RTDSP Lab 5

Loek Janssen – lfj10@ic.ac.uk
Sebastian Grubb – sg3510@ic.ac.uk

Contents

Filter design using Tustin transform

Coefficient derivation

An analogue filter based on Figure 1 was implemented in the discrete z-domain.

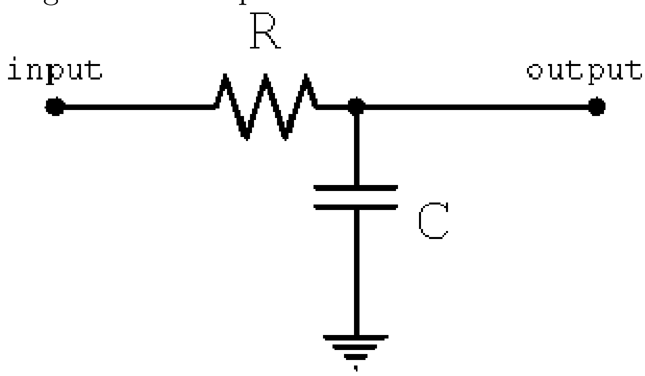


Figure 1. Low pass RC filter with R=1kΩ and C = 1µF

We know that the continuous filter in Figure 1 has equation:

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

To convert this filter to the z-domain we do the Tustin transform to transform a continuous system in s-domain to a discrete one. It is the first-order approximation of the more precise transform $z = e^{sT}$, where T is the sampling time.

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

We know that $T = \frac{1}{8000}[s]$ as the sampling frequency is 8000Hz and R=1kΩ and C = 1µF – substituting these values in we get:

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

A IIR filter's transfer function is given by:

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

Thus the coefficients are:

$b_0 = \frac{1}{17}, b_1 = \frac{1}{17}$ and $a_1 = -\frac{15}{17}$. (We also have $a_o = 1$, though this will always be true)

Implementation in C

We know that the convolution sum of an direct form I IIR filter is:

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

The idea is thus to implement this in C and use the coefficients found earlier. If we make a for loop we will have to handle the case for $i = 0$ separately to allow for one for loop to handle the rest of the convolution as one, i.e. we can rewrite it as:

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

We use the maximum between M and N and set the rest of the a and b coefficients to 0 to allow for this.

From this equation rearrangement we get the following code:

```
double base_IIR(){
      //reads the global variable sample and process it
      //using the IIR coefficients to return a filtered
      //output.
      output=b[0]*sample; // perform first coefficient multiplication
      x[0]=sample; //store input in x[0]
      //loop to multiply each x and y past values by their respective coefficients
      for (i=order;i>0;i--){
            output += x[i]*b[i]-y[i]*a[i]; //main IIR convolution sum
            y[i]=y[i-1];//shift y to represent delay element
            x[i]=x[i-1];//shift x to represent delay element
      }
      x[1]= sample; //store input in the first delayed x element
      y[1]= output; //store output in first delayed y element
      return output; //return output value
}
```

Analysis of results

Data for the frequency response was obtained using an Audio Precision APX515 unit. To be able to take full advantage of the data we also collected the data for the filter of the DSK and the codec to then subtract it from the obtained results. For example in Figure 2 the data from the left graph would be subtracted by the data of the rough graph to obtain the "correct" frequency response to be properly able compare with matlab data, thus givin

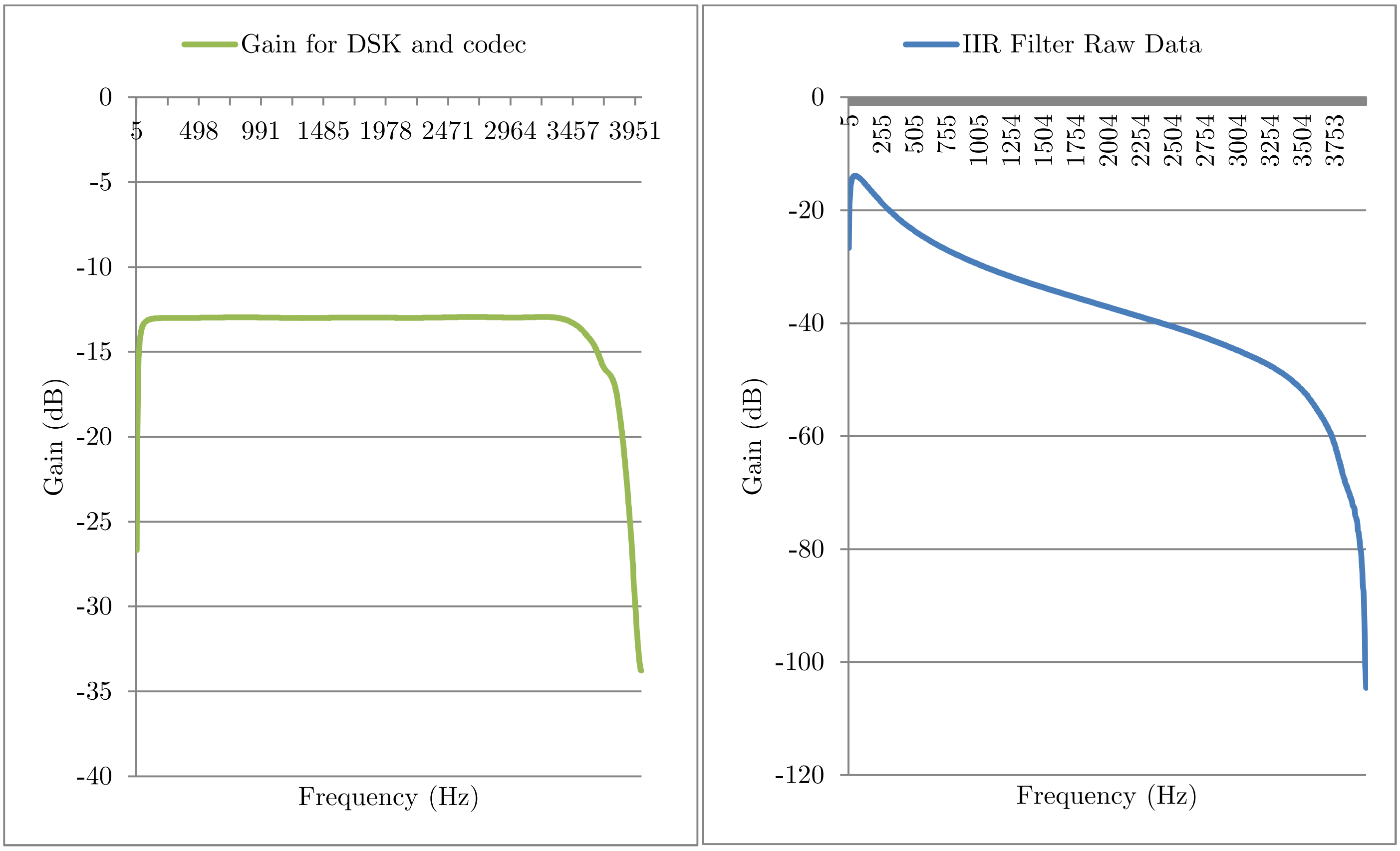


Figure 2. On the left is a graph of the frequency response for the board and employed codec. This achieved by placing the function mono_write_16Bit(mono_read_16Bit()) in the interrupt function. On the right is a graph of the unprocessed IIR filter implementation of the continuous filter represented in Figure 1.

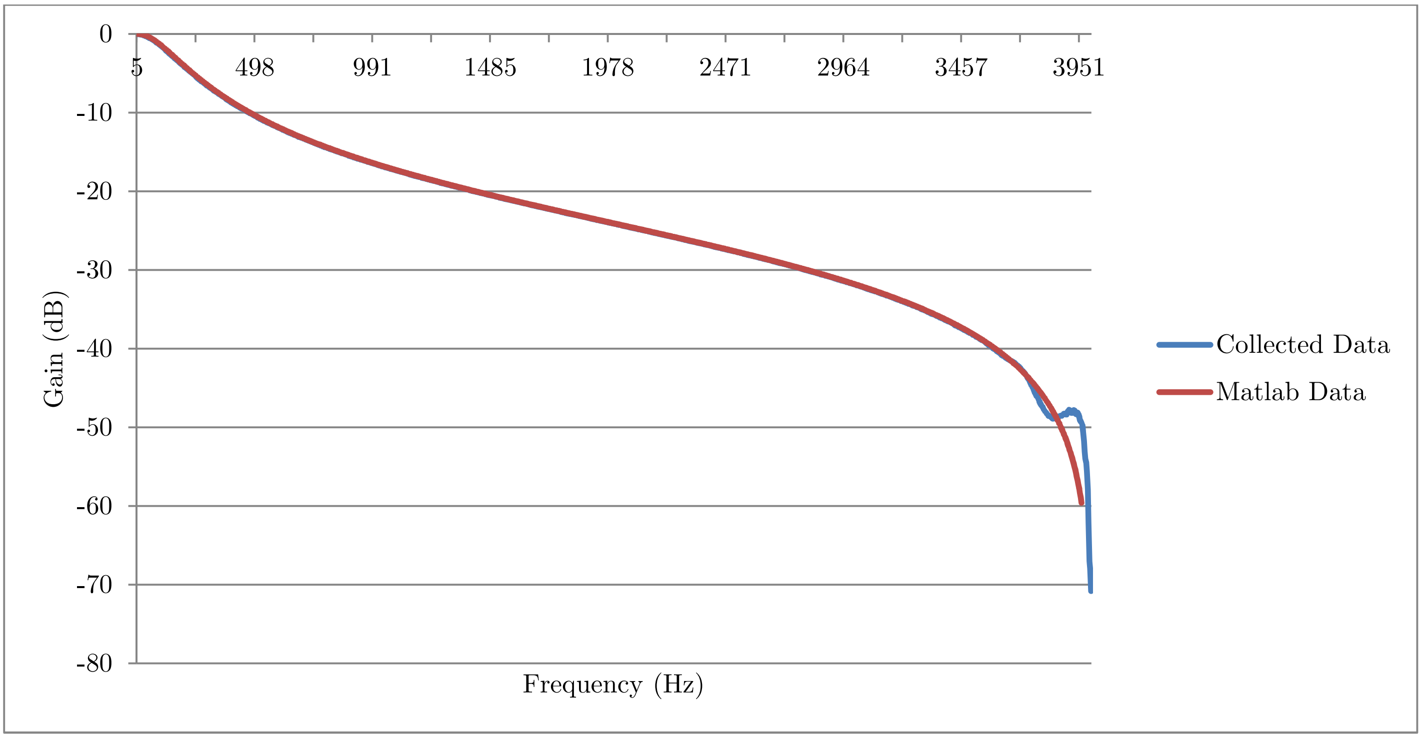


Figure 3. Graph of collected processed gain data to take into account the effects of the extra filter plotted against the expected data.

From Figure 3 we see that the discrete filter closely matched the expected output and that the results only diverge at higher frequencies. This is mainly due to the way in which the low pass filter works, which was found to exhibit divergence at high frequencies.

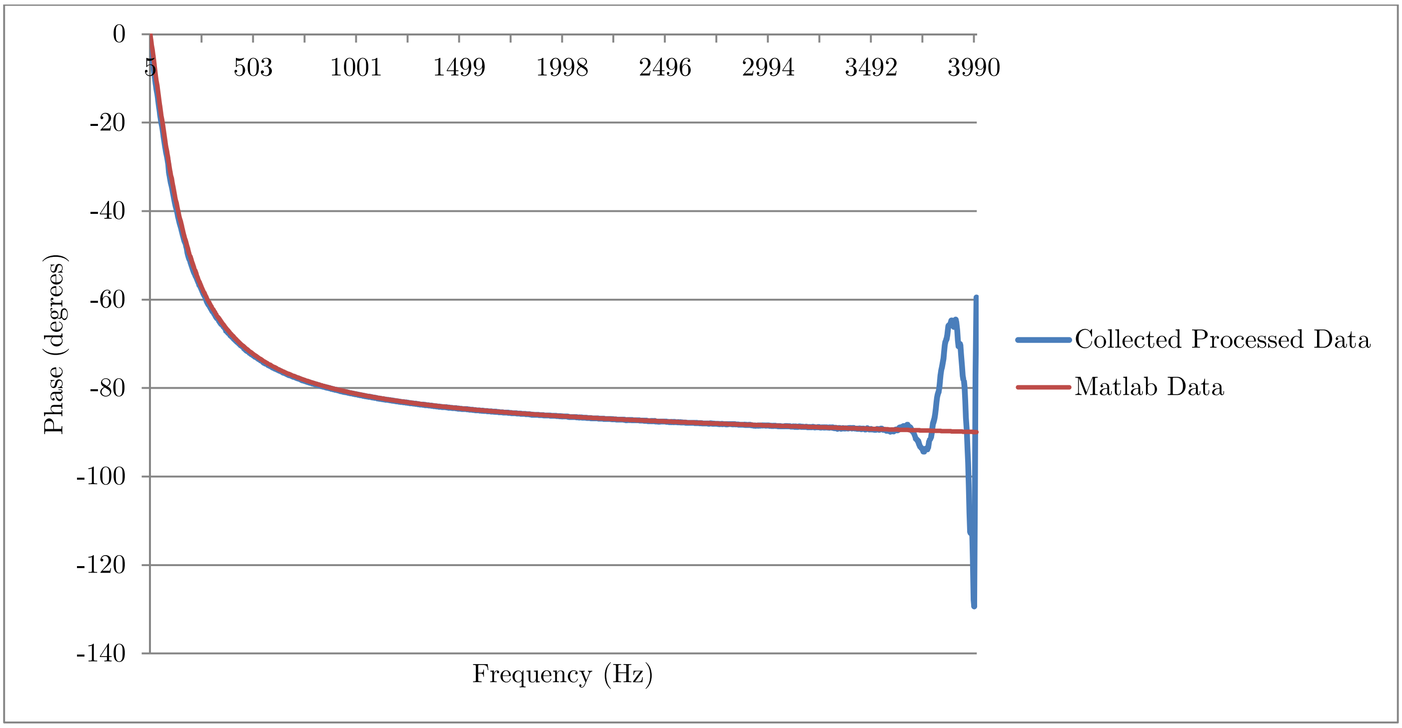


Figure 4. Graph of collected processed phase data to take into account the effects of the extra filter plotted against the expected data.

In Figure 4, which compares the expected data to the obtained one, we also find the results to match except at very high frequencies where the results diverge due to the codec filter.

The 3dB cut-off frequency was found to be 158Hz (this is taken from the processed data) which matches the cut-off frequency of the continuous filter defined by $f_{3dB\ continuous} = \frac{1}{2\pi \times RC} = \frac{1}{2\pi \times 10^{-3}} \cong 159.15Hz$ which in discrete Tustin transformed z-domain is equivalent to $\omega = 2\pi \times f_{3dB\ discrete} = \frac{2}{T}\tan^{-1}\left(\frac{2\pi \times f_{3dB\ continuous} \times T}{2}\right)$, where T is $\frac{1}{f_{samp}} = \frac{1}{8000}$. Thus substituting all the values in we get $f_{3dB\ discrete} = 158.94\ Hz$ which is a very minimal error.

We can also estimate the time constant, which in continuous domain is $\tau = RC$.

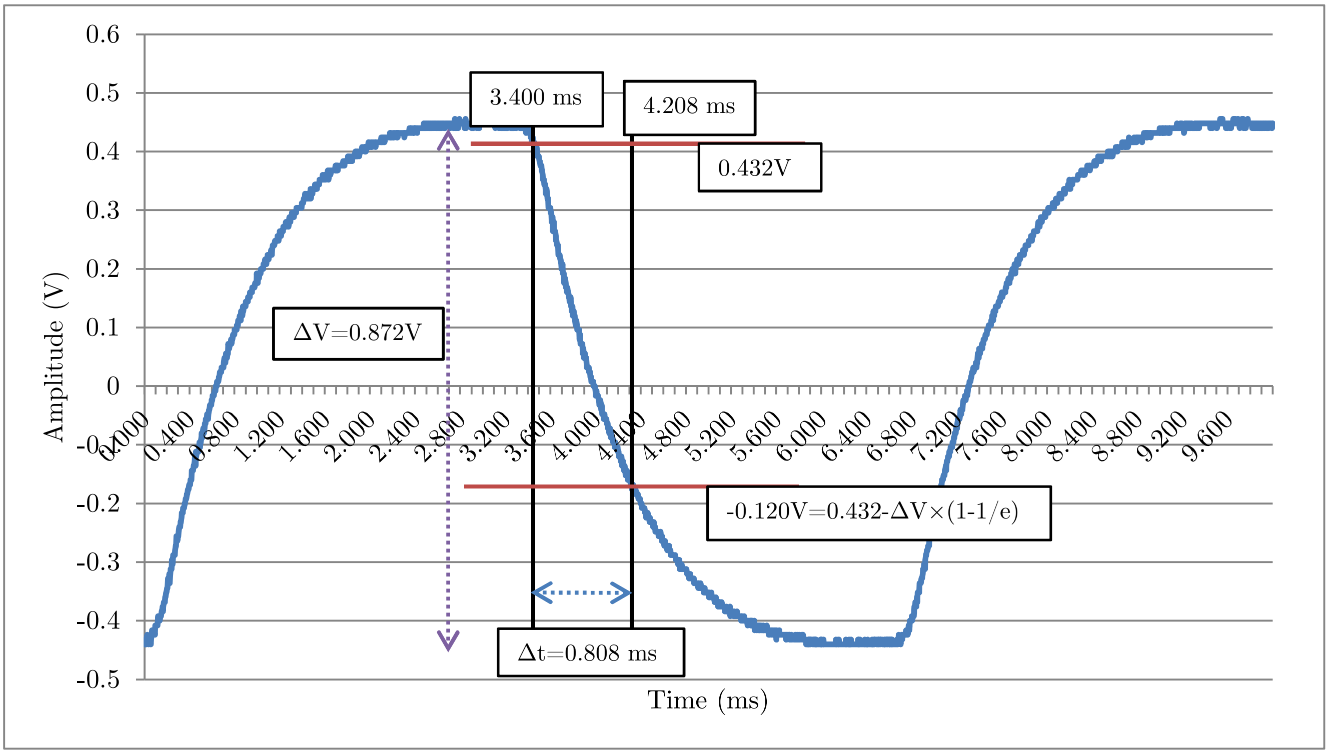
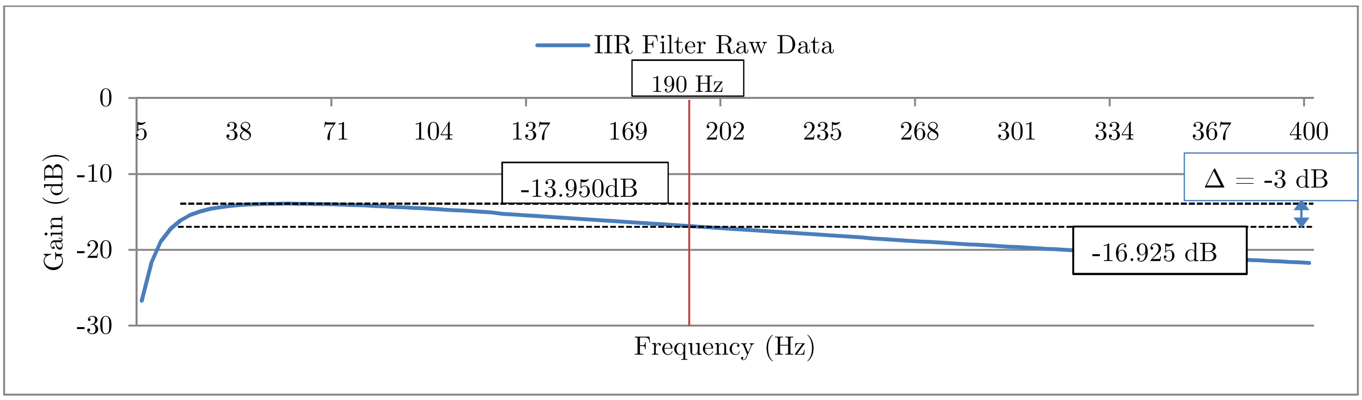


Figure 5. Graph of collected data of the output of an oscilloscope for a 150Hz square wave as input. By calculating the time for the decay to decrease by $1 - \frac{1}{e}$ a time constant, τ, can be calculated.

We find the time constant to be 0.808 ms, which is the time for the gradient to decay by $\left(1 - \frac{1}{e}\right) \cong 63.2\%$ to its final value. The expected value is $\tau = RC = 1k\Omega \times 1\mu F = 1ms$ for the continuous filter, though another expected value would be $\tau = \frac{1}{2\pi \times f_{3db}} = \frac{1}{2\pi \times 158Hz} = 1.007ms$ using the f_{3dB}if we use the adjusted data collected for the discrete filter. However the unprocessed collected data (which also takes the low and high pass board filter into account) has a 3dB point at 190Hz which gives us a time constant of $\tau = \frac{1}{2\pi \times 190Hz} = 0.8377ms$ which is closer to the observed time constant and explains the deviation from the expected time constant.



Thus the discrete filter keeps most properties of the continuous filter; however the time constant is slightly off which may be due to the way the oscilloscope collects data (the oscilloscope only gives data in 0.08V steps and time to 0.04ms steps).

Digital and Analogue response:

With the removal of the DSK effects we can now compare the frequency response of our IIR filter against that which would be expected from an analogue RC filter. From earlier we determined the transfer function of our RC filter as

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

Which for matlab calculations we convert to the frequency domain $s = j\omega$. Then in order to calculate the gain in decibels the resulting equation is

$$20\log_{10}(|\frac{1}{1 + j\omega RC}|)$$

While the phase of the RC filter is found using the matlab function angle, which merely finds the angle for complex numbers in radians. This was then converted to degrees so as to match the taken measurements. The frequency response of a RC filter was then graphed in matlab using the code shown below. The frequency was measured from 5 – 4000 Hz at intervals to create 1216 points so as to match the number of samples taken by our APX spectrum analyser.

```
%Gain stored in variable x, phase in variable 'phase'

f= 5:(125/38):4000; %From 5 to 4000 producing 1216 points
x = 20*log10(abs(1./(1+i*2*pi.*f/1000))) %Gain of filter in decibels
y = angle(1./(1+i*2*pi.*f/1000)) %Phase of filter in radians
phase = y*180/pi %Phase converted to degrees
%Plotting of gain and phase graphs
subplot(2,1,1);
plot(f,x);
grid on;
subplot(2,1,2);
plot(f,phase);
grid on;
```

The resulting data obtained is then compared on figure 5 to show the differences between the response of an analogue RC filter and of our mapped digital IIR filter, for both the expected IIR response on matlab and that measured from the DSK board using the APX. As we can see there is a significant divergence from 1000 Hz. This is most likely due to the our filter not being able to precisely match that of our analogue filter due to the nyquist frequency effects and also because of approximations made in the Tustin transform causing frequency warping. As the Digital filter must give a mirrored response around the nyquist frequency the gain value drops off.

Indeed, with the Tustin transform, as the Nyquist frequency is reached, we are effectively putting $\frac{\pi}{2}$ into a tan function: $\omega = \frac{2}{T_s}\tan\left(\frac{\omega_p T_s}{2}\right)$ thus $\lim_{\omega_p \to 2\pi\times 4000Hz} \frac{2}{T_s}\tan\left(\frac{\omega_p T_s}{2}\right) = \lim_{x\to\frac{\pi}{2}} \frac{2}{T_s}\tan(x) = \infty$ which explains why the RC filter will have a different response for near-Nyquist frequencies than the discrete filter obtained via the Tustin transform.

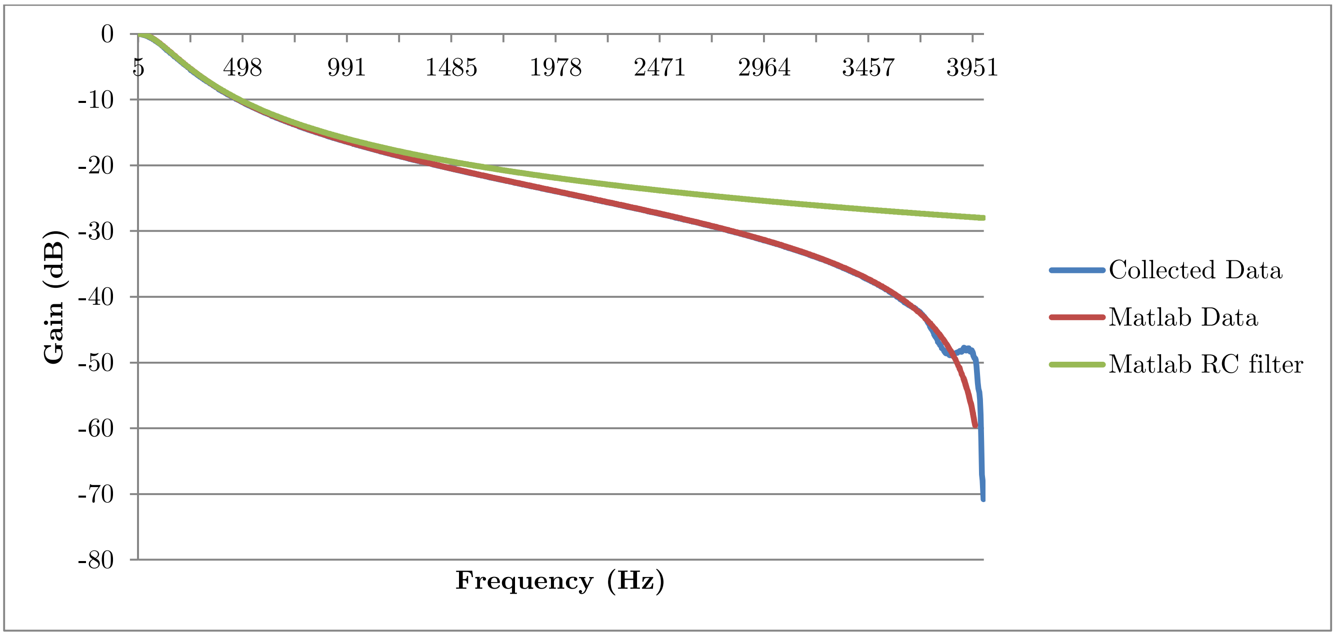


Figure 6: Graph of collected gain data against expected IIR filter and expected analogue RC filter

The phase measured data was also overlaid with our matlab simulations for an IRR and RC filter in figure 6. The shift in the difference from the matlab data is most likely due to frequency warping as discussed above, while as the filter approaches the nyquist frequency the image of reconstructed sinewave (by the DAC) is no longer cut-off by the anti-aliasing filter. With additional sinewaves of larger frequencies than 4 kHz, the phase is dramatically altered. This would also explain why matlab data for the IIR filter also fails to predict the alteration.

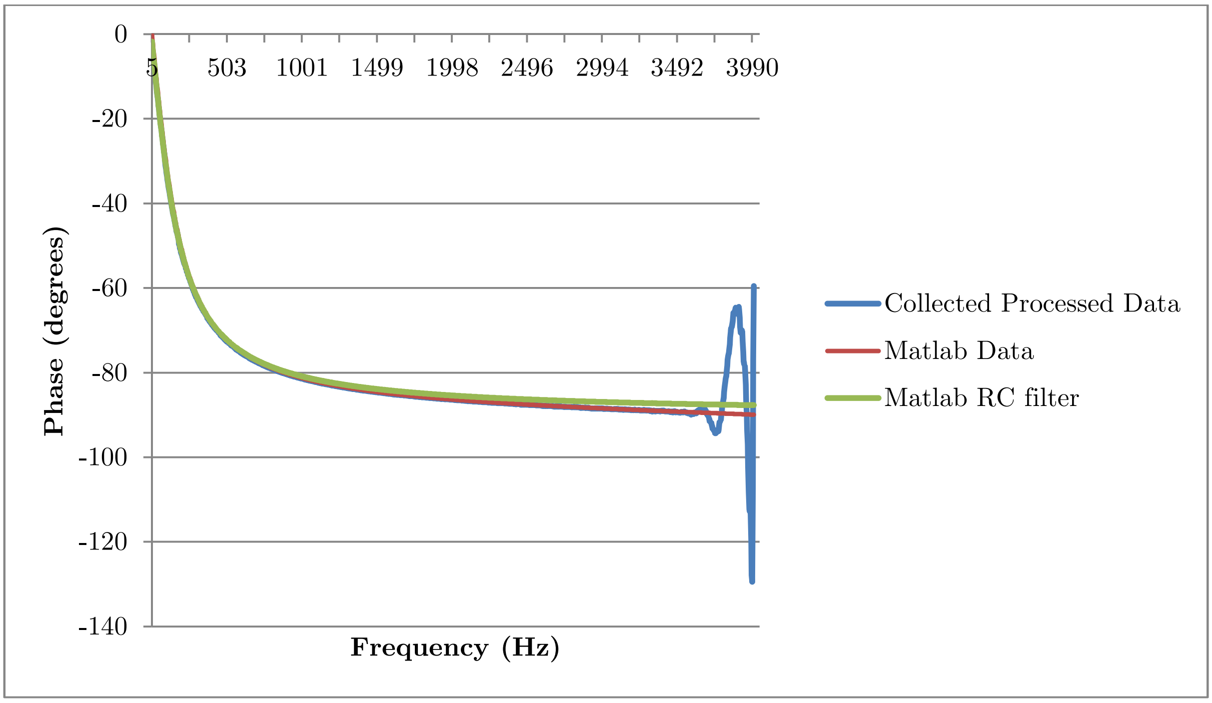


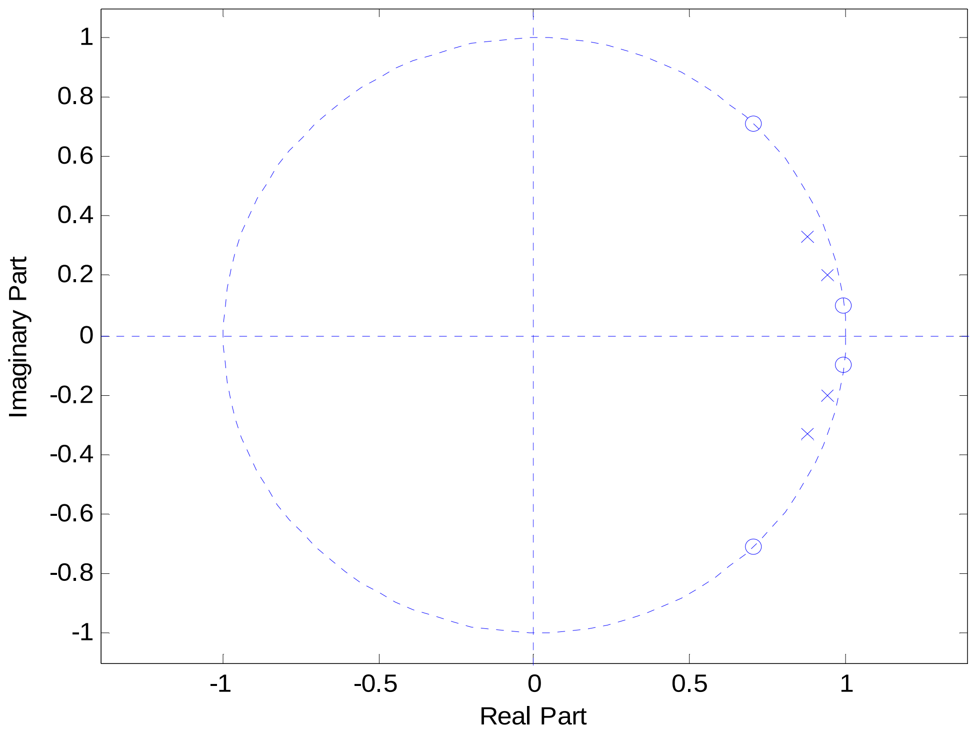
Figure 7: Graph of collected phase data against expected IIR filter and expected analogue RC filter

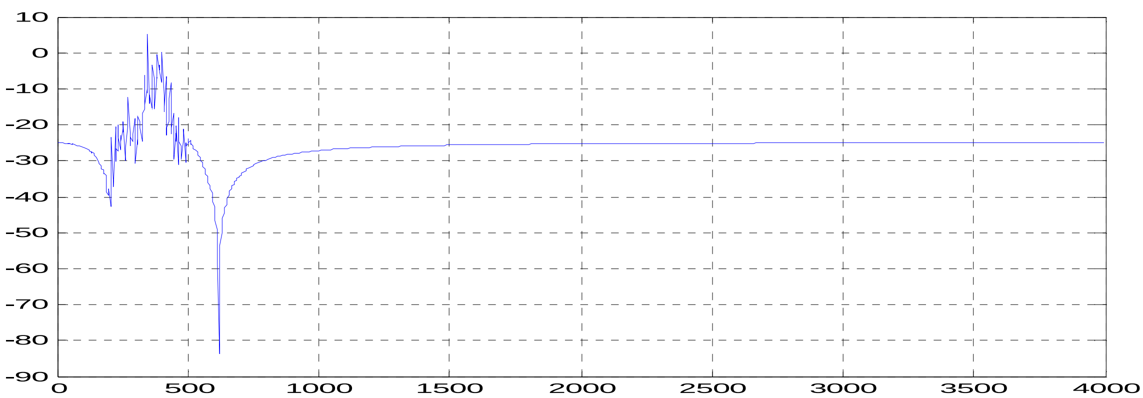
Direct form 2 implementation

IIR Elliptic Filter

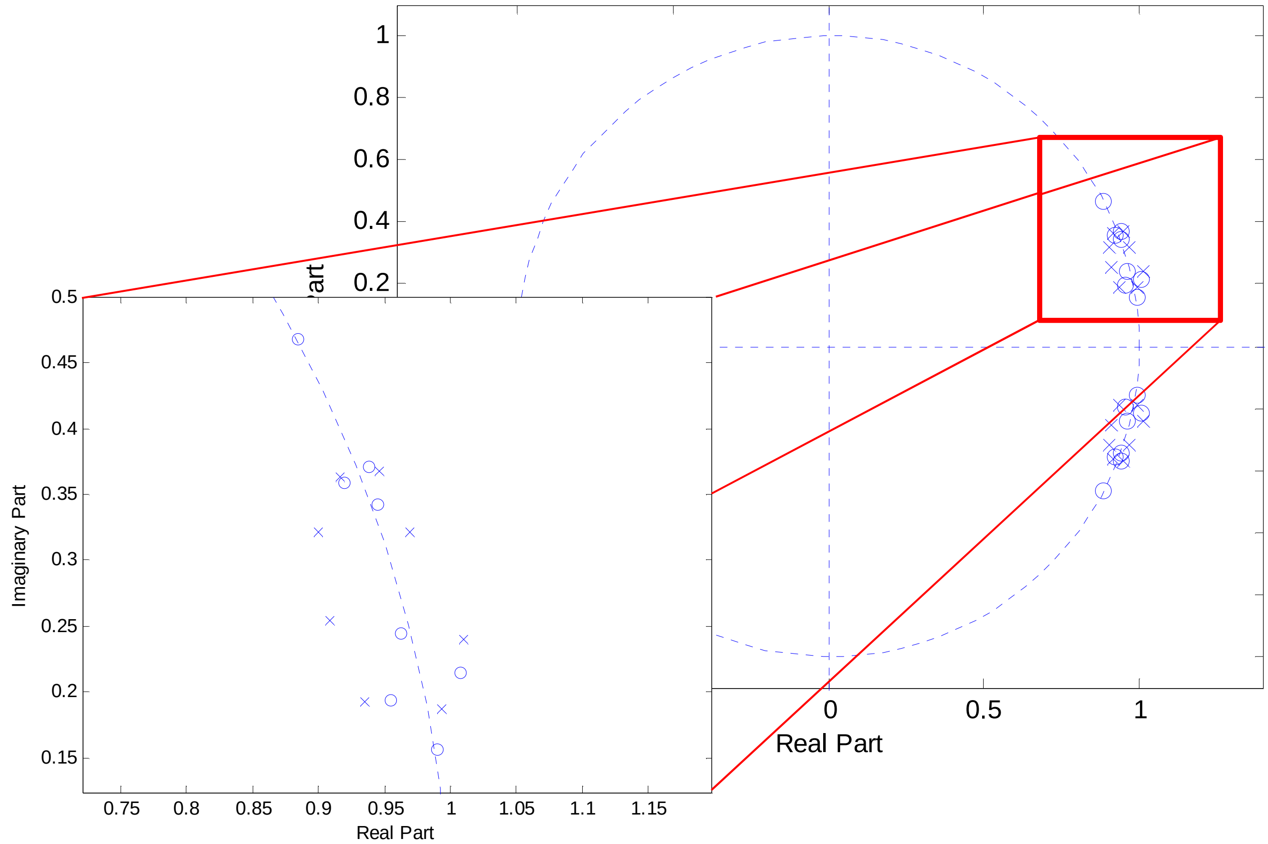
Elliptic filters have a very sharp roll-off compared to the other common IIR filter designs, however they have ripple in both the stop and passband. Like a Chebyshev the poles are placed in an ellipse in order to produce a sharper roll-off, however this is what contributes ripple in a pass band. Then with the addition of zeros in the transfer function the Elliptic filter gets an even sharper roll-off in return for introducing ripple in the stopband.

The placement of of poles and zeros is shown in figure 8 and shows the design of our specified 4th order filter using matlab (see below). The poles are in an ellipse and the zeros are placed on the unit circle. However as matlab can only place the points to the IEEE double precision floating point standard the zeros are actually just outside of the unit circle. This finite precession means that the actual filter implemented is not exactly the correct. For low order filters this does not cause too many problems but at higher order filters the poles approach the unit circle and so this imprecision causes instability, as can be seen in figure 9, which shows the gain of a 16 order elliptic filter.





Studying the z-plane of our order 16 filter in figure 10 we can quickly see why there is instability, as matlab using double floating point cannot create poles in the precise enough locations for the filter, and the rounding errors places them outside the unit circle, causing instability.



Finding values in matlab

To implement a direct form 2 implementation filter values were chosen first. To achieve this the function ellip() was used with the following requirements in mind:

Order: 4th
Passband: 280-460 Hz
Passband ripple: 0.5 dB
Stopband attenuation: 25 dB

For this, the following matlab code was written:

```
clc; %clear screen
% get coefficient values
% passband values must be normalized to Nyquist frequency
% and the function should be used such as:
% ellip(order, ripple (dB), stopband attenuation (dB), passband frequencies)
[b,a] = ellip(2,.5,25,[280/4000 460/4000]);
figure(1); %intialize first window
freqz(b,a,1200,8000); %plot graph of gain and phase
figure(2); %intialize second window
% view plot of zeros and poles to understand potential
% effect of precision of coefficients on filter performance
zplane(b,a);
formatSpec = '%1.16x,'; %set format to high precision to avoid rounding issues
fid=fopen('ccs_proj\RTDSP\coef.txt','w'); %open file
% define order of filter+1 here
```

```
% simpler and more intuitive way than
% playing around with N in C
fprintf(fid,'#define N %d\n', length(a));
%define a and b coefficient arrays
fprintf(fid,'double a[] = {');
fprintf(fid,formatSpec,a(1:length(a)-1));
fprintf(fid,'%1.16x};\n',a(length(a)));
fprintf(fid,'double b[] = {');
fprintf(fid,formatSpec,b(1:length(b)-1));
fprintf(fid,'%1.16x};\n',b(length(b)));
fclose(fid); %close file
```

Notice that the coefficient file also takes care of handling the size of N, thus avoiding the need to use calloc() in C to take into account the carrying amount of coefficients to support different orders. This also means that since at compile time the length is determined, optimisations, such as loop unrolling, can be done and this creates more efficient code.

Implementation in C

A type II direct form IIR filter resembles that of Figure 8. As can be seen only half the delay units than direct form 1 can be used which makes for more efficient code as only one delay element has to be recorded.

From Figure 11 discrete-time-domain equations can be constructed by focusing on the delay elements v_i:

$$v[n] = x[n] - a_1 v[n-1] - a_2 v[n-2] - \cdots - a_N v[n-N]$$

$$y[n] = b_o v[n] + b_1 v[n-1] + b_2 v[n-2] + \cdots + b_N v[N]$$

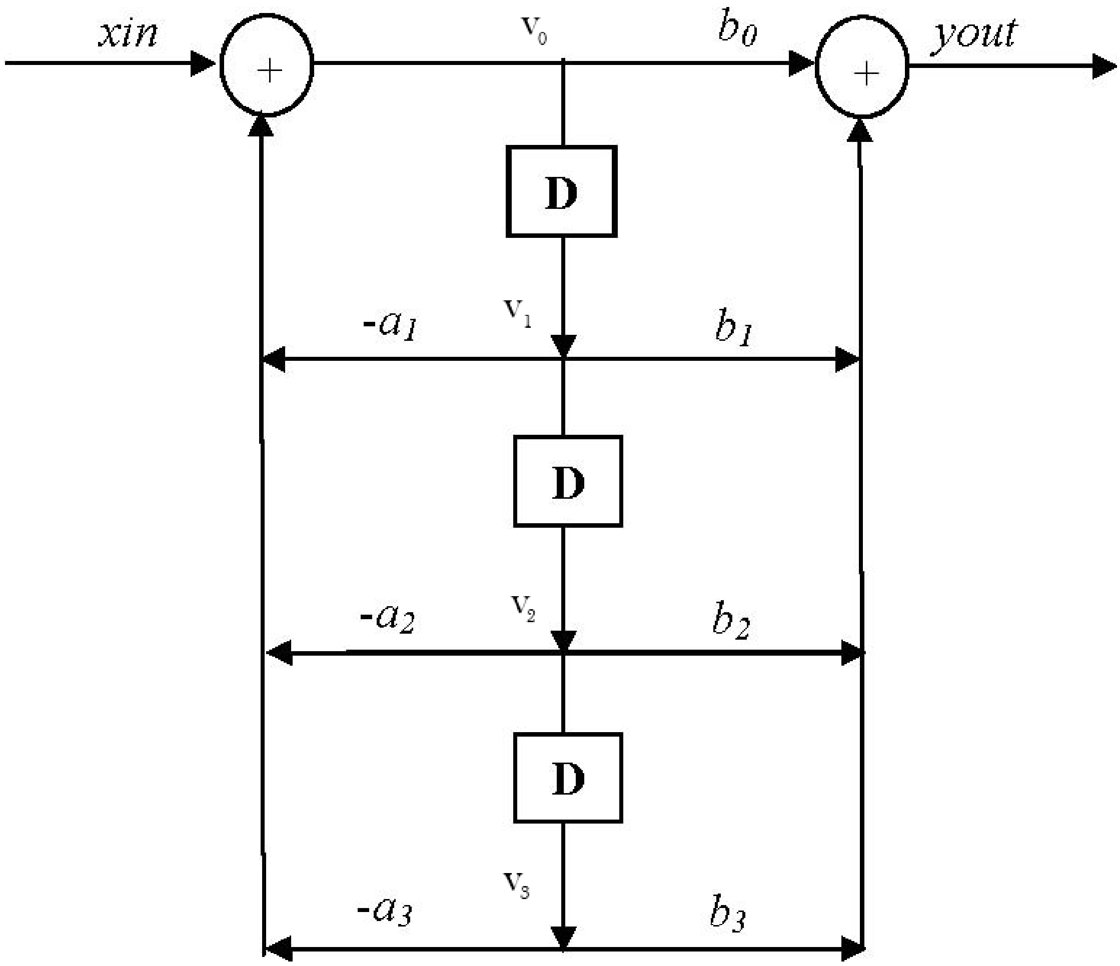


Figure 11. Image of a direct-form II IIR filter implementation. A direct form II has half the number of delay units than a direct form I IIR filter.

From this C code for a direct form II IIR filter can be made, which works by having an array v[] representing the delay elements and shifting the elements up by 1 for each iteration, thus acting as one of the delay elements.

```
double base_IIR_2(){
	//this function fulfils IIR Direct form ii
	//filter - this version does not implement a circular buffer
	//a possible efficiency tweak
	//use:reads global variable sample and returns filtered value
	v[0]=sample; //write input to v[0]
	output = 0; //reset output to accumulate result
	//loop for all values of v to accumulate them to output
	for (i=order;i>0;i--){
		v[0] -= a[i]*v[i]; //accumulate a coefficients
		output += b[i]*v[i]; //accumulate to output
		v[i] = v[i-1]; //shift v[i] data down to represent the delay elements
		//in a IIR Direct Form 2 filter
	}
	output += b[0]*v[0]; //write final values to output
	return output; //return filtered value
}
```

Performance

The frequency response of the order 4 filter was found to be nearly overlapping the frequency response expected in matlab as shown in figure 12.

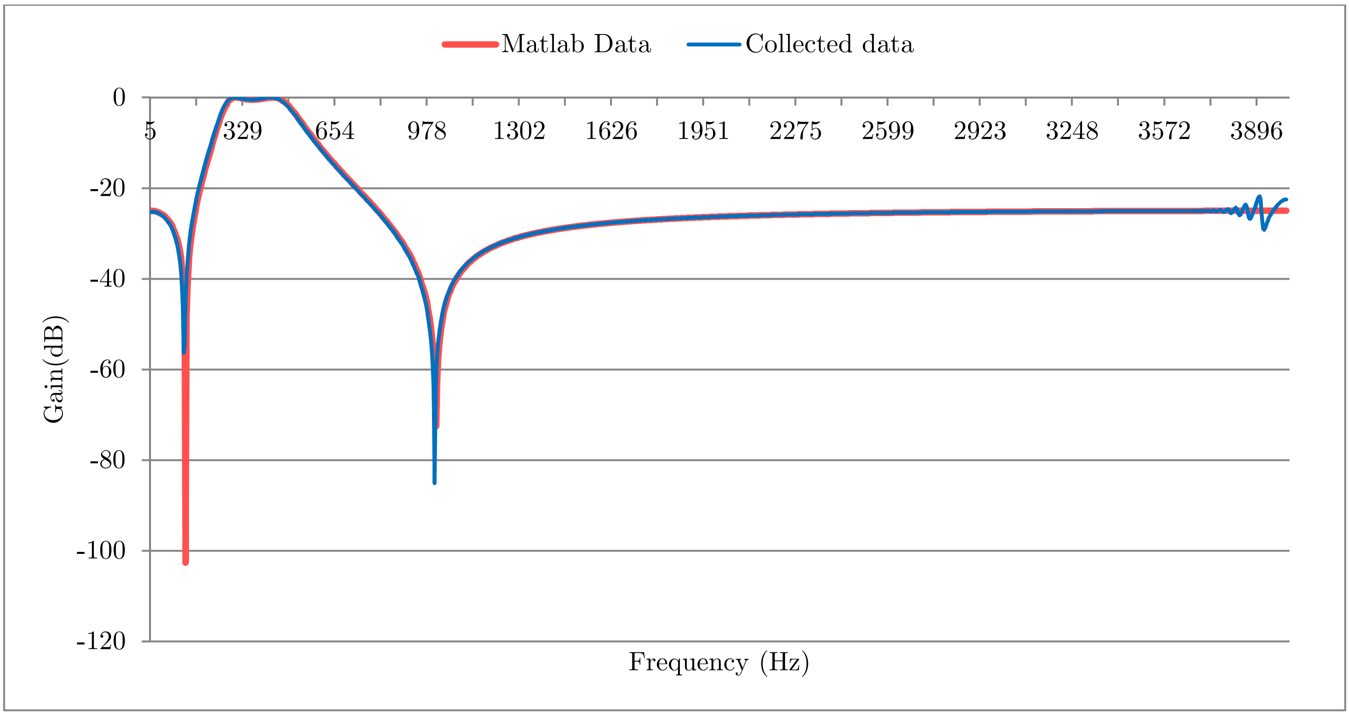


Figure 12. Graph of gain response with respect to frequency for an elliptic filter of order 4 using the given specifications for a Direct form 2.

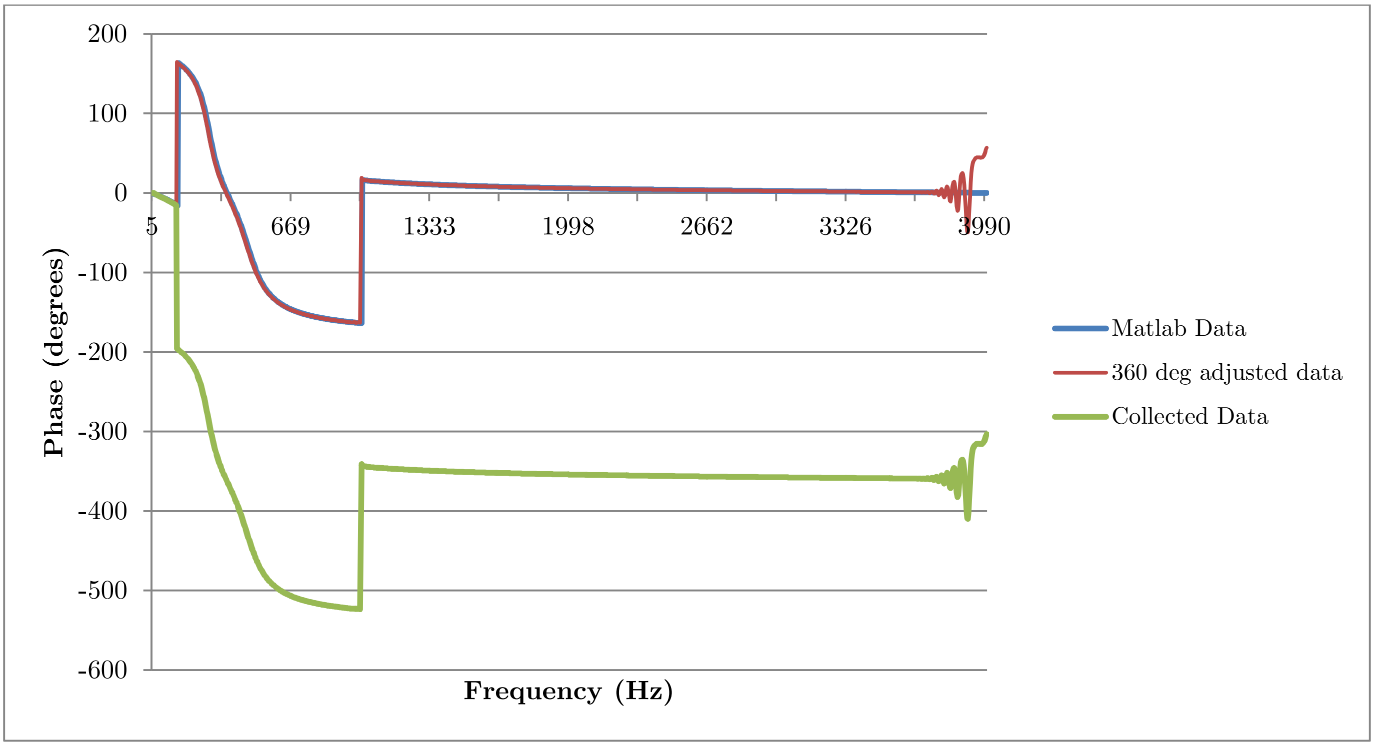


Figure 13. Graph of phase response with respect to frequency for an elliptic filter of order 4 using the given specifications. The APx500 software handles the phase data collection a bit differently than matlab explains the 360 degree phase shift in data at certain points. Wrapped back data taking into account the 360 phase shift is also displayed in green to show the overlap of the collected data and the matlab data.

Figures 12 and 13 show how the response is generally as expected using the matlab filter constructors, with only significant divergence as the frequency approaches towards the nyquist frequency. This is most likely due to the image of the reconstructed sinewave introducing errors as it is no longer cut-off by our non-ideal anti-aliasing filter, as is discussed earlier for the RC circuit.

The Passband was then studied closely with the APX and compared to the expected result, with the result shown in figure 14. It can be seen that there is almost exactly the specified ripple of 0.5 dB (just 0.006 dB out) and the Passband frequencies are 280.5 Hz and 460.2 Hz (1 d.p.). Once again showing that the filter accurately follows the specifications give, with expected results of 280 Hz and 460 Hz compared to those measured.

This was then processed to remove the DSK effects and then compared to our predicted matlab data in the passband region. As we can see in figure 15, the data lays almost on top of each other, showing that the constructed filter is exactly as predicted.

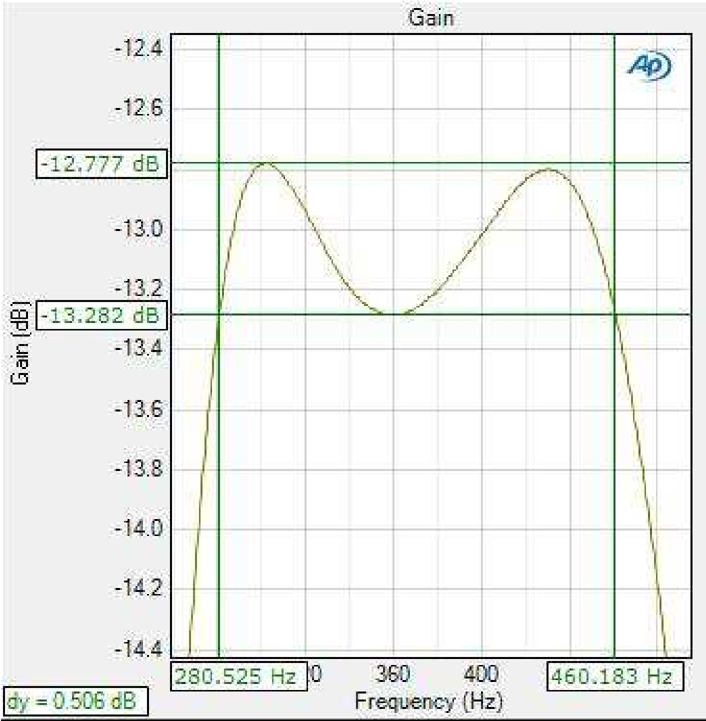


Figure 14. APX graph of Passband region, showing Ripple size and passband frequencies.

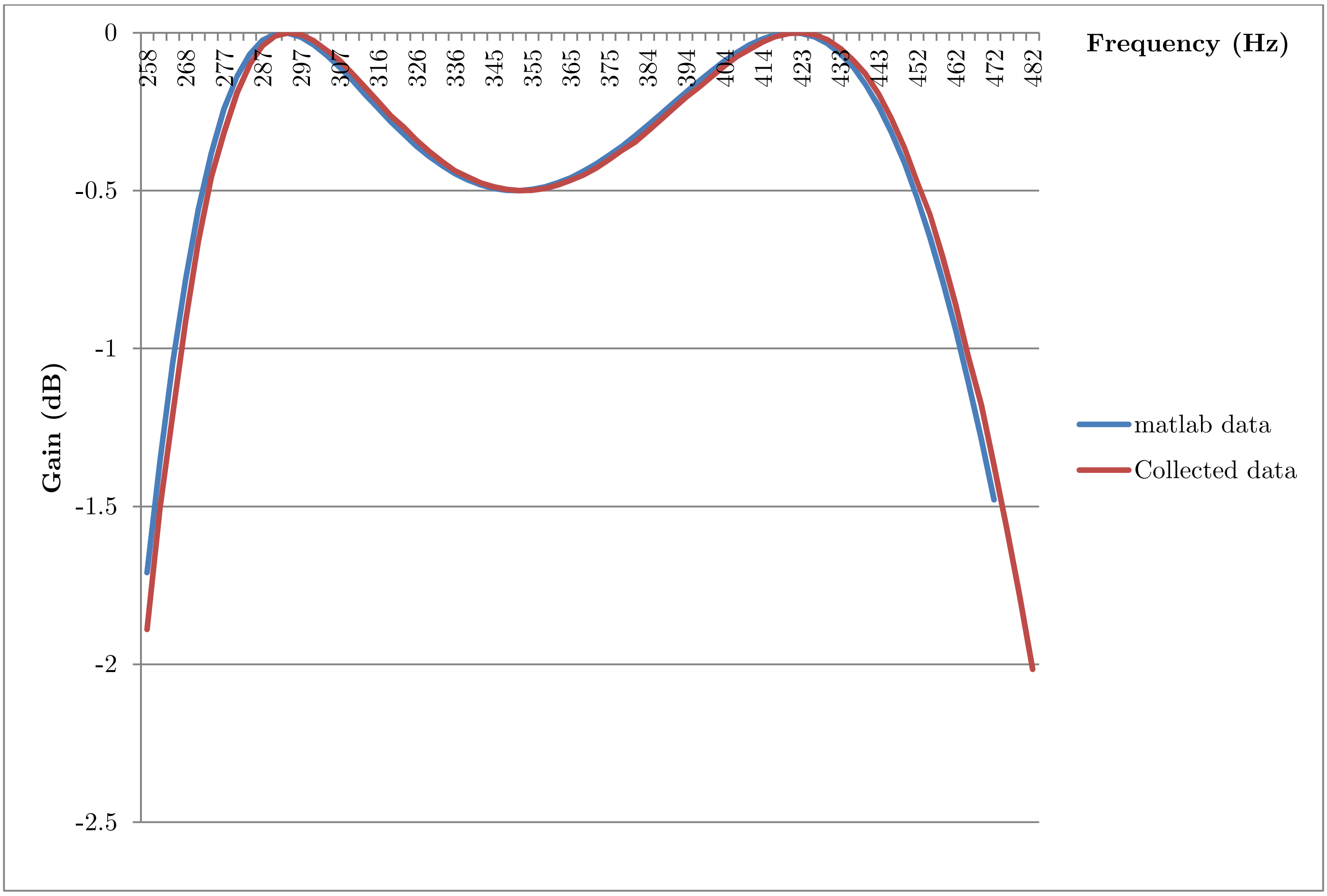


Figure 15. Graph of Matlab passband response compared to measured passband response.

IIR Direct form II transposed filter

Implementation in C

A direct form II IIR transposed filter has the form given in Figure 12.

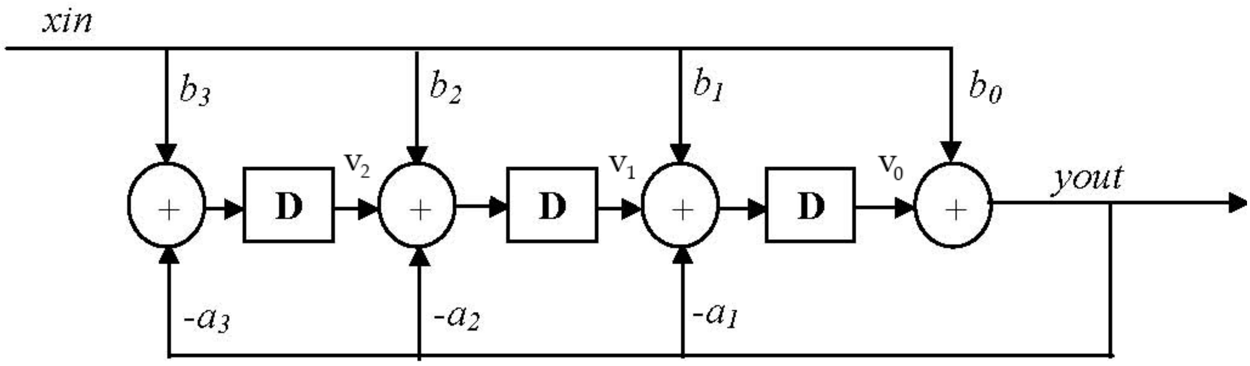


Figure 16. Diagram of a IIR Direct form II transposed filter.

From Figure 16 we can determine how each $v[n]$ delay element is related to each other as each element is related to each other in a cascade. This gives us the following set of equations, where $x[n]$ is the input and $y[n]$ is the output.

$$y[n] = v[n] + b_0 x[n]$$

$$v[n] = v[n-1] + b_1 x[n-1] - a_1 y[n]$$

$$v[n-1] = v[n-2] + b_2 x[n-2] - a_2 y[n]$$

$$\dots$$

$$v[n - (\text{order} - 1)] = b_{\text{order}} x[n - \text{order}] - a_{\text{order}} y[n]$$

From this the following code was written:

```
double base_IIR_2_trans(){
	//this function reads the global sample variable and returns
	//a filtered value(the output)
	//this function is based on equations derived from a
	//IIR Direct form II transposed filter.
	output = v[0] + b[0]*sample; //calculate output based on buffer data
	//loop to complete each iteration of v[n]=v[n-1]+b_1 x[n-1]-a_1 y[n]
	//to populate buffer
	for(i = 0;i<order-1;i++){
		// add new weighted inputs and outputs to previous buffer value v[n+1]
		v[i] = v[i+1] + b[i+1]*sample - a[i+1]*output;
	}
	//calculate first value of v which does not rely on previous versions
	v[order-1] = b[order]*sample - a[order]*output;
	return output; //return filtered value
}
```

The frequency response of the transpose filter was then measured using the spectrum analyser and then plotted against the expected response from matlab in excel. The resulting graph (figure 17) shows how accurately our filter meets the specification for an elliptic bandpass filter of order 4.

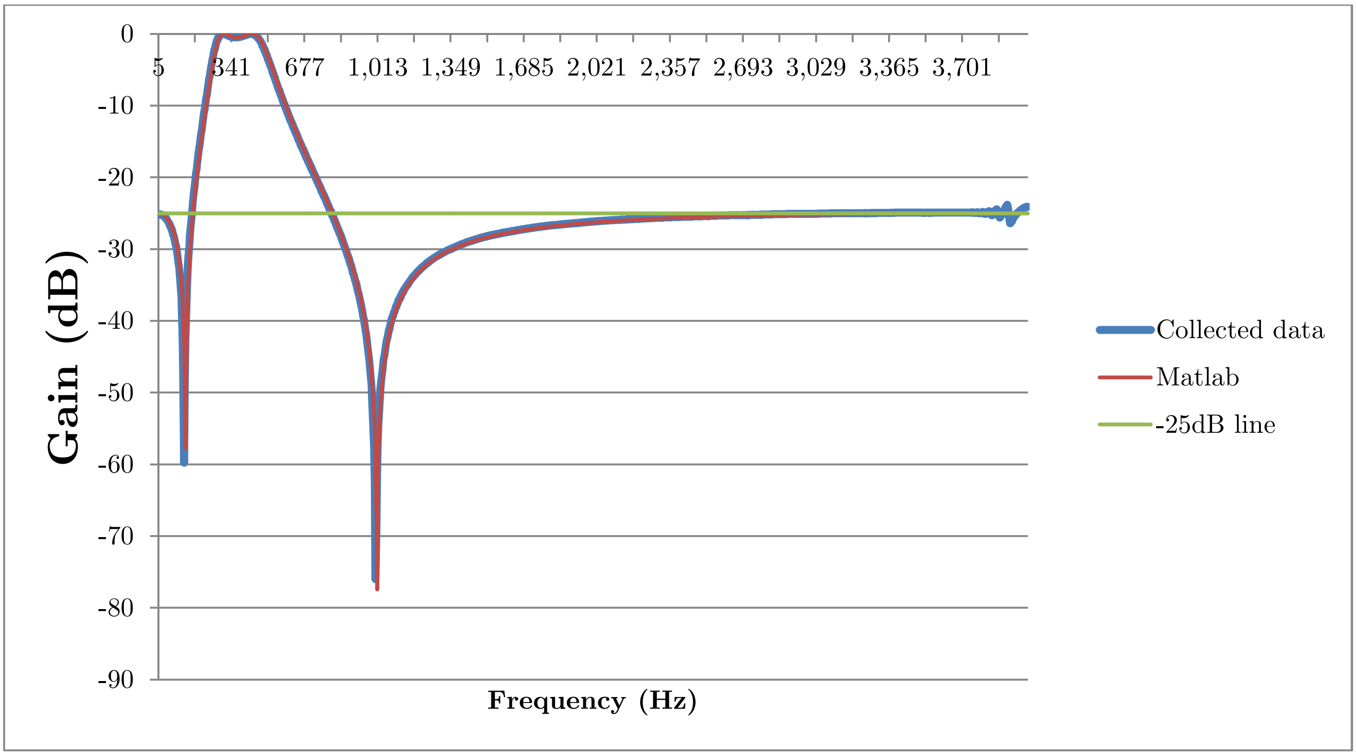


Figure 17. Graph of gain response with respect to frequency for an elliptic filter of order 4 using the given specifications for a Direct form 2 transpose filter.

In order to identify the bandpass frequencies the graph on the APX is examined closely to give the figure below, this shows both the measured frequencies and bandpass ripple. For the bandpass frequencies, 280.2 and 460.6 Hz are measured against the specified frequencies of 280 Hz and 460 Hz respectively, clearly giving a quite precise filter as planned. The bandpass ripple is once again almost spot on the specified ripple of 0.5 dB, being just .005 dB larger than expected, well within what is acceptable.

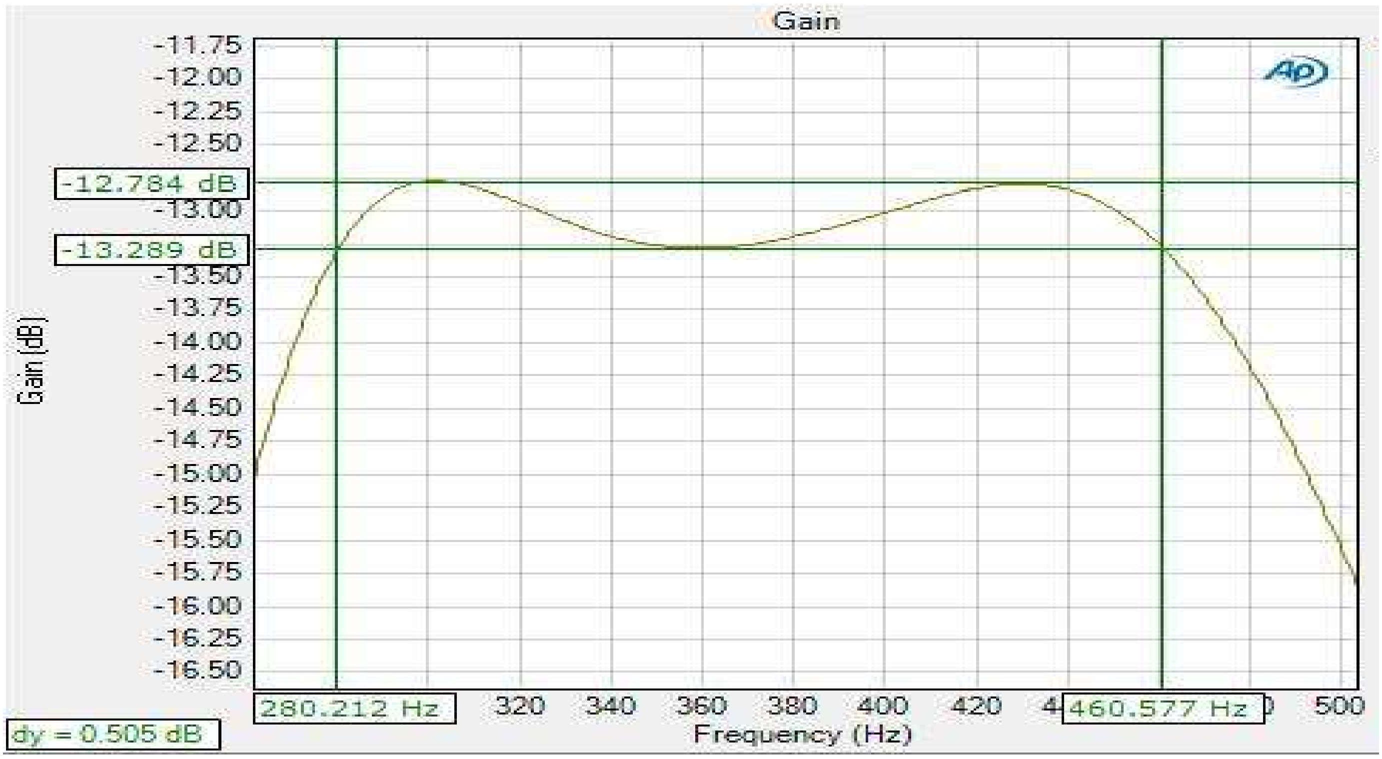


Figure 18. Graph of Passband for transpose, showing ripple and passband frequency

Finally the phase of the transposed filter was measured and processed (to remove DSK effects), this was then compared to our predicted matlab response. Similar to the non-transposed filter, the zeros do not change the phase in the same direction and so with figure 19 below the shift of -180° instead of +180° at the first zero is accounted for by adjusting the collected data up by 360°.

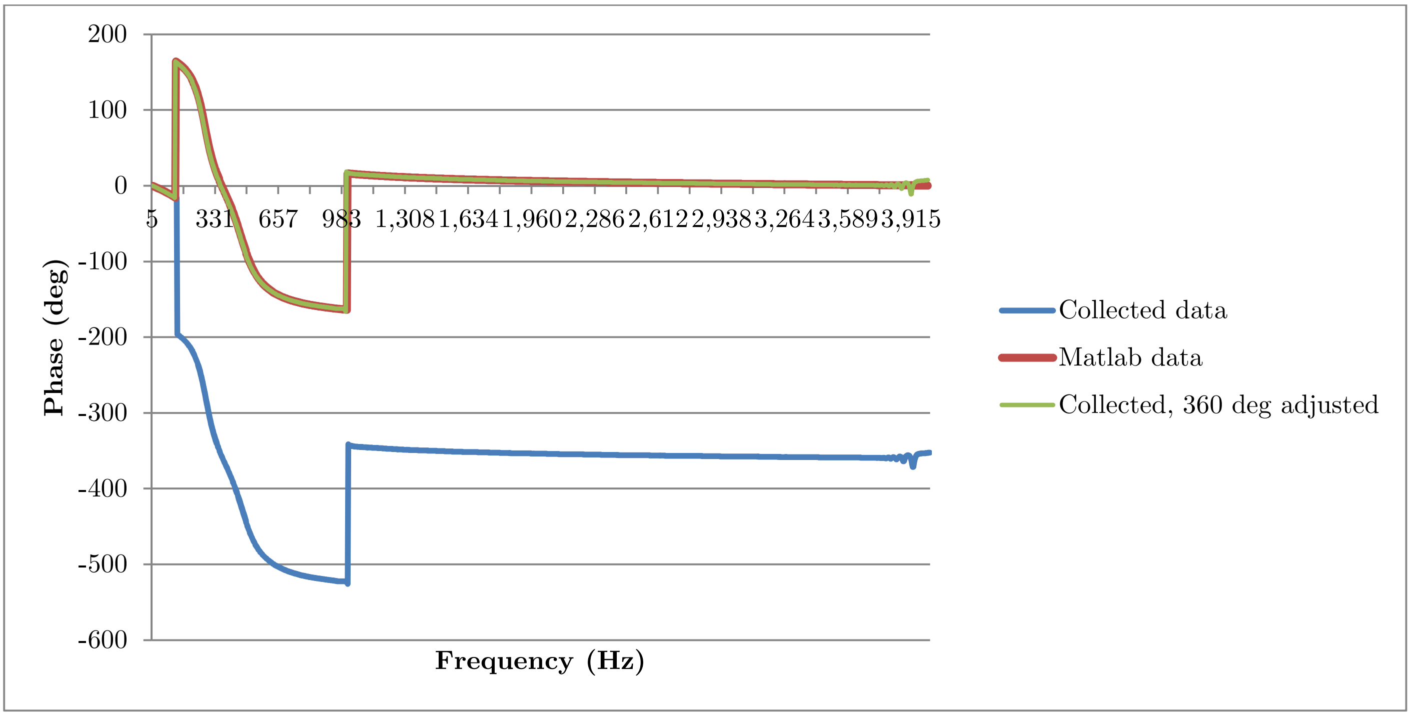


Figure 19. Graph of phase response with respect to frequency for an elliptic filter of order 4 using the given specifications for a transpose filter.

The measurements obtained were very similar, and could possibly be the same (due to the way the APX measures the exact points would not line up), as those which were found for the non-transposed direct form 2 filter that was constructed. Demonstrating how the filter response effectively remains unchanged.

Performance comparison of Direct Form 2 filters

After implementing each filter in order 4, the speed of the code was measured while adjusting the order of the filter. This was done in order to measure the number of instruction cycles between the sampling the input and writing the result of the filter to the audio port. The measurements were for when the compiler was set for no optimisation and then for optimisation level 2 for both the non-transpose and transposed direct form 2 filters.

Both sets were taken between the calls to mono_read_16Bit() and mono_write_16Bit(), with the results shown in the table below for both filter types and optimisations.

No optimisation	Direct form 2	Transpose	**Op-2**	Direct form 2	Transpose
Order	Cycles	Cycles		Cycles	Cycles
2	223	139		130	118
4	385	251		180	180
6	547	363		239	137
8	709	475		289	147
10	871	587		339	157
12	1033	699		389	167

The results were then plotted on the graphs below, with both the transpose and non-transpose for the non-optimised code shown on figure 20 below. A perfect linear relation can be found for each set of data points. This relation reflects the overhead of the code and the number of extra cycles that are due to an extra coefficient (from an increase in order). Hence we find an equation of A+Bn for how many instruction cycles are required to process a sample with a filter of order n.

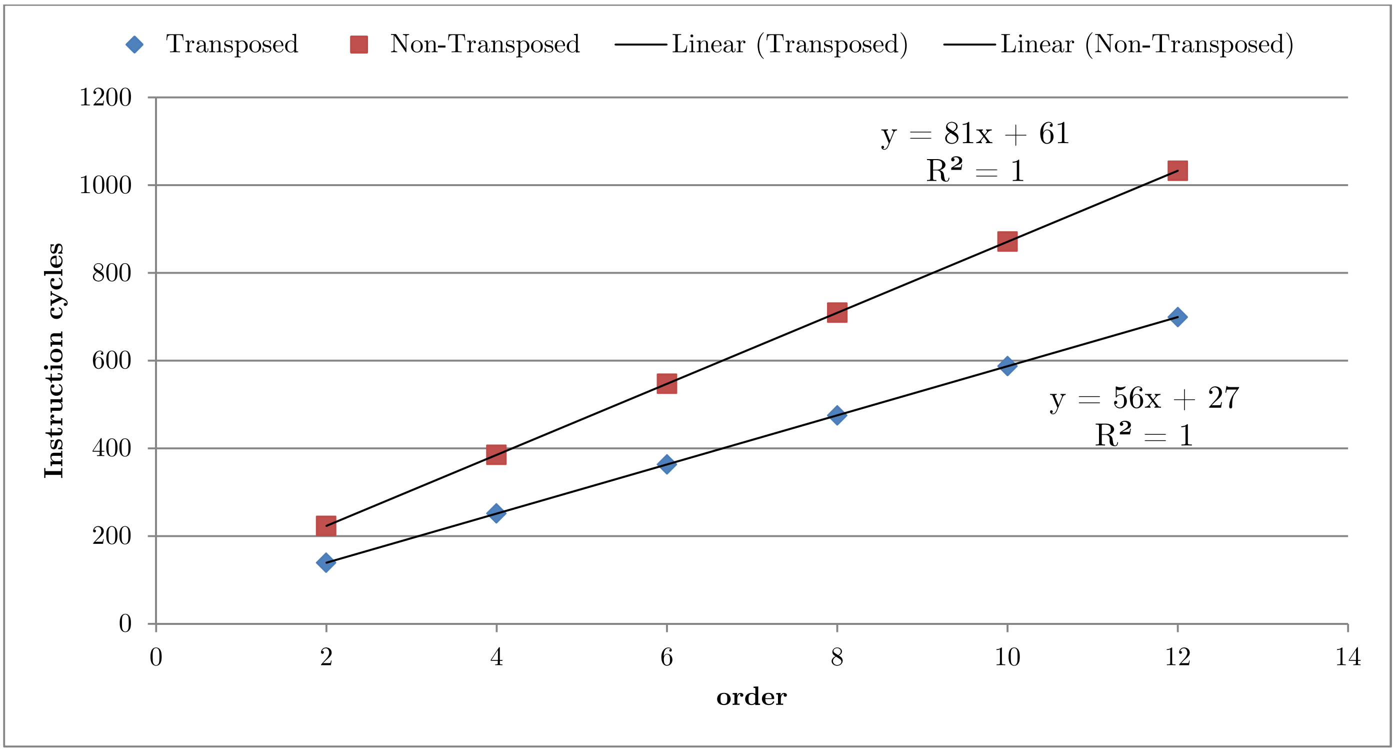


Figure 20. Plot of instruction cycles against filter order for a transposed (blue) and non-transposed (red) Direct form 2 filter for no compiler optimisation.

For the optimised level 2 results we found that that for lower orders (and so smaller loops) the optimisation does not work in a linear fashion, giving a large outlier for the transposed graph (180 cycles at order 4) while the non-transposed is also slightly non-linear at lower orders. For orders of 6 and above however a perfectly linear relationship can be measured, this can be seen in figure 21, where a linear relation is graphed for orders 6-12 giving out order to cycle relation.

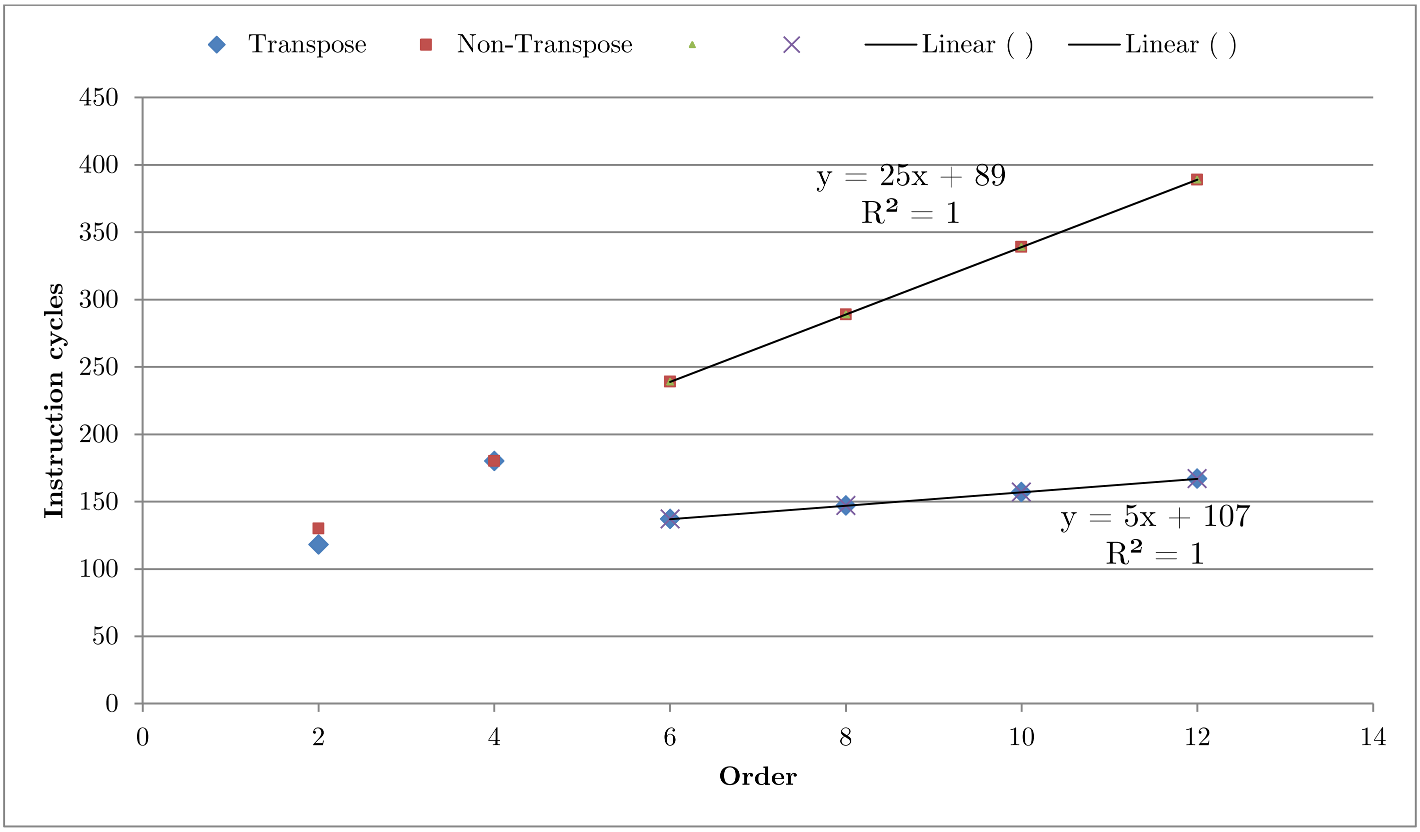


Figure 21. Plot of instruction cycles against filter order for a transposed (blue) and non-transposed filter with compiler optimisation level 2

The resulting linear functions have then been measured and placed in the table below, showing how that the transposed function is considerably faster than the normal Direct form 2. There is also a divergence as order increases as for non-optimised code the transposed rises by 56 cycles for each order while the non-transposed rises by 81 cycles per order.

Filter type and Optimisation level	Instruction cycles per sample (order n)
Direct form 2 – No optimisation	61+81n
Direct form 2 – op level 2	89+25n
Direct form 2 transposed – No optimisation	27+56n
Direct form 2 transposed – op level 2	107+5n

The reason behind this can be seen in the respective codes of the filters. Where the non-transposed code contains a for loop with three instructions whereas the transposed contains only one. Furthermore the buffer in the non-transposed must be shifted by 1 at the end of every loop adding more instructions, but this is not the case for the transposed code.

In fact to see exactly how the code optimizes from no optimization to level 2 we can check the disassembler entry which gives us the number of instructions for each section of code:

```
Breakpoints | Disassembly (base_IIR_2_trans + 0x3c)
206                for(i = 0;i<order-1;i++){
0x0000C9A8:   020000FA            ZERO.L2       B4
0x0000C9AC:   0200BB7E            STW.D2T2      B4,*+B14[187]
0x0000C9B0:   0200BA6E            LDW.D2T2      *+B14[186],B4
0x0000C9B4:   0280BB6E            LDW.D2T2      *+B14[187],B5
0x0000C9B8:   00004000            NOP           3
0x0000C9BC:   0213E05A            SUB.L2        B4,1,B4
0x0000C9C0:   0010AAFA            CMPLT.L2      B5,B4,B0
0x0000C9C4:   30001B10     [!B0]  B.S1          C$DW$L$_base_IIR_2_trans$2$E
0x0000C9C8:   00008000            NOP           5
208                   v[i] = v[i+1] + b[i+1]*sample - a[i+1]*output;
              C$DW$L$_base_IIR_2_trans$2$B, C$L10:
0x0000C9CC:   0214005B            MV.L2         B5,B4
0x0000C9D0:   0302A42A ||         MVK.S2        0x0548,B6
0x0000C9D4:   02106CA3            SHL.S2        B4,0x3,B4
0x0000C9D8:   00000001 ||         NOP
0x0000C9DC:   00000000 ||         NOP
```

Figure 22. Screenshot of a section of the assembly code running on the board for a IIR direct form 2 transposed filter implementation.

For example, we can notice that from the start to the end of the for loop of the IIR direct form II transposed filter implementation that the instruction location starts at 0x00C9A8 and ends at 0x00CA94 (see appendix for full code in loop) thus we have: $\frac{0x00CA94-0x00C9A8}{4_{\text{base } 10}} = 59_{\text{base } 10}$ 59 instructions (which do not directly translate to cycle number due to NOPs for example) for one cycle.

From this we arrive at the following table:

	IIR DF2	IIR DF2 Transposed
No op assembly lines	54	59
Number of NOP	54	34
Number of parallel instructions	4	13
Number of parallel blocks	3	8
Number of possible branches	1	1
op2_unrolled assembly lines	64	91
Lines per cycle *ROUND(ans/4)*	16	23
Number of NOP	25	25
Number of parallel blocks	12	11
Number of parallel instructions	17	23
Number of possible branches	2	7

Figure 23. Data collected for a 4th order process for IIR direct form ii transposed and non-transposed. It was found, by changing the filter order and observing for possible assembly code length changes, that loop unrolling was not done in optimisation level 2 for the IIR DF2 transposed filter loop code, possibly due to the compiler finding the branching conditions (7 in this case) more efficient than simple loop unrolling.
Note: This data is collected for the number of assembly lines between the start and the end of a C for loop, not the actual assembly loop that the compiler creates in assembly, which means that concluding that the number of lines is an exact representation of the number of cycles per loop is not exactly correct.

From the above table we notice that the transposed loop may not seem the most efficient, with a high number of lines per cycle. We do however notice that the optimisation level 2 takes greater advantage of parallel instructions, having a higher ratio of $\frac{\textit{number of parallel instruction}}{\textit{number of parallel blocks}}$, for the transposed process as well as allowing for the optimizer to have a greater number of branch conditions (denoted by **C$Lbranch#**) which the direct form 2 filter only has 2 for an optimized level 2 code whereas the direct form 2 transposed optimized assembly code has 7 different branch conditions. Thus it is possible that the single line of code in the IIR Direct Form 2 transposed filter loop is easier to optimize for the compiler than the 3 (memory interdependent) lines of code inside the IIR Direct Form 2 filter loop. Indeed the optimisation level 2 was apparently deemed so efficient for the compiler for a IIR DF2 transposed that it did no loop unrolling (in the classic sense) whereas some loop unrolling was done for the IIR DF2 implementation. This was determined by observing if the number of lines of assembly code changed by recompiling the code for different order filters.

Finally we also notice that the overhead in each implementation (as determined by the constant in the A+Bn equations) goes up as we increase the optimisation level. This is due to the compiler creating a more efficient "setup code" which allows sets the code up allowing for shorter loops to be executed later on in the code. We notice this from the assembly code of both implementations for optimisation level 2 having only a core of around 16 lines of assembly code for actual assembly loops (highlighted in red for the op2 IIR DF2 in the appendix), whereas for no optimisation the assembly looping length is closer to the number of line measured in Figure 23.

Appendix

All the following samples of codes are optimized for a 4th order IIR process.

Full assembly code for no optimisation of a IIR direct from II non-transposed filter loop:

```
188            for (i=order;i>0;i--){
0x0000C848:   0200BA6E          LDW.D2T2     *+B14[186],B4
0x0000C84C:   00006000          NOP          4
0x0000C850:   0200BB7E          STW.D2T2     B4,*+B14[187]
0x0000C854:   00100ADA          CMPLT.L2     0,B4,B0
0x0000C858:   30001C90    [!B0] B.S1         C$DW$L$_base_IIR_2$2$E (PC+228 = 0x0000c924)
0x0000C85C:   00008000          NOP          5
189                 v[0] -= a[i]*v[i]; //accumulate a coefficents
            C$DW$L$_base_IIR_2$2$B, C$L8:
0x0000C860:   02025028          MVK.S1       0x04a0,A4
0x0000C864:   020000E9          MVKH.S1      0x10000,A4
0x0000C868:   0298005B ||       MV.L2        B6,B5
0x0000C86C:   01901058 ||       MV.L1X       B4,A3
0x0000C870:   02106B65          LDDW.D1T1    *+A4[A3],A5:A4
0x0000C874:   02148BE6 ||       LDDW.D2T2    *+B5[B4],B5:B4
0x0000C878:   04981058          MV.L1X       B6,A9
0x0000C87C:   00004000          NOP          3
0x0000C880:   03109700          MPYDP.M1X    A5:A4,B5:B4,A7:A6
0x0000C884:   02240364          LDDW.D1T1    *+A9[0],A5:A4
0x0000C888:   0000E000          NOP          8
0x0000C88C:   02188338          SUBDP.L1     A5:A4,A7:A6,A5:A4
0x0000C890:   0000A000          NOP          6
0x0000C894:   02A42274          STW.D1T1     A5,*+A9[1]
0x0000C898:   02240274          STW.D1T1     A4,*+A9[0]
190                 output += b[i]*v[i]; //accumulate to output
0x0000C89C:   0200BB6E          LDW.D2T2     *+B14[187],B4
0x0000C8A0:   0282A42A          MVK.S2       0x0548,B5
0x0000C8A4:   02181058          MV.L1X       B6,A4
0x0000C8A8:   028000EA          MVKH.S2      0x10000,B5
0x0000C8AC:   00000000          NOP
0x0000C8B0:   01901058          MV.L1X       B4,A3
0x0000C8B4:   02106B65          LDDW.D1T1    *+A4[A3],A5:A4
0x0000C8B8:   02148BE6 ||       LDDW.D2T2    *+B5[B4],B5:B4
0x0000C8BC:   00006000          NOP          4
0x0000C8C0:   02109700          MPYDP.M1X    A5:A4,B5:B4,A5:A4
0x0000C8C4:   0280B76E          LDW.D2T2     *+B14[183],B5
0x0000C8C8:   0200B66E          LDW.D2T2     *+B14[182],B4
0x0000C8CC:   0000C000          NOP          7
0x0000C8D0:   0210931A          ADDDP.L2X    B5:B4,A5:A4,B5:B4
0x0000C8D4:   0000A000          NOP          6
0x0000C8D8:   0280B77E          STW.D2T2     B5,*+B14[183]
0x0000C8DC:   0200B67E          STW.D2T2     B4,*+B14[182]
191                 v[i] = v[i-1]; //shift v[i] data down to represent the delay elements
0x0000C8E0:   020C105A          MV.L2X       A3,B4
0x0000C8E4:   02106CA2          SHL.S2       B4,0x3,B4
0x0000C8E8:   0213005A          SUB.L2       B4,8,B4
0x0000C8EC:   0210C07A          ADD.L2       B6,B4,B4
0x0000C8F0:   021003E6          LDDW.D2T2    *+B4[0],B5:B4
0x0000C8F4:   02181058          MV.L1X       B6,A4
0x0000C8F8:   01907E40          ADDAD.D1     A4,A3,A3
0x0000C8FC:   00002000          NOP          2
0x0000C900:   028C2276          STW.D1T2     B5,*+A3[1]
0x0000C904:   020C0276          STW.D1T2     B4,*+A3[0]
188            for (i=order;i>0;i--){
0x0000C908:   0200BB6E          LDW.D2T2     *+B14[187],B4
0x0000C90C:   00006000          NOP          4
0x0000C910:   0213E05A          SUB.L2       B4,1,B4
0x0000C914:   0200BB7E          STW.D2T2     B4,*+B14[187]
0x0000C918:   00100ADA          CMPLT.L2     0,B4,B0
0x0000C91C:   2FFFEC10    [ B0] B.S1         C$L8 (PC-160 = 0x0000c860)
0x0000C920:   00008000          NOP          5
```

Full assembly code for no optimisation of a IIR direct from II transposed filter loop:

```
206                 for(i = 0;i<order-1;i++){
0x0000C9A8:  020000FA           ZERO.L2     B4
0x0000C9AC:  0200BB7E           STW.D2T2    B4,*+B14[187]
0x0000C9B0:  0200BA6E           LDW.D2T2    *+B14[186],B4
0x0000C9B4:  0280BB6E           LDW.D2T2    *+B14[187],B5
0x0000C9B8:  00004000           NOP         3
0x0000C9BC:  0213E05A           SUB.L2      B4,1,B4
0x0000C9C0:  0010AAFA           CMPLT.L2    B5,B4,B0
0x0000C9C4:  30001B10     [!B0] B.S1        C$DW$L$_base_IIR_2_trans$2$E (PC+216 = 0x0000ca98)
0x0000C9C8:  00008000           NOP         5
208                     v[i] = v[i+1] + b[i+1]*sample - a[i+1]*output;
            C$DW$L$_base_IIR_2_trans$2$B, C$L10:
0x0000C9CC:  0214005B           MV.L2       B5,B4
0x0000C9D0:  0302A42A ||        MVK.S2      0x0548,B6
0x0000C9D4:  02106CA3           SHL.S2      B4,0x3,B4
0x0000C9D8:  00000001 ||        NOP
0x0000C9DC:  00000000 ||        NOP
0x0000C9E0:  0211005B           ADD.L2      8,B4,B4
0x0000C9E4:  030000EB ||        MVKH.S2     0x10000,B6
0x0000C9E8:  03940942 ||        MV.D2       B5,B7
0x0000C9EC:  0310C07B           ADD.L2      B6,B4,B6
0x0000C9F0:  041401A3 ||        MV.S2       B5,B8
0x0000C9F4:  0280B96E ||        LDW.D2T2    *+B14[185],B5
0x0000C9F8:  001803E6           LDDW.D2T2   *+B6[0],B1:B0
0x0000C9FC:  0200B86E           LDW.D2T2    *+B14[184],B4
0x0000CA00:  049C6CA2           SHL.S2      B7,0x3,B9
0x0000CA04:  02025028           MVK.S1      0x04a0,A4
0x0000CA08:  0325005A           ADD.L2      8,B9,B6
0x0000CA0C:  020000E8           MVKH.S1     0x10000,A4
0x0000CA10:  02008703           MPYDP.M2    B5:B4,B1:B0,B5:B4
0x0000CA14:  02189079 ||        ADD.L1X     A4,B6,A4
0x0000CA18:  0380B76F ||        LDW.D2T2    *+B14[183],B7
0x0000CA1C:  00000000 ||        NOP
0x0000CA20:  02100365           LDDW.D1T1   *+A4[0],A5:A4
0x0000CA24:  0300B66E ||        LDW.D2T2    *+B14[182],B6
0x0000CA28:  04206CA3           SHL.S2      B8,0x3,B8
0x0000CA2C:  0182F828 ||        MVK.S1      0x05f0,A3
0x0000CA30:  0421005B           ADD.L2      8,B8,B8
0x0000CA34:  018000E8 ||        MVKH.S1     0x10000,A3
0x0000CA38:  01A07078           ADD.L1X     A3,B8,A3
0x0000CA3C:  030C0364           LDDW.D1T1   *+A3[0],A7:A6
0x0000CA40:  04189700           MPYDP.M1X   A5:A4,B7:B6,A9:A8
0x0000CA44:  0100BB6C           LDW.D2T1    *+B14[187],A2
0x0000CA48:  0082F828           MVK.S1      0x05f0,A1
0x0000CA4C:  008000E8           MVKH.S1     0x10000,A1
0x0000CA50:  0210D318           ADDDP.L1X   A7:A6,B5:B4,A5:A4
0x0000CA54:  0000A000           NOP         6
0x0000CA58:  03208338           SUBDP.L1    A5:A4,A9:A8,A7:A6
0x0000CA5C:  01845E40           ADDAD.D1    A1,A2,A3
0x0000CA60:  00008000           NOP         5
0x0000CA64:  038C2274           STW.D1T1    A7,*+A3[1]
0x0000CA68:  030C0274           STW.D1T1    A6,*+A3[0]
206                 for(i = 0;i<order-1;i++){
0x0000CA6C:  0200BB6E           LDW.D2T2    *+B14[187],B4
0x0000CA70:  00006000           NOP         4
0x0000CA74:  0210205A           ADD.L2      1,B4,B4
0x0000CA78:  0200BB7E           STW.D2T2    B4,*+B14[187]
0x0000CA7C:  0200BA6E           LDW.D2T2    *+B14[186],B4
0x0000CA80:  0280BB6E           LDW.D2T2    *+B14[187],B5
0x0000CA84:  00004000           NOP         3
0x0000CA88:  0213E05A           SUB.L2      B4,1,B4
0x0000CA8C:  0010AAFA           CMPLT.L2    B5,B4,B0
0x0000CA90:  2FFFE990     [ B0] B.S1        C$L10 (PC-180 = 0x0000c9cc)
0x0000CA94:  00008000           NOP         5
```

Full assembly code for optimisation level 2 of a IIR direct from II filter loop:

```
188              for (i=order;i>0;i--){
0x0000C8C4:   0280BA6E             LDW.D2T2      *+B14[186],B5
186              output = 0; //reset output to accumulate result
0x0000C8C8:   000004FA             ZERO.L2       B1:B0
188              for (i=order;i>0;i--){
0x0000C8CC:   00004000             NOP           3
0x0000C8D0:   01140ADB             CMPLT.L2      0,B5,B2
0x0000C8D4:   00000001 ||          NOP
0x0000C8D8:   00000001 ||          NOP
0x0000C8DC:   00000000 ||          NOP
0x0000C8E0:   70001C11      [!B2]  B.S1          C$L15 (PC+224 = 0x0000c9c0)
0x0000C8E4:   6383E42B ||   [ B2]  MVK.S2        0x07c8,B7
0x0000C8E8:   60941059 ||   [ B2]  MV.L1X        B5,A1
0x0000C8EC:   6218BE42 ||   [ B2]  ADDAD.D2      B6,B5,B4
0x0000C8F0:   638000EB      [ B2]  MVKH.S2       0x10000,B7
0x0000C8F4:   7280BB7E ||   [!B2]  STW.D2T2      B5,*+B14[187]
0x0000C8F8:   6183D028      [ B2]  MVK.S1        0x07a0,A3
0x0000C8FC:   629CBE42      [ B2]  ADDAD.D2      B7,B5,B5
0x0000C900:   618000E8      [ B2]  MVKH.S1       0x10000,A3
0x0000C904:   648C3E40      [ B2]  ADDAD.D1      A3,A1,A9
189                  v[0] -= a[i]*v[i]; //accumulate a coefficents
0x0000C908:   058403E3             MVC.S2        CSR,B11
0x0000C90C:   01981059 ||          MV.L1X        B6,A3
0x0000C910:   010000A8 ||          MVK.S1        0x0001,A2
0x0000C914:   032FCF5B             AND.L2        -2,B11,B6
0x0000C918:   04101058 ||          MV.L1X        B4,A8
0x0000C91C:   009803A2             MVC.S2        B6,CSR
           C$DW$L$_base_IIR_2$4$B, C$L13, C$L12:
0x0000C920:   00000000             NOP
0x0000C924:   B20C0274      [!A2]  STW.D1T1      A4,*+A3[0]
0x0000C928:   B31435E7      [!A2]  LDDW.D2T2     *B5--[1],B7:B6
0x0000C92C:   B28C2274 ||   [!A2]  STW.D1T1      A5,*+A3[1]
0x0000C930:   B41023E7      [!A2]  LDDW.D2T2     *+B4[1],B9:B8
0x0000C934:   02243564 ||          LDDW.D1T1     *A9--[1],A5:A4
0x0000C938:   031035E4             LDDW.D2T1     *B4--[1],A7:A6
0x0000C93C:   B11023E6      [!A2]  LDDW.D2T2     *+B4[1],B3:B2
0x0000C940:   00002000             NOP           2
0x0000C944:   03190702             MPYDP.M2      B9:B8,B7:B6,B7:B6
0x0000C948:   0210C700             MPYDP.M1      A7:A6,A5:A4,A5:A4
0x0000C94C:   B1205476      [!A2]  STW.D1T2      B2,*A8--[2]
0x0000C950:   B190A2F6      [!A2]  STW.D2T2      B3,*+B4[5]
0x0000C954:   00002000             NOP           2
0x0000C958:   030C0364             LDDW.D1T1     *+A3[0],A7:A6
0x0000C95C:   00004000             NOP           3
0x0000C960:   8087E059      [ A1]  SUB.L1        A1,1,A1
0x0000C964:   B000C31A ||   [!A2]  ADDDP.L2      B7:B6,B1:B0,B1:B0
0x0000C968:   8FFFF813      [ A1]  B.S2          C$L12 (PC-64 = 0x0000c920)
0x0000C96C:   0210C338 ||          SUBDP.L1      A7:A6,A5:A4,A5:A4
0x0000C970:   00006000             NOP           4
0x0000C974:   A10BE1A0      [ A2]  SUB.S1        A2,1,A2
           C$L14, C$DW$L$_base_IIR_2$4$E:
0x0000C978:   00000000             NOP
0x0000C97C:   020C0274             STW.D1T1      A4,*+A3[0]
0x0000C980:   031435E7             LDDW.D2T2     *B5--[1],B7:B6
0x0000C984:   028C2274 ||          STW.D1T1      A5,*+A3[1]
0x0000C988:   041023E6             LDDW.D2T2     *+B4[1],B9:B8
0x0000C98C:   00000000             NOP
0x0000C990:   011003E6             LDDW.D2T2     *+B4[0],B3:B2
0x0000C994:   00002000             NOP           2
0x0000C998:   03190702             MPYDP.M2      B9:B8,B7:B6,B7:B6
0x0000C99C:   00000000             NOP
0x0000C9A0:   01205476             STW.D1T2      B2,*A8--[2]
0x0000C9A4:   020000FB             ZERO.L2       B4
0x0000C9A8:   019062F6 ||          STW.D2T2      B3,*+B4[3]
0x0000C9AC:   0200BB7E             STW.D2T2      B4,*+B14[187]
0x0000C9B0:   00008000             NOP           5
0x0000C9B4:   0000C31A             ADDDP.L2      B7:B6,B1:B0,B1:B0
0x0000C9B8:   00AC03A3             MVC.S2        B11,CSR
0x0000C9BC:   00000000 ||          NOP
```

Full assembly code for optimisation level 2 of a IIR direct from II transposed filter loop:

```
206          for(i = 0;i<order-1;i++){
0x0000C6B4:  0580BA6E           LDW.D2T2      *+B14[186],B11
0x0000C6B8:  0183D028           MVK.S1        0x07a0,A3
0x0000C6BC:  018000E8           MVKH.S1       0x10000,A3
0x0000C6C0:  020C105A           MV.L2X        A3,B4
0x0000C6C4:  018000F8           ZERO.L1       A3
0x0000C6C8:  002C48DA           CMPGT.L2      2,B11,B0
0x0000C6CC:  3514005B     [!B0] MV.L2         B5,B10
0x0000C6D0:  20002C90  || [ B0] B.S1          C$L10 (PC+356 = 0x0000c824)
0x0000C6D4:  2180BB7C     [ B0] STW.D2T1      A3,*+B14[187]
0x0000C6D8:  3183E428     [!B0] MVK.S1        0x07c8,A3
0x0000C6DC:  3191005A     [!B0] ADD.L2        8,B4,B3
0x0000C6E0:  318000E8     [!B0] MVKH.S1       0x10000,A3
0x0000C6E4:  360D0058     [!B0] ADD.L1        8,A3,A12
208            v[i] = v[i+1] + b[i+1]*sample - a[i+1]*output;
0x0000C6E8:  00AFF058           SUB.L1X       B11,1,A1
206          for(i = 0;i<order-1;i++){
0x0000C6EC:  01046AD8           CMPLT.L1      3,A1,A2
0x0000C6F0:  A0000E90     [ A2] B.S1          C$L6 (PC+116 = 0x0000c754)
0x0000C6F4:  0604105A           MV.L2X        A1,B12
0x0000C6F8:  A007F05A     [ A2] SUB.L2X       A1,1,B0
0x0000C6FC:  A087D05A     [ A2] SUB.L2X       A1,2,B1
0x0000C700:  A0303764     [ A2] LDDW.D1T1     *A12++[1],A1:A0
0x0000C704:  00000000           NOP
          C$DW$L$_base_IIR_2_trans$3$B, C$L5:
0x0000C708:  02303764           LDDW.D1T1     *A12++[1],A5:A4
0x0000C70C:  00006000           NOP           4
          C$DW$L$_base_IIR_2_trans$4$B, C$DW$L$_base_IIR_2_trans$3$E:
0x0000C710:  02114700           MPYDP.M1      A11:A10,A5:A4,A5:A4
0x0000C714:  020C37E6           LDDW.D2T2     *B3++[1],B5:B4
0x0000C718:  032823E6           LDDW.D2T2     *+B10[1],B7:B6
0x0000C71C:  0087E058           SUB.L1        A1,1,A1
0x0000C720:  00004000           NOP           3
0x0000C724:  02110702           MPYDP.M2      B9:B8,B5:B4,B5:B4
0x0000C728:  00002000           NOP           2
0x0000C72C:  0310D31A           ADDDP.L2X     B7:B6,A5:A4,B7:B6
0x0000C730:  0000A000           NOP           6
0x0000C734:  0210C33A           SUBDP.L2      B7:B6,B5:B4,B5:B4
0x0000C738:  00002000           NOP           2
0x0000C73C:  8FFFFD10     [ A1] B.S1          C$L5 (PC-24 = 0x0000c708)
0x0000C740:  90001C90     [!A1] B.S1          0xC804 (PC+228 = 0x0000c804)
0x0000C744:  00002000           NOP           2
0x0000C748:  022836F6           STW.D2T2      B4,*B10++[1]
0x0000C74C:  02A836F6           STW.D2T2      B5,*B10++[1]
          C$DW$L$_base_IIR_2_trans$4$E:
0x0000C750:  0600BB7E           STW.D2T2      B12,*+B14[187]
          C$L6:
0x0000C754:  068403E2           MVC.S2        CSR,B13
0x0000C758:  0237CF5B           AND.L2        -2,B13,B4
0x0000C75C:  010002AA  ||       MVK.S2        0x0005,B2
0x0000C760:  009003A2           MVC.S2        B4,CSR
          C$L7:
0x0000C764:  04014701           MPYDP.M1      A11:A10,A1:A0,A9:A8
0x0000C768:  00000213  ||       B.S2          C$L8 (PC+16 = 0x0000c770)
0x0000C76C:  00303764  ||       LDDW.D1T1     *A12++[1],A1:A0
          C$DW$L$_base_IIR_2_trans$8$B, C$L8:
0x0000C770:  610BE05A     [ B2] SUB.L2        B2,1,B2
0x0000C774:  01109339           SUBDP.L1X     A5:A4,B5:B4,A3:A2
0x0000C778:  02190703  ||       MPYDP.M2      B9:B8,B7:B6,B5:B4
0x0000C77C:  030C37E6  ||       LDDW.D2T2     *B3++[1],B7:B6
0x0000C780:  032833E4           LDDW.D2T1     *++B10[1],A7:A6
0x0000C784:  712940F5     [!B2] STW.D2T1      A2,*-B10[10]
0x0000C788:  020C105B  ||       MV.L2X        A3,B4
0x0000C78C:  2003E1A3  || [ B0] SUB.S2        B0,1,B0
0x0000C790:  00000001  ||       NOP
0x0000C794:  00000001  ||       NOP
0x0000C798:  00000001  ||       NOP
0x0000C79C:  00000000  ||       NOP
0x0000C7A0:  4087E05B     [ B1] SUB.L2        B1,1,B1
0x0000C7A4:  722920F7  || [!B2] STW.D2T2      B4,*-B10[9]
0x0000C7A8:  2FFFFA13  || [ B0] B.S2          C$DW$L$_base_IIR_2_trans$4$E (PC-48 = 0x0000c750)
0x0000C7AC:  02190319  ||       ADDDP.L1      A9:A8,A7:A6,A5:A4
0x0000C7B0:  04014701  ||       MPYDP.M1      A11:A10,A1:A0,A9:A8
0x0000C7B4:  40303764  || [ B1] LDDW.D1T1     *A12++[1],A1:A0
```

```
            C$L9, C$DW$L$_base_IIR_2_trans$8$E:
0x0000C7B8:   00000001          NOP
0x0000C7BC:   00000000 ||       NOP
0x0000C7C0:   01109339          SUBDP.L1X    A5:A4,B5:B4,A3:A2
0x0000C7C4:   02190702 ||       MPYDP.M2     B9:B8,B7:B6,B5:B4
0x0000C7C8:   00000000          NOP
0x0000C7CC:   012900F5          STW.D2T1     A2,*-B10[8]
0x0000C7D0:   020C105A ||       MV.L2X       A3,B4
0x0000C7D4:   0228E0F7          STW.D2T2     B4,*-B10[7]
0x0000C7D8:   02190318 ||       ADDDP.L1     A9:A8,A7:A6,A5:A4
0x0000C7DC:   00000000          NOP
0x0000C7E0:   01109338          SUBDP.L1X    A5:A4,B5:B4,A3:A2
0x0000C7E4:   00000000          NOP
0x0000C7E8:   0128C0F5          STW.D2T1     A2,*-B10[6]
0x0000C7EC:   020C105A ||       MV.L2X       A3,B4
0x0000C7F0:   0228A0F6          STW.D2T2     B4,*-B10[5]
0x0000C7F4:   00000000          NOP
0x0000C7F8:   01109338          SUBDP.L1X    A5:A4,B5:B4,A3:A2
0x0000C7FC:   00000000          NOP
0x0000C800:   012880F5          STW.D2T1     A2,*-B10[4]
0x0000C804:   020C105A ||       MV.L2X       A3,B4
```