolation in this range gives a corner frequency of **158.4 Hz**. This is much closer to the theoretical expectation (159.2 Hz), compared to the 188 Hz seen previously before taking into account the DSK's intrinsic high-pass response. This result, when translated to time constant, yields a very accurate time constant:

$$\tau = \frac{1}{2\pi(158.4Hz)} = 1.0047ms \tag{20}$$

We superimposed the processed digital response, to compare its performance to its theoretical (digital) counterpart from MATLAB.

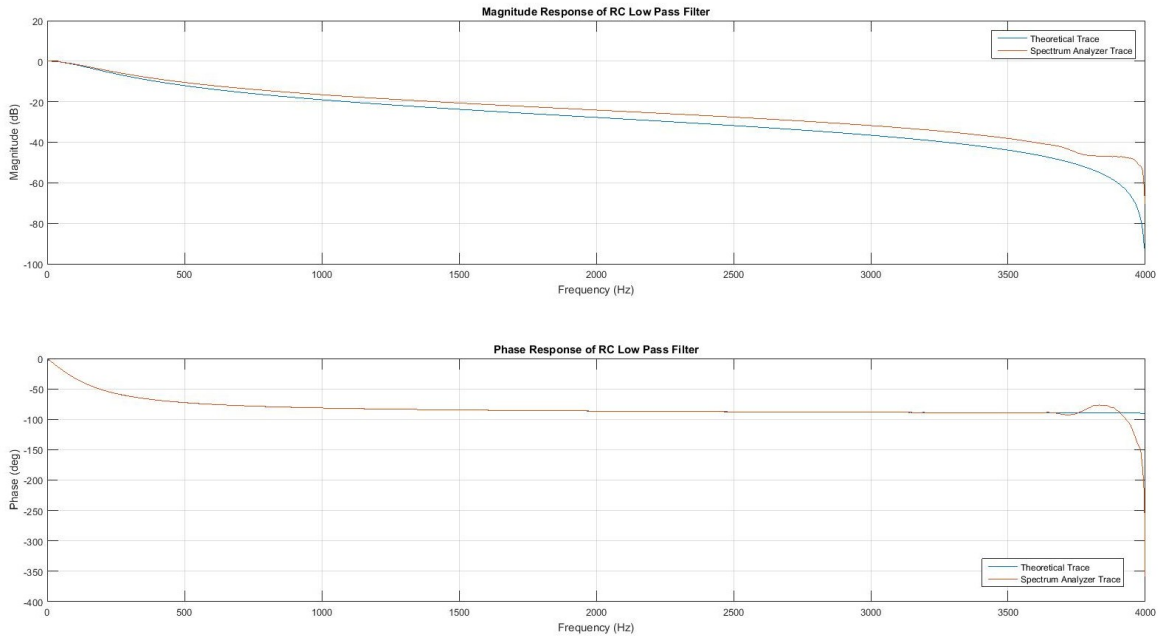


Figure 17: Measurement of Digital Response in comparison to MATLAB Response

We note that the shapes of the responses are very much as expected. However, the magnitude error between the two plots grows with frequency. For a large frequency range, the error is minimal, at less than 10 dB error. We believe this to be acceptable as the gain value is well below -20 dB for all frequencies above 1500 Hz. The phase has non-existent error until frequencies above 3600 Hz where the gain is very small and the Spectrum Analyser is more vulnerable to noise.

Finally, we also superimposed the implemented digital low-pass response, with the theoretical analogue RC filter response, and acquired the following plot:

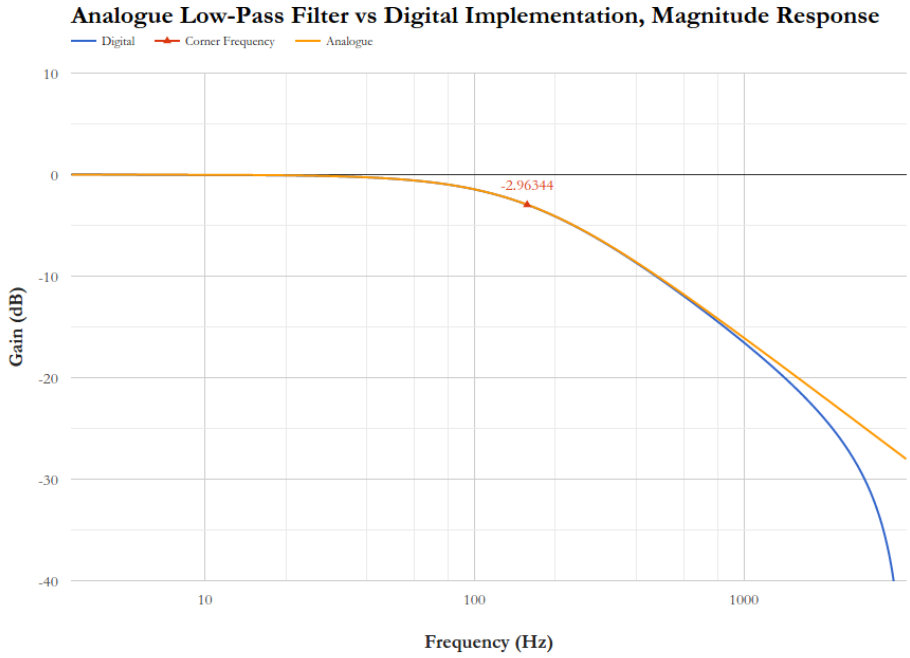


Figure 18: Comparison of Digital Implementation with Analogue Response

As seen above, our digital implementation closely matches the analogue response, and both traces have the same corner frequency, as expected. We believe the digital implementation has surpassed the original analogue RC filter's performance because it has greater attenuation at higher frequencies. This effect is due to anti-aliasing and is very desirable in a low-pass filter.

4 Direct Form II - Bandpass Filter Design

4.1 Design Specifications

The next task is to implement a bandpass IIR filter, using the Direct-Form II Implementation. This bandpass filter has to be an elliptic IIR filter, with the filter specifications noted in the following table:

Specification	**Criteria**
IIR Filter Order	4
Passband Gain	180Hz - 450Hz
Passband Ripple	0.4 dB
Stopband Attenuation	23 dB

We note that in these specifications, the transition bands, from stopband to passband and passband to stopband, have not been specified. Therefore, the last criteria, regarding stopband attenuation, will not be associated with a set frequency.

4.2 MATLAB Implementation

The `ellip()` MATLAB function is used to acquire the IIR filter coefficients. Although our design will work for any number of coefficients, we assume here, that the number of a and b coefficients are equal. The number of each coefficients a, b, are one greater than the IIR filter order.

$$\texttt{[b, a] = ellip(}\frac{order}{2}\texttt{, } R_p\texttt{, } R_s\texttt{, } W_p\texttt{);} \tag{21}$$

Arguments	**Significance**
`a`	IIR Feedback Coefficients
`b`	IIR Feedforward Coefficients
$(\frac{order}{2})$	IIR Filter Order Specification
R_p	Passband Ripple Tolerance
R_s	Stopband Ripple Tolerance
W_p	Normalized Frequency Specification

For our design specifications, the input arguments to the `ellip()` function are:

$$\texttt{order} = 4 \tag{22}$$

$$R_p = 0.4 \tag{23}$$

$$R_s = 23 \tag{24}$$

$$W_p = \frac{1}{4000}[180 \quad 450] \tag{25}$$

Below is the MATLAB code used to implement this filter:

```
%% IIR Passband Filter Design - Order 4

order = 4;
wp = [180 450];      % Filter Passband
Wp = (1/4000)*wp;    % Normalized Frequency

Rs = 23;             % Stopband Ripple
Rp = 0.4;            % Passband Ripple
Fs = 8000;           % Sampling Frequency

[b,a] = ellip(order/2, Rp ,Rs, Wp);    % Elliptical IIR Filter

% Plotting Filter response and z-plane
figure;
freqz(b,a);
figure;
zplane(b,a);
```

The following plot is the MATLAB filter frequency response, showing the bandpass nature of our filter.

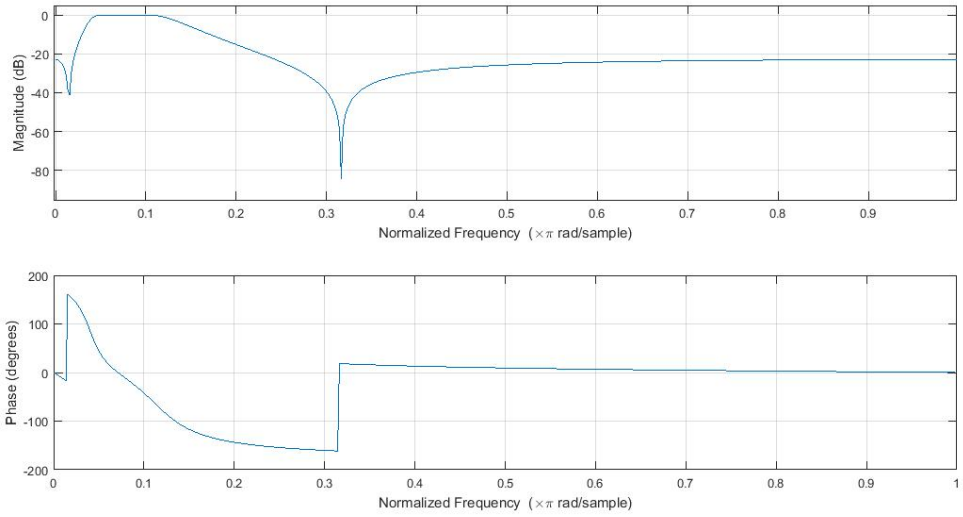


Figure 19: 4th Order Elliptic IIR Filter Designed in MATLAB

The first transition band, from stopband to passband, is fairly narrow, however, the second transition band has a large bandwidth. Our filter does reach -23 db, after the 450Hz point, however because the transition band was not specified in the design specifications, the stopband frequency is not set.

The bandpass nature of our IIR filter can further be confirmed from the pole-zero plot for our filter.

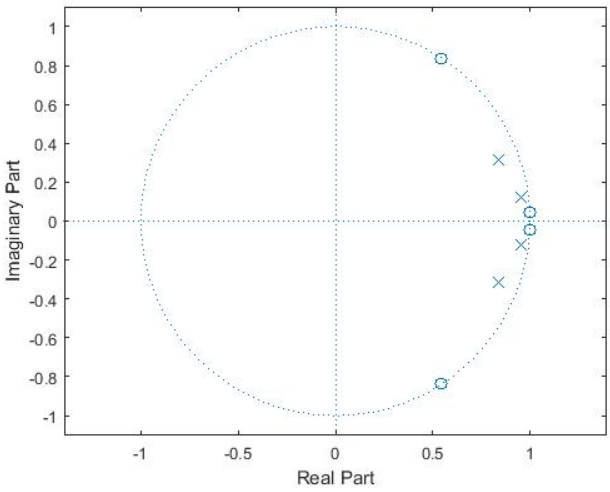


Figure 20: Pole-Zero Plot for 4th Order Elliptic IIR Filter

4.3 C Implementation

Direct Form II is an improved implementation to the previous configuration. The two delay lines have been replaced with a single delay line, following a configuration change.

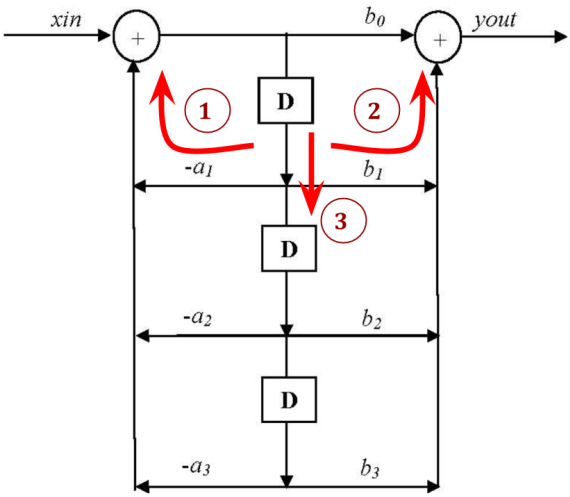


Figure 21: Direct Form II Implementation of IIR Filter

In the C code below, we use array `z` to implement the single delay line. Due to the circularness of the hardware block diagram above (figure 21), we had to break down the process into procedural software. The red arrows illustrate the sequence of steps. Multiply-accumulates (1 and 2) are performed before delays (3). When (2) is done, `yout` holds the final value to be written.

```
// Direct-Form 2 IIR filtering
double DF2_IIR(short new_input)
{
    int i;
    double output = 0.0;

    // z[0] is the top, intermediate node
    z[0] = new_input;

    // (accumulated) convolution of the left branch
    for (i = 1; i < N; i++)
    {
        z[0] -= a[i] * z[i];
    }

    output = b[0] * z[0];

    // (accumulated) convolution of the right branch
    for (i = 1; i < N; i++)
    {
        output += b[i] * z[i];
    }

    // shifting z down
    for (i = N-1; i > 0; i--)
    {
        z[i] = z[i-1];
    }

    return output;
}
```

4.4 Results Verification

Once our code for the 4th order IIR was verified to be working, we made use of the compiler optimisation level `o2`, to compare performances. We obtained a faster implementation with `o2` than with `None`, as expected.

Optimization	Clock Cycles
`None`	501
`o2`	213

In fact, for all IIR order filters greater than 4, the Direct Form II Implementation, with `o2` compiler optimisation level, is greater than `None`. (See last section for more analysis)

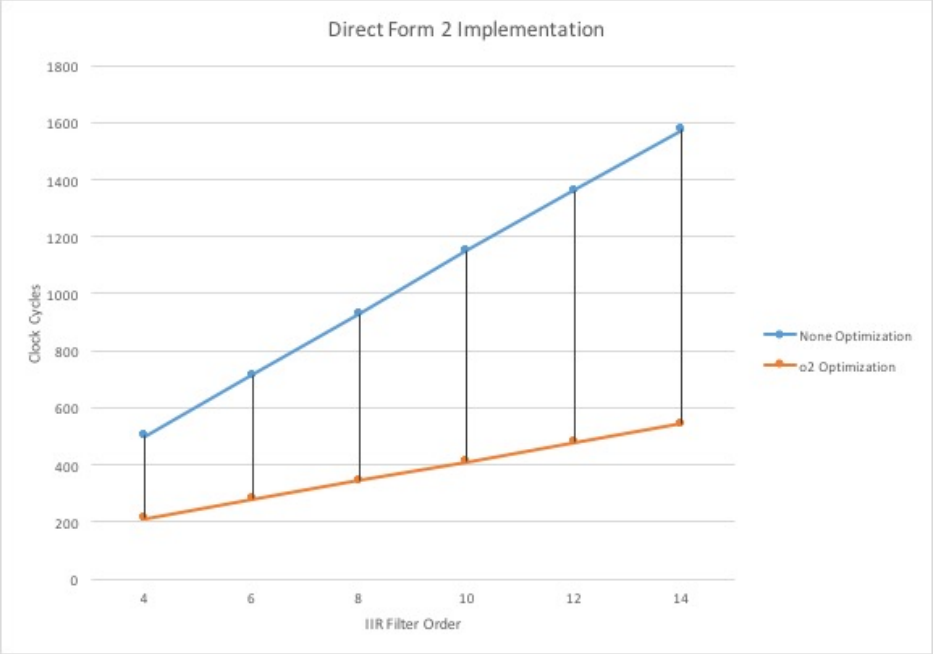


Figure 22: Clock cycle against IIR filter order for different optimisation levels

Next, we connected our DSK board to the Spectrum Analyzer, to verify the filter's frequency response. The magnitude and phase plots resemble the theoretical MATLAB plots obtained previously. The following plots serve as evidence that the design specifications have been satisfied:

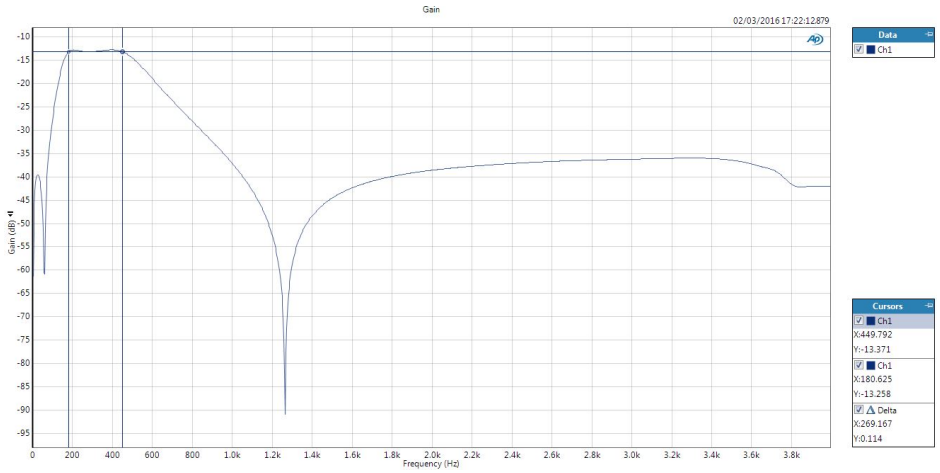


Figure 23: Direct Form II Overall Response on Spectrum Analyzer

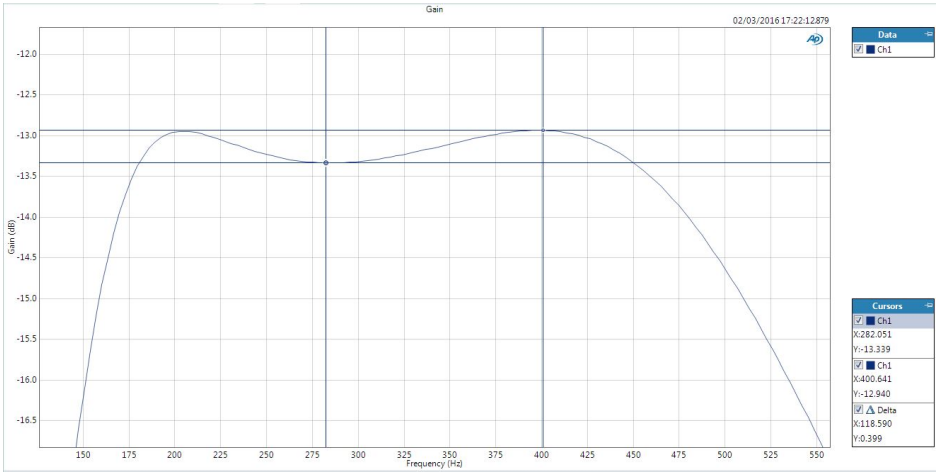


Figure 24: Direct Form II Passband Ripple $\leq 0.4dB$

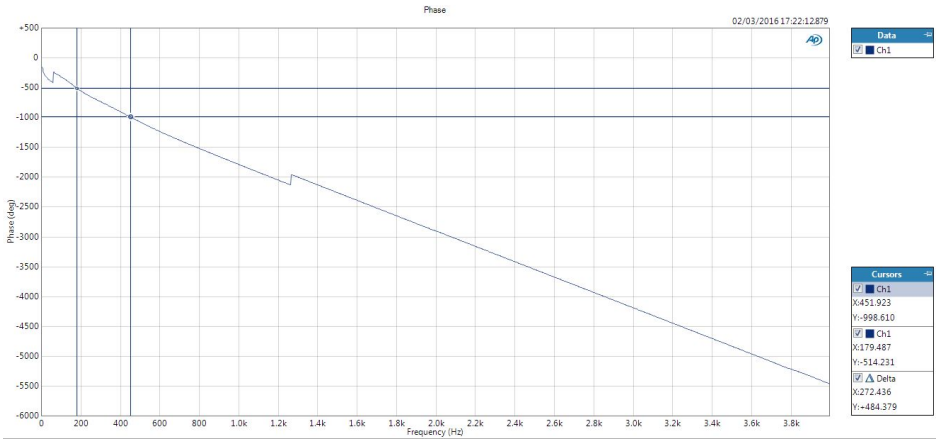


Figure 25: Direct Form II Phase Response on Spectrum Analyzer

This phase plot is the combined response of our IIR implementation, along with the DSK board response.

Lastly, we compare our IIR implementation result with the theoretical MATLAB response:

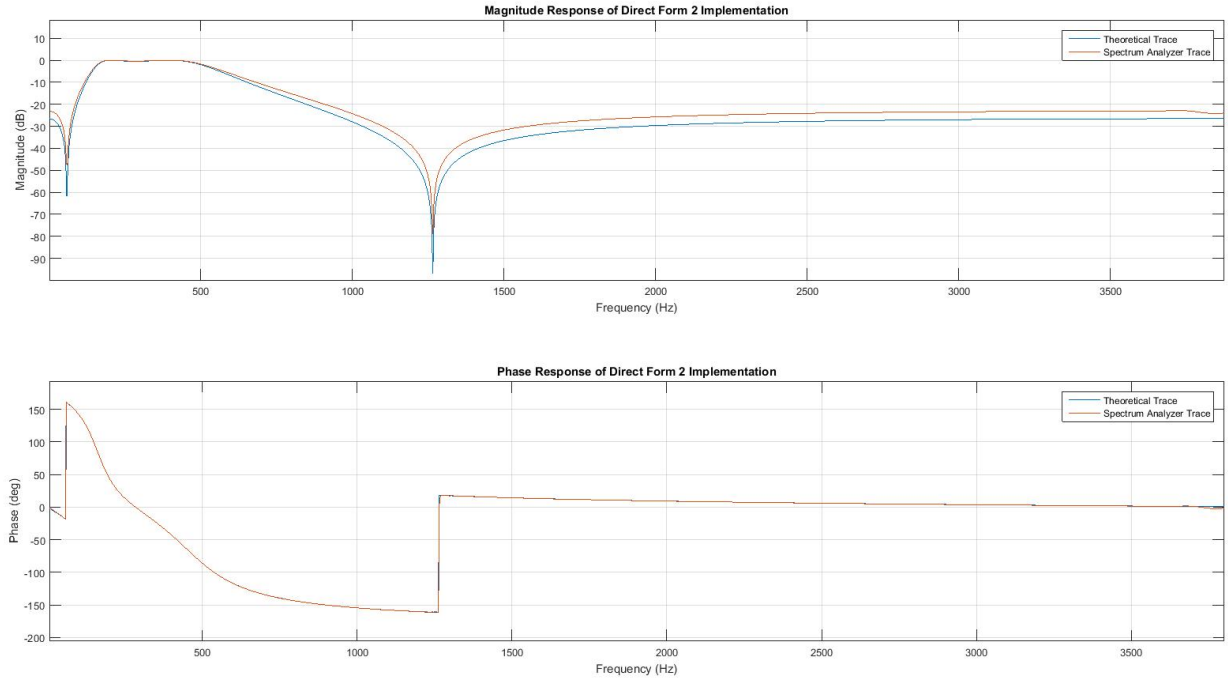


Figure 26: IIR Implementation compared with Theoretical MATLAB response

We can see that the magnitude and phase response closely resemble the expected result. Focusing particularly on the passband frequencies, the magnitude and phase are virtually indistinguishable. The phase is approximately linear in the passband. The implemented IIR also satisfies the -23 db attenuation specification.

5 Direct Form II Transposed - Bandpass Filter Design

5.1 C Implementation

The block diagram below, in figure 27, shows the Direct-Form II transposed structure. This has been transformed by *flow graph reversal* and has exactly the same function as before. The resulting structure does summation *in* the delay line. This means each delay point is an accumulated convolution.

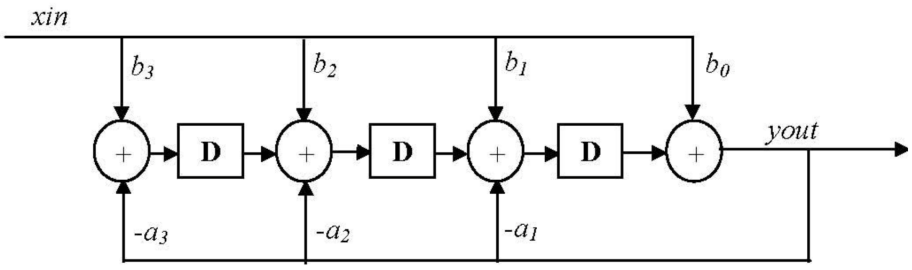


Figure 27: Direct Form II Transposed Implementation of IIR Filter

When implementing the above structure in C, we notice only 1 for-loop is needed. The C code below shows we iterate from right to left along the delay line, and shift the summation results.

```
1  // Direct-Form 2 transposed IIR filtering
2  double DF2T_IIR(short new_input)
3  {
4      int i;
5
6      double output = (new_input * b[0]) + z[0];
7
8      // iterate from right to left
9      // last summation does not have z[N-1], and is an edge case
10     for (i = 1; i < N-1; i++)
11     {
12         z[i-1] = z[i] + (new_input * b[i]) - (output * a[i]);
13     }
14
15     z[N-2] = (new_input * b[N-1]) - (output * a[N-1]);
16
17     return output;
18 }
```

5.2 Results Verification

We again made use of the compiler optimization level `o2`, to compare performances and obtained a faster implementation with `o2` than with `None`, as expected.

Optimization	Clock Cycles
`None`	236
`o2`	129

Carrying out the same analysis as before, we obtained a comparison graph, highlighting how the `o2` implementation scales better than with `None`. We combine and compare our results in section 6.1.

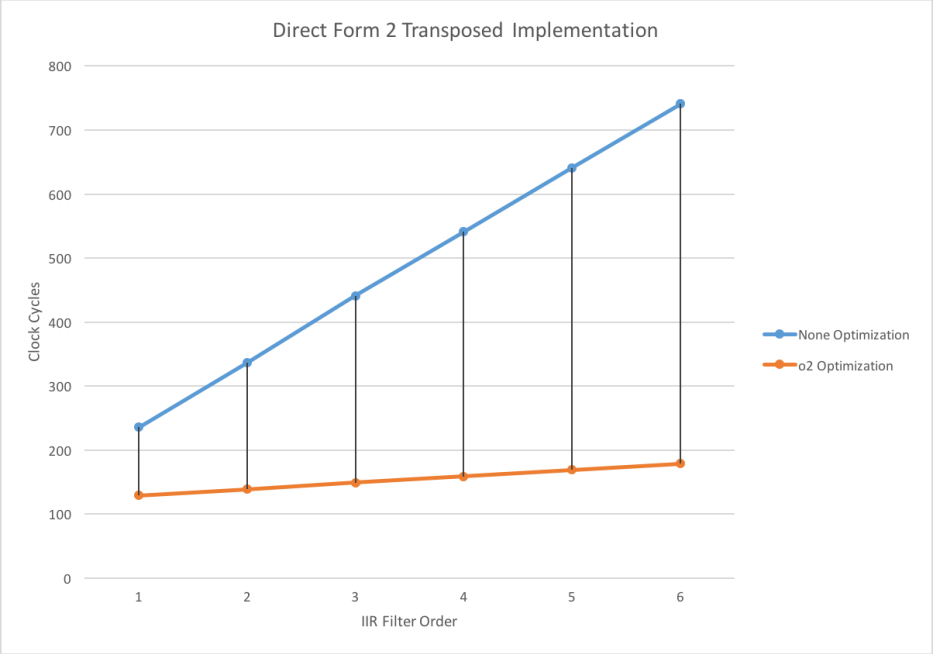


Figure 28: o2 Optimization v None, for different IIR Filter Orders

Once again, we connected our DSK board to the Spectrum Analyzer, to verify the filter's frequency response. The magnitude and phase plots very closely resemble the theoretical MATLAB plots obtained previously.

We note that the Spectrum Analyzer graphs, acquired for this implementation, are virtually identical to those obtained for the non-transposed Direct Form II implementation. This similarity indicates the successful filtering operation carried out by both our IIR algorithms.

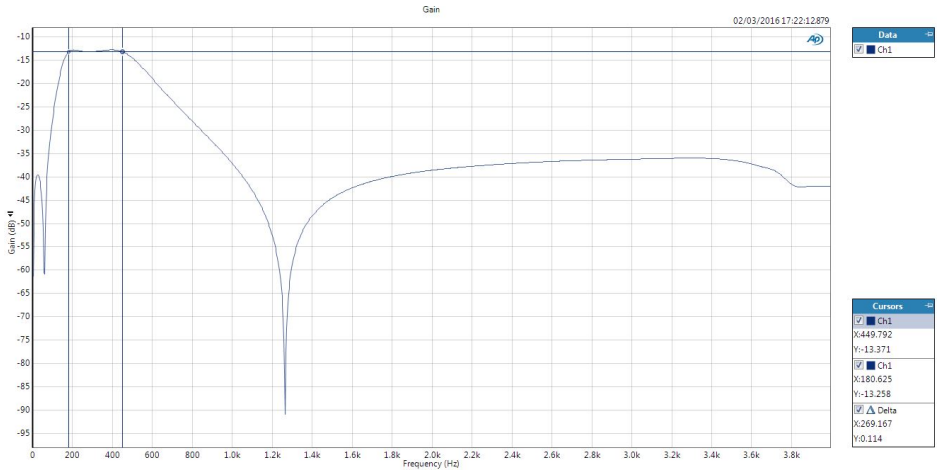


Figure 29: Direct Form II Transposed Overall Response on Spectrum Analyzer

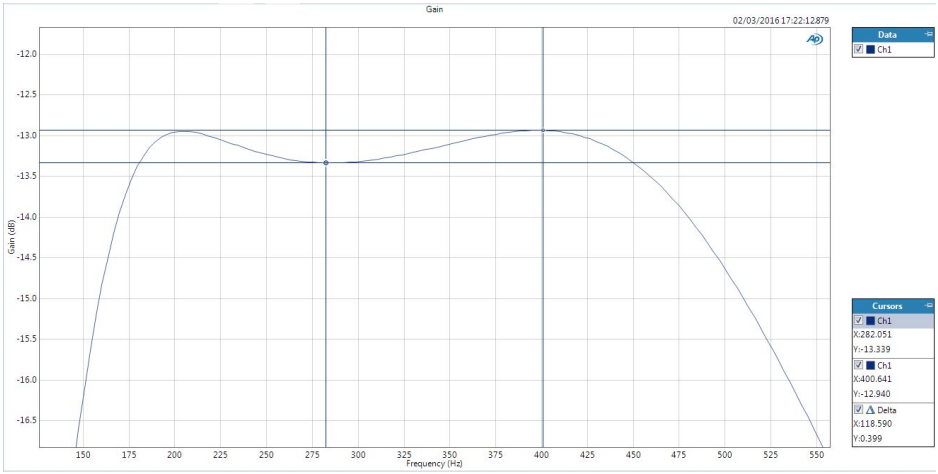


Figure 30: Direct Form II Transposed Passband Ripple $\leq 0.4dB$

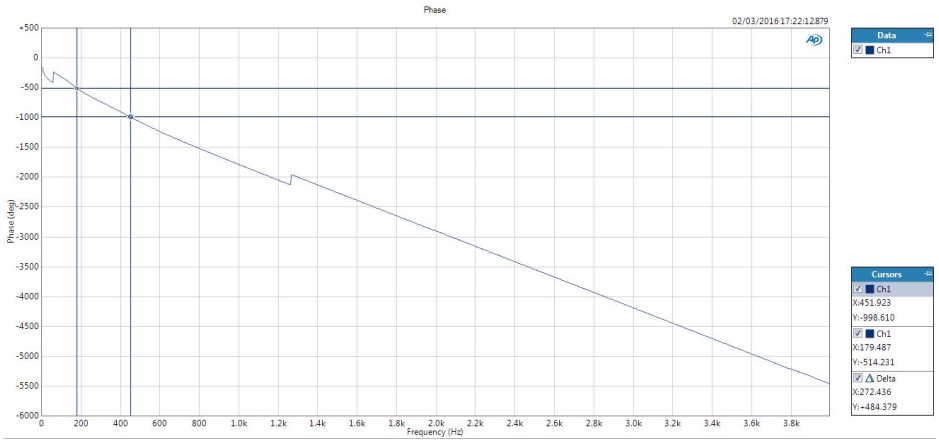


Figure 31: Direct Form II Transposed Phase Response on Spectrum Analyzer

Once again, we compare our implemented result with the theoretical MATLAB trace:

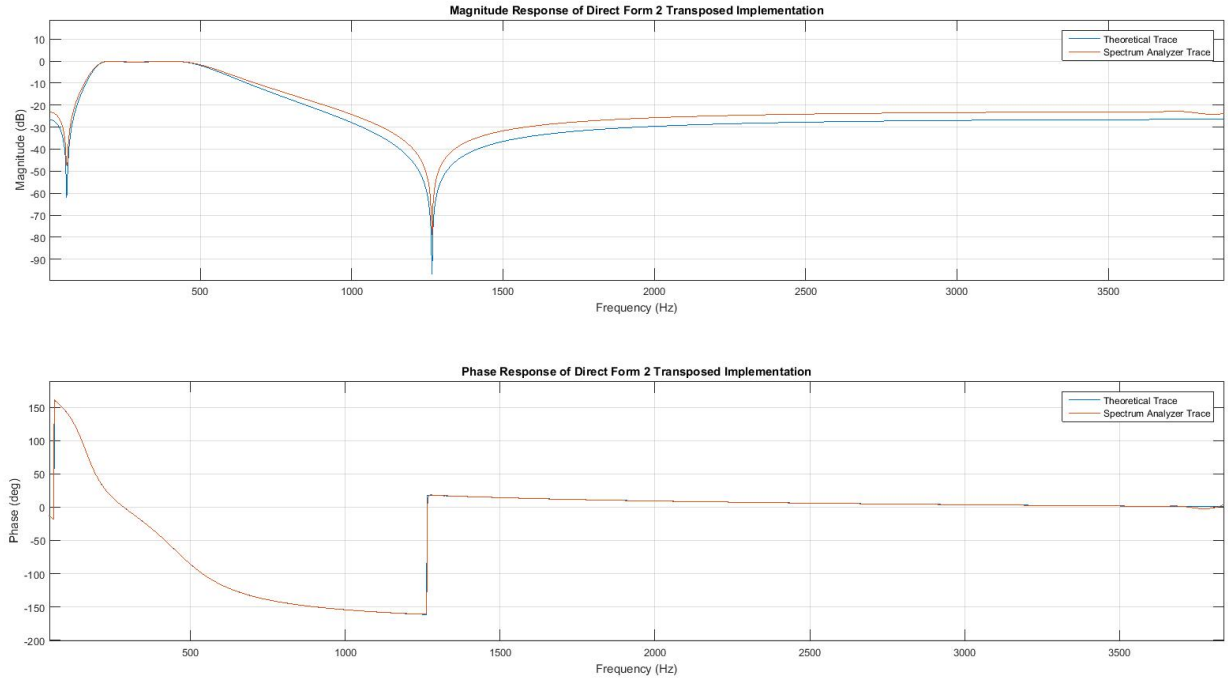


Figure 32: Implemented IIR Response compared with Theoretical MATLAB Response

The magnitude and phase response again closely resemble the expected result. The phase is approximately linear in the passband, and the implemented IIR also satisfies the -23 db attenuation specification.

6 Performance Comparison

6.1 Results Overview

To compare the performance of the two IIR filter algorithms, we used the profiling clock to measure the time taken to execute each algorithm, for different filter orders. For each order of the IIR filter, we executed the algorithm with no compiler optimisation and with `o2` optimisation. The results are presented below:

Filter Order	None	o2
4	501	213
6	715	279
8	929	345
10	1,153	412
12	1,362	478
14	1,576	544

Figure 33: Direct Form II

Filter Order	None	o2
4	236	129
6	336	139
8	441	149
10	541	159
12	641	169
14	741	179

Figure 34: Direct Form II Transposed

We set the upper bound on the order to be 16, because when we plotted the pole-zero plot for the 16th order elliptic IIR filter, we noticed that some of the poles were outside the unit circle. This rendered the filter unstable.

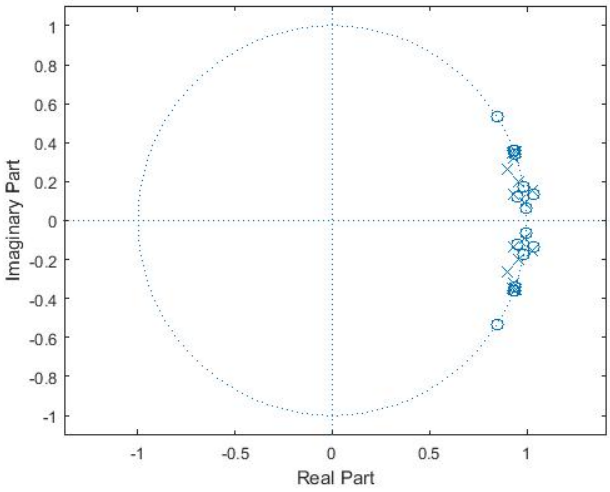


Figure 35: Pole-Zero Plot for 16th Order Elliptic IIR Filter

From the two forms of IIR Direct Form II implementations, we were able to compare their performance. We look at the unoptimised results as well as with `o2`, the best optimisation level. The combined results are shown below, in figure 36 and table 1.

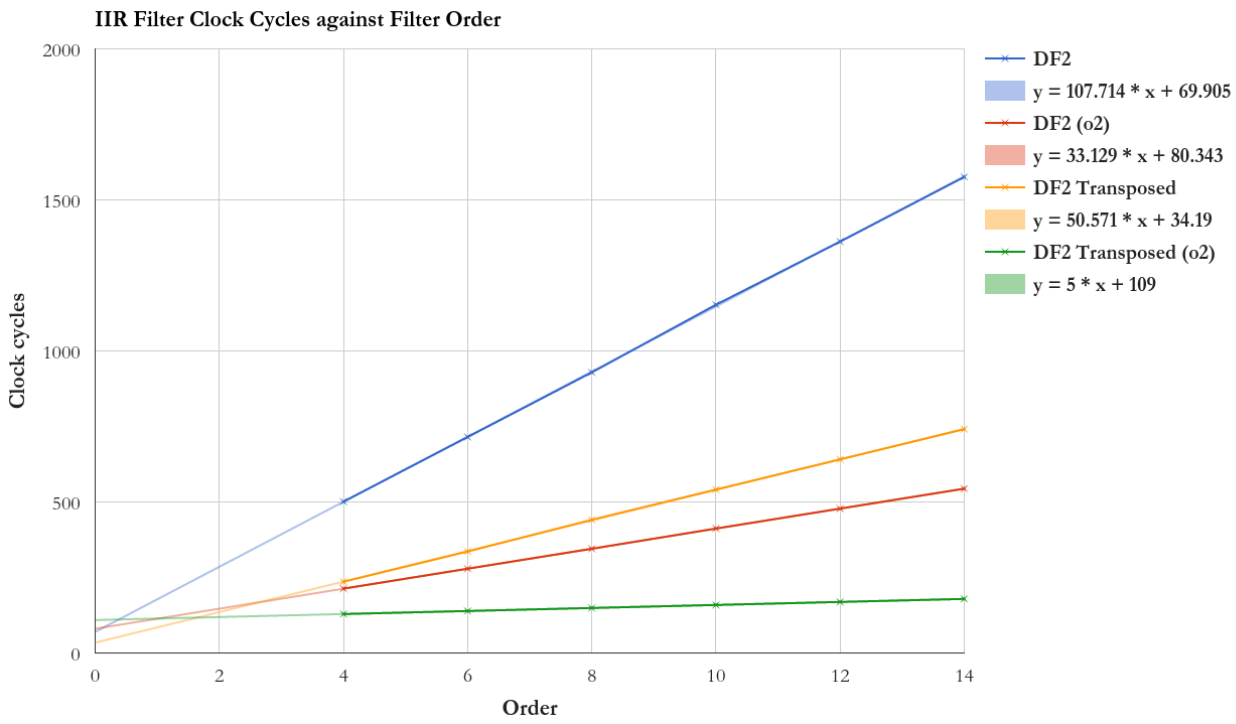


Figure 36: IIR-filtering duration against filter order

IIR Implementation & Optimisation	Filter Clock Cycles
DF2 (`None`)	70 + 108n
DF2 Transposed (`None`)	34 + 51n
DF2 (`o2`)	80 + 33n
DF2 Transposed (`o2`)	109 + 5n

Table 1: Instruction cycles per input sample for filter order n

We extrapolated the lines acquired for each graph, and used graph-plotting software to determine the trend line equation in the form $A + Bn$.

The Direct Form II does not perform as fast as the Direct Form II Transposed implementation at any optimisation level. The difference in gradient is significant at 85% for `o2`. This means the transposed implementation is 85% faster for a large filter order. We attribute this improvement to performing 1 instead of 3 C operations in a for-loop. We have chosen to ignore the fact that the optimised non-transposed implementation performs faster than the non-optimised transposed one as this is not a fair comparison.

6.2 Explanation of Bias in Relative Plot

By examining the spectrum analyzer graph closely, we noticed that the actual gain is not at `0 dB`, rather the entire frequency response has been shifted down to approximately `-12 dB`. This bias was discussed in our previous lab report in further detail, and should be consulted for further reference.

In brief, the signal from the APX500 unit first passes the potential divider circuit, at the input to the DSK board, which halves the signal amplitude. Next, as the software is configured for mono-channel read, the signal is again halved, leaving only a quarter of the input signal amplitude left. Thus, converting a gain of 0.25 into decibels, we obtain:

$$dB = 20 \cdot log_{10}(\frac{1}{4}) = -12.041dB \tag{26}$$

6.3 Conclusion

In the RC Low-Pass filter, our corner frequency was still off, by about 0.5%, even after processing out DSK intrinsic effects. We attribute this to the fact that we did not warp the frequencies in the Tustin Transform. The small error of 0.5% shows that ignoring frequency warping was valid in this case.

We conclude that, for the IIR filter orders tested, the transposed form implementation is always faster than the non transposed implementation. In the non-transposed case, convolution was done by a delay line with side multiply-accumulates. However, in the transposed structure the convolution was done by summations inside the delay line. This results in accumulated convolution when each new sample comes in. This 'readiness' is the key to why the transposed form is significantly faster.

Our results have supported this justification and our code implementations have matched our theoretical MATLAB plots.

7 Appendix - Full Readable Code

C file ran on the DSK, `intio.c`

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

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

                            Eusebius Ngemera and Prahnav Sharma

                                    LAB 5: IIR filter

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

    Performs digital filtering using an included IIR filter from a text file

 ***************************************************************************************
                 Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
                Updated for CCS V4 Sept 10
                Modified to do digital filtering using an IIR filter
 ***************************************************************************************/
/*
 *    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>

#include "iir_coef.txt"
// include MATLAB-generated IIR Filter coefficients file
// #define M (length of a[]) if different from N
// #define N (order+1)
// double a[] = {...}
// double b[] = {...}

/******************************** Global variables ***********************************/
double x[N];
double y[M];

// dynamically-allocated array
double* z;

/******************************** Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
```

```
60      interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
61  DSK6713_AIC23_Config Config = { \
62              /*****************************************************************/
63              /*   REGISTER                FUNCTION                  SETTINGS        */
64              /*****************************************************************/\
65      0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
66      0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
67      0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
68      0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
69      0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
70      0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off     */\
71      0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on     */\
72      0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit              */\
73      0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
74      0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
75              /*****************************************************************/
76  };
77
78
79  // Codec handle:- a variable used to identify audio interface
80  DSK6713_AIC23_CodecHandle H_Codec;
81
82   /**************************** Function prototypes ********************************/
83  void init_hardware(void);
84  void init_HWI(void);
85  void ISR_AIC(void);
86  // IIR implementations
87  double simple_IIR(short new_sample);
88  double DF2_IIR(short new_input);
89  double DF2T_IIR(short new_input);
90  /********************************* Main routine ***********************************/
91  void main(){
92
93      // initialize board and the audio port
94       init_hardware();
95
96       // dynamic memory allocation, with zero initiliasing
97      z = (double*)calloc(N, sizeof(double));
98      memset(z, 0, sizeof(z));
99
100      // initialize hardware interrupts
101     init_HWI();
102
103     // loop indefinitely, waiting for interrupts
104     while(1)
105     {};
106 }
107
108 /********************************* init_hardware() ********************************/
109 void init_hardware()
110 {
111     // Initialize the board support library, must be called first
112     DSK6713_init();
113
114     // Start the AIC23 codec using the settings defined above in config
115     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
116
117     /* Function below sets the number of bits in word used by MSBSP (serial port) for
118     receives from AIC23 (audio port). We are using a 32 bit packet containing two
119     16 bit numbers hence 32BIT is set for  receive */
120     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
121
122     /* Configures interrupt to activate on each consecutive available 32 bits
```

```
123     from Audio port hence an interrupt is generated for each L & R sample pair */
124     MCBSP_FSETS(SPCR1, RINTM, FRM);
125
126     /* These commands do the same thing as above but applied to data transfers to
127     the audio port */
128     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
129     MCBSP_FSETS(SPCR1, XINTM, FRM);
130
131 }
132
133 /********************************* init_HWI() ************************************/
134 void init_HWI(void)
135 {
136     IRQ_globalDisable();            // Globally disables interrupts
137     IRQ_nmiEnable();                // Enables the NMI interrupt (used by the debugger)
138     IRQ_map(IRQ_EVT_RINT1,4);        // Maps an event to a physical interrupt
139     IRQ_enable(IRQ_EVT_RINT1);        // Enables the event
140     IRQ_globalEnable();                // Globally enables interrupts
141
142 }
143
144 /************************** INTERRUPT SERVICE ROUTINE ****************************/
145 void ISR_AIC(void)
146 {
147     // reads mono-channel input as type Int16 and outputs the filtered output
148     // as type Int16 in the mono-channel output
149
150     mono_write_16Bit( (short) DF2T_IIR( mono_read_16Bit() ));
151     // in-line coding is used to reduce time spent in the Interrupt
152 }
153
154 /************************** IIR Filter Implementations ***************************/
155 // IIR assuming a and b are of different lengths, a[M] and b[N]
156 double simple_IIR(short new_sample)
157 {
158     int i;                    // for-loop iteration index
159     double sum = 0.0;        // data type matches IIR coefficients
160
161     // shifting x, the past inputs
162     for (i = N-1; i > 0; i--)
163     {
164         // move data one index up, discarding last (oldest) input
165         x[i] = x[i-1];
166     }
167     // prepend new sample, to be the first element
168     x[0] = new_sample;
169
170     // Convolution between b coefficients and past inputs
171     for (i = 0; i < N; i++)
172     {
173         sum += b[i] * x[i];
174     }
175
176     // Convolution between a coefficients and past outputs
177     // a[0] = 1, convolution starts from a[1]
178     for (i = 1; i < M; i++)
179     {
180         sum -= a[i] * y[i];
181     }
182
183     // shifting y, the past outputs
184     for (i = M-1; i > 0; i--)
185     {
```

```
186         // move data one index up, discarding last (oldest) output
187         y[i] = y[i-1];
188     }
189 
190     // prepend final output value to y buffer
191     // y index matches a coefficient array
192     y[1] = sum;
193 
194     return sum;
195 }
196 
197 // Direct-Form 2 IIR filtering
198 double DF2_IIR(short new_input)
199 {
200     int i;
201     double output = 0.0;
202 
203     // z[0] is the top, intermediate node
204     z[0] = new_input;
205 
206     // (accumulated) convolution of the left branch
207     for (i = 1; i < N; i++)
208     {
209         z[0] -= a[i] * z[i];
210     }
211 
212     output = b[0] * z[0];
213 
214     // (accumulated) convolution of the right branch
215     for (i = 1; i < N; i++)
216     {
217         output += b[i] * z[i];
218     }
219 
220     // shifting z down
221     for (i = N-1; i > 0; i--)
222     {
223         z[i] = z[i-1];
224     }
225 
226     return output;
227 }
228 
229 // Direct-Form 2 transposed IIR filtering
230 double DF2T_IIR(short new_input)
231 {
232     int i;
233 
234     double output = (new_input * b[0]) + z[0];
235 
236     // iterate from right to left
237     // last summation does not have z[N-1], and is an edge case
238     for (i = 1; i < N-1; i++)
239     {
240         z[i-1] = z[i] + (new_input * b[i]) - (output * a[i]);
241     }
242     z[N-2] = (new_input * b[N-1]) - (output * a[N-1]);
243 
244     return output;
245 }
```