# Imperial College London

## Real-Time Digital Signal Processing

## Laboratory 5

*Ahmad Moniri, CID: 00842685*

*Pranav Malhotra, CID: 00823617*

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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 offences policy

Signed: Ahmad Moniri, Pranav Malhotra

April 29, 2016

# Contents

# 1 Introduction

In laboratory 4, Finite Impulse Response (FIR) filters were introduced. Filter implementation was discussed and efficiency of different programming techniques was considered. In laboratory 5, Infinite Impulse Response (IIR) filters are introduced. IIR filters can be implemented in multiple ways; namely, the direct form II realisation and the transposed form realisation. Both these implementations are considered in this laboratory. The concept of dynamic memory allocation is also introduced; a programming technique that provides the programmer a greater control over the use of resources on the $TMS320C6713$ chip.

# 2 Infinite Impulse Response (IIR) Filters

An IIR filter, as the word infinite suggests, has an impulse response that does not reach zero in a finite duration of time. This has an important implication; the memory in the system is infinite. **This does not mean that an infinite amount of memory is required on the $TMS320C6713$ $DSK$ board. The infinite memory is implemented in the form of a feedback loop.** The difference equation of an IIR filter is listed in equation (1).

$$y(n) = \sum_{i=0}^{M} b_i x(n-i) - \sum_{i=1}^{N} a_i y(n-i) \tag{1}$$

Taking the z-transform of equation (1) will return the transfer function of the IIR filter. The transfer function is shown in equation (2).

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

**As opposed to FIR filters that do not provide the flexibility to place poles arbitrarily in the z-plane, both poles and zeros can be placed arbitrarily when IIR filters are designed.** The freedom to place poles at an arbitrary location presents both pros and cons. Desired frequency responses can be attained with significantly less coefficients than if poles were constrained to being placed at the origin of the z-plane. However, IIR filters will be unstable if poles are placed outside the unit circle. Even if the poles are deliberately placed within the unit circle, instability may occur if the number system used to implement the filtering process does not possess enough precision to accurately represent all filter coefficients, samples and outputs; the position of the poles could drift out of the unit circle.

## 2.1 Bilinear Transform

As opposed to FIR filters, IIR filters can be implemented digitally or with actual electrical components; electrical components such as capacitors and inductors used to build analogue filters have infinite impulse response and thus cannot be used to design FIR filters. Analogue filter design is an extremely developed field and multiple filters have been designed to balance trade offs and optimise certain parameters. For example, the elliptical filter has an equirriple in the pass band while the butterworth filter has a maximally flat magnitude response. For this reason, digital IIR filters are based off classical analogue filters.

Classical analogue filters can be described mathematically in the s-plane. To find the frequency response of the analogue filter, the substitution $s = e^{j\omega}$ is used. A transformation is required to map the filter from the s-plane to the z-plane; alternatively, the mapping can be viewed as converting a continuous time analogue filter to a discrete time digital filter. Once the filter has a representation in the z-plane, the inverse z-transform can be used to evaluate the difference equation for the filter. The difference equation will provide the time-domain representation of the digital filter. Finally, the output can be computed by evaluating the difference equation in the form of multiply accumulate operations.

There are certain conditions on the function that maps the filter from the s-plane to the z-plane. Firstly, the mapping has to preserve the stability of the filter. This means that all points in the left-half of the s-plane have to be mapped to points inside the unit circle in the z-plane.

The transform is expected to be non-linear. Consider the frequency response of an analogue filter. It is evaluate from $-\infty < \omega < \infty$. However, digital filters have frequency responses that are periodic. The frequency

response is only evaluated in the fundamental frequency range, $-\pi < \omega < \pi$. To preserve the integrity of the filter, the mapping from the s-plane to the z-plane has to map the entire continuous range of frequencies to the fundamental frequency range of the digital filter. It is clear that the mapping will be non-linear.

**The bilinear transform stated in equation (3) shows the mapping that will be used to convert the analogue filter into a digital filter.**

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{3}$$

To understand the key characteristics of the bilinear transform, s and z are first defined in equations (4) and (5) respectively.

$$s = \sigma + j\Omega \tag{4}$$

$$z = re^{j\omega} \tag{5}$$

Substituting, equation (5) into equation (3) yields

$$\begin{aligned} s &= \frac{2}{T}\frac{re^{j\omega}-1}{re^{j\omega}+1} \\ &= \frac{2}{T}\Big(\frac{r^2-1}{1+r^2+2rcos\omega} + \frac{2rsin\omega}{1+r^2+2rcos\omega}\Big) \end{aligned} \tag{6}$$

Finally, comparing equation (6) to the form presented in equation (4)

$$\sigma = \frac{2}{T}\frac{r^2-1}{1+r^2+2rcos\omega} \tag{7}$$

$$\Omega = \frac{2}{T}\frac{2rsin\omega}{1+r^2+2rcos\omega} \tag{8}$$

**From equation (7), if $r < 1$, then $\sigma < 0$ and if $r > 1$, then $\sigma > 0$. Clearly the left-half of the s-plane maps to inside the unit circle in the z-plane and the right-half of the s-plane maps to outside the unit circle in the z-plane; the bilinear transforms satisfies the criteria that the mapping from the s-plane to the z-plane should preserve stability of the filter**.

**To study the non-linearity of the mapping, the frequency response is considered.** The frequency response is evaluated along the $j\omega$ axis in the s-plane and along the unit circle in the z-plane. Consequently, $\sigma = 0$ and $r = 1$.

$$\begin{aligned} \Omega &= \frac{2}{T}\frac{sin\omega}{1+cos\omega} \\ &= \frac{2}{T}tan\frac{\omega}{2} \\ \omega &= 2tan^{-1}\frac{\Omega T}{2} \end{aligned} \tag{9}$$

**Equation (9) provides the relationship between the frequency variables in the two domains. The initial conjecture that the mapping will be non-linear is verified. The entire frequency range in the s-plane is mapped to the fundamental frequency interval in the z-plane. The frequency is said to be warped**.

This has a very important implication. The shape of the frequency response in the analogue domain and in the digital domain will be the same however the axis will be scaled. T**he non-linear mapping means that the corner frequencies may shift. This problem can be circumvented by a process referred to be as pre-warping. This will allow the corner frequency to be placed at a specific frequency in the digital domain.** For example, if we wish to implement a single-pole filter digitally with a corner frequency of $\omega = \omega_c$, the analogue filter should be designed with a corner frequency at $\Omega = \frac{2}{T}tan(\frac{\omega_c T}{2})$. Once the bilinear transformation is applied, the analogue corner frequency will map exactly to the desired digital corner frequency. In similar fashion, multiple pole/zero systems can be designed in the analogue domain to compensate for the warping effect of the bilinear transformation.

# 3 Single Pole Filter

## 3.1 Design of Digital Filter

In laboratory 5, the analogue filter presented in figure 1 has to be implemented digitally.

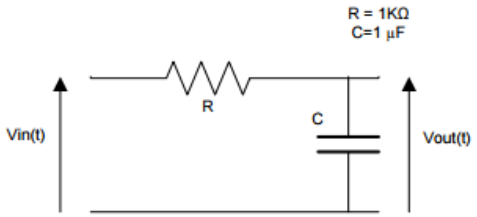


Figure 1: Single pole RC filter

The first step to converting the analogue filter to a digital one is to compute the transfer function of the analogue filter. This is presented in equation (10).

$$\begin{aligned} V_{out}(s) &= \frac{1/sC}{R + 1/sC} V_{in}(s) \\ &= \frac{1}{1 + sRC} V_{in}(s) \\ H(s) &= \frac{V_{out}(s)}{V_{in}(s)} = \frac{1}{1 + sRC} \end{aligned} \tag{10}$$

It is clear that the filter is a low-pass filter. At low frequencies, $H(s) \approx 1$ while at high frequencies, $H(s) \approx 0$.

The next step is to apply the bilinear transformation, presented in equation (3) to the transfer function presented in equation (10). This yields the transfer function of the digital filter.

$$\begin{aligned} H(s) &= \frac{1}{1 + (\frac{2}{T}\frac{z-1}{z+1})RC} \\ &= \frac{zT + T}{T(z+1) + 2(z-1)RC} \\ &= \frac{zT + T}{z(T + 2RC) + (T - 2RC)} \\ &= \frac{T + Tz^{-1}}{(T + 2RC) + (T - 2RC)z^{-1}} \qquad (11) \\ &= \frac{\frac{T}{T+2RC} + \frac{T}{T+2RC}z^{-1}}{1 + \frac{T-2RC}{T+2RC}z^{-1}} \qquad (12) \end{aligned}$$

Equation (12) shows the general form of a single pole analogue filter that has been mapped to the z-plane using the bilinear transformation. In laboratory 5, the values of R and C have been defined to be $1k\Omega$ and $1\mu F$ respectively; the sampling frequency is set to $8kHz$. These are substituted into equation (12) and the final transfer function is presented in equation (13).

$$H(z) = \frac{\frac{1}{17} + \frac{1}{17}z^{-1}}{1 - \frac{15}{17}z^{-1}} \tag{13}$$

Finally, taking the inverse z-transform the difference equation presented in equation (14) is obtained.

$$y(n) = \frac{1}{17}x(n) + \frac{1}{17}x(n-1) + \frac{15}{17}y(n-1) \tag{14}$$

**As discussed in section 2.1, the mapping from the analogue domain to the digital domain is non-linear. The effects of frequency warping were discussed and it was mentioned that the bilinear**

**transformation will shift corner frequencies. The shifting of the corner frequencies can be compensated for when designing the analogue filter through a process referred to as pre-warping. This was however not considered when mapping the single-pole filter in figure 1 to the digital domain since the ratio of the corner frequency to the sampling frequency is small enough that the assumption $tan(\Omega T/2) \approx \Omega T/2$ is valid.**

To check the validity of the assumption, the analogue corner frequency is first calculated in equation (15).

$$\begin{aligned} \Omega_c &= \frac{1}{RC} = 1000\ rad \\ F_c &= 159.15Hz \end{aligned} \tag{15}$$

Next, the digital corner frequency is computed using equation (9).

$$\begin{aligned} \omega_c &= \frac{2}{T} tan^{-1}(\frac{\Omega_c T}{2}) = 998.70\ rad\ s^{-1} \\ f_c &= 158.94Hz \end{aligned} \tag{16}$$

**Equation (16) shows that the assumption is valid and pre-warping does not have be applied in this case.** Figure 2 shows the ideal frequency response of the RC filter once the bilinear transform has been applied and the analogue filter has been mapped to a digital filter.

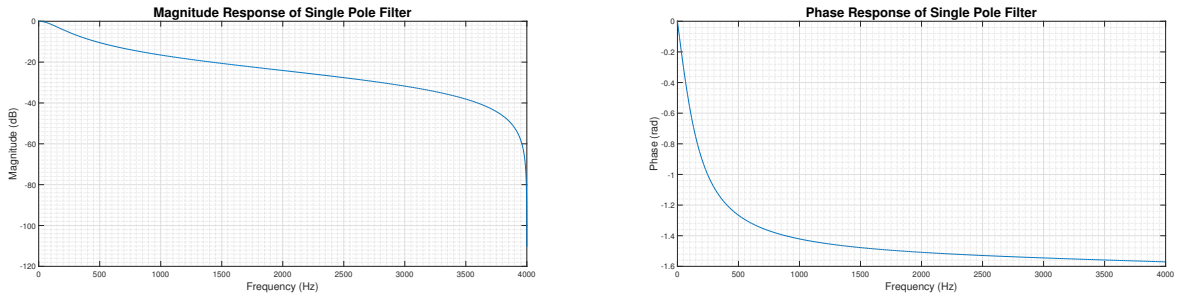


Figure 2: Ideal frequency response of the designed filter

**Figure 3 shows that the corner frequency of the digital filter is indeed at $158Hz$.**

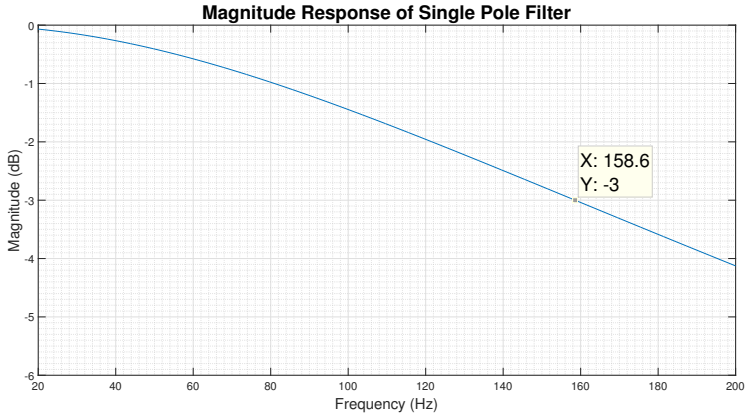


Figure 3: Ideal frequency response of the designed filter

## 3.2 Ideal corner frequency of digital filter

In section 3.1, the bilinear transform was utilised to convert a single pole analogue filter into a digital filter. The filter was presented in equation (14) in the form of a difference equation. To implement the digital filter on the $TMS320C6713\ DSK$ board, the code presented in listing 1 was used.

```
// defining filter coefficients and
// initialising variables to hold inputs and outputs
// as global variables
const double a = 0.8823529411764706;
const double b = 0.05882352941176471;
double x[2] = {0};
double y = 0;

void ISR_filter(void){
    // reading of new sample
    x[0] = mono_read_16Bit();

    // output of filter is calculated using RC_filter function
    y = RC_filter();

    // calculated output written to audio codec
    mono_write_16Bit((short)y);
}

double RC_filter(void){
    // output is computed in accordance with filter's difference equation
    y = b*(x[0] + x[1]) + a*y;

    // position of buffer is updated
    x[1] = x[0];

    return(output);
}
```

Listing 1: C code to implement single pole digital filter

The code required to implement the single pole filter is extremely simple. The order of the filter is 1 thus only 3 elements need to be stored. The array of doubles, x, needed to store previous inputs, is defined as a global variable of size 2; both elements in the array are initialised to 0. Observing the form of the equation (14), it is clear that the variable y, which is used to store the current output can be reused when calculating the next output.

As in the laboratory 4, the filter coefficients are stored as double-precision floating point numbers. As discussed in section 2, the digital representation of numbers may result in poles drifting out of the unit circle. This should be carefully considered as implementing the filter designed in section 3.1 requires representing the filter coefficients, calculated to be fractions, as double-precision floating point numbers. **In this case however, the error incurred does not result in instability as the pole is located a significant distance away from the unit circle[1].**

Each time a new sample is read, the samples within the buffer are rearranged such that the index of the array represents the relative delay between the current sample and the stored sample. In laboratory 4, this structure was referred to as a simple non circular buffer**. The overhead for rearranging elements was significant in the previous laboratory session because the number of filter coefficients was in excess of 200. In this case however, a simple buffer does not require much computation complexity to be maintained.**

The output of the filtering process is performed with just 1 line of code. This is again possible because the number of filter coefficients is extremely small.

## 3.3 Verification of Time Constant

An important design parameters in the design of analogue filter is the time constant. The time constant for the analogue filter presented in figure 1 is calculated in equation (17).

$$\tau_a = RC = 1.000ms \tag{17}$$

Notice that the time constant is the inverse of the corner frequency, scaled by $2\pi$, which was calculated in equation (15). The frequency warping introduced due to the non-linear nature of the bilinear transform will

[1] Higher order filters are more suspectile to rounding errors.

slightly change the time constant of the digital filter. Equation (18) shows the time constant of the digital filter. It is calculated using the corner frequency stated in equation (16).

$$\tau_d = \frac{1}{2\pi(158.94)} = 1.001ms \tag{18}$$

**To verify that the time constant for the digital filter matches the theoretical value of** $1.001ms$**, a square wave is used as an input to the filter.** The external connections of the $AIC23$ audio codec, presented in figure 4, show the presence of a high-pass filter at the input.

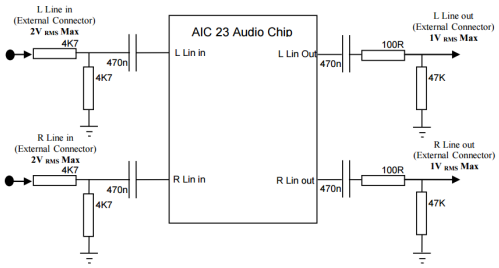


Figure 4: AIC23 audio chip external connections[1]

The high-pass filter has the following transfer function, where $R_1 = 100\Omega$, $R_2 = 47k\Omega$ and $C = 470nF$:

$$H(s) = \frac{V_o(s)}{V_i(s)} = \frac{R_2Cs}{1 + (R_1 + R_2)Cs} \tag{19}$$

The corner frequency of the filter is:

$$f_p = \frac{1}{2\pi(R_1 + R_2)C} = \frac{1}{2\pi(100 + 47 \cdot 10^3)(470 \cdot 10^{-9})} = 7.1895Hz \tag{20}$$

**To avoid distortion from the high-pass filters on the** $AIC23$ **audio codec, a square wave of frequency** $150Hz$ **is used as an input.** The oscilloscope traces obtained are presented in figure 5.

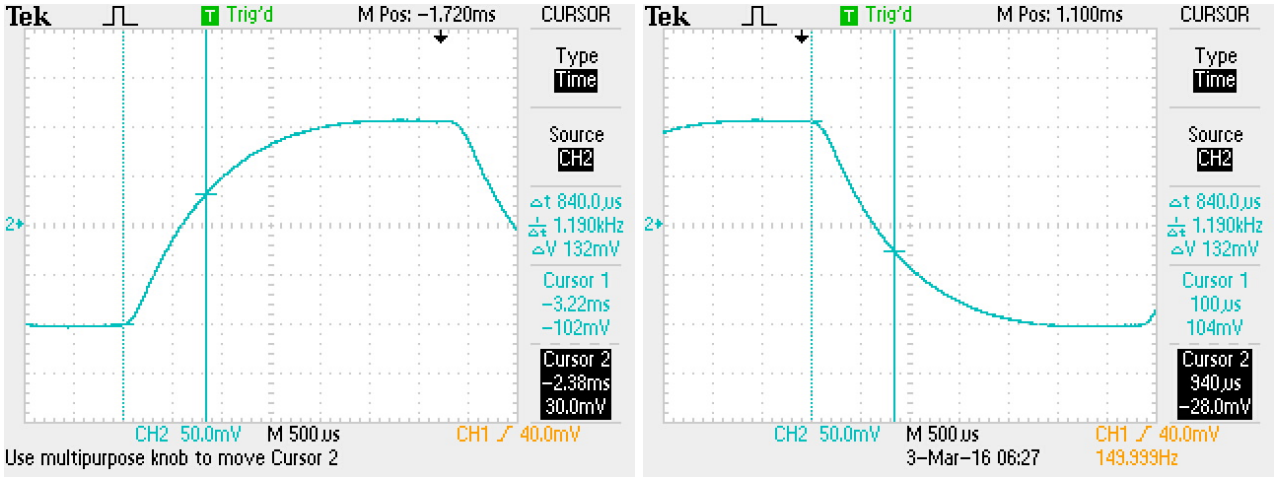


Figure 5: Oscilloscpoe traces used for calculation of time constant of digital filter

The time constant of the circuit physically represents the time taken for the system's step response to reach 63.2%. The peak-to-peak voltage of the output wave is $220mV$ and thus a voltage change of $132mV$ is equivalent to a 63.2% increase/decrease.

**From figure 5, it is clear that the time constant of the digital filter is** $0.840ms$**. This is slightly off from theoretical value of the digital filter of** $1.001ms$**.**

**Note that the frequency of** $150Hz$ **is perfect to measure the time constant of the digital filter. A square wave of a lower frequency would be distorted due to the high-pass nature of the filters**

**connected to the $AIC23$ audio codec. A square wave of a higher frequency would not allow the output of the digital filter to settle close to its asymptotic value. In the analogue domain, this is equivalent to the capacitor not charging or discharging fully. Using an input square wave of $150Hz$ provides more than enough time[2] for the output to settle to very near its asymptotic value.** This is shown in figure 6.

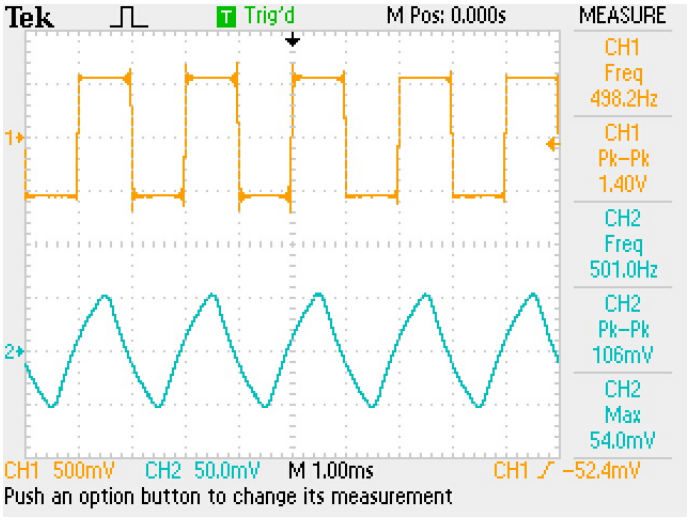


Figure 6: Oscilloscope trace when a high frequency square wave is used as an input

## 3.4 Frequency Response

Before the frequency response of the designed single pole filter can be studied, it is important to realise that the $TMS320C6713$ $DSK$ will have a non-flat frequency response even if no filtering is applied. This is due to the 2 analogue filter, 2 digital filters and multiple network of resistors that are present on the board. For accurate analysis, the frequency response of the $DSK$ board has to be taken into account. Figure 7 shows the response of the $DSK$ board.

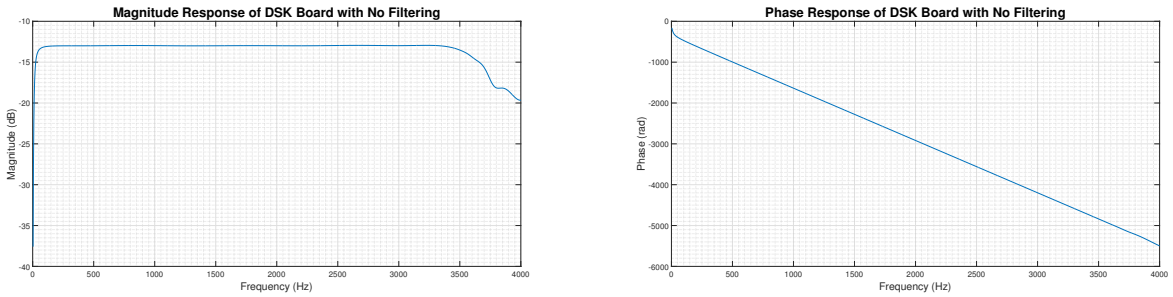


Figure 7: Frequency response of $TMS320C6713$ $DSK$ board without any filter

Pass band gain of $-12dB$ corresponds to a gain of approximately $1/4$. This is exactly as expected.
Next, figure 8 shows the frequency response of the filter designed in section 3.1. Note that the magnitude response in the pass band is centred about $-12dB$. This is discrepancy comes about because of the inherent frequency response of the $DSK$ board as discussed above.

[2] A frequency of $150Hz$ corresponds to a period of $6.66ms$. This is greater than 5 time constants than thus the output should have settled to within 99% of its asymptotic value.

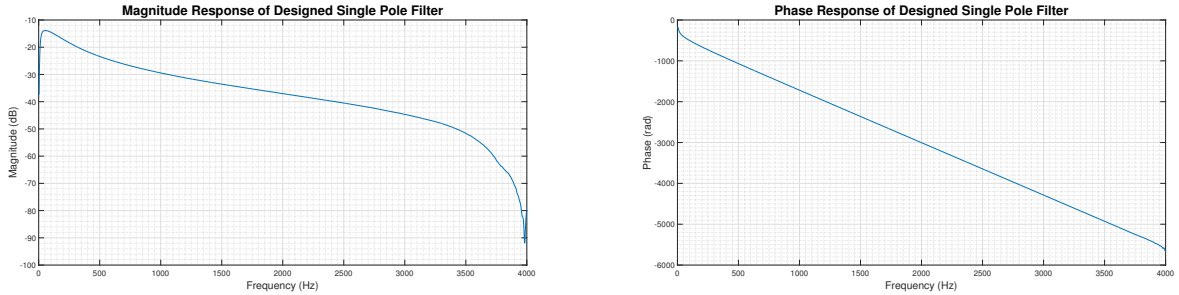


Figure 8: Frequency response of the filter designed in section 3.1

Finally, figure 9 shows the frequency response once the offset introduced by the $DSK$ board is taken into account.

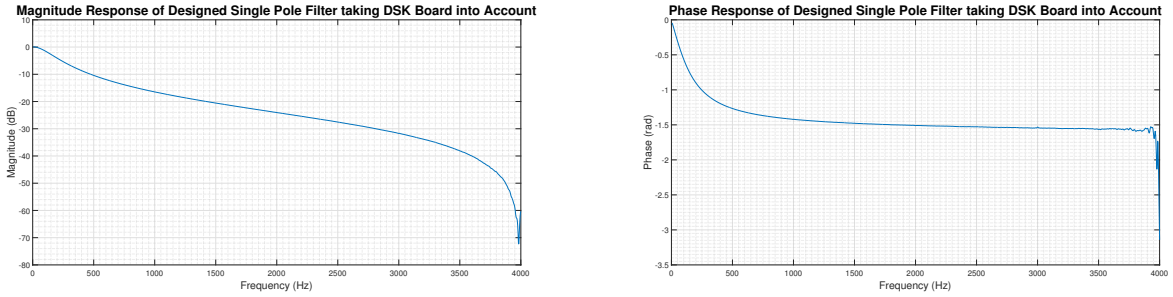


Figure 9: Frequency response of filter taking $TMS320C6713$ $DSK$ board's inherent response into account

Figure 10 shows the actual corner frequency of the filter offset introduced by the $DSK$ board is taken into account.

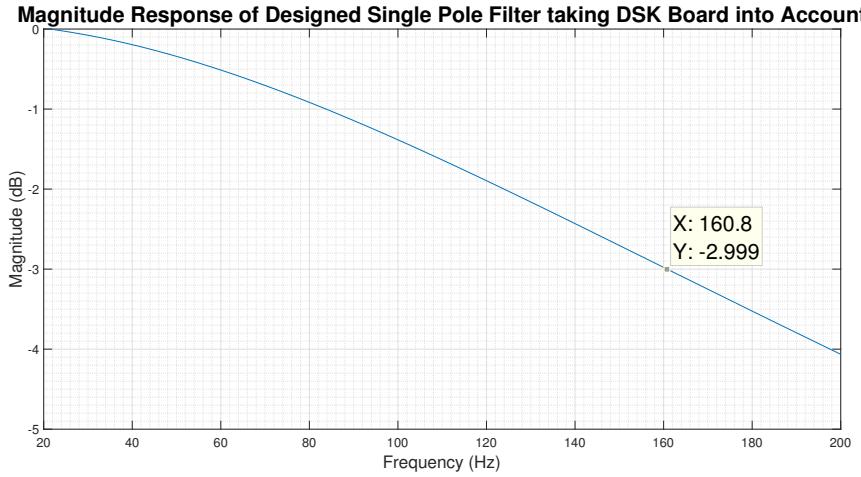


Figure 10: Frequency response of filter taking $TMS320C6713$ $DSK$ board's inherent response into account

**In section 3.3, the time constant of the digital filter was calculated to be $0.840ms$. This is slightly off from the theoretical value for the digital filter which is $1.001ms$. The difference can be attributed to the inherent frequency response of the DSK board. Using $f_c = 160.8Hz$, the time constant of the filter comes to $0.990ms$. This is significantly closer to the theoretical value of $1.00ms$ and confirms that the digital filter designed in section 3.1 has been implemented correctly on the $DSK$ board. Table 1 summarises the data obtained.**

| | Analogue Filter | Ideal Digital Filter | Digital Filter without taking DSK offset into Account | Digital Filter taking DSK offset into account |
|---|---|---|---|---|
| Corner Frequency / Hz | 159.15 | 158.94 | 189.47 | 160.80 |
| Time Constant / ms | 1.000 | 1.001 | 0.840 | 0.990 |

Table 1: Corner frequencies and time constants overview

Lastly, figure 11 shows how the frequency response of the digital filter compares to that of the analogue RC filter. The graph shows that the bilinear transfomation has been very effectively in mapping the analogue filter to the digital domain. **Near $4kHz$, both magnitude and phase response diverge slightly from the ideal response due to the anti-aliasing filter.**

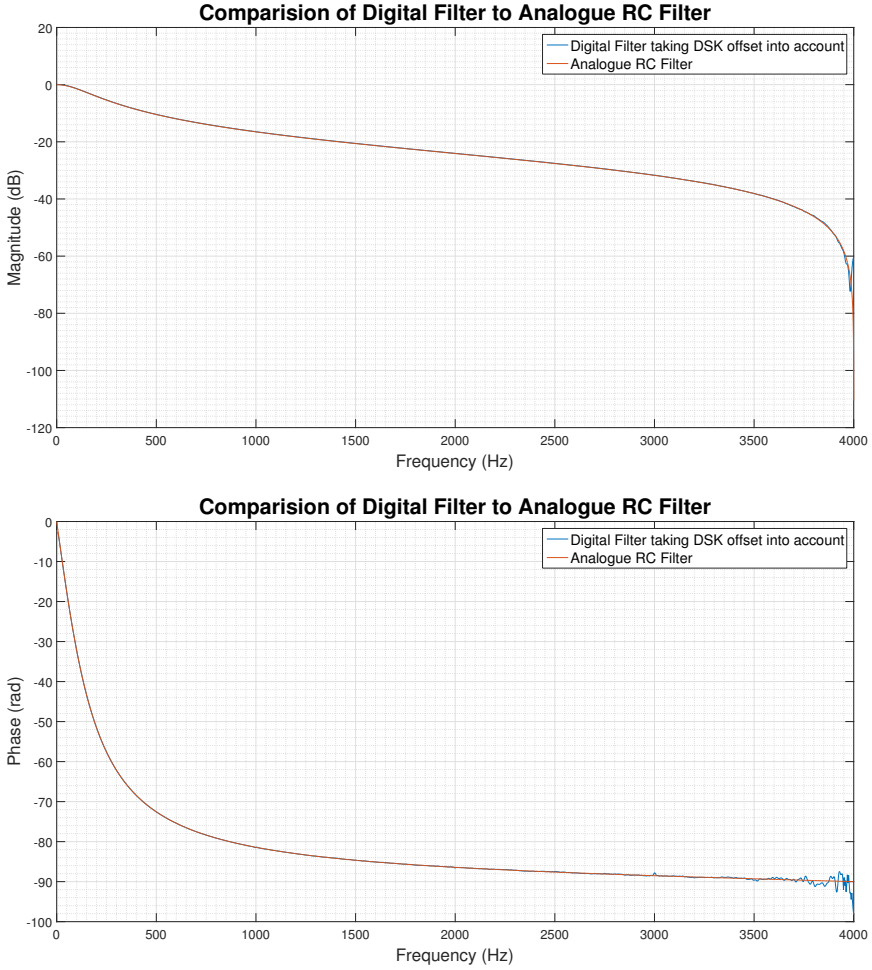


Figure 11: Comparision of frequency response of the digital filter and the analogue filter

# 4 Bandpass Filter

In laboratory 5, an elliptical filter has to be designed. The specifications for the filter are:

- Order: 4th
- Passband: $180Hz - 450Hz$
- Passband ripple: $0.4dB$
- Stopband attenuation: $23dB$

The filter is designed using the MATLAB code presented in lisiting 6.

```
fsamp = 8000;                   % defines the sampling frequency
order = 4;                      % defines the order of the digital filter
passband_ripple = 0.4;          % defines the passband ripple
stopband_attenuation = 23;      % defines the stopband attenuation of the filter
Wp = [180 450]/(0.5*fsamp);     % normalising edge frequencies of passband


% ellip function designs filter of order 2n
[b, a] = ellip(order/2, passband_ripple, stopband_ripple, Wp, "bandpass");
```

Listing 2: MATLAB code to generate coefficients for an elliptical filter

The generated filter's response is shown in figure 12.

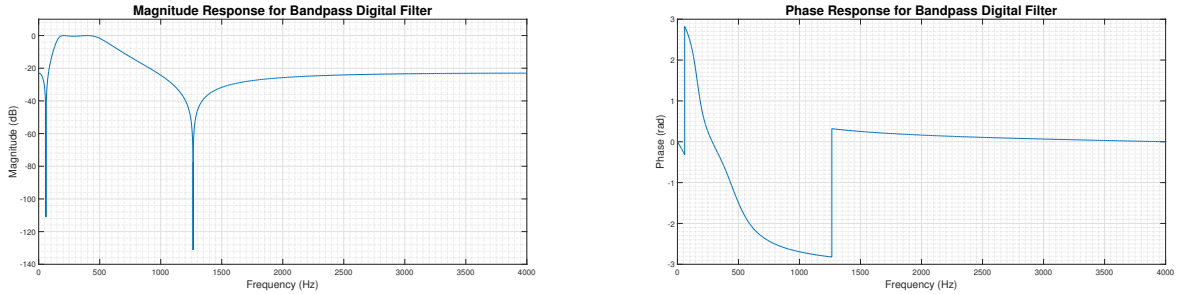


Figure 12: Theoretical frequency response of digital filter

The first thing to notice is that the filter does not have a linear phase in the pass band. **A linear phase response is characteristic of FIR filters and a non-linear phase response is expected.** Secondly, the filter's magnitude response meets all the specifications set out in the laboratory script. Figure 13 shows the magnitude response at specific frequencies.

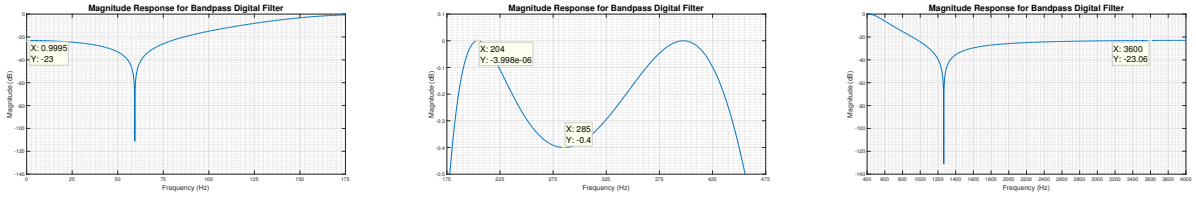


Figure 13: Theoretical pass band and stop band frequency response of filter designed on MATLAB

# 5 Implementation of Bandpass Filter: Direct Form II Realisation

An IIR filter can be implemented in multiple ways. The filter designed in section 4 will be implemented using the direct form II realisation. Before the filter can be implemented in C, it is important to understand how the output is calculated.

The signal flow graph for the direct form II realisation of an IIR is shown in figure 14. For simplicity, a $3^{rd}$ order system is considered.

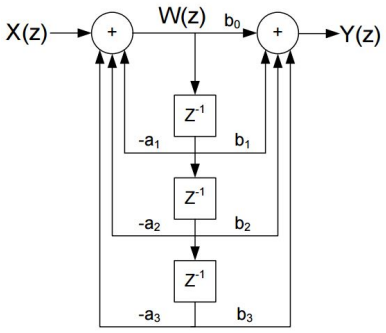


Figure 14: Direct form II realisation of an IIR filter

The direct form II realisation of a $3^{rd}$ order IIR filter can be mathematically expressed in equations (21) and (22).

$$w(n) = x(n) - a_1x(n-1) - a_2x(n-2) - a_3x(n-3) \tag{21}$$

$$y(n) = b_0w(n) + b_1w(n-1) + b_2w(n-2) + b_3x(n-3) \tag{22}$$

**The equations above show that the direct form II structure requires computing two variables, $w$ and $y$ through multiply accumulate operations. These are however largely independent and thus parallelism can be employed.**

The C code to implement the direct form II realisation of the IIR filter is presented in listing 3.

```
1  void ISR_filter(void){
2      // read sample from the line input
3      short sample = mono_read_16Bit();
4
5      // compute the next output sample using the designed filter
6      double output = direct_form_2(sample);
7
8      // write output to the line output
9      mono_write_16Bit((short)output);
10 }
11
12 // Function to implement an IIR filter using the Direct Form II realisation
13 // Function uses a simple buffer
14 double direct_form_2(short sample){
15
16     // declare variable for loops
17     int i;
18
19     // declare variable to store output
20     double output = 0;
21
22     // x[0] will contain (after for loop): sample minus all 'a' coefficient terms
23     x[0] = sample;
24
25     for (i = N; i > 0; i--){
26         x[0] -= a[i]*x[i];      // accumulate 'a' coefficient terms
27         output += b[i]*x[i];    // accumulate 'b' coefficient terms
28         x[i] = x[i-1];          // shift the buffer along
29     }
30
31     // compute output by summing the accumulation of 'a' coefficients (stored in x[0])
32     // and 'b' coefficients (already stored in output)
33     output += b[0]*x[0];
34
35     return(output);
36 }
```

Listing 3: `direct_form_2`

The function `direct_form_2` utilises a simple buffer. The memory required for the buffer is dynamically allocated using the `calloc` function. The `calloc` function initialises the buffer elements to 0, and thus zeroing is not required. **The simple buffer requires a great deal of overhead to maintain because complete reorganisation of the buffer, each time a new sample is read, is required. The simple buffer however can be implemented very easily in C.**

**As discussed in the previous report, the use of a circular buffer can significantly improve performance.** Complete reorganisation of the buffer is no longer required. A pointer is used to signify at which point in the array the newest sample is currently stored. The pointer is wrapped around when the end of the array is reached. Moreover, by doubling the size of the buffer, the use of `if` statements, likely to cause pipeline stalls, can be eliminated. With a double sized buffer, there is no danger of overflow in the `for` loop.

**As such, the code presented in listing 4 utilises a double sized circular buffer. The function `direct_form_2_circ` also utilises pointers to improve performance.** In the previous report, it was established that the use of pointers can be used to improve performance. In assembly, pointers can be referenced and incremented/decremented in one instruction. Optimisation level 2 seeks to optimise code to use pointer automatically however often the compiler is not able to identify areas where such optimisation can be utilised. As such, pointers are used explicitly in the function `direct_form_2_circ`.

```
1  void ISR_filter(void){
2      // read sample from the line input
3      short sample = mono_read_16Bit();
4
5      // compute the next output sample using the designed filter
6      double output = direct_form_2_circ(sample);
7
8      // write output to the line output
9      mono_write_16Bit((short)output);
10 }
11
12 // Function to implement an IIR filter using Direct Form II
13 // Function uses a double-sized circular buffer
14 // Pointers used for optimisation and code portability
15 // Requires:
16 //      GLOBAL: double *index;
17 //      MAIN:   x = (double *)calloc(2*(N+1), sizeof(double));
18 //              index = x;
19
20 double direct_form_2_circ(short sample){
21
22     // declare variable for for-loop
23     int i;
24
25     // declare variable to store output
26     double output = 0;
27
28     // declare pointers for filter coefficients
29     const double *ptr_a = a;
30     const double *ptr_b = b;
31
32     // *index should contain (after for-loop): sample minus all 'a' coefficient terms
33     // update *(index+N+1) to prevent the need for wrap-around in the loop
34     *index = *(index+N+1) = sample;
35
36     for (i = 1; i <= N; i++){
37         *index -= (*(ptr_a+N+1-i) * *(index+i));    // accumulate 'a' coefficient terms
38         output += *(index+i) * *(ptr_b+N+1-i);      // accumulate 'b' coefficient terms
39     }
40
41     // update *(index+N+1)
42     *(index+N+1) = *index;
43
44     // compute output by summing the accumulation of 'a' coefficients (stored in *index)
45     // and 'b' coefficients (already stored in output)
46     output += *index * *ptr_b;
47
48     // increment buffer pointer
49     index++;
50
51     // wrap the pointer around if it gets to half the size of the double-buffer
52     if(index == &x[N+1])
53         index = x;
54
55     return(output);
56 }
```

Listing 4: `direct_form_2_circ`

## 5.1 Direct Form II Realisation: Analysis of Performance

In order to profile the performance of the direct form II realisation of the IIR filter, the clock cycles required to calculate the output were measured. Breakpoints were inserted after reading the newest sample but before writing the output to the $AIC23$ audio codec. Performance of the algorithm is measured for various orders of filters[3]. The results obtained are presented in table 2.

| | `direct_form_2` | | `direct_form_2_circular` | |
|---|---|---|---|---|
| Order | No Optimisation | Level 2 | No Optimisation | Level 2 |
| 2 | 135 | 112 | 170 | 129 |
| 4 | 245 | 177 | 250 | 181 |
| 6 | 356 | 241 | 330 | 232 |
| 8 | 466 | 306 | 411 | 284 |
| 10 | 576 | 370 | 491 | 335 |
| 12 | 686 | 435 | 571 | 387 |
| 14 | 796 | 500 | 651 | 439 |
| 16 | 907 | 564 | 731 | 490 |
| 18 | 1017 | 629 | 812 | 542 |
| 20 | 1127 | 693 | 892 | 593 |

Table 2: Direct form II performance

Figure 15 maps the performance of the two functions `direct_form_2` and `direct_form_2_circ`.

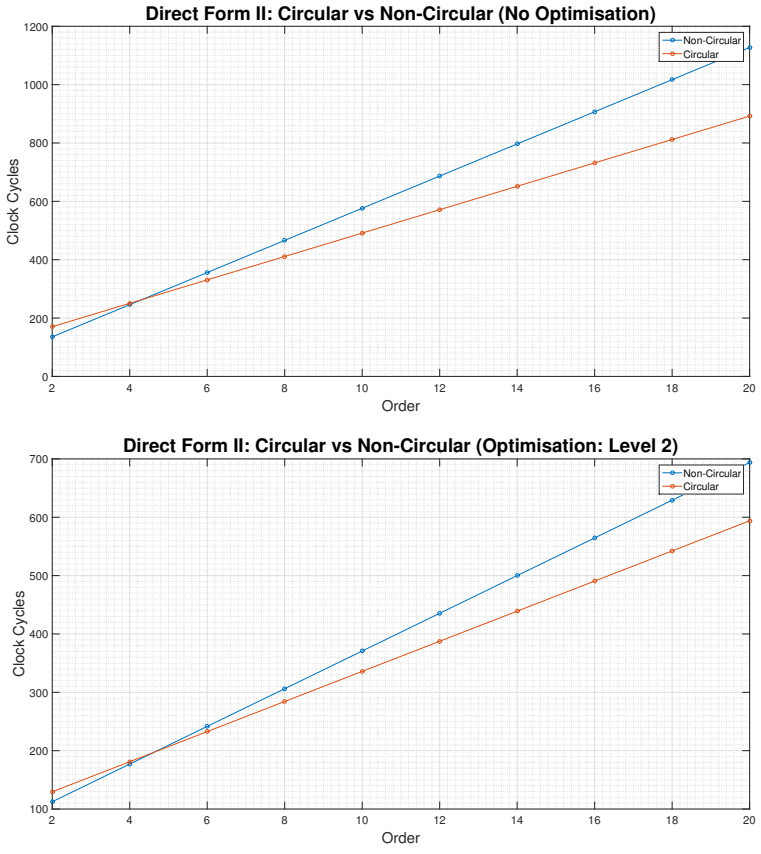


Figure 15: Performance of direct form II realisation of IIR filter

**From the data obtained, it is evident that the implementation of the circular buffer significantly improves the performance of the filtering process. Using a circular buffer requires half the number of memory access operations than if a simple buffer was used. As such, the circular buffer implementation is expected to be significantly faster.**

[3]The filter became unstable after order $N = 14$. This will be discussed later. Nonetheless, the number of clock cycles was measured.

**This is however not true when small filter orders are utilised. When a $2^{nd}$ order filter is used, both functions require similar number of memory access operations however the circular buffer implementation requires more calculations outside the `for` loop. This includes the `if` statement required to implement the wrapping around and other calculations required to determine the buffer location that the newest sample has to be written to. When a $4^{th}$ order filter is implemented, even though the circular buffer implementation requires less memory access operations than the simple buffer implementation, the overhead required to maintain the circular buffer still results in the simple buffer implementation being slightly faster.**

**The clear benefits of utilising a circular buffer become obvious when filters of order greater than 4 are implemented.** This can be understood mathematically from equations (25) and (26). The circular buffer implementation has a smaller gradient than the non circular buffer and thus scales better with filter order. However its y-intercept is significantly larger. The large y-intercept accounts for the slow performance at small filter orders.

$$T_{Non-Circular} = 55.1N + 25.4 \tag{23}$$

$$T_{Circular} = 40.1N + 90.2 \tag{24}$$

When compiler optimisation is utilised, the performance of both functions increases. This is exactly as expected. The increase in performance is more significant at higher orders. This is also exactly as expected. **The optimisation requires rearranging of the `for` loop and this has some overhead. However, once the rearrangement of the `for` loop has been completed, arbitrarily increasing the order of the filter does not present a significant increase in the number of clock cycles required.** Once again, this can be understood mathematically; the gradient of both the circular and the non circular implementation has decreased.

$$T_{Non-Circular,\ Optimisation} = 32.3N + 47.8 \tag{25}$$

$$T_{Circular,\ Optimisation} = 25.8N + 77.9 \tag{26}$$

# 6 Implementation of Bandpass Filter: Direct Form II Transposed

An discussed in section 5, an IIR filter can be implemented in multiple ways. The filter designed in section 4 will now be implemented using the transposed form realisation. Before the filter can be implemented in C, it is important to understand how the output is calculated.

The signal flow graph for the transposed form realisation of an IIR is shown in figure 16.

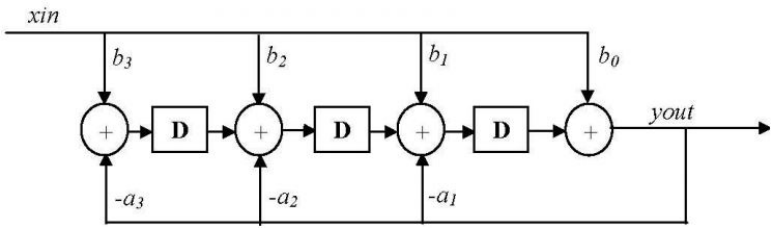


Figure 16: Direct form II transposed realisation of an IIR filter

**The first thing to notice is that the transpose form explicitly reveals the recursive nature of IIR filters.** In section 2, it was highlighted that an IIR filter has infinite memory. This is implemented by feeding the output back into the system. In the direct form II realisation, this feedback was less explicit.

**Note that the transposed form realisation of an IIR filter is expected to perform faster than the direct form II realisation explored in section 5. Although the transposed form does not allow for parallelised computations like the direct form II, the transposed form algorithm does not require excessive shifting of samples within the buffer. The buffer is updated continuously with intermediate values that are calculated. Elements in the buffer are overwritten not shifted.** The code to implement the transposed form realisation of the IIR filter designed in section 3.1 is presented in listing 5.

```
1  void ISR_filter(void){
2      // read sample from the line input
3      short sample = mono_read_16Bit();
4
5      // compute the next output sample using the designed filter
6      double output = transposed_form(sample);
7
8      // write output to the line output
9      mono_write_16Bit((short)output);
10 }
11
12 // Function to implement an IIR filter using Transposed Form
13 // Buffer size is equivalent to order of filter
14 // 1 less element in the buffer as compared to direct form II implementation
15 // Pointers used for optimisation and code portability
16 // Requires:
17 //      GLOBAL: double *index;
18 //      MAIN:   x = (double *)calloc(N, sizeof(double));
19 //              index = x;
20
21 double transposed_form(const short sample){
22
23     // declare variable for for-loop
24     int i;
25
26     // declare pointers for filter coefficients
27     const double *ptr_a = &a[1];
28     const double *ptr_b = &b[1];
29
30     // declare variable to store output
31     const double output =  b[0]*sample + x[0];
32
33     // compute next buffer values for the next function call
34     for (i = 0; i < N-1; i++){
35         x[i] = x[i+1] + *(ptr_b++)*sample - *(ptr_a++)*output;
36     }
37
38     x[N-1] = *(ptr_b)*sample - *(ptr_a)*output;
39
40     return(output);
41 }
```

Listing 5: `transposed_form`

**After a new sample is read, it is stored in a temporary variable called `sample`. It is important to note that the buffer `x` in the transformed form realisation does not contain past input samples. The buffer holds intermediate values that will be used to calculate the output. The intermediate values that are to be held in the buffer are calculated within the `for` loop. Notice that the last element in the buffer is calculated outside the `for` loop because it's mathematical structure is different from the other elements.**

Note that the compiler deals very well with variables defined as `const`. They have been utilised to improve code performance

## 6.1 Transposed Form Realisation: Analysis of Performance

As in the case of the direct form II filter, the performance of the transposed form realisation will be analysed and a relationship will be obtained in the form $A + BN$.

Table 3 provides the data obtained when the transposed form filter was run on the $DSK$ board.

| | `transposed_form` | |
|---|---|---|
| Order | No Optimisation | Level 2 |
| 2 | 101 | 103 |
| 4 | 162 | 144 |
| 6 | 224 | 185 |
| 8 | 285 | 227 |
| 10 | 347 | 268 |
| 12 | 408 | 309 |
| 14 | 469 | 350 |
| 16 | 531 | 391 |
| 18 | 592 | 433 |
| 20 | 654 | 474 |

Table 3: Transposed form performance

From the data presented in table 3, equations (27) and (28) are obtained. It is evident that optimisation decreased the gradient of the slope and thus the algorithm scales better when the filter order is increased.

$$T_{Transposed,\ No\ Optimisation} = 30.7N + 40.0 \tag{27}$$

$$T_{Transposed,\ Optimisation} = 20.6N + 62.2 \tag{28}$$

Figure 17 presents the data obtained in table 3 graphically.

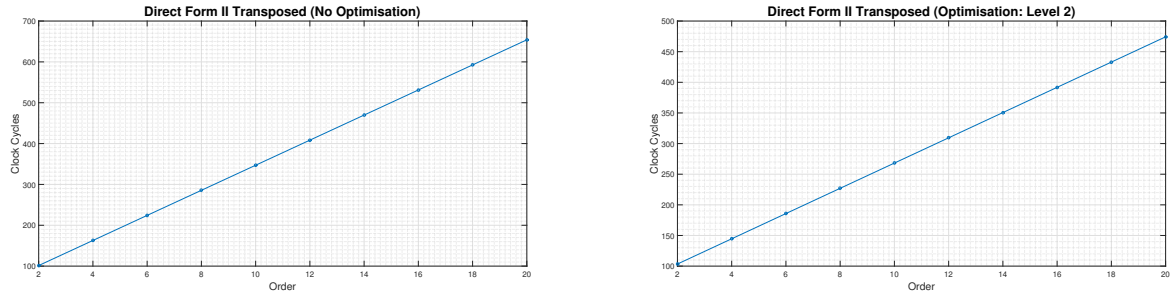


Figure 17: Performance of transposed form realisation of IIR filter

# 7 Comparison of Direct Form II and Transposed Form

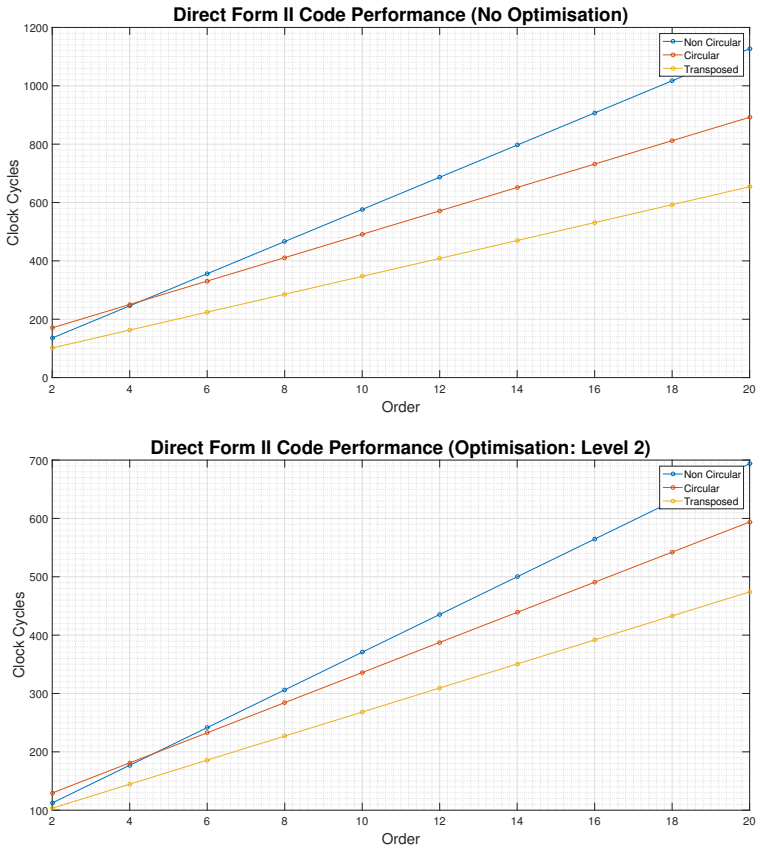


Figure 18: Comparison of direct form II and transposed form realisations of bandpass IIR filter

Finally, figure 18 shows how the direct form II and transposed form compare with one another. **It is clear that the transposed form realisation performs significantly better than both implementations of the direct form II realisation.** When optimisation is used, the gradients change slightly, however the general trend remains consistent.

The results obtained are exactly as expected. The transposed form is expected to perform better than the direct form II realisation because of the way the buffer is implemented. The direct form II uses a buffer that stores previous input samples. To keep track of the relative delay between the current sample and past samples requires overhead in terms of computational complexity. The simple buffer keeps track of the relative delay by complete reorganisation of the buffer each time a new sample is read. The circular buffer keeps track of relative delays by changing the position of the pointer such that it points to the newest sample in the buffer.

The transposed form removes this overhead. The buffer in the transform form implementation keeps track of intermediate values. The buffer is overwritten by new values rather than being shifted around. This is clearly simpler to implement as noted in listing 5; only 1 line of code is required within the `for` loop.

**Having a smart buffer, like the one used to implement the transposed form, is quintessential. Memory access operations are very expensive and smart buffers significantly reduce the number of memory accesses.**

**In addition, the code composer environment is able to significantly speed up code that make use of the `const` keyword. In the transposed form, the input sample and the calculated output, both required for each iteration of the `for` loop, are declared as constants; their value does not change during the function call.** This again helps to speed up the performance of the code. This is not possible with the direct form II implementation

# 8 Frequency Spectrum Analysis of the Bandpass Filter

As in section 3.4, for accurate analysis of the bandpass filter's frequency response, it is important to remove the offset introduced by the $DSK$ board. Figure 19 shows the frequency response of the bandpass filter. The frequency response of the $4^{th}$ order filter is graphed. **Note that the frequency response for the direct form and the transposed form is equivalent. This is as expected as the two implementations might have slightly different mathematical equations, but they implement the same filter and thus produce identical outputs.**

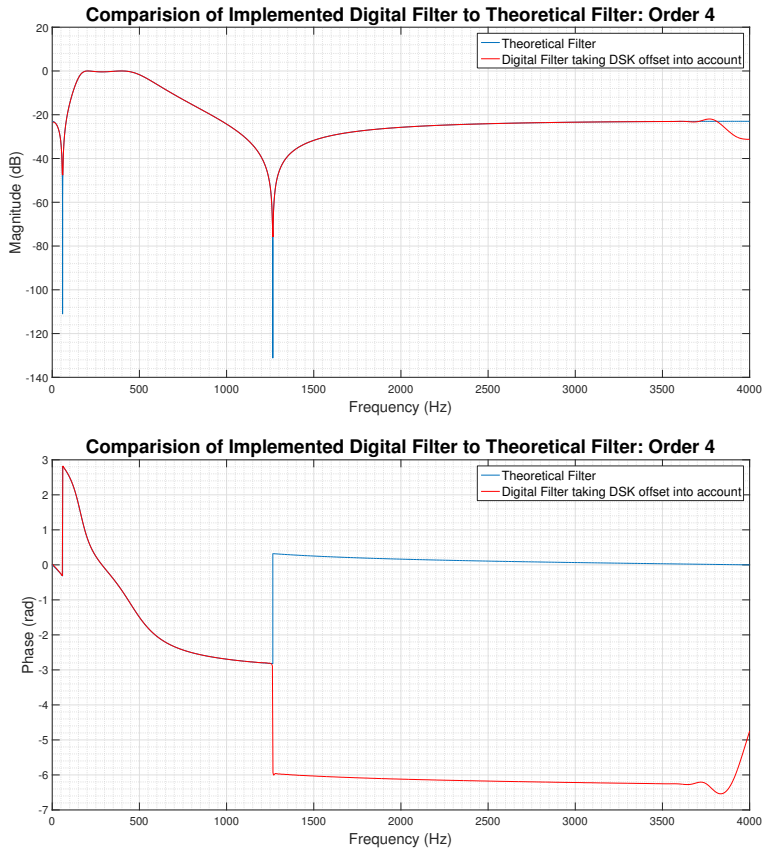


Figure 19: Comparison of theoretical response and response of digital filter on $DSK$ board

Superimposed on the graphs are the original filter response that was designed on MATLAB. Figure 19 confirms that the filter correctly implements the filtering process that it was designed for. It meets all the specifications that were set out in the laboratory 5 script. **Note the discrepancy in the phase comes from the way that the phase is unwrapped and does not mean that the filter is not performing as expected. Near $4kHz$, both magnitude and phase response diverge slightly from the ideal response due to the anti-aliasing filter.**

# 9 Stability of Elliptic Filter

**The method used to design the elliptic filter on MATLAB should theoretically produce stable filters. However due to finite memory constraints, the filter coefficients incur rounding errors. This causes poles and zeros to drift away from their actual position on the z-plane.**

In MATLAB there are two methods of using the `ellip` function to design a filter:

- Producing filter coefficients
- Producing poles and zeros

**Since the filtering process is going to be implemented in the time domain using multiply accumulate operations, filter coefficients are produced.** This method of filter design is referred to as the

Transfer Function (TF) design methodology. In contrast, the Zero-Pole-Gain (ZPK) design method[4] produces the position of poles and zeros on the complex plane. The ZPK method is less susceptible to the drifting of poles and zeros as it is less sensitive to finite precision effects. The ZPK design method uses a biquad implementation as opposed to the direct form I/II implementation of the filter. See Appendix A for the filter co-efficients that MATLAB uses for the TF and ZPK design (for N = 14).

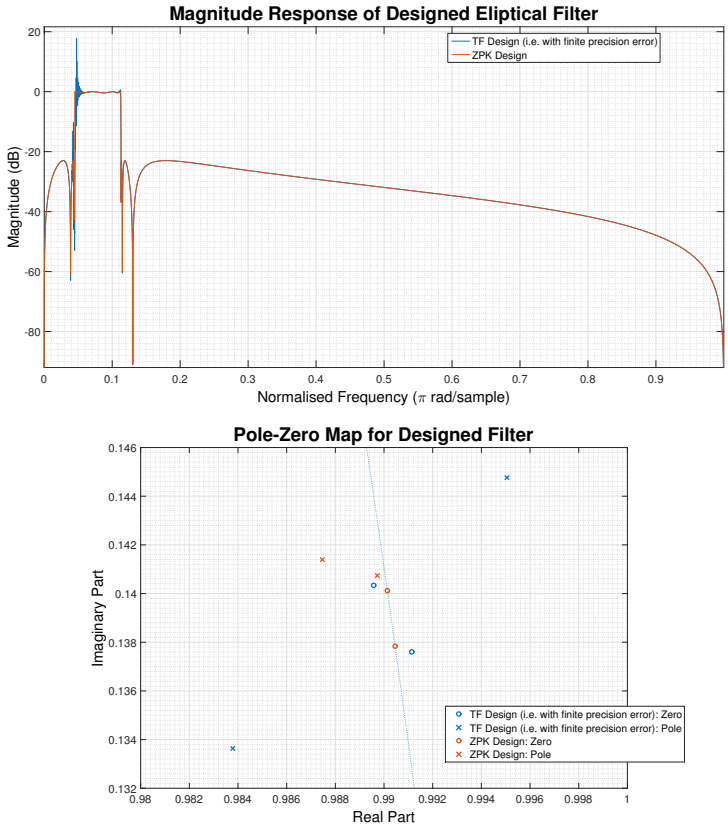


Figure 20:

**While increasing the order of the bandpass elliptical filter, it was observed that at N = 14, the filter became unstable.** Figure 20 shows the magnitude response of the TF design and the *ideal* ZPK design. The difference in frequency response is clear. **The pole-zero plot shows that a pole has drifted out of the unit circle when the TF methodology was used.**

In the design of IIR filters such as elliptic filters, poles placed near the edge of the unit circle, when high-order filters are designed, may drift out of the unit circle due to finite precision effects; the resulting filter will become unstable.

# 10 Conclusion

This report presented the theory behind IIR filters and the discussed generation of filters using the MATLAB functions. A detailed discussion about implementing IIR filters on the $TMS320C6713\ DSK$ board is presented. Compiler optimisations and different filter realisations were considered. Lastly the frequency spectrum of the designed filter was observed. The designed filter met all specifications listed in laboratory 5. It was noted that for filter of high orders, instability might occur. The reasons for instability were discussed and an alternative design methodology was explored.

# References

[1] Instruments, T. (2004). TLV320AIC23B, Stereo Audio CODEC, Data Manual. Retrieved February 04, 2016, from `http://www.ti.com/lit/ds/symlink/tlv320aic23b.pdf`

[4] This design methodology is referred to as the ZPK method, where K is used to signify gain.

# A Comparison of TF and ZPK Design Methodologies

```
fsamp = 8000;                   % defines the sampling frequency
order = 4;                      % defines the order of the digital filter
passband_ripple = 0.4;          % defines the passband ripple
stopband_attenuation = 23;      % defines the stopband attenuation of the filter
Wp = [180 450]/(0.5*fsamp);     % normalising edge frequencies of passband


% ellip function designs filter of order 2n
[b, a] = ellip(order/2, passband_ripple, stopband_ripple, Wp, 'bandpass');

% Zero-Pole-Gain design
[z,p,k] = ellip(order/2, passband_ripple, stopband_ripple, Wp, 'bandpass');
sos = zp2sos(z,p,k);
```

Listing 6: MATLAB code to compare TF and ZPK design methodologies

```
% TF Design (i.e. with finite precision error)
% --------------------------------

Numerator:
   0.022497433087628631
  -0.25933372947560179
   1.3578118108020327
  -4.226033152763061
   8.5301676784053004
 -11.199751059033469
   8.3225929854672227
   0.00000000000001374747433382150562
  -8.3225929854672369
  11.199751059033458
  -8.5301676784052791
   4.2260331527630433
  -1.3578118108020254
   0.25933372947560007
  -0.022497433087628457
Denominator:
     1
   -13.294797732225605
    82.492398684774983
  -316.62130686437945
   839.79221253282833
 -1628.2539304905699
  2379.8327340155715
 -2663.6585199936212
  2294.1995683454156
 -1513.1751220849169
   752.34382398208277
  -273.43561499303894
    68.673685494538049
   -10.668642850705483
     0.77351195490957214
```

```
% --------------------------------
% ZPK Design
% --------------------------------
--------------------------
Section #1
--------------------------
Numerator:
 0.022497433087628631
-0.000000000000000002497716820884913 3
-0.022497433087628624
Denominator:
 1
-1.8251410899353726
 0.87188613187592889
Gain:
1
--------------------------
Section #2
--------------------------
Numerator:
 1
-1.8349000478343571
 0.99999999999999845
Denominator:
 1
-1.8271480289125788
 0.93329715718977257
Gain:
1
--------------------------
Section #3
--------------------------
Numerator:
 1
-1.9852013132563306
 1.0000000000000002
Denominator:
 1
-1.9467131215714355
 0.96888932683670648
Gain:
1
--------------------------
Section #4
--------------------------
Numerator:
 1
-1.8709521194001697
 0.99999999999999944
Denominator:
 1
-1.8666888467656142
 0.98811071015479846
Gain:
1
--------------------------
Section #5
--------------------------
Numerator:
 1
-1.9809095990696028
 0.99999999999999822
Denominator:
 1
-1.9749354125434055
 0.99508487403232637
Gain:
1
--------------------------
Section #6
--------------------------
Numerator:
 1
-1.8750257859014998
 0.99999999999999911
Denominator:
 1
-1.8747176390255129
 0.99844165338195889
Gain:
1
--------------------------
Section #7
--------------------------
Numerator:
 1
-1.9802697540207115
 0.99999999999999711
Denominator:
 1
-1.9794535934716841
 0.99936635319119782
Gain:
1
--------------------------
Output Gain:
1
```

Figure 21: Comparison of TF and ZPK filter design methodologies