*Imperial College London*

Mattin Mir-Tahmasebi (mm5213) and Ahmed Ibrahim (aai13)

March 2016

RTDSP Lab 5 Report

*Real-Time Implementation of IIR Filters*

Contents

[Single pole RC filter 2](#_Toc444866150)

[Direct I Form IIR filter Implementation in C 4](#_Toc444866151)

[RC Filter Analysis 4](#_Toc444866152)

[Comparison of Digital and Analogue response 8](#_Toc444866153)

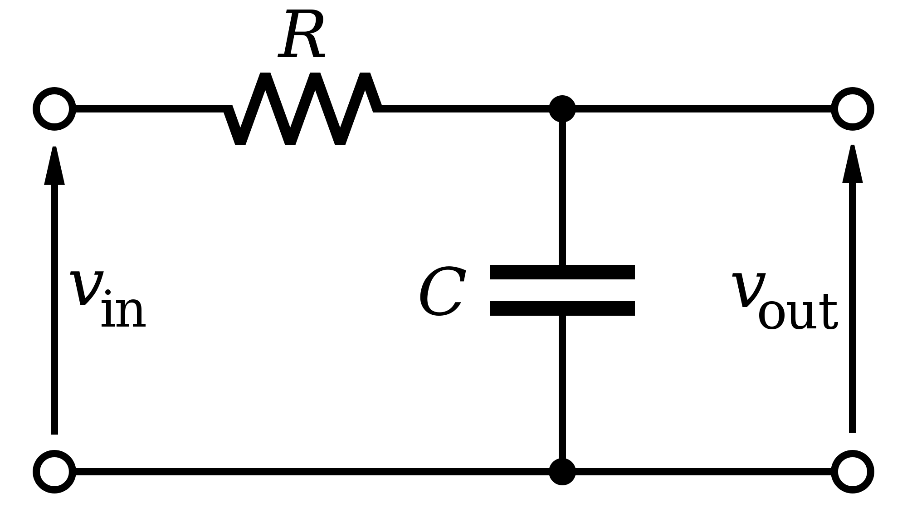
[bandpass filter: direct form ii 11](#_Toc444866154)

[Direct Form II Analysis 13](#_Toc444866155)

[Bandpass filter: direct form ii transposed 18](#_Toc444866156)

# Single pole RC filter

Initially, a single pole filter, consisting of a simple RC filter, must be designed. This will be a first-order low pass filter. The mathematical representation of the filter in the s domain, shown below, will be mapped in the discrete time domain using the Tustin (bilinear) transform.



*Figure 1 Simple low pass RC filter [[1]](#footnote-1)*

The transfer function for this filter can be found to be:

Using the Tustin transform we can map this function to the discrete time domain by substituting:

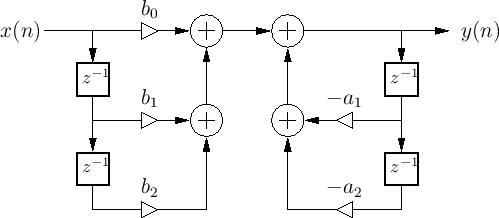
The transfer function now becomes:

Therefore:

Using the values given in the specification (R = 1KΩ, C = 1µF) and the sampling frequency, the transform becomes.

To implement this filter in C, the characteristic transfer function of an IIR filter must be used to give a difference equation in the time domain[[2]](#footnote-2).

In Direct I form



*Figure 2 Direct-Form I*

A corresponding difference equation can be found:

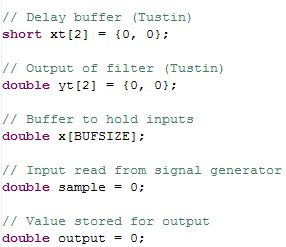
Manipulating the discrete time transfer function to have it in the form of an IIR filter:

The corresponding difference equation is:

## Direct I Form IIR filter Implementation in C

Using the coefficient values ( the filter was implemented in C.

It makes use of two buffers, xt[] and yt[] (*Figure 3*) which hold the history of input and outputs, respectively, where the 0th index represents the newest sample, and increasing indices indicate samples from further back in time.



*Figure 3 The global variables that have been instantiated*

The algorithm begins by shifting the past inputs back in the buffer to make room in xt[0], which is initialised to the incoming sample. The filter is then applied using the coefficients calculated earlier and assigned to the start of the output buffer. That value is also assigned to the next element in the buffer for use in the next iteration of the filter, and is finally assigned to the output. The filter function is seen in Figure 3.

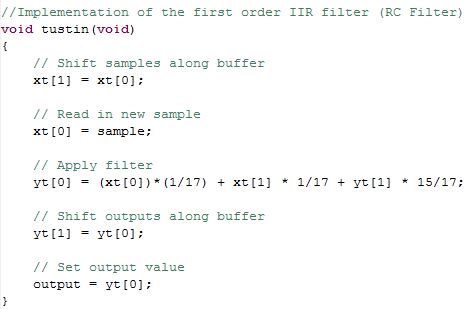
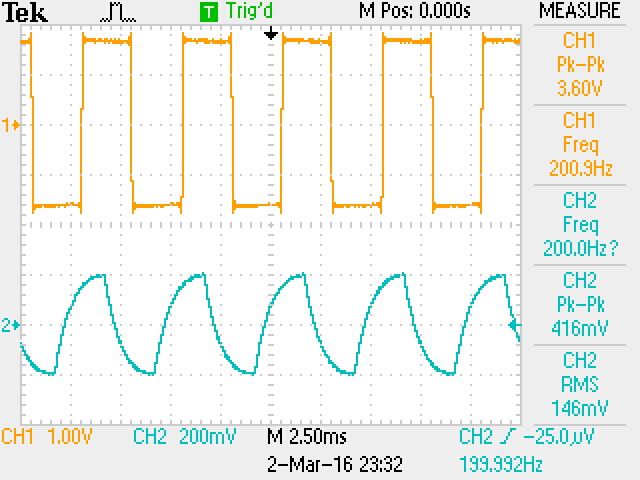


Figure RC filter C implementation

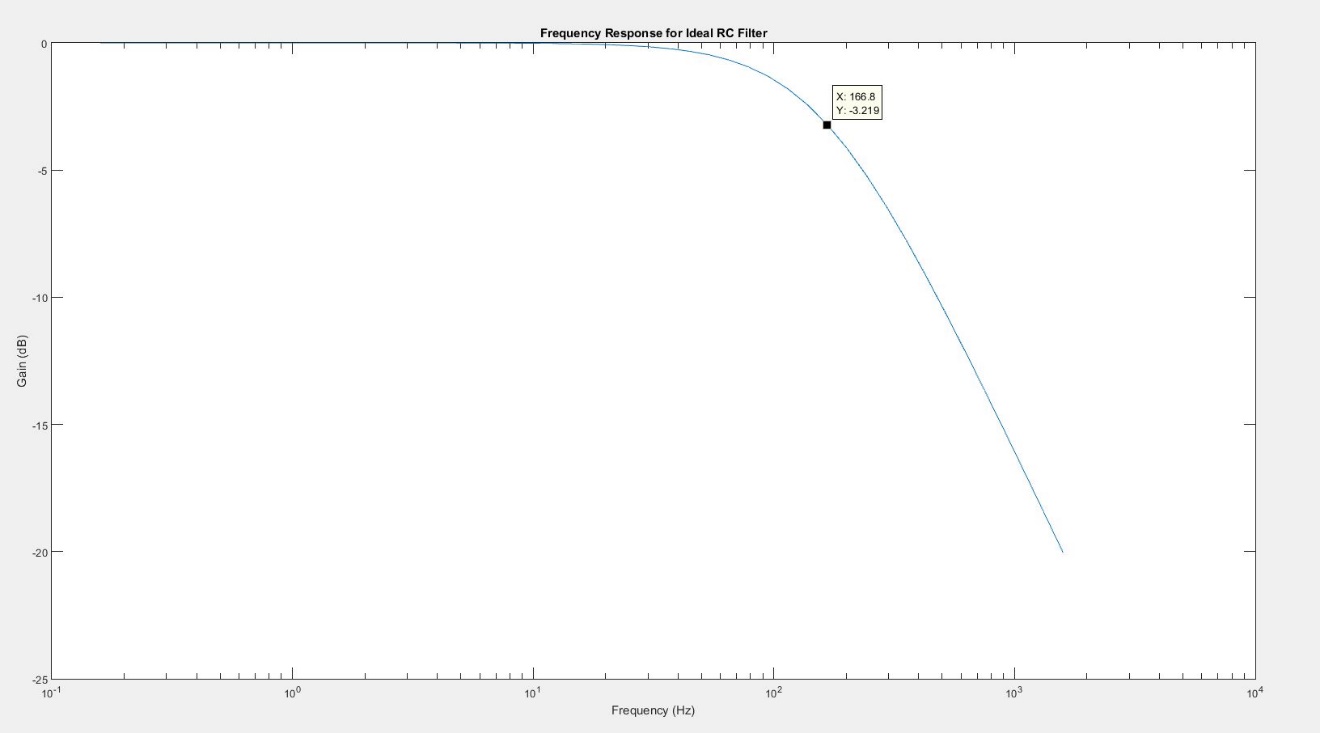
## RC Filter Analysis

The filter is then implemented in the DSK. Driving the input with a low frequency square wave *Figure 4*, the time constant of this RC filter can be found.



*Figure 5 RC filter output for a square wave input*

An input frequency of 150Hz is used for this filter since it is well below the cut off frequency for the first order filter (*Figure 5*) () and is also above the roughly 7Hz cut-off of the high pass filter present at the output of the AIC23 Codec.



*Figure 6 RC filter ideal response showing the roughly 160Hz cut-off frequency*

The time constant of the filter can be found by driving the input with this 150Hz square wave. Using the output given in Figure 6, the time constant for the RC filter can be found. The time constant can be found through by example, using the relationship where τ = RC in this case.

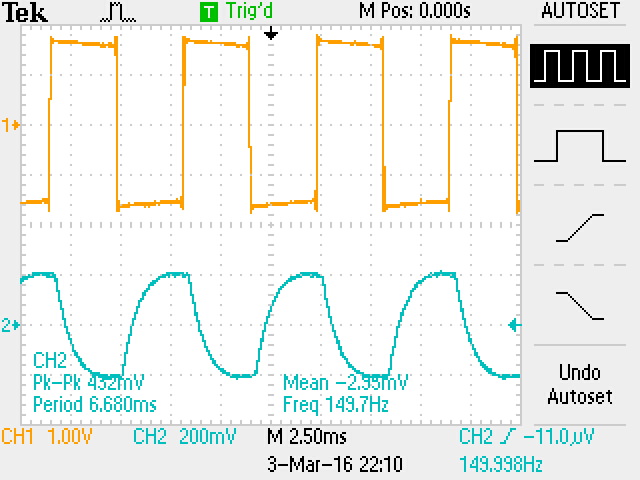


Figure 150Hz square wave input with its corresponding output

By calculating the time for the output voltage to decay by one time constant , which is when the output voltage decreases by 63.2% to 36.8% of its initial peak value. An example is shown on Figure 7 below.

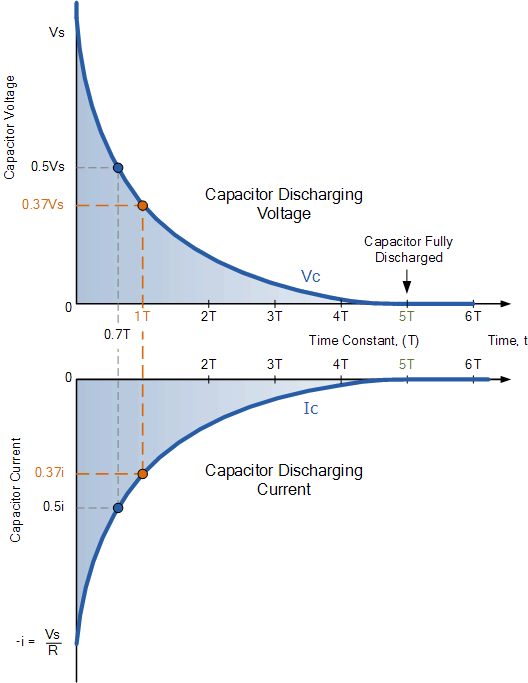
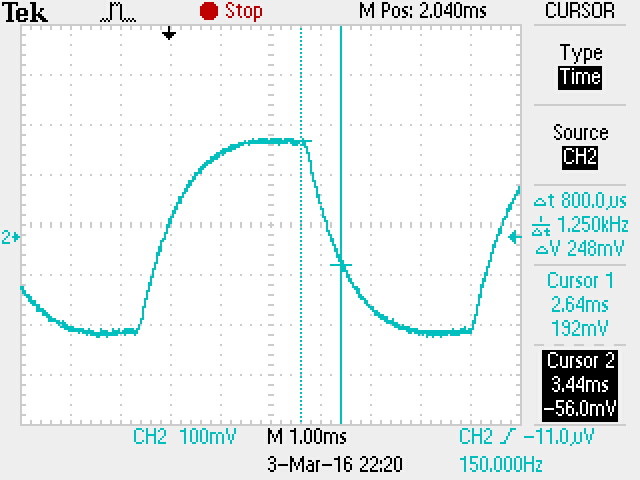


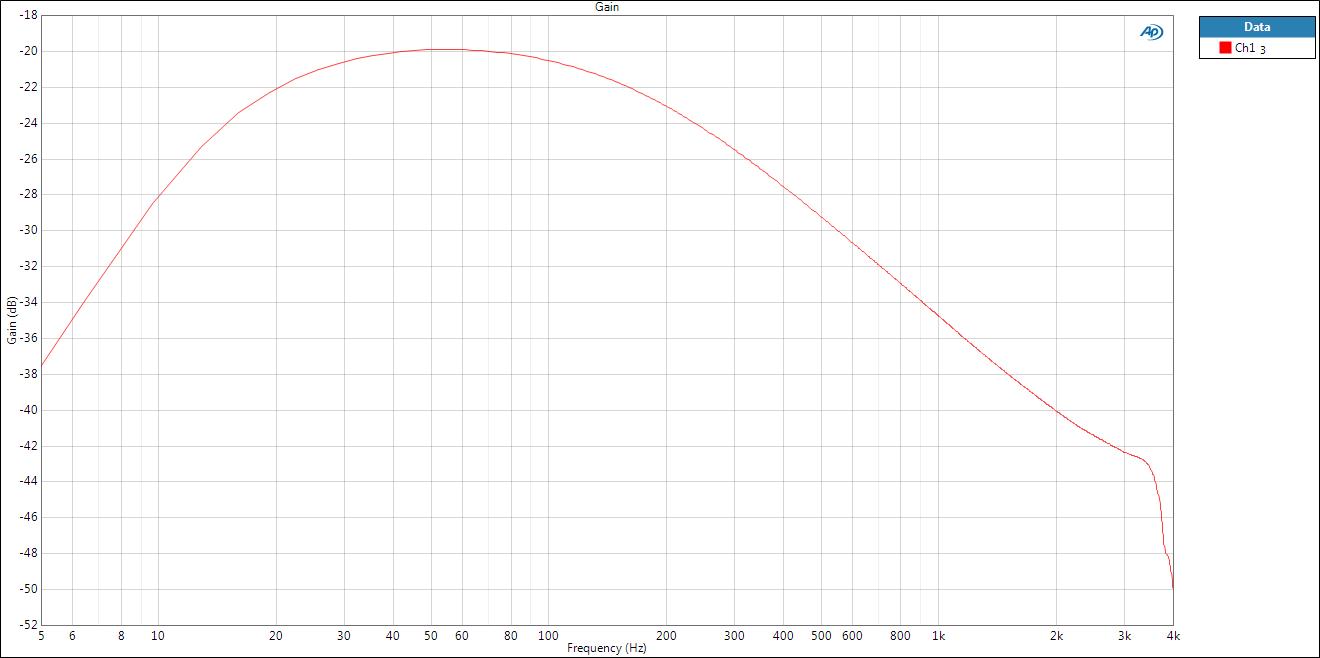
Figure Capacitor discharging voltage example noting the time taken for discharging to 1 time constant of its initial peak value (~37%) [[3]](#footnote-3)

Using the output trace, the time constant can be found (*Figure 8*).



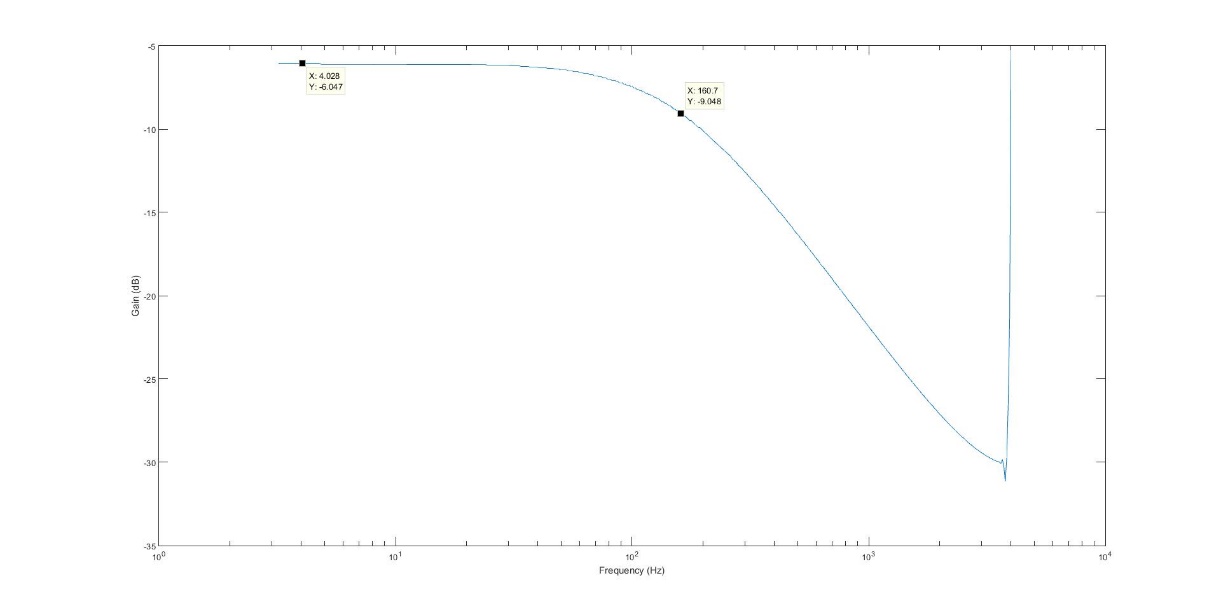
*Figure 9 Finding the time constant using the output voltage trace*

The value of the time constant is found to be 0.8ms. This is below the expected value of 1ms (. This can be explained due to the effects of high pass and reconstruction filters present in the DSK (see next subsection). The DSK attenuates the response expected from the ideal RC filter and gives a response as seen in *Figure 9*.



*Figure 10 Response of the RC filter as given by the spectrum analyser*

The time constant can be found for the ideal filter by subtracting the no filter response with the response of the ideal filter, this is given in *Figure 5*.



*Figure 11 Adjusted response for the RC filter taking into account the attenuation of the DSK.*

Finding the cut-off frequency for this gives a time constant of 0.99 which is very close to the expected value of 1ms.

## Comparison of Digital and Analogue response

The ideal response of the filter is shown in Figure 11. The actual response of the filter as given by the spectrum analyser is shown in *Figure 9*. It can be seen that the response does not closely match the expected magnitude response given by MATLAB in *Figure 5*. This can be explained when looking at the spectrum analyser magnitude response when no filter function is used. The response for this is shown in *Figure 11* below.

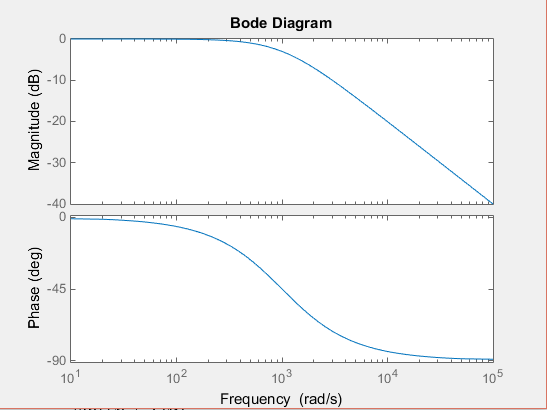
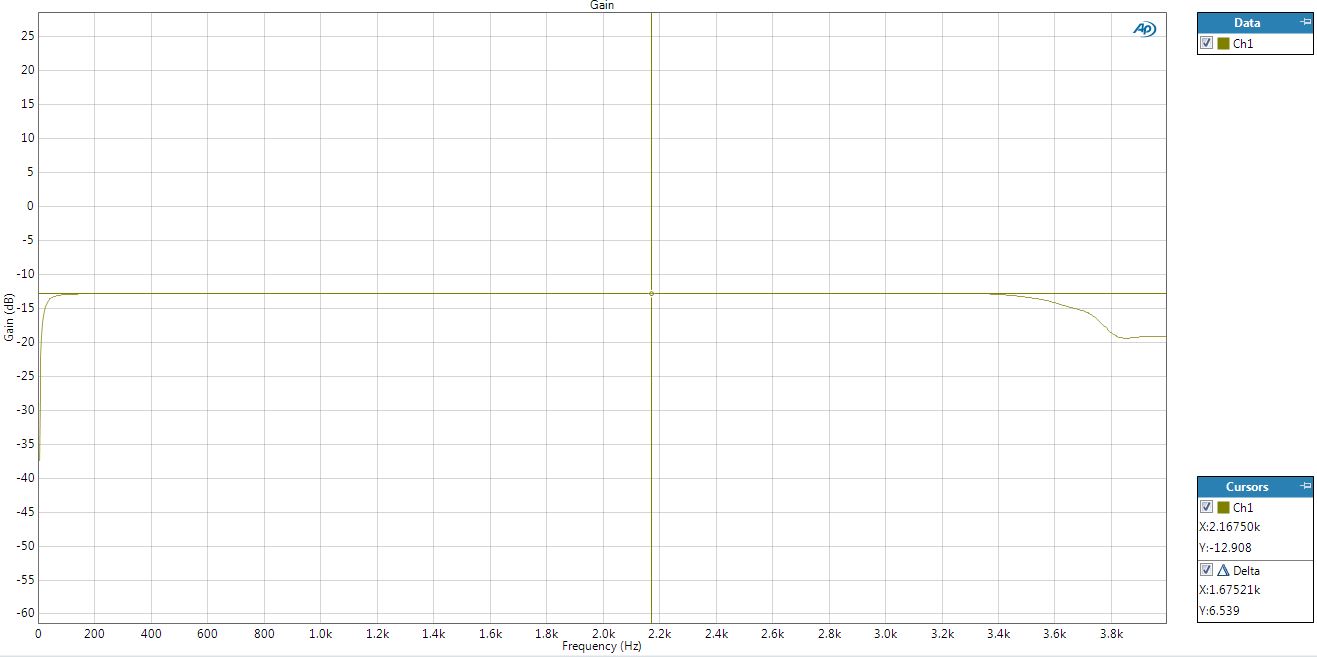


Figure Bode plot of ideal RC filter



*Figure 13 Spectrum analyser magnitude response when no filter function is used*

As can be seen there is attenuation at low frequencies (~<10Hz) this is due to the high pass filter present at the output of the AIC23 Codec (Figure 12. The transfer function for this filter is . The magnitude response of this filter can also be plotted and is shown in *Figure 13*. With a cut-off frequency of 7Hz this shows the attenuation of the RC filter at low frequencies. At high frequencies there is sharp drop due to the reconstruction filter with a cut-off frequency of , this is shown clearly as well on the actual RC filter plot (*Figure 9*). The -12dB attenuation is due to the input to the AIC23 Codec being halved due to the potential divider and the fact that the two inputs to the spectrum analyser are being averaged. Since only one line is used the input is halved further.

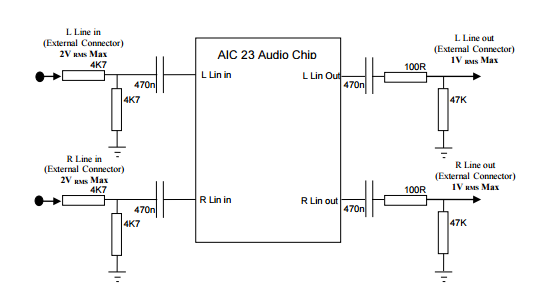
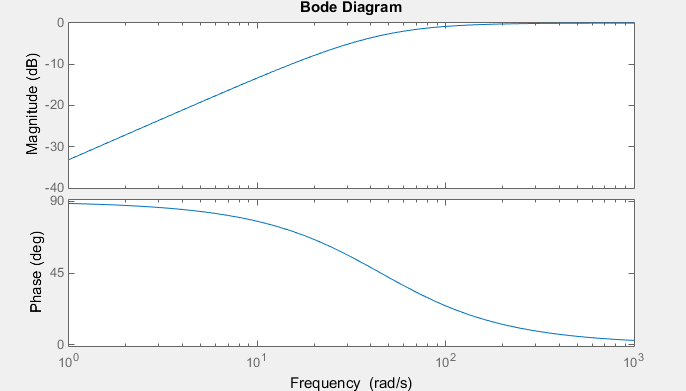
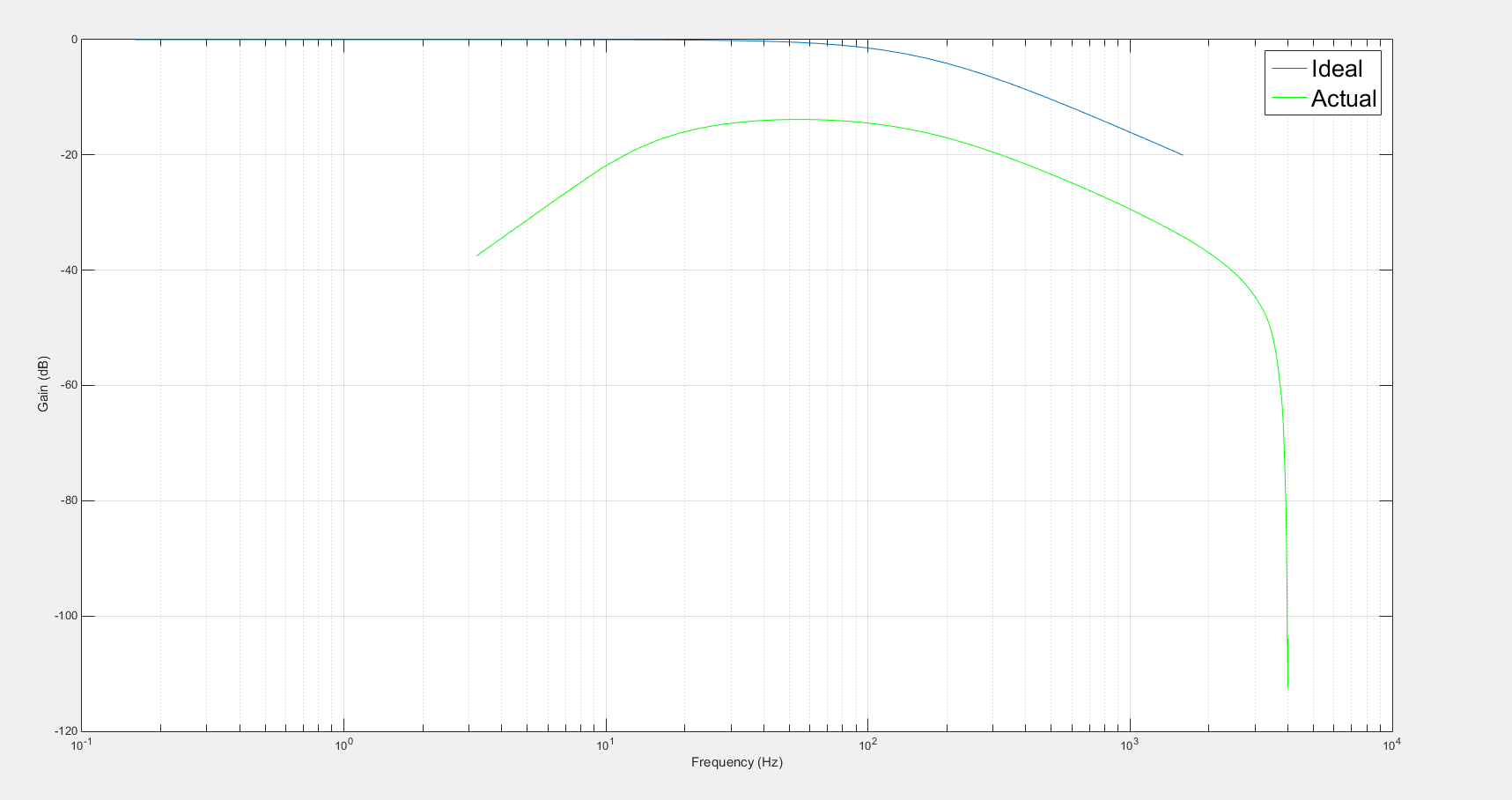


Figure AIC 23 Codec Audio Chip



*Figure 15 Bode plot for the high pass filter at the output of the AIC23 Codec*

Superimposing the spectrum analyser results with the ideal analogue filter results, the attenuation from the reconstruction and high pass filters are seen clearly (*Figure 14*). The attenuation brought about by the hardware is also seen. The reconstruction filter attenuation at 4Khz can be seen when plotting the digital filter. The actual spectrum analyser results again can be superimposed onto the ideal digital filter response seen. This is shown clearly in Figure 15.



*Figure 16 Actual spectrum results superimposed onto the ideal magnitude response*

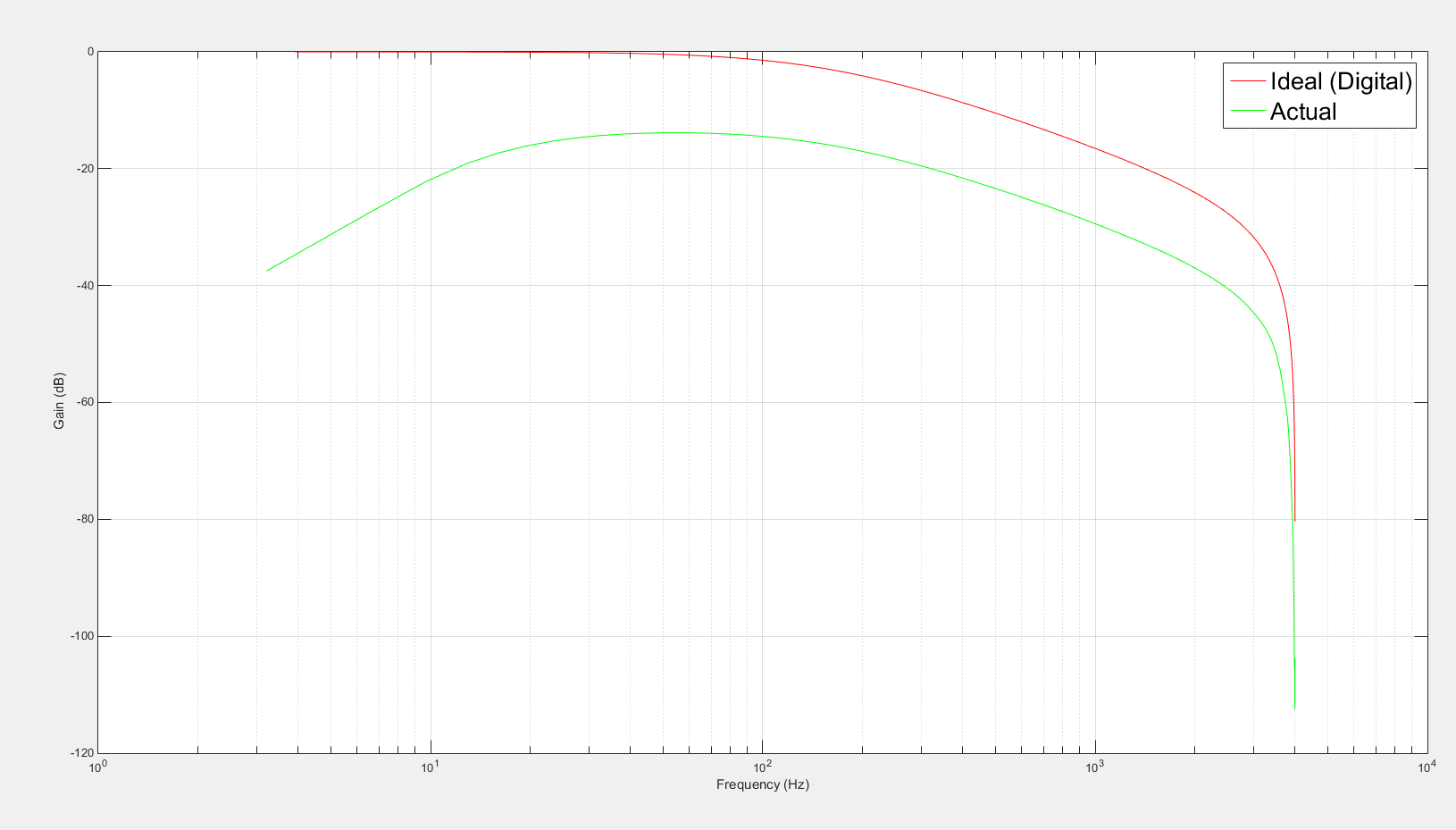


Figure Actual spectrum superimposed onto the ideal digital response

# Bandpass Filter: Direct Form II

An elliptic bandpass filter is designed with the following specifications[[4]](#footnote-4):

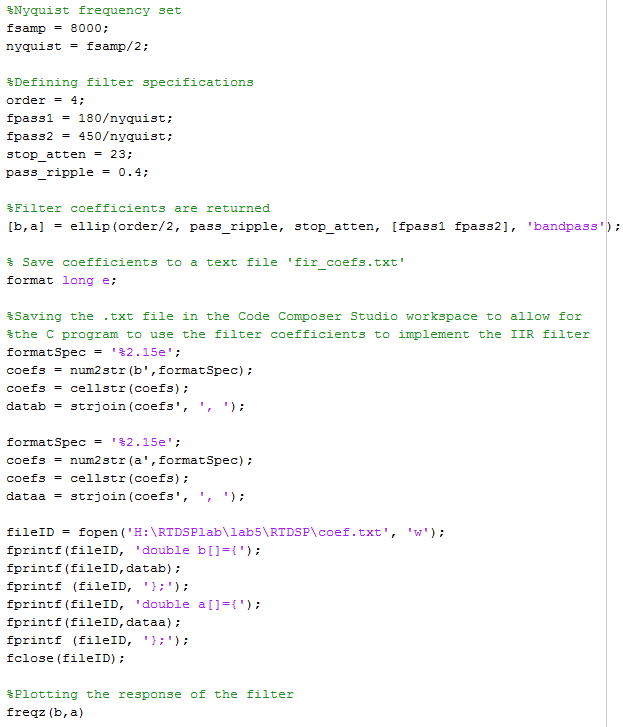
|  |  |
| --- | --- |
| Order | 4th |
| Passband | 180-450Hz |
| Passband Ripple | 0.4dB |
| Stopband Attenuation | 23dB |

Table Filter Specifications

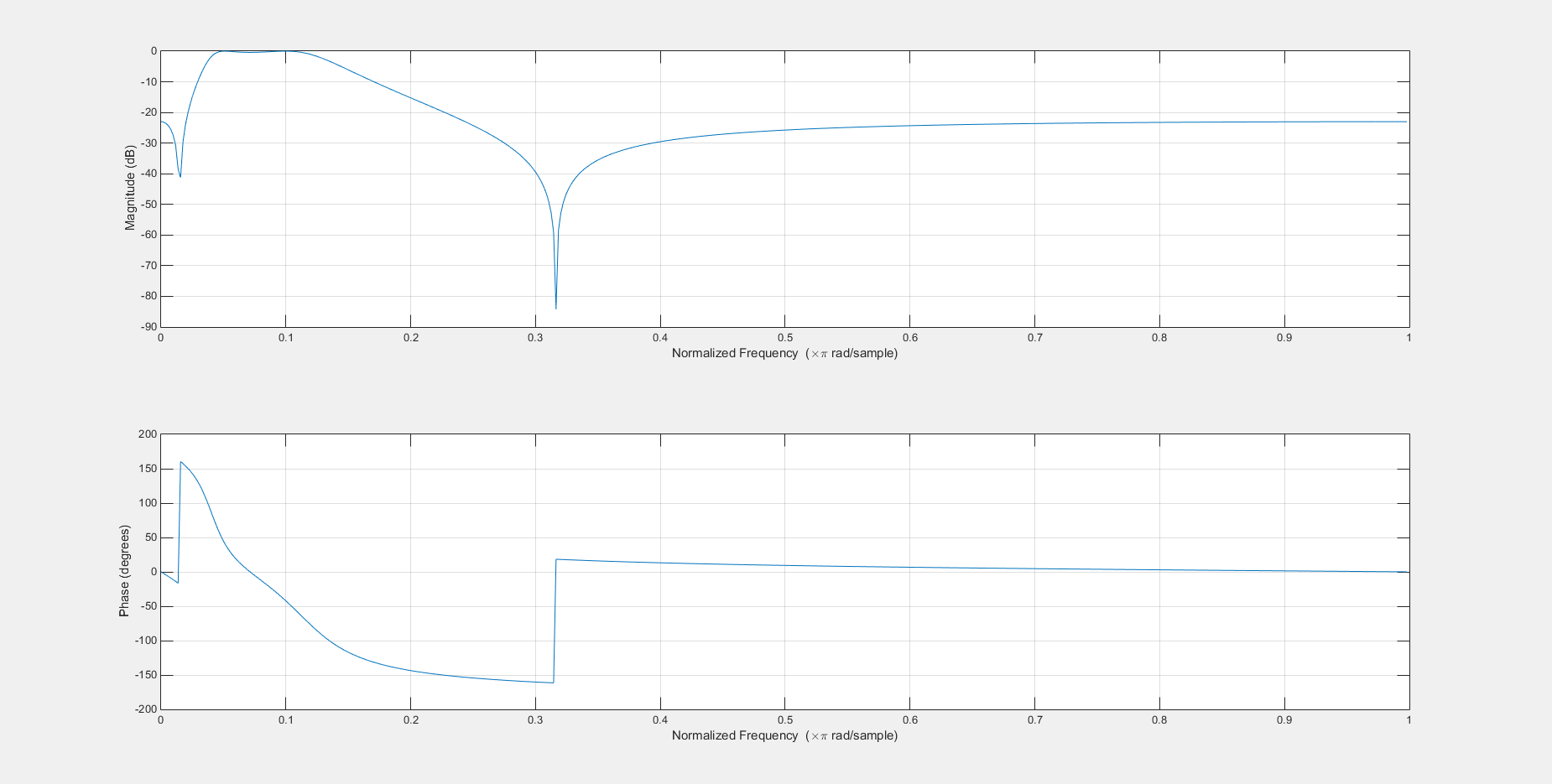
An elliptic filter has ripple in both pass-band and stop-band. This allows the filter to have a an extremely fast transition (faster than a Chebyshev) between the pass band and stop band[[5]](#footnote-5). An elliptic filter has the following response [[6]](#footnote-6)

Where is the response of a Chebyshev filter, and is the ripple factor. Elliptic filters are generally used where there requires to be cut-offs at frequencies very close to particular frequencies.

Assuming a sampling frequency of 8Khz, the elliptic filter can be designed in MATLAB using the program shown in *Figure 16*. The MATLAB program returns a set of filter coefficients that can be used to implement the IIR filter in Direct form II. The response of the filter is shown in *Figure 17*. It can be clearly seen from the response that the filter has a sharp roll-off at the cut-off frequency.



*Figure 18 The design of the filter in MATLAB*



*Figure 19 Filter frequency response*

To implement the direct form 2 filter, we consider each adder shown in Figure 19 as two separate variables, left and right.

For every new sample that is received, the existing samples in the buffer are shifted down, and the relevant filter coefficients are applied, the results of which are added to the respective variable: left for the a coefficients, and right for the b coefficients.

The new sample is then added to the left adder, which is multiplied by b[0] and added to right to create the final value of the output. Lastly, the current value of left is assigned to the first element of the buffer, to be ready for the next cycle. The C function used is shown in Figure 18.

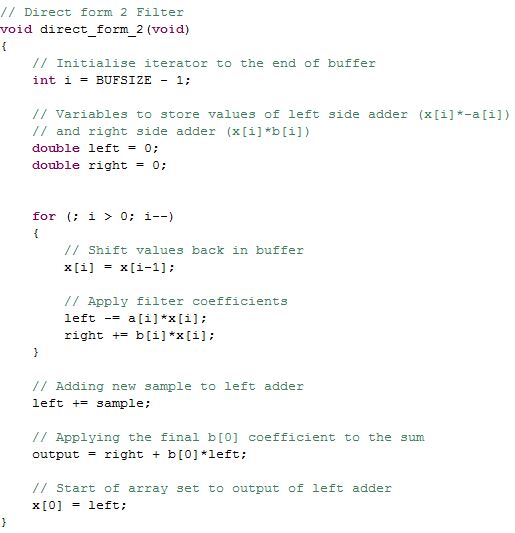


Figure Direct form II C implementation

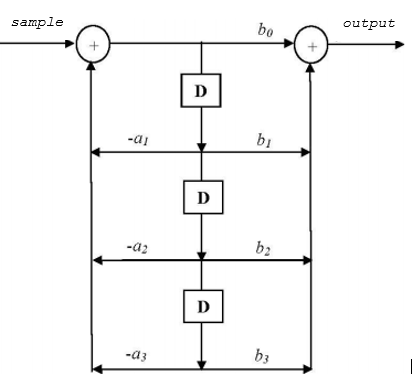


Figure Direct II Form

## Direct Form II Analysis

The spectrum given in Figure 20, shows that the magnitude response is similar to the response given by MATLAB. The pass-band ripple as seen in Figure 21, is measured to be 0.390dB, well within the specified limits. The actual spectrum can be superimposed onto the spectrum given by MATLAB (Figure 22). It can be seen that the actual spectrum clearly matches the input spectrum, however at low frequencies, there is attenuation. The plot also is taken at a relative level, eliminating the 12dB attenuation performed by the hardware. The low frequency attenuation as explained earlier, is due to the high pass filter present at the output of the AIC23 codec. The high frequency attenuation, again as explained earlier, is due to the reconstruction anti-aliasing filter present in the DSK. The phase response shown in Figure 23 is however quite different. There is an additional phase shift of again due to the hardware. Again the AIC23 codec gives this additional shift, this can be seen when looking at the phase response using the spectrum analyser without a filter implemented (Figure 24).

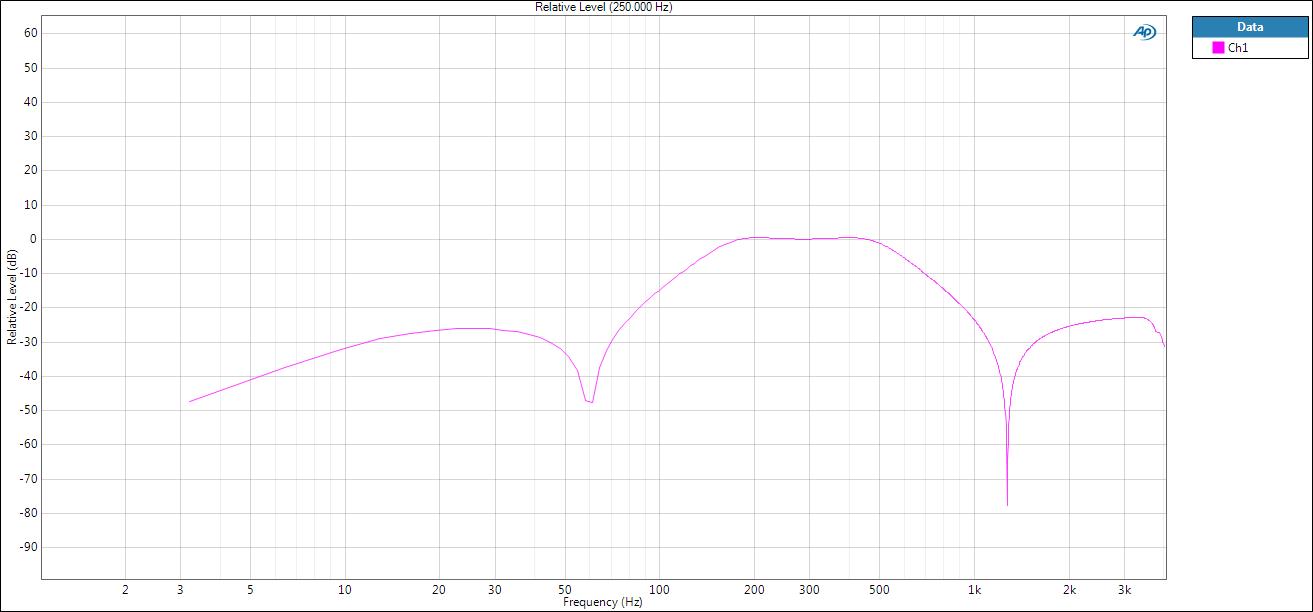


Figure Magnitude response (250Hz relative level)

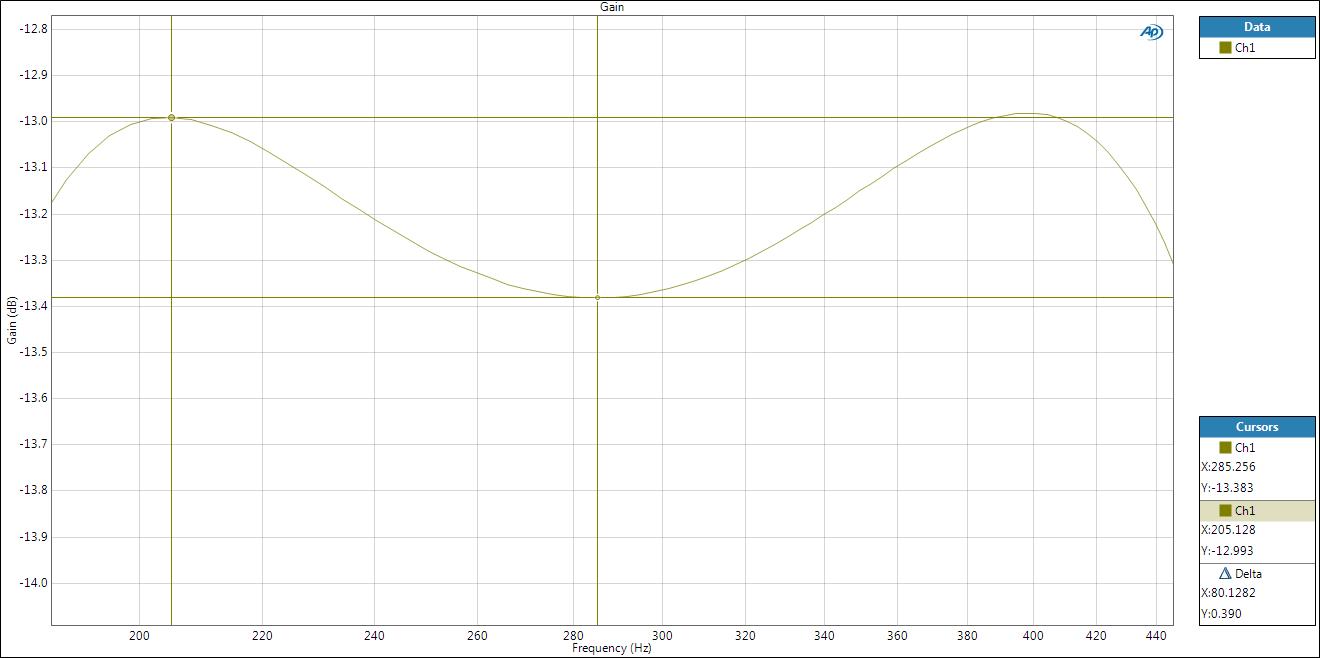


Figure Spectrum Analyser Response showing pass-band ripple within the specified limits

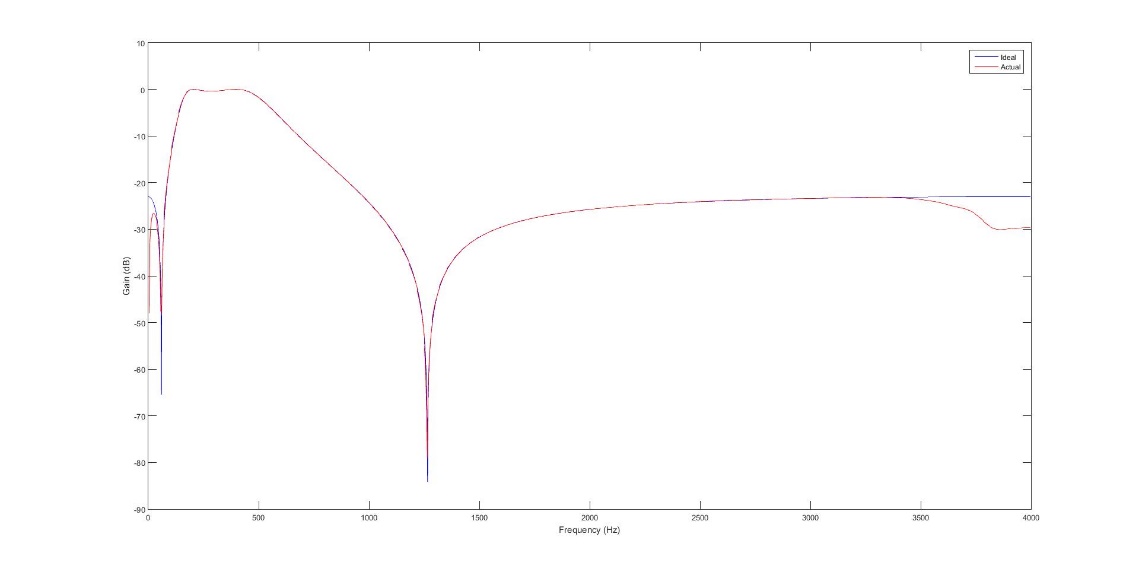


Figure Actual spectrum superimposed onto the ideal magnitude response. Note that the actual spectrum is used at a relative level.

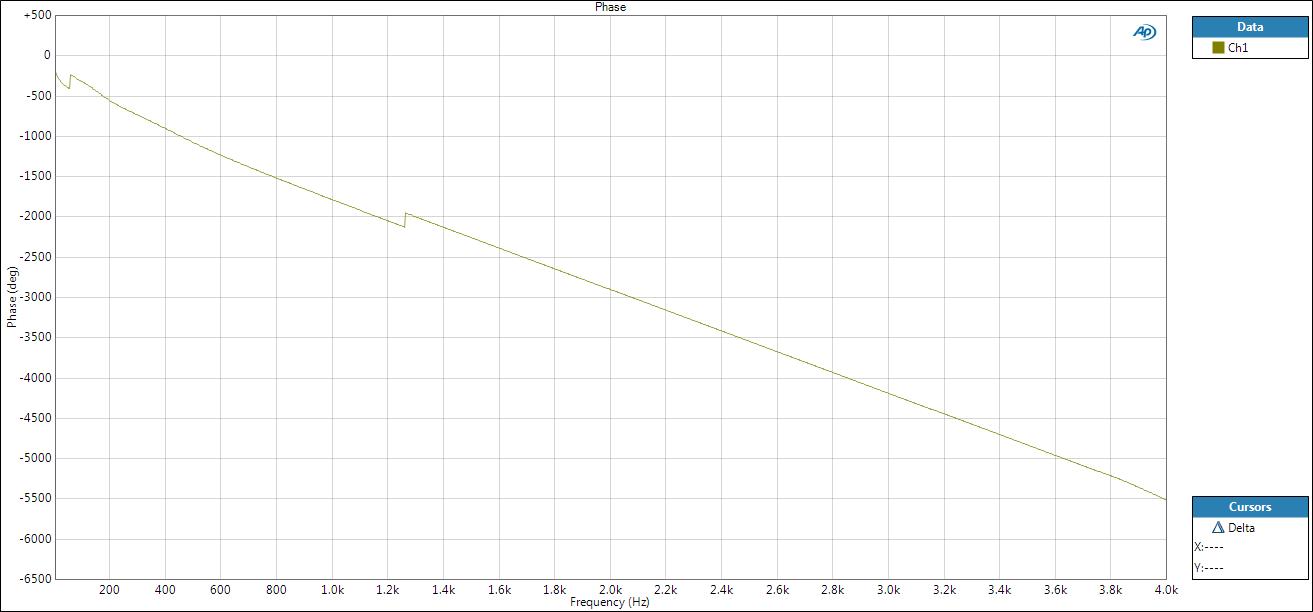


Figure Actual Phase response of the IIR filter showing an additional phase shift

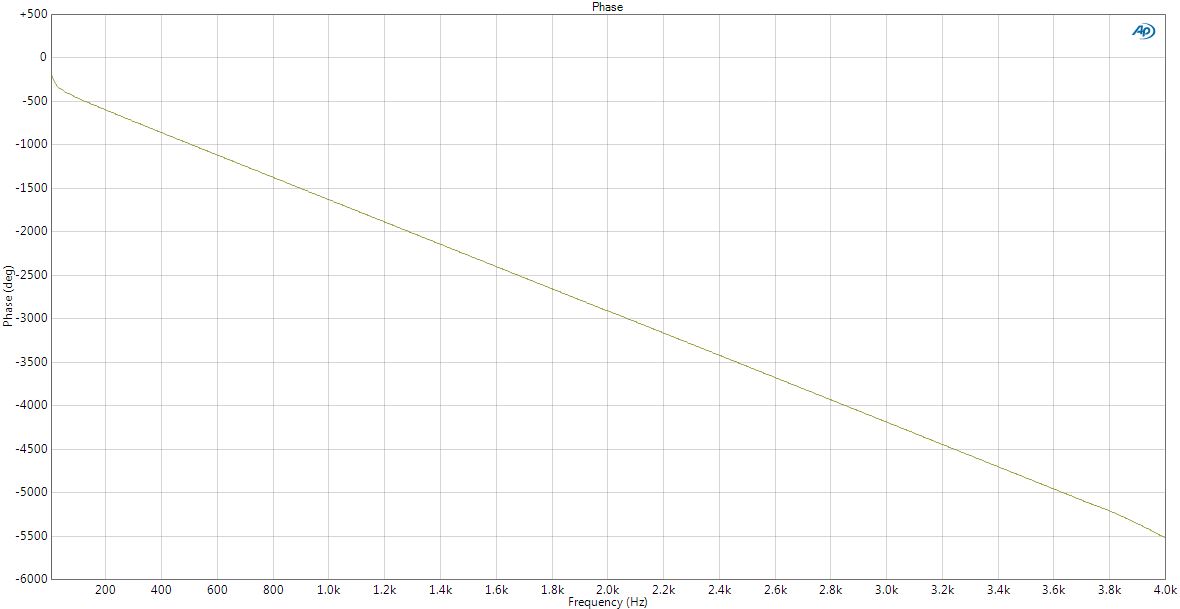


Figure Phase response when no filter function is used

Another interesting effect can be seen when increasing the order of the filter. As the filter order is increased past 12, the response given by MATLAB shows significant distortion to the required magnitude and phase response. This distorted frequency response is shown in Figure 25. This distortion is due to the filter becoming unstable at higher orders. This can only be explained by looking at the placement of the poles and zeros of the filter.

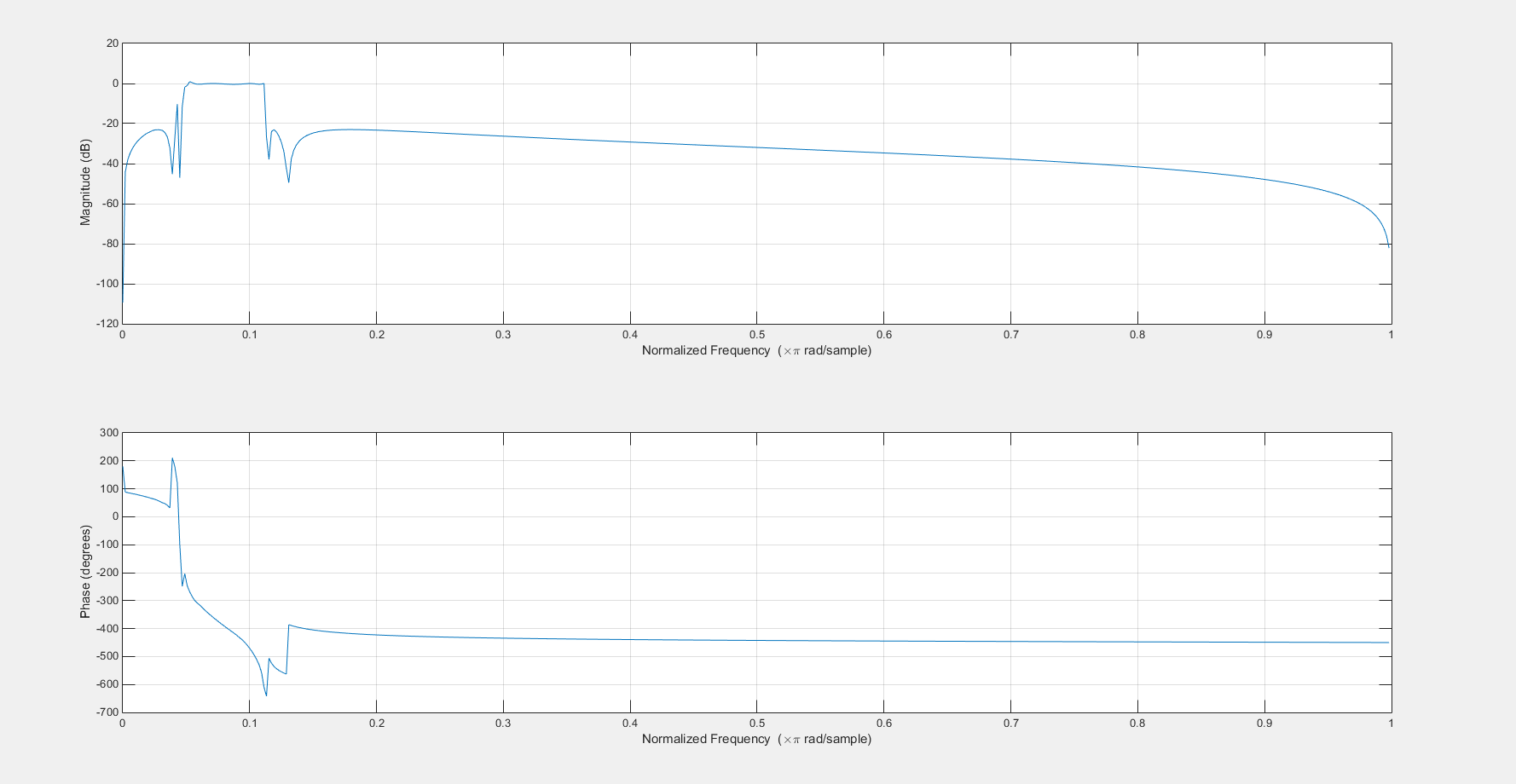


Figure Distorted frequency response of 14th order elliptic filter

When using a 4th order filter the poles of the filter are placed well within the unit circle allowing for inherent stability. The 4th order filter, from MATLAB is given by the transfer function:

Has the z-plane plot as shown in Figure 26 below:

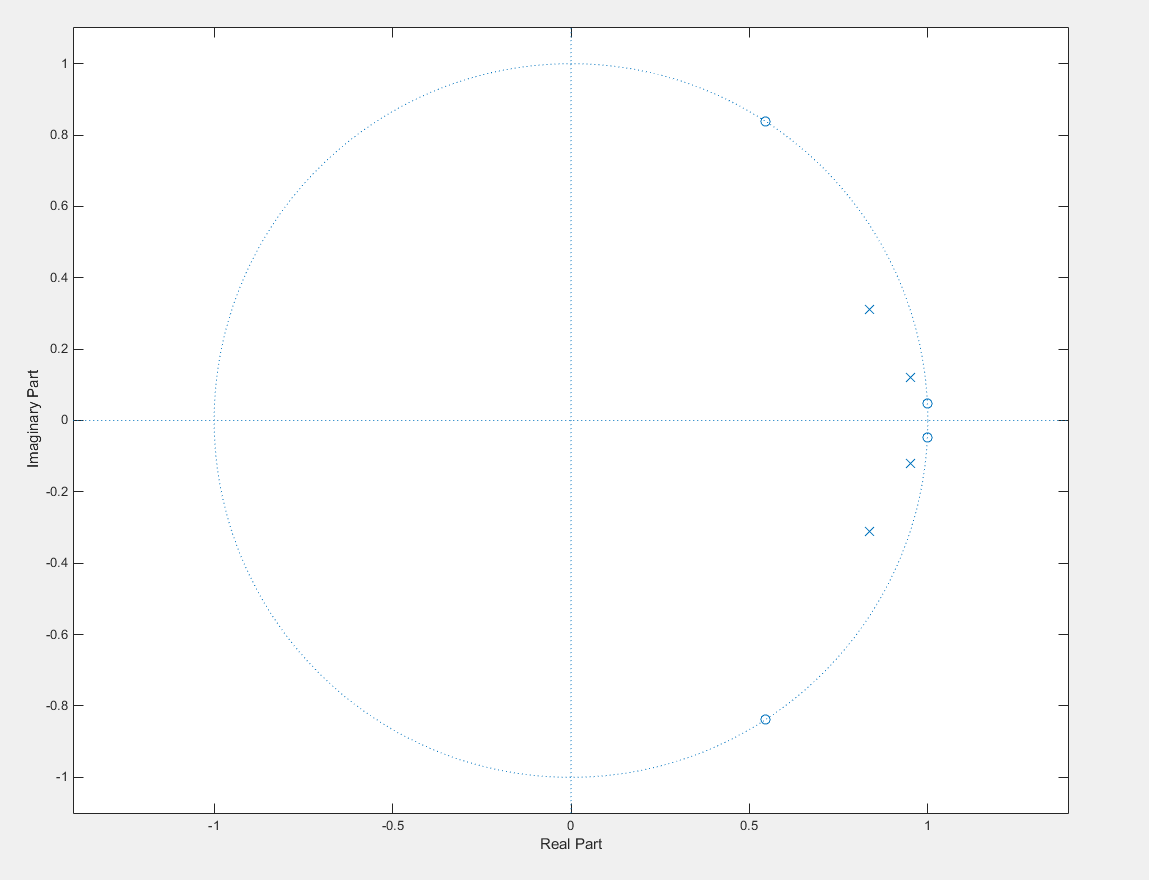


Figure z-plane plot of a 4th order elliptic filter

The plot clearly shows that the zeros are within the unit circle. When using a higher order filter however, to achieve the steep roll-off, the poles are moved closer to the unite circle. An example is shown with an 8th order elliptic filter on Figure 27 below.

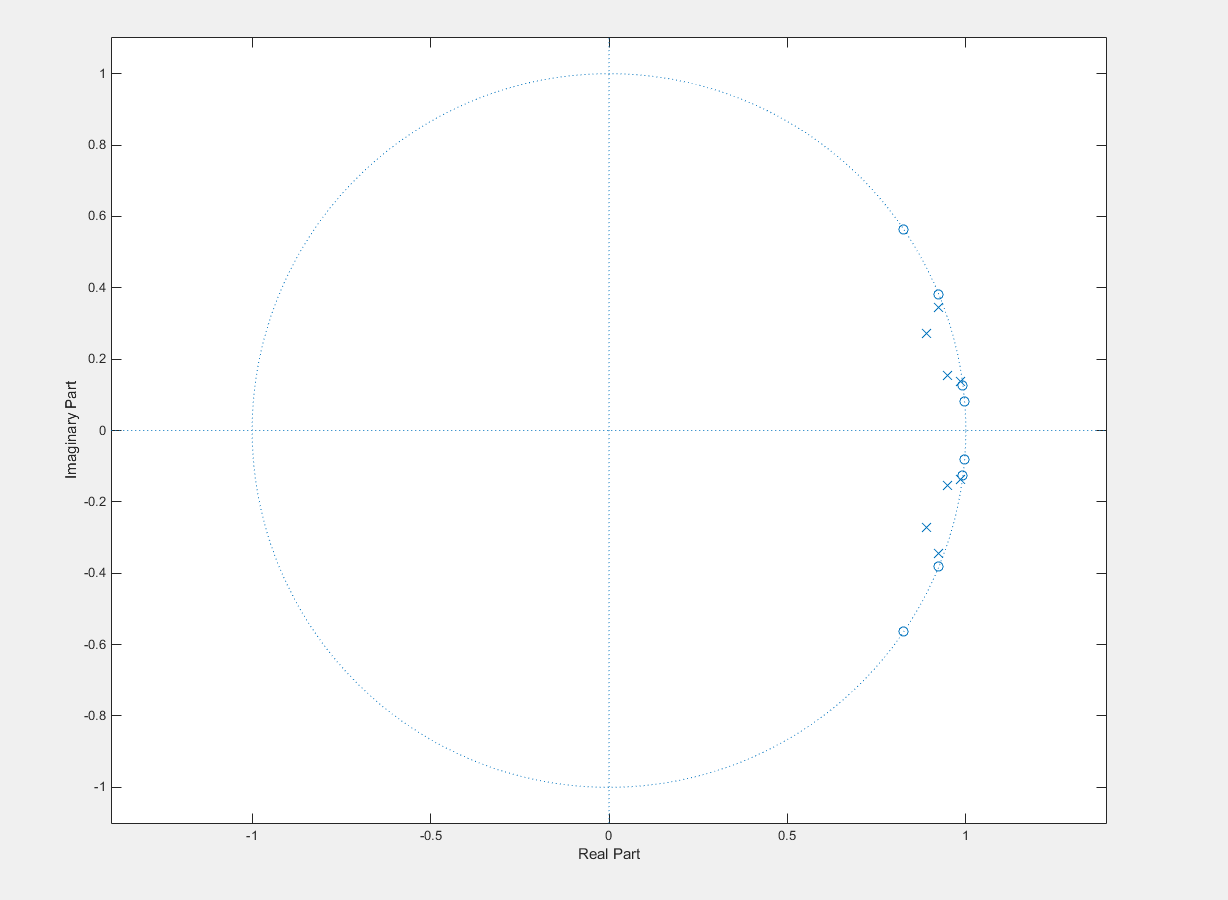


Figure Z plane plot for an 8th order elliptic filer

This however causes problems as the filter order increases. When reaching a filter of order 14 the poles are placed outside the unit circle causing instability. This is due to the double floating point precision used by MATLAB causing rounding and quantisation errors which wrongly calculates the position of the poles of the filter to be outside of the unit circle. This can be shown in Figure 28 when using a 14th order filter.

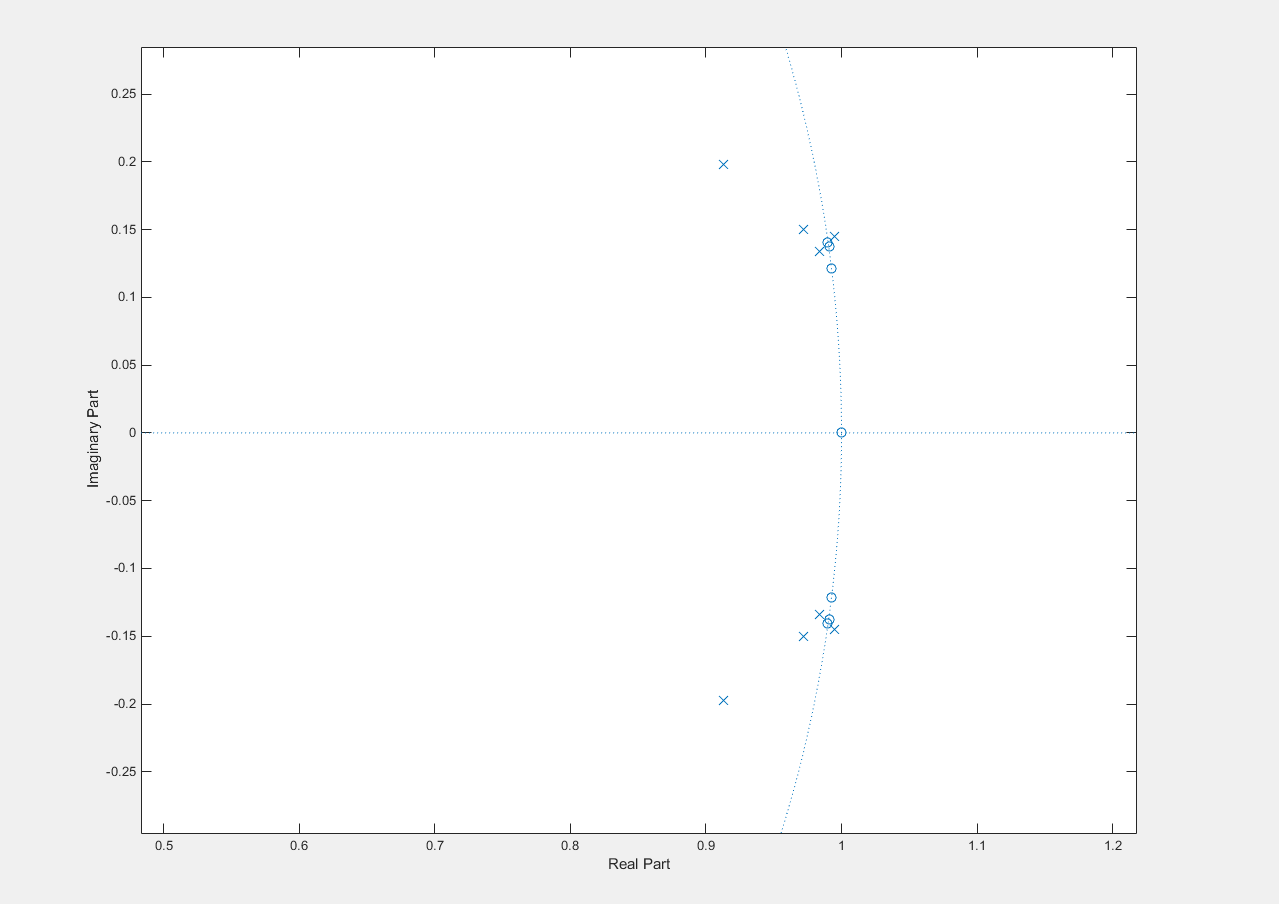


Figure z-plane plot of a 14th order filter clearly showing the zeros are placed outside the unit circle

To determine the linear equation that describes the relationship between filter order and number of cycles taken for each algorithm, the algorithms were run at several different order levels, and the time taken, in cycles, was recorded. This process was repeated for no optimisation and optimisation level 2 for two algorithms (see next section for the second algorithm).

The performance of this direct form II IIR filter was found for a number of filter orders. The results were collated in *Table 2*. A graph can also be plotted from these results (*Figure 29*), showing a clear linear relationship. When no optimization is used, it is clear that the number of instruction cycles of a filter of order n has the linear relationship 76n + 30. When using optimization 2, the relationship is of the form 6.2n +109.2

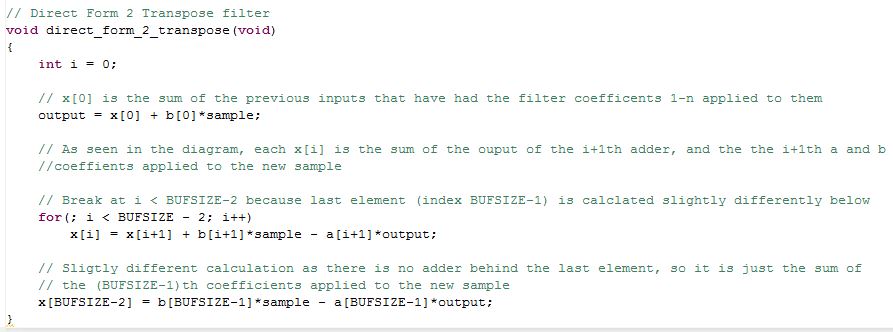
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order** | **4** | **6** | **8** | **10** | **12** |
| **No optimisation** | 334 | 486 | 638 | 790 | 942 |
| **Optimisation 2** | 136 | 145 | 157 | 171 | 185 |

*Table 2 Performance (in cycles) for a direct form II IIR filter while varying the filter order*

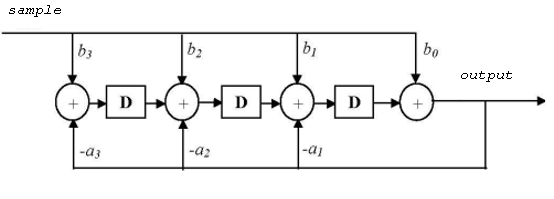
*Figure 31 Graph of number of cycles vs filter order for a direct form II IIR filter*

# Bandpass filter: Direct Form II Transposed

The IIR filter is now implemented in direct form II transposed structure. This form works by summing the chained outputs of previous adders. The output is initialised to be the sum of the first output and the first b coefficient applied to the new sample, as shown in *Figure 31* The for loop in the function sets the value of the output of each adder (represented by x[i]) as the sum of the output of the previous adder, and the ith a and b coefficients applied to the new sample. The reason it only runs until BUFSIZE-2 is that the last element, i.e. the last adder (position BUFSIZE-1 in the buffer), has no adder previous to it, so its output is calculated slightly differently: the last b coefficient applied to the new sample, minus the last a coefficient applied to the output. *Figure 30* shows the function implemented in C.

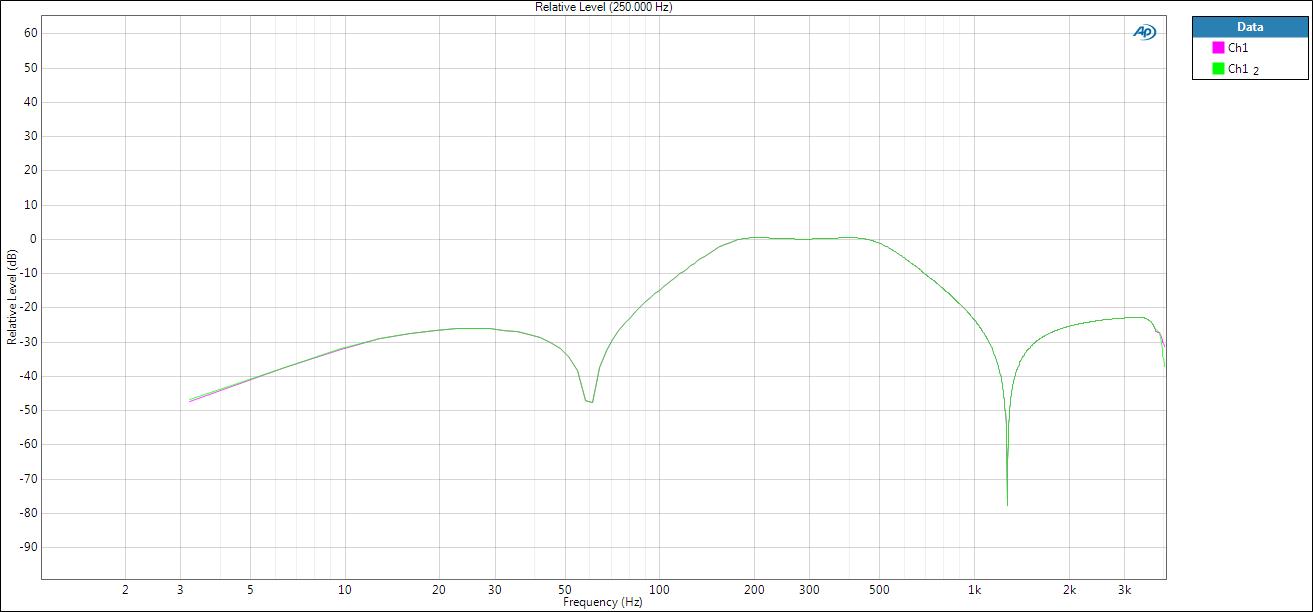


*Figure 32 Direct form II transposed function*

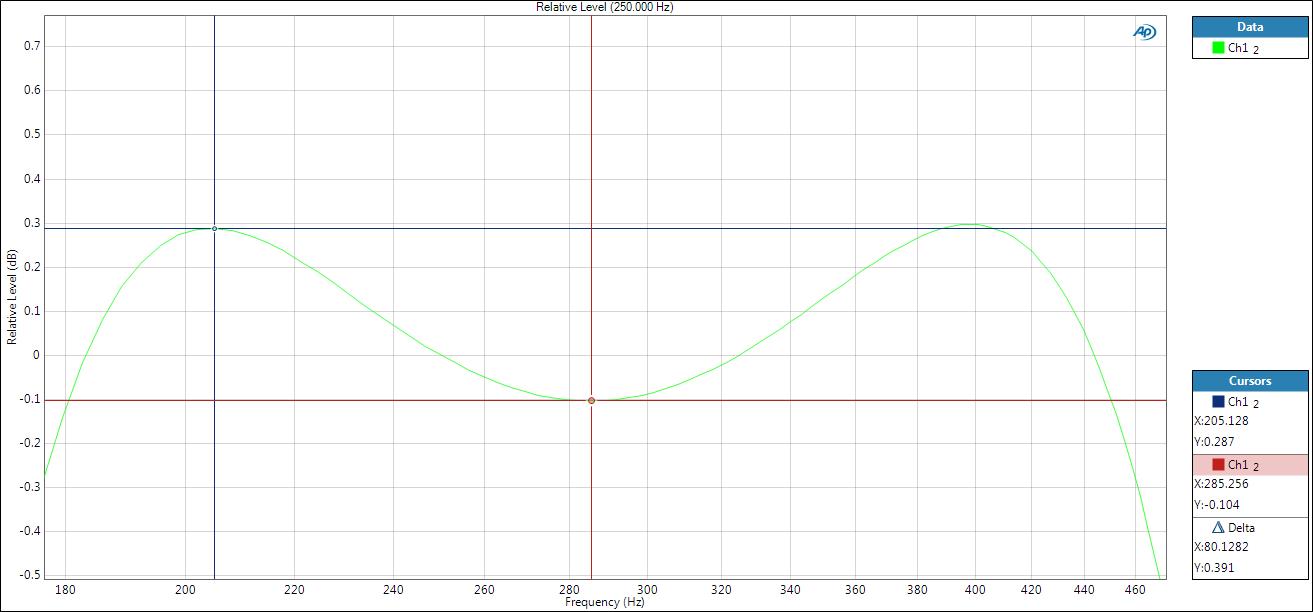


*Figure 33 Direct form II transposed form structure of the IIR filter*

When the C code is implemented, the response of the filter (*Figure 32*), closely matches that of the previous non transposed Direct Form II IIR filter. Similar characteristics are also seen due to the hardware attenuation. As can be seen from *Figure 32*, again there is attenuation at low frequencies (~7Hz) and at close to the Nyquist frequency. The ripple is calculated and is found to be well within the specification (0.391dB, shown in *Figure 33*). There is however a slight difference in responses at the close to the Nyquist frequency compared to that of the non-transposed IIR filter. The convolution summation now implemented is [[7]](#footnote-7):

**

*Figure 34 Response of Direct form II transposed*

**

*Figure 35 Pass-band ripple, within specification*

Using the data collected in *Table 3*, the graph in Figure 34was produced, showing that there is, indeed, a linear relationship. This is especially true for the non-optimised processes, where the number of cycles increased by exactly the same amount for each data point (152 for Direct Form 2 (*Figure 29*) and 118 cycles for Direct Form 2 Transposed). The optimised processes' first two data points are slightly less linear, but from the this data point onward, they appear to settle into a linear trend, growing at the same number of cycles (14 cycles for Direct Form 2 and 10 cycles for Direct Form 2 Transposed).

For the Direct Form 2 Transposed, the relationship between order and number of cycles is 59n + 38 for a non-optimised function. For direct form 2 transposed when using optimisation level 2, the relationship between order and number of cycles is 5.2n + 107.8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order** | **4** | **6** | **8** | **10** | **12** |
| **No optimisation** | 274 | 392 | 510 | 628 | 746 |
| **Optimisation 2** | 129 | 138 | 160 | 170 | 180 |

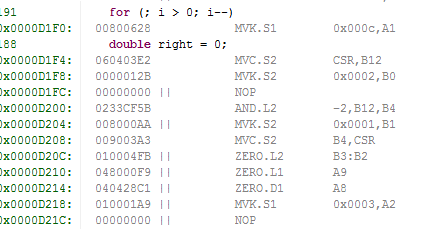
*Table 3 Results for the Direct Form II transposed IIR filter implementation*

Figure Direct Form II transposed graph of number of cycles vs filter order showing a linear trend

# Performance Comparison

Using the previous results and the general algorithms, the performance of the two filters can be compared. The transposed form is seen to be faster than the non-transposed form. Looking at the non-optimised plots, the transposed for has a much lower x coefficient than the non-transposed form. This means that it will stay consistently faster as order is increased. This is due to the differences in their for loops: in the non-transposed algorithm, we are performing three distinct operations (two multiplications and a shift of the buffer), whereas in the transposed version, we are performing similar operations, but have grouped them into one statement. Since the instructions are not being optimised, the compiler is less efficient at managing its memory with the non-transposed algorithm, since it is not aware that the values it is calculating will be summed at the end. In the transposed algorithm, however, this is more clearly communicated to the compiler.

Looking at the equations for the -o2 plots, there is slightly more overhead in the Direct Form 2 algorithm than the transposed version (109.2 vs 107.8). Looking at the C code, the non-transposed algorithm has more initialisations, which is likely what is contributing to that increase, however, it is difficult to accurately predict why the optimised code behaves as it does, since the compiler alters the code in non-obvious and sometimes counter-intuitive ways to maximise speed. Looking at the disassembly, as shown in *Figure 36*, in Code Composer (for a 12th order filter), we find that the transposed form makes use of 42 parallel-execution instructions in its for loop, as opposed to the non-transposed form's 28, which may be a contributing factor, as it reduces the number of cycles needed to run one iteration of the loop.



*Figure 37 Example of the parallel-execution instructions, preceded by ||*

1. https://upload.wikimedia.org/wikipedia/commons/thumb/e/e0/1st\_Order\_Lowpass\_Filter\_RC.svg/2000px-1st\_Order\_Lowpass\_Filter\_RC.svg.png [↑](#footnote-ref-1)
2. Real Time Digital Signal Processing Lab 5 Coursework notes – Implementation of IIR Filters page 5 [↑](#footnote-ref-2)
3. http://www.electronics-tutorials.ws/rc/rc5.gif?81223b [↑](#footnote-ref-3)
4. Real Time Digital Signal Processing Lab 5 Coursework notes – Implementation of IIR Filters page 6 [↑](#footnote-ref-4)
5. http://www.radio-electronics.com/info/rf-technology-design/rf-filters/elliptic-cauer-rf-filter-basics.php [↑](#footnote-ref-5)
6. http://www.fizyka.umk.pl/~daras/pfc/filtr\_eliptyczny.pdf [↑](#footnote-ref-6)
7. Real Time Digital Signal Processing Section 6 - Filters and their Design P11 [↑](#footnote-ref-7)