Imperial College London

Real Time Digital Signal Processing

Lab 5 Report

Andrew Zhou, CID: 00938859

Jagannaath Shiva Letchumanan, CID: 00946740

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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.
Signed: **Andrew Zhou, Jagannaath Shiva Letchumanan**

March 3, 2017

Contents

1 Background and Introduction

In the previous experiment, finite impulse response (FIR) filters were studied in depth and implemented on the DSK board. In this laboratory session, infinite impulse response (IIR) filters will be studied in a similar manner, yet again using interrupts instead of polling due to their many advantages.

As previously stated, FIR filters are filters whose impulse response is zero outside a finite duration. IIR filters on the other hand, have impulse responses which do not become exactly zero outside a finite duration. These filters therefore have infinite memory and can be implemented in an analogue fashion (with capacitors and inductors). In fact, almost all analogue filters are IIR. This is in contrast to FIR filters which have finite memory and can therefore only be implemented digitally.

IIR filters are mathematically expressed by a weighted sum of the current input, previous inputs and previous outputs:

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

As a difference equation this becomes:

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_Mx[n-M] - a_1y[n-1] - a_2y[n-2] - ... - a_Ny[n-N] \tag{2}$$

Taking Z-transforms:

$$Y(z) = b_0X(z) + b_1X(z)z^{-1} + ... + b_MX(z)z^{-M} - a_1Y(z)z^{-1} - a_2Y(z)z^{-2} - ... - a_NY(z)z^{-N} \tag{3}$$

Therefore:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1z^{-1} + ... + b_Mz^{-M}}{1 + a_1z^{-1} + a_2z^{-1} + ... + a_Nz^{-N}} \tag{4}$$

It is clear from (4) that in the Z-domain, an IIR is composed of two polynomials: one as a numerator and one as a denominator. Consequently, an IIR can have zeros anywhere in the Z-domain, much like an FIR, but unlike an FIR, it can also have poles anywhere, as opposed to only at the origin.

The flexible positioning of poles represents an extra degree of freedom, and enables one of the biggest advantages that IIR filters have over FIR filters: increased efficiency of implementation. When certain filter specifications such as passband, stopband, ripple and roll-off, must be met, they can be done so by a lower order (N in equations (1-4)) IIR than FIR. In other words, an IIR filter requires a much lower number of taps for accurate frequency

selectivity compared to an FIR filter. When implemented on a DSP chip, this implies a correspondingly less costly complexity, and as seen in further experimentation in sections 3 and 4, can be achieved in O(N).

One drawback of IIR filters when compared to FIR filters is that they are not inherently stable as poles can be located outside the unit circle and thus exclude the unit circle from the region of convergence.

As a result, precision is of utmost importance for IIR filters (as demonstrated in section 5). Since an IIR filter has an infinite length impulse response, the errors accumulate and can hence become large enough to cause the output to fluctuate or behave unlike expected, even for a small filter order. This is due to the fact that rounding errors may cause a pole that is just on the unit circle to appear outside thereby causing instability.

Another difference from FIR filters is related to linear phase filters. A major property of linear phase FIR filters is that they have mirror zero pairs, i.e., if z_0 is a zero then $\frac{1}{z_0}$ will also be a zero. For a linear phase IIR filter, poles would also have mirror pairs, i.e., for a pole inside the unit circle, there will be a mirror image outside the unit circle that will cause instability. Hence, linear phase IIR filters will not be stable.

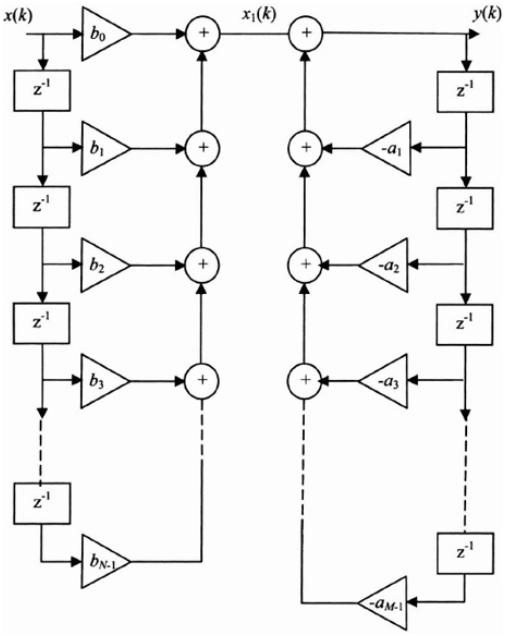

Figure 1: Direct Form I implementation of IIR filter [1]

Figure 1 above shows the direct form I implementation of an IIR filter and is obtained directly from the difference equation in equation (2) above. The all-zero filter is implemented first, followed by the all-pole filter. However, this implementation requires a large number of delay elements and can hence be optimised by placing the all-pole filter first, and this new

implementation is called direct form II (Figure 2).

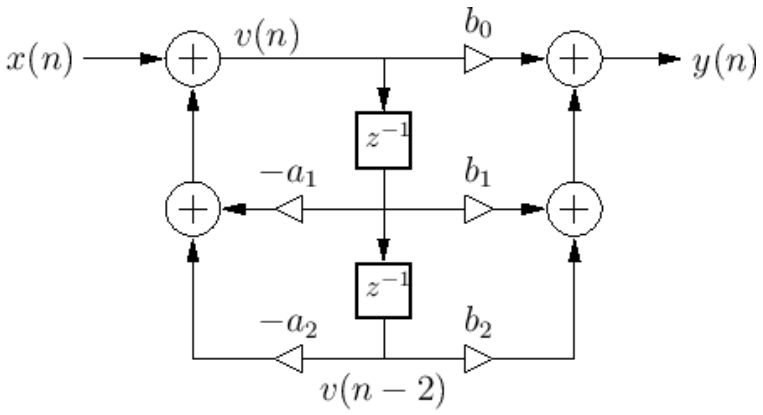

Figure 2: Direct Form II implementation of IIR filter [2]

This implementation is a lot more efficient as it has a minimum number of delay elements $max(M, N)$ where M is the order of the all-zero filter and N, the order of the all-pole filter. Hence, it is called a canonical form. Further efficiency can be achieved by using the Transposed direct form II implementation (Figure 3). This is obtained by using the *transposition or flow-graph reversal theorem*:

- Reversing the direction of all branches
- Swapping the input and output

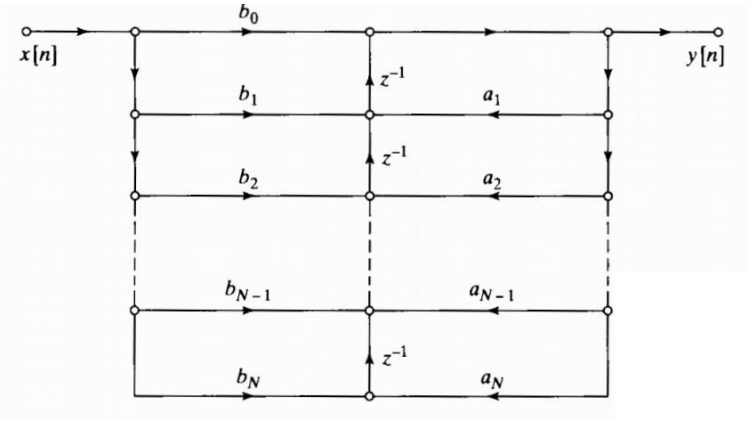

Figure 3: Transposed Direct Form II implementation of IIR filter [3]

The above figure shows the transposed direct form II implementation of the IIR filter. This is commonly used to increase the efficiency of the implementation (refer to section 4 later).

Unfortunately, the structures discussed above (and used in this lab session) are extremely sensitive to parameter quantization. This is an issue if realised on a computer, where the accuracy of filter coefficients is limited by word length or length of register used to store coefficients. This leads to imprecise pole and zero locations, leading to a transfer function that is not exactly as desired. The sensitivity of the filter frequency response characteristics to quantization of the filter coefficients however, is minimised by realising a filter having a large number of poles and zeros as an interconnection of second-order filter sections (bi-quad) [4]. Instead, IIR filters are implemented using the *cascade* or *parallel* forms which connect the bi-quad sections in series (cascade form - Figure 5) or in parallel (parallel form - Figure 4).

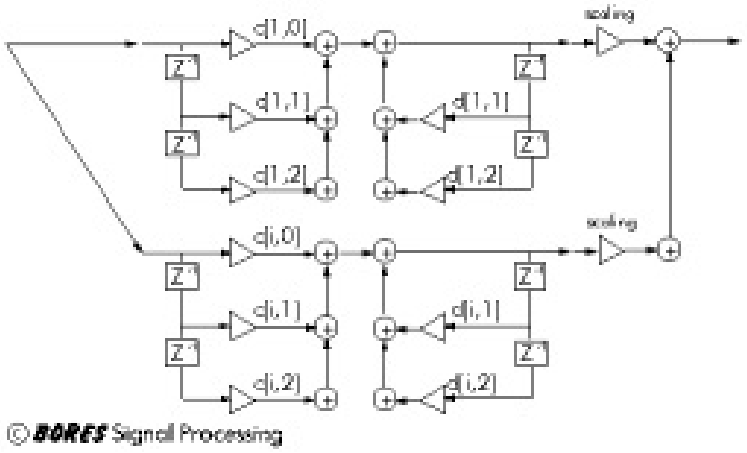

Figure 4: Parallel Form Implementation of IIR filter [5]

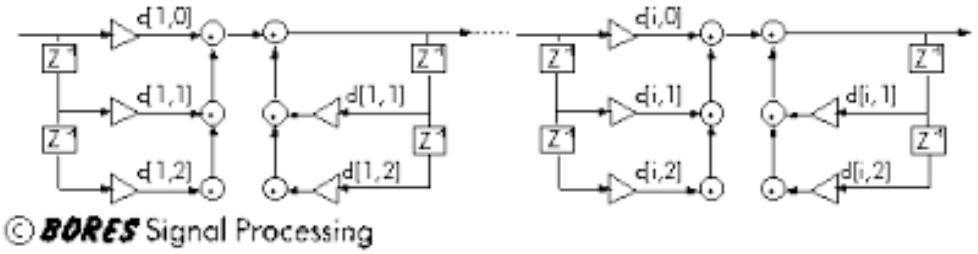

Figure 5: Cascade Form Implementation of IIR filter [6]

2 Single-pole filter

In general, a continuous-time analogue filter can be mapped to a discrete-time digital filter by the *Tustin transform*, a type of the bilinear transform. This is a very widely used transform that avoids the aliasing problem of *impulse-invariant transformations*. This uses a first order approximation of the natural logarithm function (or an approximation of an integral using trapezoids) that is an exact mapping of the s-plane to z-plane, i.e. continuous time to discrete time. There are two elements to the transform: frequency warping and a mapping from the s-plane to z-plane. The mapping takes the form (Proof in Appendix 1):

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

Assume T=2. Rearranging for z, separating s into real and imaginary parts ($s = \sigma + j\Omega$) and analysing with consideration to the unit circle:

$$z = \frac{1+s}{1-s} = \frac{1+\sigma+j\Omega}{1-\sigma-j\Omega} = re^{j\theta}$$

Thus the mapping can be broken down as follows:

- $\sigma > 0 \implies r > 1$: right half plane of s maps to exterior of unit circle in z
- $\sigma = 0 \implies r = 1$: imaginary axis in s maps to unit circle in z
- $\sigma < 0 \implies r < 1$: left half plane of s maps to interior of unit circle in z

Frequency warping takes the form as follows (Proof in Appendix 1):

$$\Omega_c = tan(\omega/2)$$

where ω is the digitised cut-off frequency of the filter as given, as a proportion of the sampling frequency, and Ω is the analogue cut-off frequency to be implemented to obtain the desired transfer function. This represents the relationship between frequency in the continuous-time filter actually implemented and frequency in the discrete-time filter, which is to be implemented. This is shown graphically in Figure 6 where ω is the x axis and Ω is the y-axis.

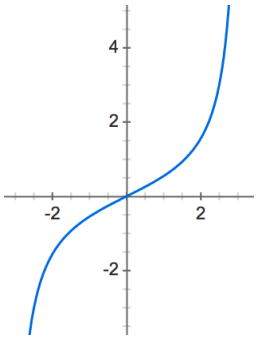

Figure 6: Single Pole Filter

From this, it is clear that at low frequencies, the relationship between the pre-warp and post-warpped frequencies is linear, and thus it is acceptable to neglect frequency warping in cases where sampling frequency is large compared to the cut-off frequency.

Therefore the full procedure for applying the Tustin transform is to firstly warp the frequency (by swapping s in the analogue filter transfer function with $\frac{s}{\Omega_c}$) and then to apply the substitution of $s = \frac{2}{T}\frac{z-1}{z+1}$.

2.1 Design

The first task of the experiment is to map the following continuous time analogue filter (Figure 7) into a discrete time version (as an IIR filter):

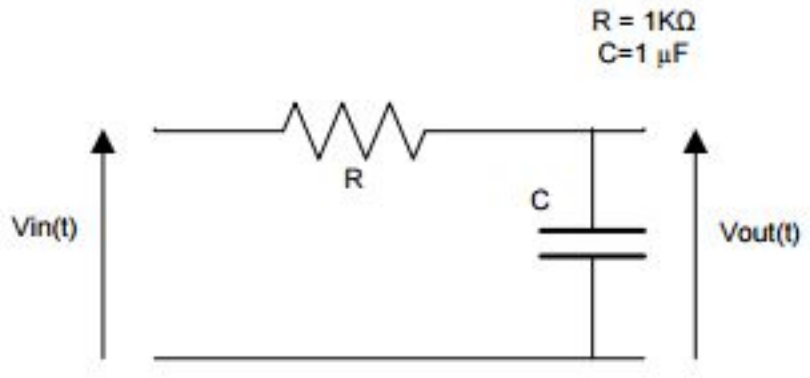

Figure 7: Single Pole Filter

This is a low-pass filter with transfer function:

$$\frac{V_{out}}{V_{in}} = \frac{1}{1 + j\omega RC} \tag{5}$$

In the Laplace domain the transfer function is:

$$\frac{V_{out}}{V_{in}} = \frac{1}{1 + sRC} \tag{6}$$

As the above analogue filter and IIR filters both have infinite impulse responses, the Tustin transform can be applied to convert the filter to a discrete time representation. As described above, frequency warping must first be applied, and thus the corner frequency must be calculated:

$$\omega = \frac{1}{RC} = 1000 rad/s$$

$$f_0 = \frac{1}{2\pi RC} = \frac{1000}{2\pi} = 159.15 Hz$$

Since this is low compared to the sampling frequency of 8kHz, the mapping from s-plane to z-plane will be good even without frequency warping. This statement is further proven in the following equation:

$$\begin{aligned}\Omega &= \frac{2}{T}tan(\frac{\omega}{2}) \\ &= \frac{2}{\frac{1}{8000}}tan\left(\frac{\frac{159.15}{8000} * 2\pi}{2}\right) \\ &= 16000tan(0.0624) \\ &= 1001.272rad/s\end{aligned} \tag{7}$$

Thus the angular frequency output of the warping is very close to the analogue angular frequency of the RC circuit in Figure 7 above, and is ignored from further calculations involved in the application of the Tustin transform. The next step is to apply the mapping from s-plane to z-plane:

$$s = \frac{2}{T}\left(\frac{z-1}{z+1}\right) \tag{8}$$

Combining equations (6) and (8) yields

$$\begin{aligned}\frac{V_{out}}{V_{in}} &= \frac{1}{1 + \frac{2RC}{T}\frac{z-1}{z+1}} \\ &= \frac{z+1}{z+1+\frac{2RC}{T}(z-1)}\end{aligned}$$

Substituting $R = 1000\Omega$, $C = 1\mu F$ and $T = \frac{1}{8000Hz} = 125\mu s$ yields:

$$\begin{aligned}\frac{V_{out}}{V_{in}} &= \frac{z+1}{z+1+16(z-1)} \\ &= \frac{z+1}{17z-15} \\ &= \frac{1+z^{-1}}{17-15z^{-1}}\end{aligned}$$

Thus there is a zero at $z = -1$ and a pole at $z = \frac{15}{17}$. The digitised RC filter results in the following zero-pole plot:

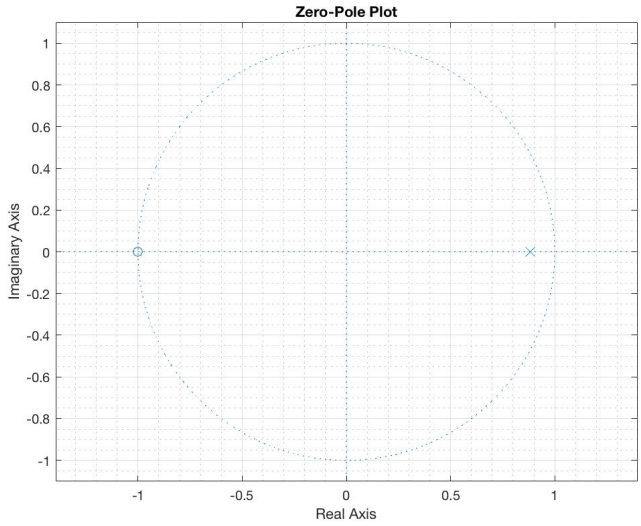

Figure 8: Zero-Pole plot of the digitised RC filter

In difference equation form,

$$y[n] = \frac{1}{17}x[n] + \frac{1}{17}x[n-1] + \frac{15}{17}y[n-1]$$

where y is the output and x, the input.

The filter is first implemented in MATLAB to obtain a theoretical response for the analogue filter, along with the input and output high pass filters of the AIC chip. The input high pass filter has a cut-off frequency of 9.675Hz (to three decimal places), which was obtained using the line input impedance of the AIC chip (found to be $35k\Omega$) from the datasheet [7].

$$f = \frac{1}{2\pi * RC} = \frac{1}{2\pi * 35000 * 470 * 10^{-9}} = 9.675Hz$$

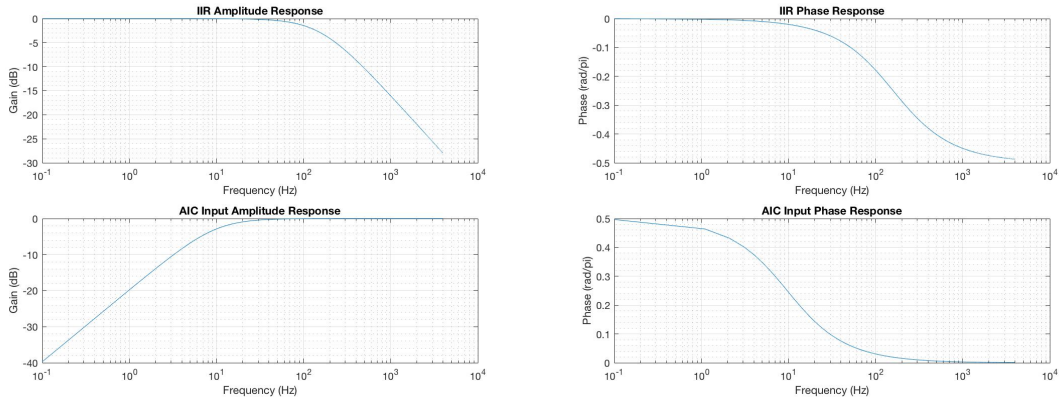

Figure 9: Amplitude and Phase responses of the low-pass RC filter and the high-pass input filter of the AIC

Similarly, the output high-pass filter has a cut-off frequency of 7.189Hz (to three decimal places).

$$f = \frac{1}{2\pi * RC} = \frac{1}{2\pi * (47000 + 100) * 470 * 10^{-9}} = 7.189Hz$$

Combining, the above three filters together, the amplitude and phases responses in Figure 10 were obtained. The amplitude response is that of a band-pass filter from around 20Hz to 130Hz (approximately) with the low frequency slope as 40dB/decade and the high frequency decay as -20dB/decade.

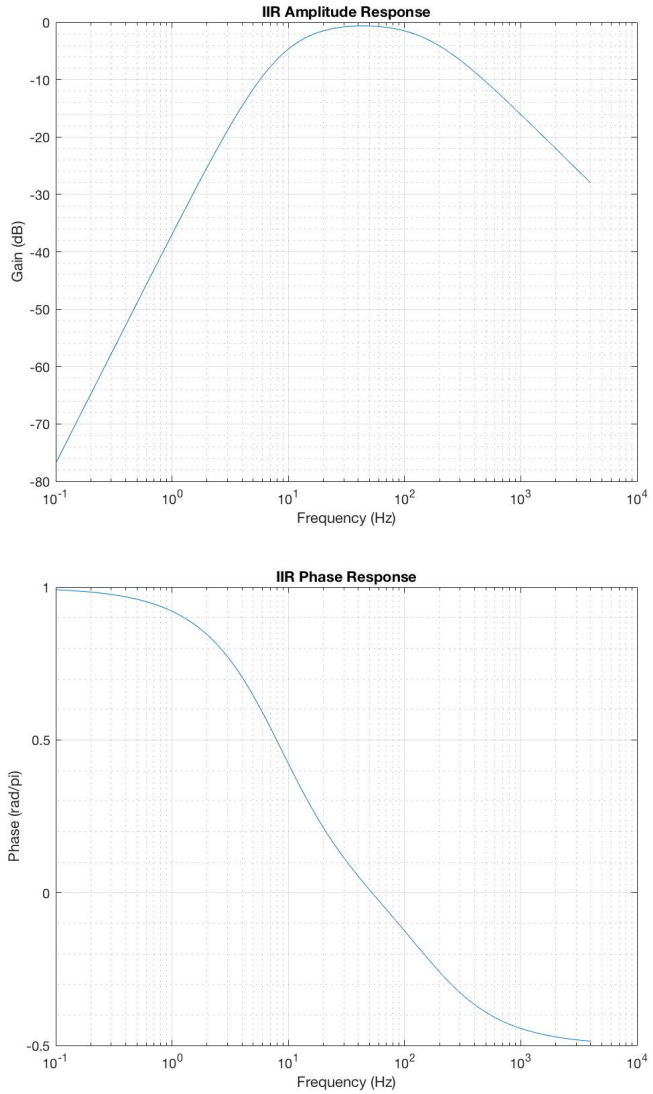

Figure 10: Amplitude and Phase responses of the combined filter

2.2 Digital Implementation

In order to implement this filter digitally, the time-domain difference equation must be used:

$$y[n] = \frac{1}{17}x[n] + \frac{1}{17}x[n-1] + \frac{15}{17}y[n-1]$$

As the software cannot handle divisions, these coefficients were implemented with 8 digits of precision after the decimal point as can be seen in Listing 1 below:

```
//initialisations
double y[2] = {0};          //output buffer
double x[2] = {0};          //input buffer

//Interrupt Service Routine
void ISR_AIC()
{
  double a1 = 0.05882353, a2 = 0.88235294; //locally specified
    coefficient values
  x[1] = x[0];                //time shifting of current samples at
    index 0
  y[1] = y[0];
  x[0] = mono_read_16Bit();        //reading in current input
  y[0] = a1*(x[0]+x[1])+a2*y[1];       //calculation of current output
  mono_write_16Bit((short)y[0]);       //writing to output
}
```

Listing 1: RC filter code implementation

The buffers are specified with size 2 as only the current and the last value of input and output are required, i.e., $x[n]$, $x[n-1]$, $y[n]$ *and* $y[n-1]$. These are stored in the zeroth and first indices of the corresponding buffers respectively.

In the ISR, the elements are first shifted as they become the past values. Next, the current input is read and this, along with the past input and past output, is used to generate the current output, which is stored in $y[0]$. Similar to the previous laboratory sessions, the signal generator is used to create the input, which is read by `mono_read_16Bit()` and the output is written by `mono_write16Bit()`.

Before the implementation was run, the configuration file was changed to allow for a dynamic heap in the Internal RAM (IRAM) of the DSK6713 chip. This is because, henceforth, the data will no longer be stored in static arrays, but rather in dynamic arrays which are created during run time. This above configuration change enables the use of the `malloc()` function or the `calloc()` function which allow the dynamic creation of arrays based on the array size and size of the data type. Hence, the array size and array itself are made customisable, which is a desirable feature.

Upon implementing the code on the DSK, the filter was first verified by plotting a sine wave at 100Hz. The output was of the expected form and amplitude, i.e. approximately half of the input due to the quarter gains in both left and right lines which when combined, result in a gain of a half. This confirmed the correct operation of the code.

In order to determine the time constant, the correct range of frequencies had to be chosen. If the frequency is too high, the capacitor will not charge and discharge completely during each cycle as the time period is a lot smaller than the time constant (second image in Figure 11). Similarly, for very low frequencies, the lower harmonics (square wave is the sum of infinite odd harmonic sinusoids) will be cut out by the high pass filters resulting in a poorly shaped square wave input.

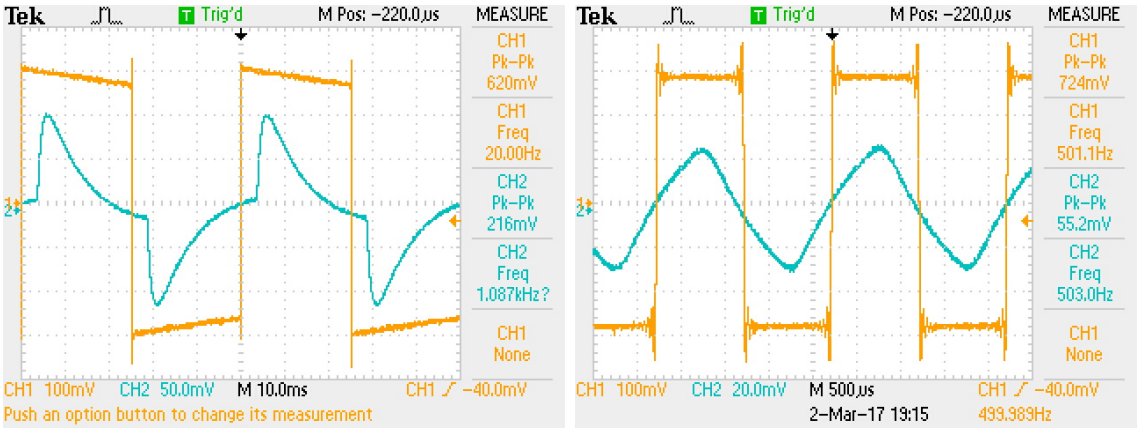

Figure 11: 20Hz and 500Hz square wave inputs with corresponding outputs

As 150Hz is quite close to the cut-off frequency, it was chosen as a compromise between the two extremes. For this case, the output seemed to grow just enough so that it almost saturates as can be seen in Figure 12 below:

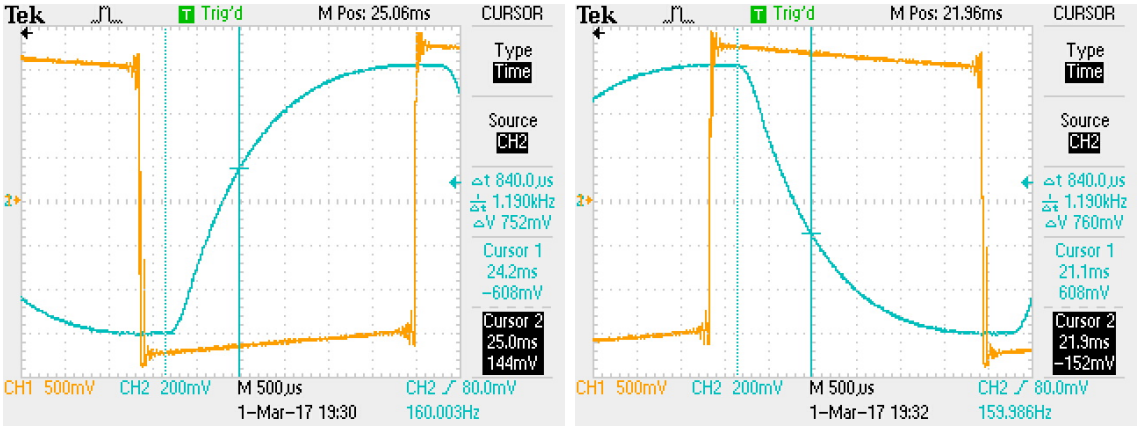

Figure 12: Exponential increase and decrease of output

The images indicate that the output grows when the input decays, which is contrary to what is expected. This is because there is a negative gain in the circuit that results in the output being flipped. So, the output is in fact decaying (for the figure on the left) and growing (for the figure on the right) after a small amount of time (delay in circuit). From the figures, it can be seen that the time constant is $840\mu s$ (using the fact that in one time

constant the signal grows and decays to 63% of its peak to peak value), in contrast to the 1ms expected value. This is due to the non-ideality in the mapping from the analogue to the digital domain by using the bilinear transform and also the non-idealities in the hardware implementation, as explained shortly.

The amplitude and phase responses for the filter were then measured using the APX520 spectrum analyser and can be seen in the figure below:

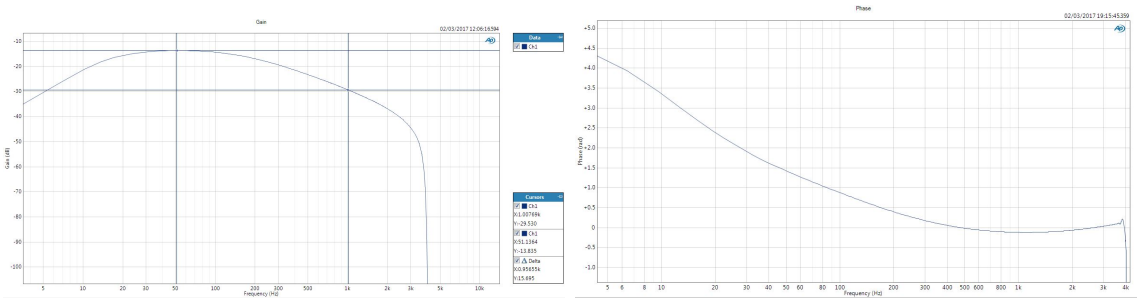

Figure 13: Amplitude and Phase responses from the spectrum analyser

These are quite similar to the theoretical plots obtained for the combined filter in Figure 10 above. As expected, the amplitude response has shifted down to -13.8dB (approximately 1/4 gain:-12dB) due to the fact that only one channel is used, which has quarter gain. The reason it isn't exactly -12dB is due to further gains in the circuit that make the output smaller than half (1.47V output for 3.3V input in section 3.2). This was further analysed by implementing an all-pass filter and finding the amplitude and phase responses as can be seen in Figure 14.

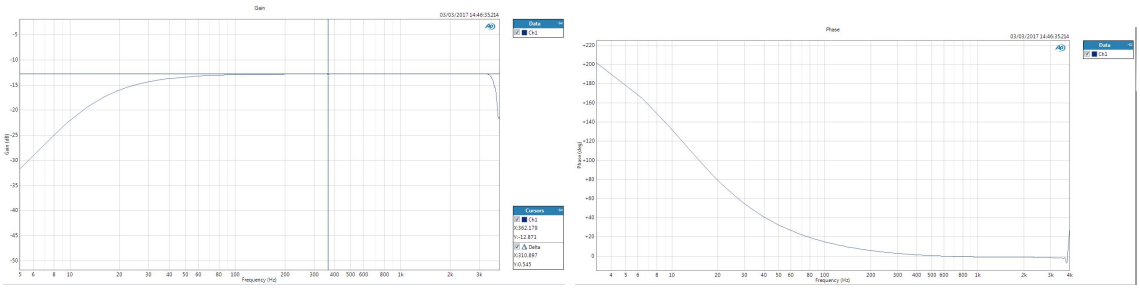

Figure 14: Amplitude and Phase responses of all-pass filter from the spectrum analyser

As per the above responses, the gain in the passband is in fact -12.8dB indicating that the remaining -1dB is from the digital implementation of the RC filter. In addition, the high-pass response can be observed in both the amplitude and phase responses along with the anti-aliasing filter at the end.

By placing cursors on different frequencies on the high frequency side of the amplitude response in Figure 13, the difference in gain was found to be identical to that of similar

frequencies in the theoretical response. However, for the low frequency side, the corner frequency differed from expected as the cursor differences did not yield similar results.

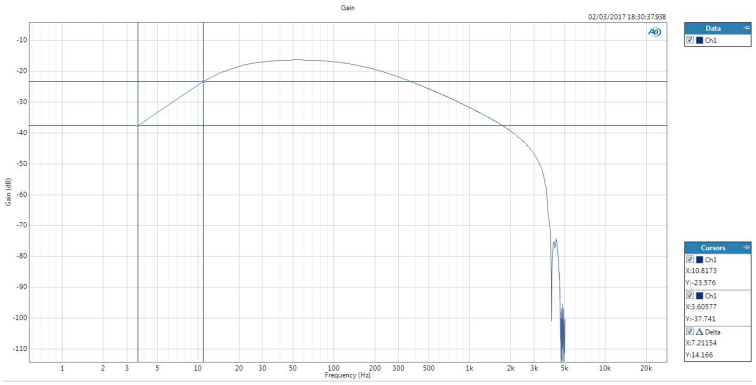

Figure 15: Low frequency response cursors

The difference between cursors at 3Hz and 10Hz was 14dB which is quite similar to a slope of 40dB/decade (on comparing the slope to a 40dB/decade plot). Similarly, the phase response in Figure 13 is also similar to the theoretical response in Figure 10. Hence, the experimental response mirrors the expected theoretical response.

Nevertheless, the spectrum analyser gives a corner frequency of 180Hz, as the 3dB difference occurs there, instead of 160Hz. This further explains the deviation in experimental time constant of 0.84ms from the theoretical value of 1ms.

3 Bandpass Filter: Direct form II

3.1 Filter Coefficients

An elliptic bandpass filter with the following specifications was to be implemented:

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

The filter coefficients required to achieve these specifications can be calculated in MATLAB using the `ellip` function. This function takes in order the inputs:

- `n`, where `2n` is the desired order of the filter. The reason for this is that an elliptic filter always has an even number of poles
- `rp`, the desired passband ripple specified in decibels
- `rs`, the desired stopband attenuation in decibels
- `pb`, a 2 element vector [w1 w2], where $w1 < w2$, of the passband frequencies normalised with respect to the Nyquist frequency
- the type of filter as a string (low, high, stop or bandpass)

As the sampling frequency to be used is 8000Hz, `pb = [200 450]/4000`.

The function returns two 2n+1 element row arrays `a` and `b` containing the coefficients of the filter. These correspond to the same a and b as in equation (1). Note that the first element of `a` will always be 1 as it corresponds to the weighting of the current input.

This is implemented as shown in Listing 2 below:

```
%calculate filter coefficients
n = 7;
rp = 0.3;
rs = 20;
pb = [200 450]/4000;
[b,a] = ellip(n,rp,rs,pb,'bandpass');
```

Listing 2: Calculation of elliptic filter coefficients in MATLAB

These coefficients are saved to a .txt file called iir_coef.txt, formatted in the syntax of C code through use of a series of `fprintf` functions. This allows the file to be read in code composer by an `#include` statement. It is important to note that the coefficients are saved to double precision, as unlike FIR filters, precision errors compound infinitely for IIR filters due to the feedback.

The amplitude response, phase response and zero-pole plots of the resulting filter are displayed below in Figure 16.

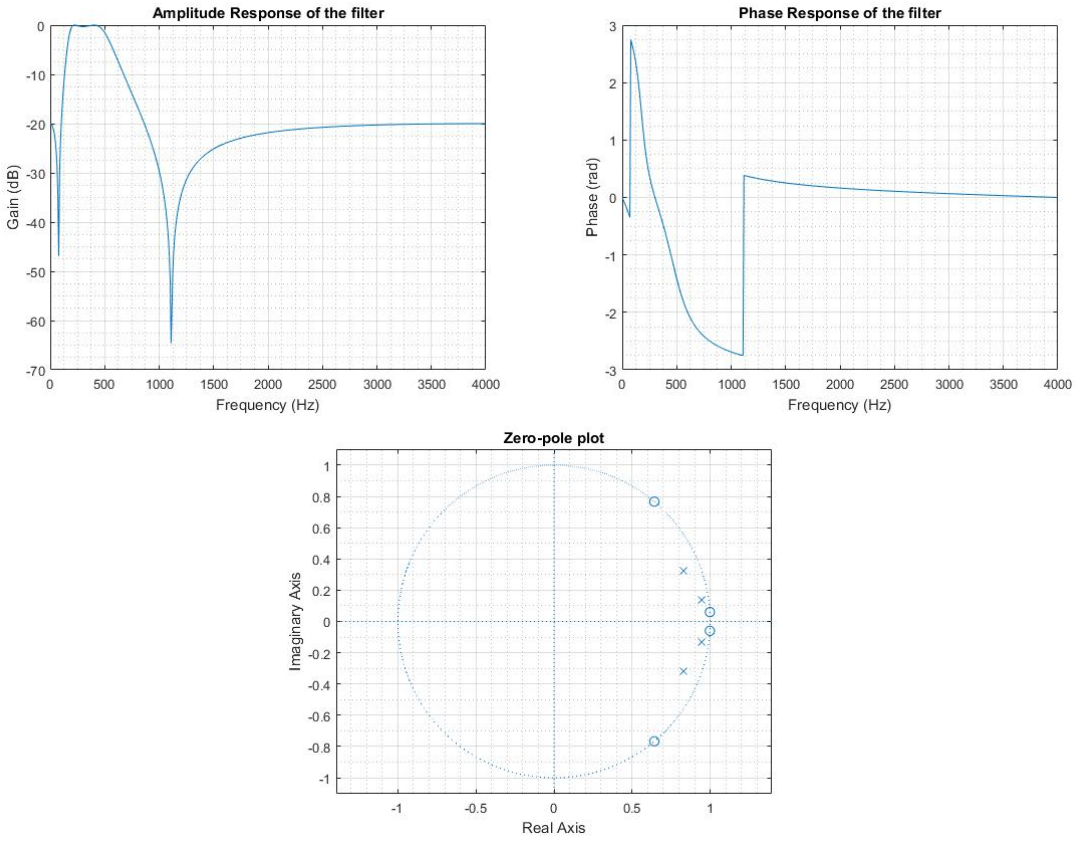

Figure 16: Amplitude response, phase response, and zero-pole plot of desired fourth order filter

From the amplitude response it is clear that the passband specifications are met. However, it was observed that the higher frequency transition band was quite large (450Hz to 2500Hz approximately) before settling down to the required stop-band attenuation.

The pole-zero plot is as expected. There are zeros on either side of the passband. One is at 78Hz and the other at 1109Hz as can be seen in the amplitude response and the zero-pole plot (in Figure 16). The poles are on an arc around those points in the unit circle that represent the passband. In the case of a fourth order filter, there are not enough poles to see the arc.

The next step was to implement the IIR algorithm in the hardware using Code Composer.

3.2 IIR Algorithm

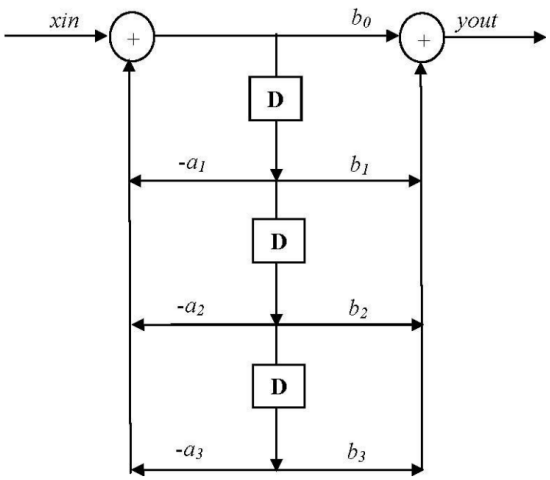

Figure 17: Direct Form 2 structure of third order IIR filter

Figure 17 shows the signal flow graph for the direct form 2 structure of a third order IIR filter, which can be used to construct the filtering algorithm. Physically, this diagram represents applying the all-pole filter to the input first, and then implementing the all-zero filter. The filter coefficients a_i and b_i are extracted from the iir_coef.txt file, as described in section 3.1, using the line `#include "iir_coef.txt"`.

In order to implement the filter, an array `x` of size N+1 (where N is the order of the filter), must be initilised. In this specific case, the order of the filter is N=4, and so `x` is of size 5. The memory of `x` is dynamically allocated using the `calloc()` function (Listing 4). The array `x` is used to store the values in the middle column in Figure 17. Instead of hard coding the value of N, it was calculated using the size of one of the arrays containing a set of filter coefficients. Therefore the size of either `a` or `b` in the text file `iir_coef.txt` could be used. `a` was chosen in this specific case (Listing 3).

```
1 #define N sizeof(a)/sizeof(double)-1 //definition of N before the main
    function
2 double* x;     //initialisation of pointer x to first element of array
```

Listing 3: Initialisation of N and declaration of x

```
1 x = (double *)calloc(N+1, sizeof(double)); //initialisation of x using
    calloc()
```

Listing 4: Initialisation of x in the `main()` function

The filtering algorithm implemented firstly reads the input from the signal generator as `input`, and assigns it to the first element of array `x`, `x[0]`. The input is not directly assigned to `x[0]` from `mono_read_16Bit()` as the number of clock cycles taken is not to include the

time taken to read a sample or write to the output. As explained earlier, this array stores the values in the middle column of Figure 17. Initially, three `for` loops were used. The first loop performed the duty of the first feedback sum (all-pole filter) where the current input is summed with the product of the negative of the coefficients in the array `a[]` and corresponding `x` values, thus calculating the result of the leftmost addition block in Figure 17. From this point onwards, all `x` values correspond to the values in the middle column. The second `for` loop performed the duty of the second sum (all-zero filter) where the output is calculated through a weighted sum of `x` values, where the weights are the corresponding values in the array `b[]`. The last `for` loop would shift all values of `x` downwards, i.e., a delay of one sample, in order to prepare to receive the next input. This is a fairly naive implementation of the algorithm (Listed in Appendix 2), and can be optimised.

The first optimisation made to this algorithm was to concatenate the first and second `for` loops described above into one (Listed in Appendix 3). This offered a slight improvement in efficiency, as shown later in Table 1. In the same vein, the `for` loop which shifts the samples can also be implemented into the same loop, further optimising the code. This particular algorithm is shown in Listing 5:

```
void ISR_AIC()
{
  int i;
  double y=0; //initialising output variable
  double input = mono_read_16Bit(); //read input value

  x[0] = input; //initilaise output of first adder block
  for (i=N;i>0;i--)
  {
    x[0] -= x[i]*a[i];//all-pole accumulation
    y += b[i]*x[i];//all-zero accumulation
    x[i] = x[i-1];//time shift
  }
  y += b[0]*x[0];//assign output
  mono_write_16Bit((short)y);//write to output
}
```

Listing 5: C implementation of direct form II IIR filter

Figure 18 below validates the correct operation of this band-pass filter implementation by using inputs of different frequencies.

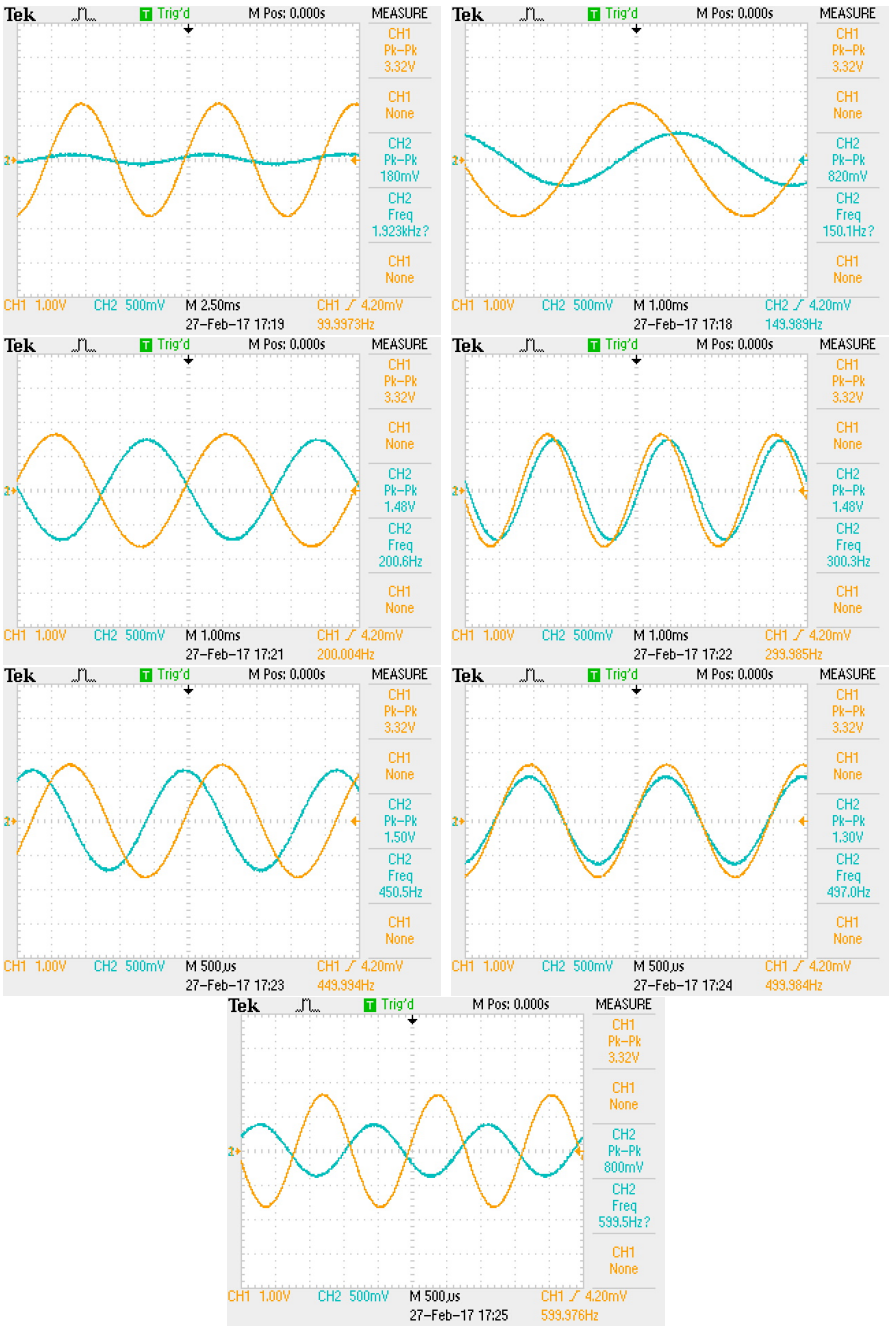

Figure 18: Scope traces for inputs at 100, 150, 200, 300, 450, 500 and 600Hz

As evidenced by the fairly constant amplitudes of the output waveforms (channel 2, blue), the passband ranges from 200Hz to 450Hz. The amplitudes at 100Hz, 150Hz, 500Hz and 600Hz however are subjected to varying levels of attenuation. Therefore, the filter operates as desired.

Further testing was carried out using the APX520 spectrum analyser. The amplitude and phase responses obtained from a frequency sweep are shown in Figure 19:

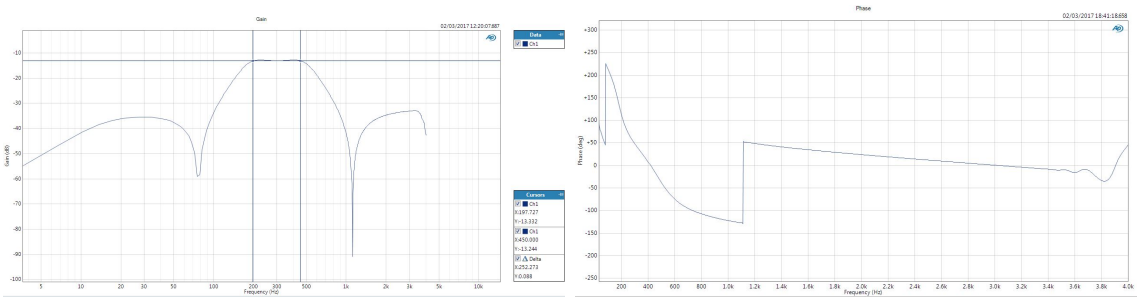

Figure 19: Amplitude and Phase responses obtained from the APX520

Note that the gain of the passband is at roughly -13dB (explained in section 2.2 page 13). By comparing with the MATLAB plots from Figure 16, it can be observed that the results are similar. For the phase response, the 'input-output' excess option was used on the APX520 in order to achieve the same response plotted by MATLAB. The difference between the modes [8] for plotting phase is described below:

- **Relative to Ch1**
  - In this mode, the absolute phase of channel one is subtracted from the absolute phase of channels numbered greater than 1. The result is plotted against frequency for each channel numbered 2 and above, showing the phase difference (from channel 1) for each channel. Since channel 1 is used as the reference, it is not plotted in this result. This mode shows "unwrapped" phase differences.
- **Input-to-output**
  - In the input-to-output mode, the absolute phase of each channel, from device input to device output, is plotted against frequency, "unwrapped." Input-to-output mode includes device delay.
- **Input-to-output, wrapped**
  - This mode shows the same result as input-to-output phase, but "wrapped" within the range of -180° to +180°.
- **Input-to-output, excess**
  - This mode shows the input-to-output phase (unwrapped), but removes the linear component (the average group delay of the system), leaving the "excess phase."

As a result, using the *input-output,excess* mode gets rid of the average delay cause by the system and hence, what remains is the phase response of the implemented filter.

3.3 Higher order filters

To find the coefficients of higher order filters, MATLAB was again utilised. After verifying the correct operation of these filters by sweeping through frequencies, the number of instruction cycles required per cycle using the algorithm with 2 `for` loops was recorded. This is detailed in Table 1 below.

Table 1: Instruction cycles against filter order, 2 `for` loops

Filter Order	4	6	8	10	12
Instruction Cycles No Optimisation	512	722	932	1142	1352
Instruction Cycles -O2	236	298	360	422	484

When no optimisation is used, the number of cycles and filter order follow the linear relationship $92 + 105N$, where N is the filter order. At optimisation level -O2, the relationship is $112 + 31N$. As expected, optimisation level -O2 consistently improved the required cycle numbers. Interestingly, the optimisation level seems to reduce the efficiency of the code for very low order filters, as the constant term in the relationship is higher at -O2, but the scaling of clock cycles against order is drastically lowered. In reality, this would not occur as the lowest order elliptic filter that can be implemented is of order 2 (as the number of poles has to be even). These results can be seen in the first image in Figure 20.

When using the algorithm with one `for` loop (Listing 5) the results were much more desirable, as shown in Table 2:

Table 2: Instruction cycles against filter order, optimised with one `for` loop

Filter Order	4	6	8	10	12
Instruction Cycles No Optimisation	413	583	753	923	1093
Instruction Cycles -O2	211	253	301	349	397

Without optimisation, the number of cycles follows the equation $73 + 85N$ where N is the order of the filter. With the optimisation level set to -O2 however, the number of cycles follows $115 + 24N$. However, this was not the case for the very first jump from N=4 to N=6, where the relationship is $127 + 21N$.

As with the code with 2 `for` loops, the constant in the relationship is higher, meaning low order filters will be less efficient with optimisation set to -O2, but scale much more desirably with filter order. Yet again, in reality, the order cannot go below 2 and hence, -O2 optimisation is always faster. These results can be seen in the second image in Figure 20

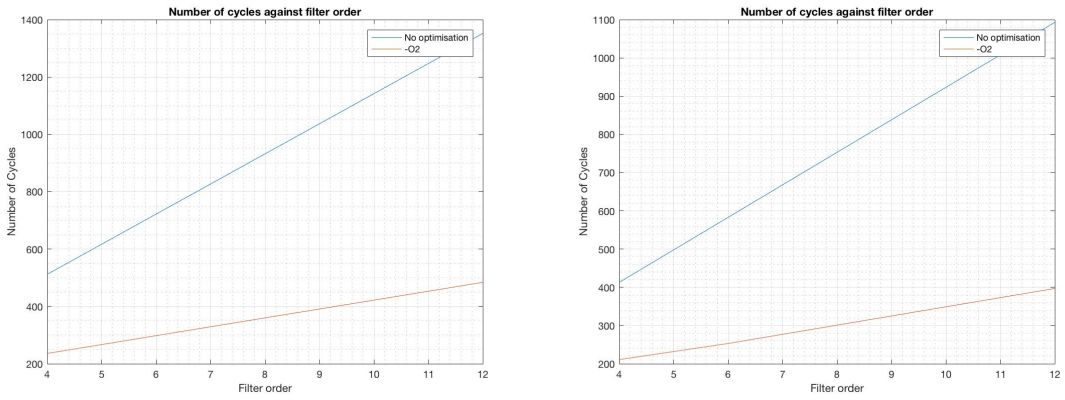

Figure 20: Number of cycles against filter order for 2 `for` loops and 1 `for` loop

As the order of the filter was increased, the number of poles in the arc increases, thereby increasing the occurrence of ripples (not in magnitude). However, the cut-off on either side of the passband gets larger as seen for 6th order and 12th order filters in Figure 21. This is a major difference between elliptic and other analogue filters such as Chebyshev and Butterworth filters as elliptic filters use zeros along with poles to obtain sharper cut-offs than 20dB/decade. Nevertheless, this leads to ripple in the passband and stopband.

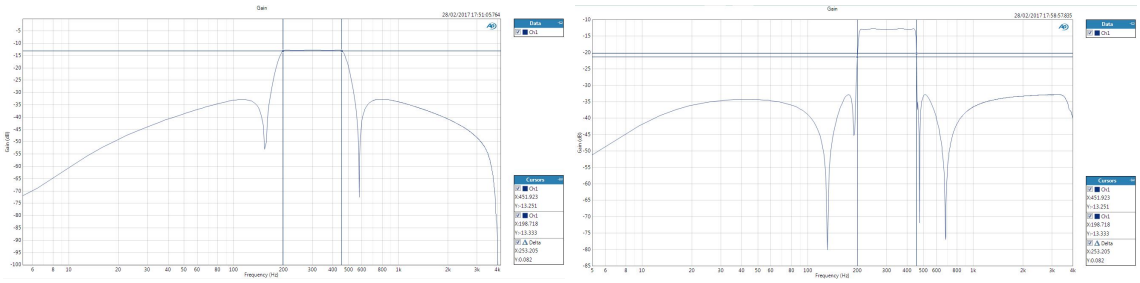

Figure 21: Amplitude responses of 6th and 12th order elliptic filters

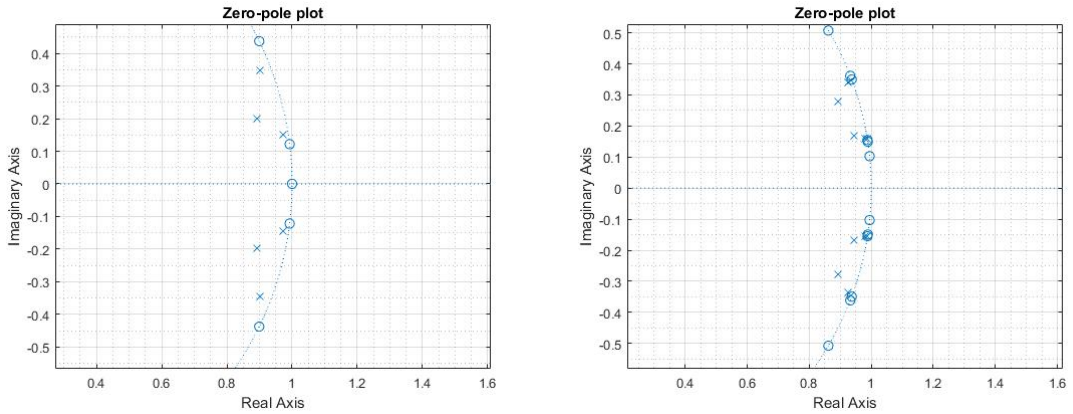

Figure 22: Zero-Pole plots of 6th and 12th order elliptic filters

Figure 22 shows the pole-zero plots for the 6th and 12th order elliptic filters. As stated in section 3.1 earlier, the poles are seen to be in an arc around the points in the unit circle which represent the passband frequencies.

The filter was only tested up to an order of 12, as beyond that, the filter becomes unstable due to a pole being placed outside the unit circle in the z-plane. This can be determined easily by using the `isstable` function in MATLAB. This function takes in input the filter coefficients, `b` and `a`, and returns a boolean specifying whether or not the filter is stable. The case of N=14 is shown in Figure 23, where it is evident that there is a pole output the unit circle.

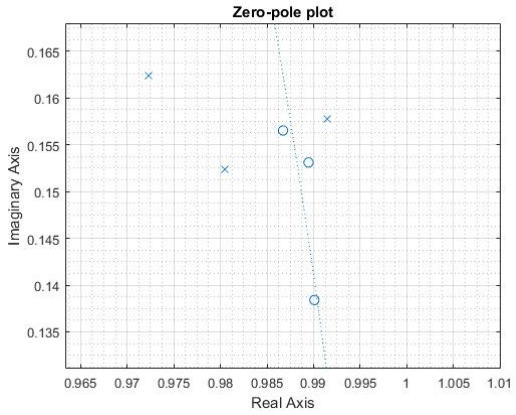

Figure 23: Zero-Pole plots for a 14th order elliptic filter

The method that MATLAB applies to design an elliptic filter should theoretically always give a stable filter. This is not the case in practice however, due to the quantisation effect of the computer as alluded to in section 1. This problem can be circumvented by designing the filter by its positioning of poles and zeros rather than its transfer function. This design methodology cannot be used however, as the algorithm used to filter input signals requires multiplication by filter coefficients, therefore, the transfer function methodology must be used.

4 Bandpass filter: Direct form II transposed

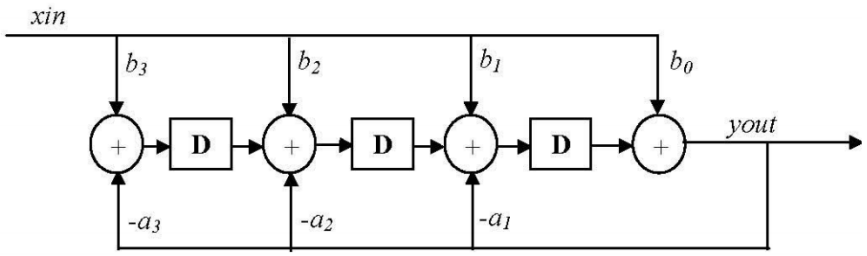

Figure 24: Direct form II transposed structure

The direct form II transposed structure (Figure 24) allows for an even more efficient implementation of the IIR filter. This is due to the fact that unlike the non-transposed direct form II, there is no need to shift values. The only operation required to be implemented in a loop therefore is the weighted summation of the current input, current output and the delayed, weighted previous element (as represented by the middle row of Figure 3), with the weights being the elements of the arrays `a` and `b` respectively. The omission of the costly shifting operation means that this is a simple MAC operation, which is particularly advantageous when applied in the context of a DSP chip. Array `x` is again used to store the intermediate values between input and output, which are generated by the summation blocks in Figure 24. This array is of length N+2 and is allocated memory on the fly, much like in the non-transposed form. This is shown in Listing 6 (Note that Figure 24 is for a filter of order 3.)

```
x = (double *)calloc(N+2, sizeof(double)); //initialisation of x using
    calloc()
```

Listing 6: Initialisation of x in the main function for transposed form

Thus the algorithm can be implemented as in Listing 7:

```
//Interrupt service routine implementing a fourth order IIR filter
void ISR_AIC()
{
  int i;
  double input = mono_read_16Bit();//read input value

  x[0] = x[1] + b[0]*input;//assign output
  for (i=1;i<=N;i++)
  {
    x[i] = x[i+1] + (b[i]*input)-(a[i]*x[0]);//perform all-zero and all
    -pole accumulations
  }
  mono_write_16Bit((short)x[0]);//write to output
}
```

Listing 7: IIR direct form II transposed

This algorithm reads the input from the signal generator as `input`, multiplies it by the first coefficient of the all zero filter, `b[0]`, adds it to the second element of `x`, `x[1]`, and assigns this as the first element of `x`, `x[0]`, to be passed as the output. This is done prior to the rest of the algorithm as the current output is required for the other terms in `x`. The signal is correctly filtered due to the construction of the other values of `x` by the `for` loop. The equation in the `for` loop emulates the summation blocks in Figure 24, starting from right to left. Once the loop has been completed, the value stored in `x[1]` is the value required to produce the next output. The reason that `x` is of size $N + 2$ is to allow the last element of `x` to be processed within the `for` loop, instead of doing so outside. The extra last element of `x` is always kept at zero. Another optimisation that was performed was making N a pre-processor directive (`#define`) instead of an `int`. This is because, whenever N occurs in the code, it is replaced by its definition and hence, this saves memory.

The performance of this code was tested as before in section 3. The improvement in efficiency is apparent from Table 3.

Table 3: Instruction cycles against filter order

Filter Order	4	6	8	10	12
Instruction Cycles No Optimisation	282	392	502	612	722
Instruction Cycles -O2	108	119	129	139	149

When no optimisation is used, the number of instructions cycles and filter order are linearly related by $62 + 55N$, where N is the filter order. When optimisation is set to -O2, the relationship becomes $89 + 5N$. These equations plotted in Figure 25 below:

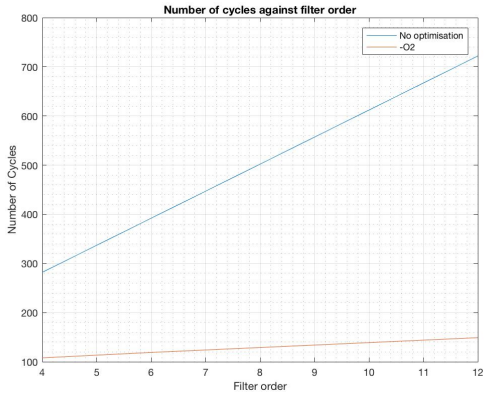

Figure 25: Number of cycles against filter order

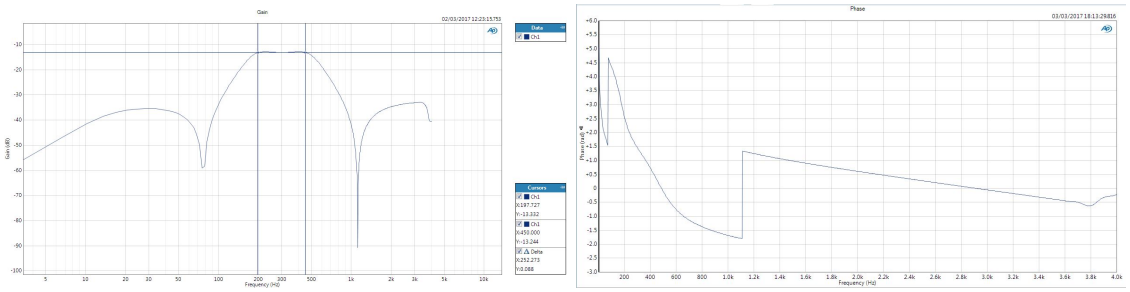

Figure 26: Amplitude and Phase responses obtained from the APX520

When comparing the amplitude and phase response for the transposed implementation (in Figure 26) to those of the non-transposed implementation (in Figure 19), it can be seen that they are identical. Hence, the change in implementation did not affect the response of the filter.

As stated earlier in this section, the transposed form eliminates the need for shifting operations; the shifting is inherently implemented in the calculations, which saves some overhead and as a result, improves efficiency. This was confirmed by comparing Table 3 with Table 2. Moreover, the number of cycles taken by the transposed implementation only increases by 5 with an increase in order, compared to the increase in 24 with the order for the non-transposed implementation, further confirming the improvement in efficiency.

5 Importance of Precision

As stated in section 1, precision is extremely important for an IIR filter. Coefficient quantisation could cause the output to become unstable as a pole on the unit circle may appear outside. In order to experimentally determine this, the precision of the coefficients and that of the array `x` are changed to `float` from `double`.

For filter orders up to 6, the output was identical to that of double precision. However, from order 8 onwards, the output started becoming unstable as seen in the blue scope trace in Figure 27.

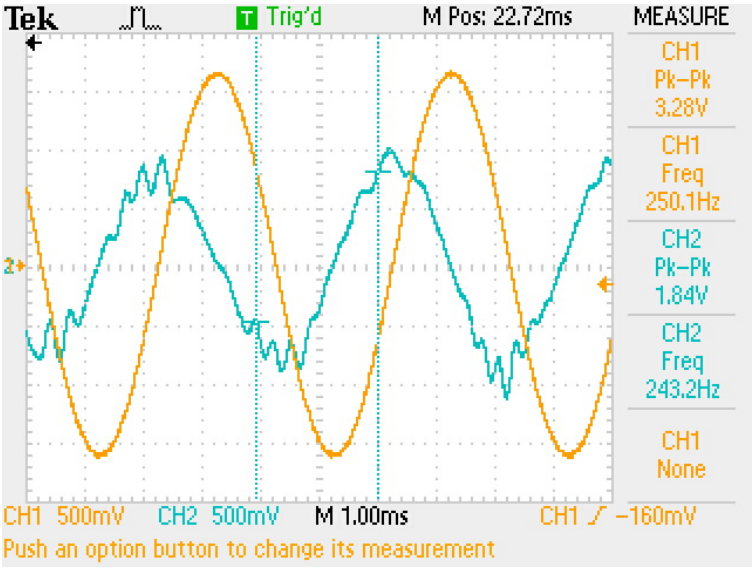

Figure 27: Output for order 8 filter with single precision coefficients

Moreover, no output was obtained for filters of order 8 and above if the precision was limited while generating the .txt files in MATLAB as the rounding caused the output to grow exponentially till it could not be sustained. Hence, the precision in MATLAB was set to 15 digits after the decimal point to avoid this issue.

Comparing this to the results of the previous laboratory session with FIR filters, it can be seen that a very small order such as 8 causes instability in IIR filters while orders as large as 500 do not cause the errors in FIR filters. Therefore, it is clear that IIR filters are more likely to become unstable which is a disadvantage.

References

[1] Mohammed Najim. Structure of iir filters. `https://www.safaribooksonline.com/library/view/digital-filters-design/9781905209453/ch007-sec003.html`. [Online; accessed 2017-03-01].

[2] Julius O. Smith. *Introduction to Digital Filters with Audio Applications, September 2007 Edition*. 2007. https://ccrma.stanford.edu/ jos/filters/Direct_Form_II.html [Online; accessed 2017-03-01].

[3] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1989. http://d1.ourdev.cn/bbs_upload782111/files_24/ourdev_523225.pdf [Online; accessed 2017-03-02].

[4] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications, 3rd Edition*. 1996. http://itl7.elte.hu/ zsolt/Oktatas/editable_Digital_Signal_Processing_Principles_Algorithms_and_Applications_Third_Edition.pdf [Online; accessed 2017-03-02].

[5] BORES Signal Processing. Introduction to dsp - iir filters: parallel and cascade iir structures, 2009.

[6] BORES Signal Processing. Introduction to dsp - iir filters: parallel and cascade iir structures, 2009.

[7] Texas Instruments. *TLV320AIC23b Stereo Audio Codec Data Manual*. 2004. [Online; accessed 2017-02-07].

[8] Audio Precision. Audio precision apx500 version 4.2 user manual, 2016.

Appendices

A Proof of the Bilinear Transform

Consider an analogue linear filter with transfer function in the Laplace domain:

$$H(s) = \frac{b}{s+a}$$

This can also be expressed as a differential equation:

$$\frac{dy(t)}{dt} + ay(t) = bx(t)$$

Integrating yields:

$$y(t) = \int_{t_0}^{t} y'(\tau)d(\tau) + y(t_0)$$

where $y'(t)$ denotes the derivative of $y(t)$.
Approximating this integral by the trapezoidal formula at $t = nT$ and $t_0 = nT - T$

$$y(nT) = \frac{2}{T}[y'(nT) + y'(nT - T)] + y(nT - T)$$

Thus the differential equation evaluated at $t = nT$ yields:

$$y'(nT) = -ay(nT) + bx(nT)$$

Substituting this expression for the derivative into the approximation of the integral yields:

$$(1 + \frac{aT}{2})y(n) - (1 - \frac{aT}{2})y(n-1) = \frac{bT}{2}[x(n) + x(n-1)]$$

The z-transform of this difference equation is:

$$(1 + \frac{aT}{2})Y(z) - (1 - \frac{aT}{2})z^{-1}Y(Z) = \frac{bT}{2}(1 - z^{-1})X(z)$$

The transfer function of the equivalent transfer function is therefore:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(bT/2)(1 + z^{-1})}{1 + aT/2 - (1 - aT/2)z^{-1}}$$

or equivalently:

$$H(z) = \frac{b}{\frac{2}{T}(\frac{1-z^{-1}}{1+z^{-1}}) + a}$$

Clearly the mapping from s-plane to the z-plane is:

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

The frequency response is obtained by evaluating the function $s = \sigma + j\Omega$ around the unit circle i.e. $z = e^{j\omega}$:

$$\begin{aligned} s &= \frac{2}{T}\frac{z-1}{z+1} \\ &= \frac{2}{T}\frac{e^{j\omega}-1}{e^{j\omega}+1} \\ &= \frac{2}{T}\left(j\frac{sin\omega}{1+jcos\omega}\right) \end{aligned}$$

Consequently,

$$\sigma = 0$$

$$\begin{aligned} \Omega &= \frac{2}{T}\frac{sin\omega}{1+cos\omega} \\ &= \frac{2}{T}tan(\frac{\omega}{2}) \end{aligned}$$

Rearranging,

$$\omega = 2tan^{-1}\frac{\Omega T}{2}$$

This frequency mapping is extremely non-linear and is known as frequency warping. The mapping is graphed below in Figure 28:

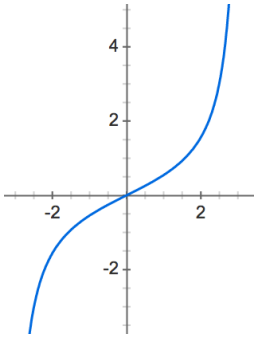

Figure 28: Mapping of frequency response

Thus the higher the desired analogue frequency, the more different the actual implemented analogue frequency will be.

B 3 Loop naive implementation of non-transposed direct form IIR

```
void ISR_AIC()
{
  int i;
  double y=0;//initialise output
  double input = mono_read_16Bit();//read input value
  for (i=N;i>0;i--)
  {
    x[i] = x[i-1];//shift samples
  }
  x[0] = input;
  for (i=1;i<=N;i++)
  {
    x[0] -= x[i]*a[i];//all-pole accumulation
  }

  for (i=0;i<=N;i++)
  {
    y += b[i]*x[i];//all-zero accumulation
  }
  mono_write_16Bit((short)y);//write to output
}
```

C 2 Loop optimised implementation of non-transposed direct form IIR

```
void ISR_AIC()
{
  int i;
  double y=0;//initialise output
  double input = mono_read_16Bit();//read input values

  x[0] = input;
  for (i=1;i<=N;i++)
  {
    x[0] -= x[i]*a[i];//all-pole accumulation
  }
  for (i=N;i>0;i--)
  {
    y += b[i]*x[i];//all-zero accumulation
    x[i] = x[i-1];//shift samples
  }
  y += b[0]*x[0];//calculate output value
  mono_write_16Bit((short)y);//write to output
}
```