REAL-TIME DSP LABORATORY5:
Infinite Impulse Response (IIR) Filters on the C6713 DSK

Contents

1 Introduction

Infinite duration impulse response, or IIR, filters cannot be implemented in the same non-recursive ways as FIR filters. However, when an IIR filter has a rational transfer function, it can be implemented recursively, using difference equations, or their state-space equivalents. The implementation of difference equations in discrete-time systems, using coded algorithms, is entirely analogous to the implementation of differential equations in continuous-time systems using op-amps, capacitors, inductors, and resistors. In this lab, you will study

- IIR filter realizations,
- IIR filter design, and
- IIR filter implementation on the DSK, using C and assembly language programs.

2 Rational Transfer Functions

Consider an IIR filter with rational transfer function $H(z)$:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b[0] + b[1]z^{-1} + \ldots + b[M]z^{-M}}{1 + a[1]z^{-1} + \ldots + a[N]z^{-N}}. \tag{1}$$

Recall that $H(z)$ describes the input-output relation $Y(z) = H(z)X(z)$. From this we get the following relationship:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{B(z)}{A(z)} \Rightarrow Y(z) = H(z)X(z),\ A(z)Y(z) = B(z)X(z), \text{ and } A(z)H(z) = B(z). \tag{2}$$

The transform equation

$$(1 + a[1]z^{-1} + \ldots + a[N]z^{-N})Y(z) = (b[0] + b[1]z^{-1} + \ldots + b[M]z^{-M})X(z) \tag{3}$$

describes the difference equation

$$\begin{aligned} y[n] + a[1]y[n-1] + \ldots + a[N]y[n-N] &= b[0]x[n] + b[1]x[n-1] + \ldots + b[M]x[n-M] \\ &\text{or} \\ \sum_{i=0}^{N} a[i]y[n-i] &= \sum_{i=0}^{M} b[i]x[n-i]. \end{aligned} \tag{4}$$

If the sequence $\{x[n]\}$ is the unit pulse sequence $\{\delta[n]\}$, then $\{y[n]\}$ is called the unit pulse response sequence $\{h[n]\}$. Then, from eqn (4), $\{h[n]\}$ satisfies the following difference equation

$$\sum_{i=0}^{N} a[i]h[n-i] = b[n]. \tag{5}$$

The Z-transform of this is $A(z)H(z) = B(z)$, which is eqn(2). The output of the filter $H(z)$ at time n is

$$y[n] = -\sum_{i=1}^{N} a[i]y[n-i] + \sum_{i=0}^{M} b[i]x[n-i] \tag{6}$$

which illustrates that the output at time n is a linear combination of the previous N outputs and $M+1$ most recent input samples. It is this difference equation in eqn (6) that we will code on the DSK.

3 Direct Realizations of IIR Filters

3.1 IIR Direct Form I Realization

For convenience, we will assume that $N = M$. The most basic way to implement an IIR filter is to set up two array buffers, each represented by the delay operator z^{-1} in Figure 1, to hold the most recent M inputs and $N-1$ outputs. The output is then calculated as in Figure 1, which is the *direct form I* realization of eqn (6).

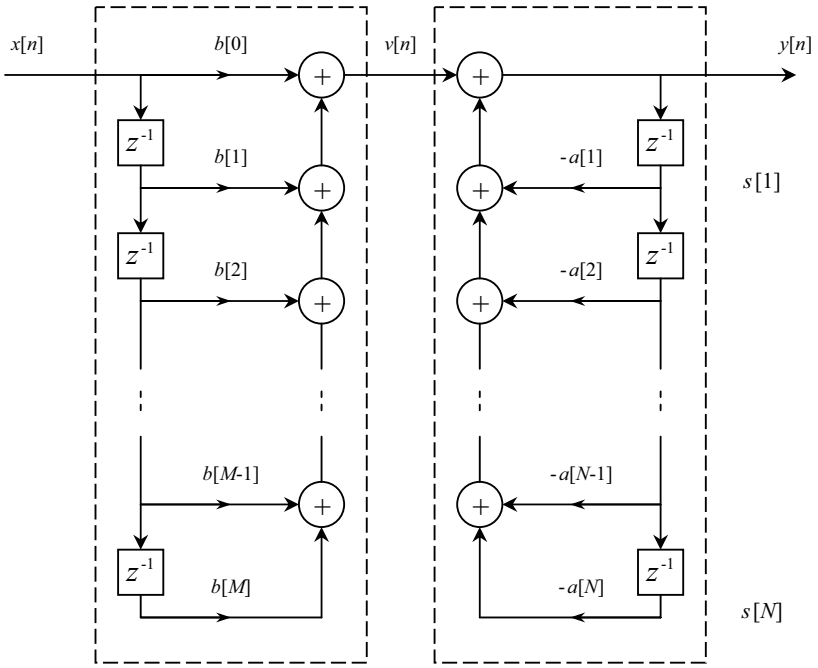

Figure 1: Direct Form I Realization. Adapted from [5].

Another way to calculate the output would be to calculate the intermediate sum

$$v[n] = \sum_{i=0}^{M} b[i]x[n-i] \tag{7}$$

and use it to calculate

$$y[n] = v[n] - \sum_{i=1}^{N} a[i]y[n-i]. \tag{8}$$

The direct form I realization requires $N + M + 1$ ($2N + 1$ operations when $M = N$) multiply and accumulate (MAC) operations and $N + M + 1$ memory blocks and data transfers. The $N + M + 1$ MAC operations are inherent to the algorithm and cannot be reduced. However, the number of memory blocks can be reduced from $N + M + 1$ to N, essentially cutting the memory storage requirement in half when $M = N$. This leads to the direct form II algorithm.

3.2 IIR Direct Form II Realization

A more memory efficient way to implement an IIR filter is the *direct form II* realization shown in Figure 2. You should be able to argue from linearity and shift-invariance that Figures 1 and 2 are equivalent.

In direct form II realizations, the output at time n is calculated from the input at time n and a linear combination of the N filter states, namely $s_1[n], s_2[n], \ldots, s_N[n]$. Once the filter output, $y[n]$, is calculated, the filter states are updated for the next input. The direct form II realization is implemented in hardware as follows [6]:

1. Compute $v[n]$.
$$v[n] = x[n] - \sum_{i=1}^{N} a[i]s_i[n]$$
2. Compute the output $y[n]$.
$$y[n] = b[0]v[n] + \sum_{i=1}^{N} b[i]s_i[n]$$
3. Update state variables starting with state N and working backwards through the states.
$$\begin{aligned} s_N[n+1] &= s_{N-1}[n] \\ s_{N-1}[n+1] &= s_{N-2}[n] \\ &\vdots \\ s_2[n+1] &= s_1[n] \\ s_1[n+1] &= v[n] \end{aligned}$$

The direct form II method reduces the number of memory blocks required, but it still requires N data transfers to compute the output $y[n]$. This can be eliminated with the direct form II transpose. You should be able to argue from linearity that Figures 2 and 3 are input-output equivalent.

The *direct form II transpose* realization is illustrated in Figure 3. It is implemented in hardware as follows [6]:

1. Compute $y[n]$.
$$y[n] = b[0]x[n] + s_1[n]$$
2. Update state variables starting with state 1 and working forwards through the states.
$$\begin{aligned} s_1[n+1] &= b[1]x[n] - a[1]y[n] + s_2[n] \\ s_2[n+1] &= b[2]x[n] - a[2]y[n] + s_3[n] \\ &\vdots \\ s_{N-1}[n+1] &= b[N-1]x[n] - a[N-1]y[n] + s_N[n] \\ s_N[n+1] &= b[N]x[n] - a[N]y[n] \end{aligned}$$

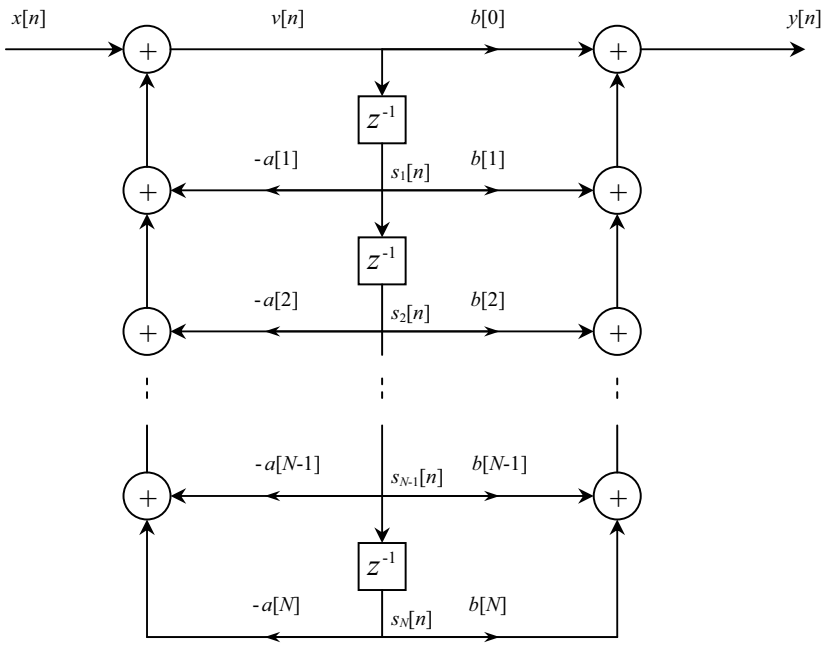

Figure 2: Direct Form II Realization ($M = N$). Adapted from [5] and [6].

4 Cascaded Second-Order Systems (Biquads)

The direct forms suffer from numerical problems (quantization errors and overflow) and are not preferred in many applications [3]. Instead, higher order systems may be broken up into a cascade of second-order systems (sections) known as *biquads* [6]. The second-order systems are generated by combining real pairs and complex conjugate pairs of the poles and zeros. These second order systems are normalized so that the leading coefficient of the denominator is one in each section. If there are N_s biquads in a cascade of second-order system, then eqn(1) may be written as [3]

$$H(z) = G \prod_{k=1}^{N_s} \frac{b[0,k] + b[1,k]z^{-1} + b[2,k]z^{-2}}{1 + a[1,k]z^{-1} + a[2,k]z^{-2}}. \tag{9}$$

The coefficients $b[i,k]$ and $a[i,k]$ will be coded as a two-dimensional array in C. In this lab, we will design biquads with $b[0,k] = 1$ for all k. This is done to reduce the number of multiplications required. The cascading of biquads (with $b[0,k] = 1$) is illustrated in Figure 4 below.

The factored gain of the cascaded system is G, which will scale the input into the first biquad. As we will see, this gain is typically less than 1, and does not determine the maximum gain of the filter. The output of the first biquad is the input to the second biquad. The output of the second biquad is the input to the third biquad and so on. The output of the final biquad is the filtered output $y[n]$. The implementation of the individual biquads may be any of the direct forms.

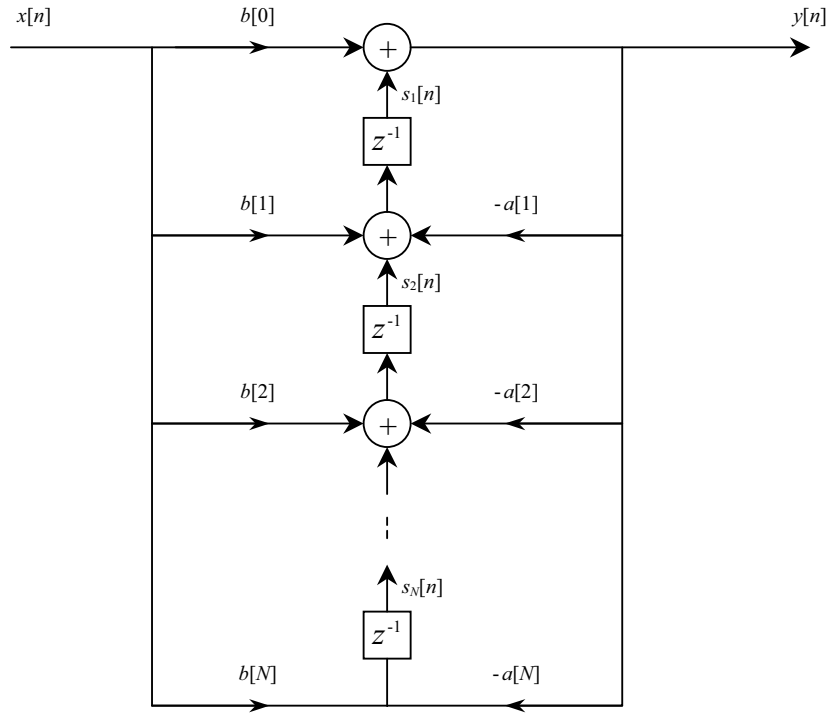

Figure 3: Direct Form II Transpose Realization ($M = N$). Adapted from [5] and [6].

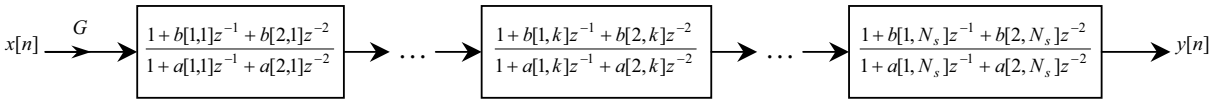

Figure 4: N_s Second-Order Sections. Adapted from [3].

5 IIR Filter Design

Since IIR filters are not designed under a general linear phase constraint like FIR filters, we will not be concerned with preserving symmetry conditions in our impulse response. There are various way to design digital filters, but we will only explore a few here. In particular, we will design rational analog functions and covert them to digital via impulse invariant and bilinear Z-transformations. Before we start with analog designs, let's try to develop some intuition on how poles and zeros of a transfer function affect the magnitude of the complex frequency response.

5.1 Pole/Zero Placement Design of IIR Filters

In the last lab, we explored how zero placements affect the frequency response. Consider the complex Z-plane shown in Figure 5.

We use the Z-plane to plot the poles and zeros of the filter transfer function $H(z)$. For causal stable systems, all of the poles must lie in the unit disc, and for minimum-phase systems, all

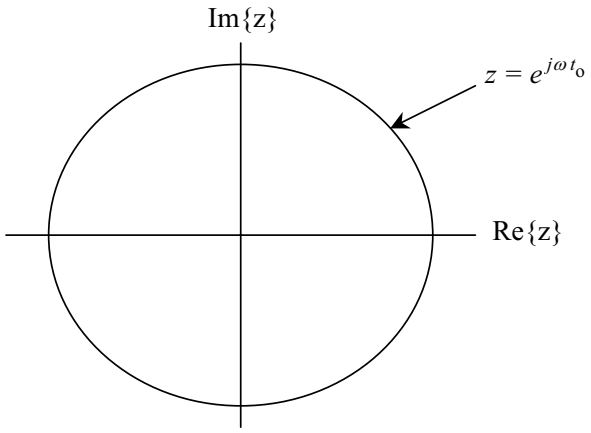

Figure 5: Z-Plane With Unit Disc $z = e^{j\omega t_o}$.

of the zeros must also lie in the unit disc ($|z| < 1$). When we evaluate $H(z)$ on the unit circle ($z = e^{j\omega t_o}$), we get the complex frequency response of our filter, $H(e^{j\omega t_o})$. At a frequency ω for which $e^{j\omega t_o}$ gets close to a zero in the Z-plane, we observe an attenuation in the magnitude response, $H(e^{j\omega t_o})$. If the zero has a magnitude less than one, then the phase response will not vary much at frequencies close to the zero (minimum phase). If the zero has a magnitude greater than one, then the phase response will vary much at frequencies close to the zero. As a frequency on the unit circle gets close to a pole in the Z-plane, we observe an amplification in the magnitude response of $H(e^{j\omega t_o})$. To see this, consider a transfer function with two poles (one minimum phase and one non-minimum phase) and one pole. Let the minimum phase zero, z_1, be at 500Hz and have a magnitude of .9 (i.e. $z_1 = .9e^{j2\pi 500 t_o}$). Let the non-minimum phase zero, z_2, be at 2500Hz and have a magnitude of 1.2 (i.e. $z_2 = 1.2e^{j2\pi 2500 t_o}$). Let the pole, p_1, be at 1500Hz and have a magnitude of .7 (i.e. $p_1 = .7e^{j2\pi 1500 t_o}$). For real filters, poles and zeros must occur in complex conjugate pairs, so our filter would be of the form

$$H(z) = \frac{(1 - z_1 z^{-1})(1 - \overline{z_1} z^{-1})(1 - z_2 z^{-1})(1 - \overline{z_2} z^{-1})}{(1 - p_1 z^{-1})(1 - \overline{p_1 z^{-1}})}. \tag{10}$$

If we let $t_o = .125$ms and multiply out eqn(10), then $H(z)$ becomes

$$H(z) = \frac{1 - 0.7445 z^{-1} + 0.7226 z^{-2} - 1.6508 z^{-3} + 1.1664 z^{-4}}{1 - 0.5358 z^{-1} + 0.49 z^{-2}}. \tag{11}$$

Using the the homebrew function `plotZTP.m` (located on the class webpage), the pole-zero diagram, complex frequency response, and impulse response may be plotted by typing the following in the MATLAB workspace.

```
>>b=[1 -0.7445 0.7226 -1.6508 1.1664]; % numerator coefficients
>>a=[1 -0.5358 0.49]; % denominator coefficients
>>tmax=30; % display tmax samples of the impulse response
>>fs=8000; % set sampling frequency
>>plotZTP(b,a,tmax,fs) % Analyze filter H(z)
```

The output of `plotZTP.m` is shown in Figure 6.

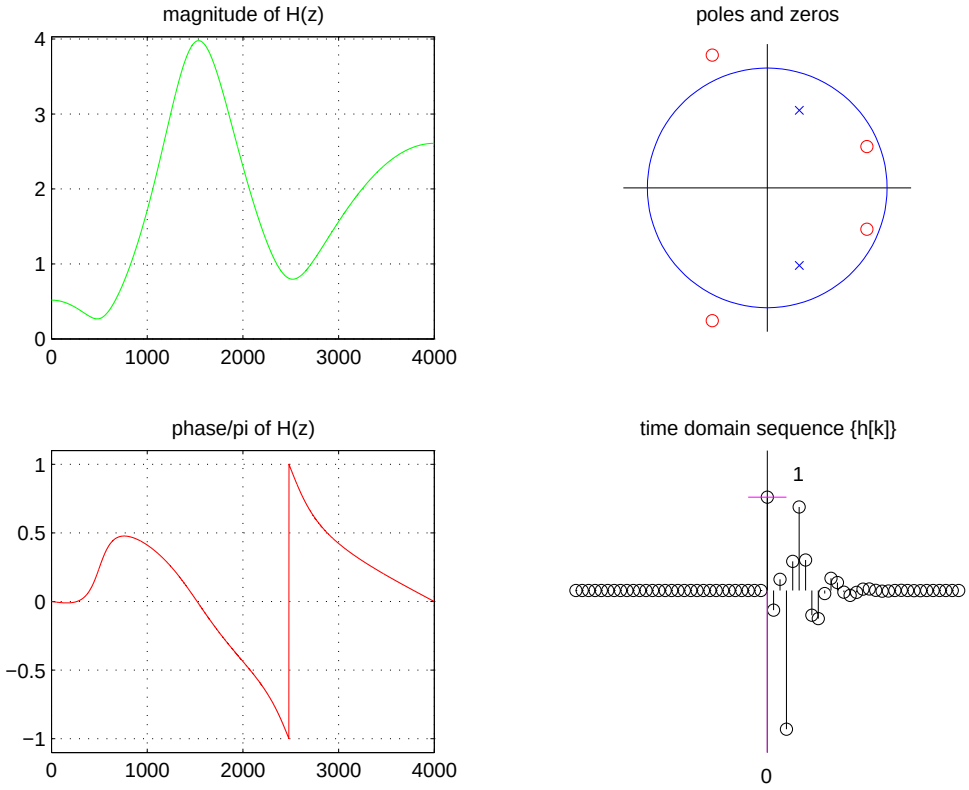

Figure 6: $H(z)$ as defined in eqn(11).

You will explore this concept in more depth later in this lab.

5.2 Standard Second-Order Section

The standard analog second-order section is

$$H_c(s) = \frac{\Omega_n^2}{s^2 + 2\xi\Omega_n s + \Omega_n^2} = \frac{\Omega_n^2}{(s + \Omega_n e^{j\cos^{-1}\xi})(s + \Omega_n e^{-j\cos^{-1}\xi})}, \tag{12}$$

where ξ is the damping coefficient and Ω_n is the natural undamped frequency (in radians per second). If $\xi = 0$, the system is undamped and eqn (12) is the transfer function of a sinusoid with frequency Ω_n. The system is underdamped when $0 < \xi < 1$, critically damped when $\xi = 1$, and overdamped when $\xi > 1$. To explore this in continuous-time, download the homebrew MATLAB function (from the class webpage) for plotting a Laplace transform pair, `plotLTP.m`, which is the analog version of the plot Z-transform pair tool, `plotZTP.m`.

5.3 Digital Filter Design Using Impulse Invariance and Bilinear Z-transforms

The classical way to design digital filters is to first design an analog filter and then convert it to a digital filter using a transformation. In this lab, we will only explore the impulse invariant and bilinear Z-transforms.

In the last lab, we explored impulse invariance from the point of view of an impulse response. In short, if $h(t) \longleftrightarrow H(j\omega)$ is the impulse response of an analog filter, then the corresponding digital filter will have impulse response

$$t_o h(nt_o) = h[n] \longleftrightarrow H(e^{j\omega t_o}) = \sum_r H\left[j\left(\omega + r\frac{2\pi}{t_o}\right)\right] = \sum_r H\left[j2\pi\left(f + \frac{r}{t_o}\right)\right], \tag{13}$$

where t_o is the sampling interval of the digital filter. If the original analog filter, $H(j\omega)$, is approximately bandlimited to $\frac{\omega_o}{2} = \frac{2\pi f_o}{2} = \frac{\pi}{t_o}$, then the spectrum of our digital filter, $H(e^{j\omega t_o})$, will be the same as the spectrum of our analog filter on the Nyquist band ($\frac{-\pi}{t_o} < \omega < \frac{\pi}{t_o}$). Of course, the difference will be that our digital filter will alias every $\frac{2\pi}{t_o}$ radians per second in the frequency domain.

To convert a rational analog filter to rational digital filter in the frequency domain, we must write out the partial fraction expansion of analog filter. Consider the analog filter

$$H_c(s) = \frac{B(s)}{A(s)} = \frac{b[0]s^M + b[1]s^{M-1} + \ldots + b[M-1]s + b[M]}{a[0]s^N + a[1]s^{N-1} + \ldots + a[N-1]s + a[N]}. \tag{14}$$

Equation (14) may then be written in terms of its partial fraction expansion as

$$H_c(s) = \sum_{k=1}^{N} \frac{\gamma_k}{s - \alpha_k}, \tag{15}$$

where the residues (γ_k) and poles (α_k) are calculated from $B(s)$ and $A(s)$. For examples on doing this in MATLAB, see [2]. From eqn(15), the impulse response may be written as

$$h(t) = \sum_{k=1}^{N} \gamma_k e^{\alpha_k t} u(t). \tag{16}$$

Using the impulse invariance eqn (13), the corresponding discrete-time impulse response is

$$h[n] = t_o \sum_{k=1}^{N} \gamma_k e^{\alpha_k n t_o} u[n], \tag{17}$$

which has the transfer function

$$H(z) = \sum_{k=1}^{N} \frac{\gamma_k t_o}{1 - e^{\alpha_k t_o} z^{-1}}. \tag{18}$$

Using equations (15) and (18), a transformation between an analog and digital filter may be made using impulse invariance. However, if the analog filter is not bandlimited, then the corresponding digital filter will have aliasing effects in the Nyquist band. This analog to digital mapping is not unique and is therefore not invertible.

When aliasing in the Nyquist band is an issue, the bilinear Z-transformation may be used. A bilinear Z transformation is derived by mapping the entire s-plane into $\frac{-j\pi}{t_o} < \text{Im}(s) < \frac{j\pi}{t_o}$ to prevent aliasing on the Nyquist band [1]. This newly mapped s-plane, s', is then mapped to the z-plane via the transformation $z = e^{s't_o}$. The general bilinear Z transform is

$$s = \frac{2}{t_o}\frac{1 - z^{-1}}{1 + z^{-1}}. \tag{19}$$

For a derivation of this, refer to [1]. The bilinear transformation results in frequency warping from mapping the frequency range $-\infty < \omega < \infty$ of the analog filter to the Nyquist band $\frac{\pi}{t_o} < \omega < \frac{\pi}{t_o}$ of the corresponding digital filter. This is due to the nonlinear mapping between the s-plane and z-plane. To design a digital filter using a bilinear Z transformation, the cutoff and stopband frequencies in your digital filter must be pre-warped to corresponding frequencies in the analog filter [1]. If ω represents the desired frequency in a digital filter, then its corresponding pre-warped analog frequency Ω will be [1]:

$$\Omega = \frac{2}{t_o}\tan\left(\frac{\omega t_o}{2}\right). \tag{20}$$

5.4 MATLAB's SPTOOL For IIR Filter Design

In the last lab, we used the MATLAB signal processing tool, SPTOOL, to design FIR filters. In SPTOOL, there are four types of IIR filter available (in addition to the three types of FIR filter), namely Butterworth IIR, Chebyshev Type I IIR, Chebyshev Type II IIR, and elliptic IIR filters. These filters may be exported from SPTOOL to the MATLAB workspace. If the filter is exported as `filt1`, as in the last lab, then the numerator and denominator coefficients may be accessed by typing the following in the MATLAB workspace

```
>>b=filt1.tf.num; % numerator coefficients
>>a=filt1.tf.den; % denominator coefficients.
```

There is a pre-developed MATLAB m-file (`IIR_cof_gen.m`) that will create a `.cof` file, which we will include in our generic IIR based C code. This will be demonstrated later. In general, we will only implement these filters using cascaded second-order sections for numerical stability. When designing second-order sections, the transfer function must be proper[1].

[1] A proper transfer function is a transfer function where the order of the denominator polynomial, $A(z)$, is greater than or equal to the order of the numerator polynomial, $B(z)$. This means that the number of poles in a transfer function must be greater than or equal to the number of zeros. In the SPTOOL designs that we do, this will always be the case.

5.5 All-Pass Filters

All-pass filters are filters that have a constant magnitude for all frequencies. In the case of digital filters, this means that the they have a constant magnitude on the Nyquist band[2]. Complex digital all-pass systems are of the form [3]

$$H(z) = \frac{z^{-1} - \overline{a}}{1 - az^{-1}}. \tag{21}$$

Digital all-pass systems are systems that have pole - zero pairs that (in their polar representations) are at the same frequency, but have magnitudes that are reciprocals of one another. The complex number, a, is of the form $a = re^{j\omega t_o}$. In eqn (21), r is the magnitude of the pole and $\frac{1}{r}$ is the magnitude of the zero. For real filters, poles and zeros must appear in complex conjugate pairs (if they are not real to begin with), so most all-pass systems are cascades of eqn (21). The phase of an all-pass filter will be rather linear except around the frequencies associated with the pole - zero pairs. The applications of these type of filters is for altering the phase response of a system at a given frequency. Also, they are used in minimum-phase/all-pass decompositions. For examples of this, refer to [3].

6 IIR Filter Implementations

As with the FIR filter designs in MATLAB, an m-file has been developed that will create `.cof` files, which will be included at the beginning of our C coded algorithms. Download the file `IIR_cof_gen.m` from the class webpage. Use `>>help` and `>>type` in MATLAB to learn how to use this file.

6.1 Direct Form I implementation on the C6713 DSK *

When implementing a direct form I realization on the C6713 chip (with the assumption $M = N$), eqn (4) would be re-written as

$$y[n] = b[0]x[n] + \sum_{i=1}^{N}(b[i]x[n-i] - a[i]y[n-i]). \tag{22}$$

This may be coded in C as illustrated in Figure 7.

Assignment

1. Download the files `IIR_directI.c` and `IIR_LPF1800Hz.cof` from the webpage. Build this project and implement it on the DSK.

[2]Digital all-pass filters alias white in frequency.

```
// IIR_directI.c  IIR filter using Direct Form I
// assume M=N
/*1 */ #include "DSK6713_aic23.h"
/*2 */ #include "IIR_LPF1800.cof"     // LPF @ 1800 Hz coefficient file
/*3 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;// sampling frequency of codec
/*4 */ short x[N+1] = {0};             // input samples
/*5 */ short y[N+1] = {0};             // output samples
/*6 */ int yn;
/*7 */
/*8 */ interrupt void c_int11()        // ISR
/*9 */ {
/*10*/   int i;
/*11*/   yn=0;
/*12*/   x[0] = input_left_sample();   // input sample
/*13*/   yn += (b[0]*x[0]>>15);        // b[0]*x[n]
/*14*/
/*15*/   for (i = 1; i <= N; i++)
/*16*/     yn += ((b[i]*x[i]>>15) - (a[i]*y[i]>>15));
/*17*/   for (i = N; i > 0; i--)
/*18*/     {
/*19*/       x[i]=x[i-1];              // update buffers
/*20*/       y[i]=y[i-1];
/*21*/     }
/*22*/   y[1]=yn;                      // store current output as y[n-1]
/*23*/   // for next output
/*24*/   output_left_sample((short)yn);// output final result for time n
/*25*/   return;                       // return from ISR
/*26*/ }
/*27*/
/*28*/ void main()
/*29*/ {
/*30*/   comm_intr();                  // init DSK, codec, SP0
/*31*/   while(1);                     // infinite loop
/*32 */ }
```

Figure 7: Listing of `IIR_directI.c`.

6.2 Direct Form II implementation on the C6713 DSK *

Assignment

2. Create a project that implements a fixed-point direct form II realization of an IIR filter. Use the `.cof` file `IIR_LPF1800Hz.cof` from the direct form type I realization to test your program. Explain your algorithm and include a copy of your C code.

6.3 Second-Order Section implementation on the C6713 DSK *

MATLAB has a built-in function that will generate second order sections from the poles and zeros of a transfer function. In MATLAB, we code $B(z)$ as the vector $\mathtt{b} = [b[0], b[1], \ldots b[N]]$ and

$A(z)$ as the vector `a` $= [1, a[1], \ldots a[N]]$. The poles and zeros of a transfer function are found using the built-in MATLAB function `tf2zp` (*transfer function to zero pole*) by typing

```
>>[z,p,k]=tf2zp(b,a);
```

in the MATLAB workspace. From here, the biquads are formed by using the built-in MATLAB function `zp2sos` (*zero pole to second-order systems*) by typing

```
>>[H,G]=zp2sos(z,p,k);
```

in the MATLAB workspace. The matrix `H` contains the filter coefficients in Figure 4, and the constant `G` is the overall gain of the system.

The homebrew function `IIR_cof_gen.m` will create the `.cof` files that are to be included in `IIR_sos.c`. Once these biquads are created, they may be implemented in C code as illustrated in Figure 8. NB: The homebrew function `IIR_cof_gen.m` is only meant to be used with `IIR_sos.c` and will not work with direct form implementations mentioned above.

```
// IIR_sos.c  IIR filter using cascaded second-order sections (biquads)
// Each biquad is coded as a Direct Form II transpose realization
/*1 */ #include "DSK6713_aic23.h"
/*2 */ #include "IIR_LPF1800.cof"     // LPF @ 1800 Hz coefficient file
/*3 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  // sampling frequency of codec
/*4 */ short state[Ns][2] = {0};    // filter states in each biquad
/*5 */
/*6 */ interrupt void c_int11()  // ISR
/*7 */ {
/*8 */  int i, input;
/*9 */  int un, yn;
/*10*/
/*11*/  input = input_left_sample();// input from codec
/*12*/  un = ((int)input*G)>>15;    // scale input
/*13*/
/*14*/  for (i = 0; i < Ns; i++)    // repeat for each biquad
/*15*/   {
/*16*/    yn = un + state[i][0];    // output at current biquad
/*17*/                              // update states in current biquad
/*18*/    state[i][0] = ((b[i][0]*un)>>15) - ((a[i][0]*yn)>>15) + state[i][1];
/*19*/    state[i][1] = ((b[i][1]*un)>>15) - ((a[i][1]*yn)>>15);
/*20*/    un=yn;
/*21*/   }
/*22*/    output_left_sample((short)yn);// output final result for time n
/*23*/    return;     // return from ISR
/*24*/ }
/*25*/
/*26*/ void main()
/*27*/ {
/*28*/   comm_intr();               // init DSK, on-board codec, SP0
/*29*/   while(1);     // infinite loop
/*30*/ }
```

Figure 8: Listing of `IIR_sos.c`.

Assignment

3. Create a floating-point version of `IIR_sos.c`. Create an IIR filter using SPTOOL and use `IIR_cof_gen.m` to create a floating-point `.cof` file. Use `plotZTP.m` to analyze your filter. Describe the filter that you designed. Include a copy of your C code, and a copy of the Figure generated by `plotZTP.m`. Comment on your results.

4. Use a bilinear Z-transform to convert the standard second-order section in eqn (12) to a digital filter. Implement it in on the DSK and using impulses as an input. NB: This can be done in C code by creating a counter. Once the counter reaches a certain value, set the current input equal to one and reset the counter. For all other values not equal to the counter limit, set the input to zero. Provided that your system is damped ($\xi > 0$), the output should disappear after a few milliseconds. Choose an appropriate counter so the system has reached steady-state before another impulse is applied. Use the CCS help files to research the structure of if-else statements. For this part, do not use $\xi = 0$ (we will explore this next). Implement under damped, critically damped, and over damped systems. Comment on your results. Include a copy of the code used and your derivation of the digital filter $H(z)$ using the bilinear Z-transform.

5. Re-implement the previous program with $\xi = 0$, except use only one impulse at time 0 as your input. Choose ω_n in your digital filter to generate a 2kHz sinusoid. Be sure to use eqn (20) to choose the analog filter frequency Ω_n in eqn (12). Briefly outline the steps required to generate the 2kHz sinusoid from the analog standard second-order section. Comment on your results and give as much intuition as possible.

6. NOTE: do two out of four of the following: Design lowpass, highpass, bandstop, and bandpass filters using Butterworth, Chebyshev I, Chebyshev II, and elliptic filters in MATLAB. Mix and match these filter types so that you create two filter types using two methods. Use the file `IIR_cof_gen.m` to create header files that contain the second-order sections of the filters. Use `plotZTP.m` to analyze these filters and implement them on the DSK. Comment on the pass and stop bands of these filters (Butterworth, Chebyshev I, Chebyshev II, and elliptic filters). Include a copy of the `plotZTP.m` plots and give the specifications of each filter. Comment on your results.

6.4 IIR Filter Design - Comb Filter *

Suppose that we would like to design a simple comb filter that passes one tone well and suppresses other tones. This could be done by designing a filter with the transfer function

$$\begin{aligned} H(z) &= \frac{1}{(1-\alpha e^{j\omega_1 t_o} z^{-1})(1-\alpha e^{-j\omega_1 t_o} z^{-1})} \\ &= \frac{1}{1-2\alpha\cos(\omega_1 t_o) z^{-1} + \alpha^2 z^{-2}} \\ &= \frac{1}{1-2\alpha\cos(2\pi f_1 t_o) z^{-1} + \alpha^2 z^{-2}} \end{aligned} \tag{23}$$

where $0 \leq \alpha < 1$ for this to be stable.

Assignment

7. (a) Design the comb filter given in eqn(23) for the that passes a 500Hz tone. Use `plotZTP.m` to examine this filter. What value did you use for α.

   (b) Using the signal generator as your input source, listen to the output of the comb filter at various frequencies. What can you say about the intensity (volume) of sinusoids at and near 500Hz?

   (c) Using C code, generate the input sequence $x[n] = A(\cos[2\pi f_1 n/f_s] + \cos[2\pi f_2 n/f_s])$, where $A < 16384 = 2^{15}$ is the amplitude, $f_1 = 500$Hz is the frequency passed by the comb filter, f_2 is varied, and $f_s = 8$kHz is the sample rate of the codec. (NB: This is the beating between tones algorithm from lab 2 implemented using the codec.) Instead of reading in values from the codec at each interrupt, use C code to create the current input sample $x[n]$. Filter this signal using your comb filter designed in part (a). Observe and listen to both the filtered and unfiltered versions of $x[n]$ on the oscilloscope and with headphones. Do this for various values of f_2 between 2kHz and 500Hz. What can you say about the filtered output as $f_2 \to f_1$? Does this filter have a linear phase? Comment on your results and give as much intuition as possible.
   HINT: To listen to the filtered and unfiltered versions of the beating pattern in the same program, use an if-else conditional statement that tests a global variable. If the variable is a binary one (TRUE), then filter, and if the variable is a binary zero (FALSE), output the input sample (as in the straight wire program).

6.5 IIR Filter Design - Improved Notch Filter *

In the last lab, we designed a notch filter that knocked out a specific frequency, but the filter response greatly suppressed tones near the notch frequency. This filter can be improved by cascading the FIR notch filter with the IIR comb filter examined above. The resulting transfer function is

$$H(z) = \frac{1 - 2\cos(2\pi f_1 t_o)z^{-1} + z^{-2}}{1 - 2\alpha\cos(2\pi f_1 t_o)z^{-1} + \alpha^2 z^{-2}} \tag{24}$$

where f_1 is the notch frequency. This a second order IIR system (one biquad) and may easily be implemented using any of the IIR filter realizations mention above.

Assignment

8. Design a notch filter as described by eqn (24) for the codec that notches a 500Hz tone. Use `plotZTP.m` to examine this filter. What is the value of α? Repeat the experiment from question 6 part (c) above. What can you say about the filtered output as $f_2 \to f_1$? Comment on your results and give as much intuition as possible. How does this notch filter compare to the notch filter from the FIR lab?

7 End Notes

7.1 Advanced Lab Suggestions

- An advanced topics would be to code IIR filters using the C6000 fixed- and floating-point DSP assembly instructions.
- Any desired frequency response of a filter may be obtained by summing up a given number of all-pass filters in parallel. The idea is to have signals add constructively and destructively to produce a given magnitude at a given frequency. An advanced topic would be to develop a way to design a filter using only all-pass filters.
- Designing digital filters using covariance invariant (and other) transformations between analog and digital filters [4]. For a list of other transformations, refer to [1, Ch. 10]
- Many guitar effects processors use simple difference equations to create their effects. Some of these effects include: reverb, delay, flanger, chorus, distortion, phaser and pitch-shifter. An advanced lab would be to implement these and other effects using the DSK.

References

[1] Jackson Leland B. *Digital Filtering and Signal Processing*. Kluwer Academic Publishers, Norwell, MA, 1996.

[2] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 4: Polynomials, Laplace Transforms, and Analog Filters in MATLAB*, 2001.

[3] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Uper Saddle River, NJ, 1989.

[4] J. Perl and L.L. Scharf. Covariance-invariant digital fltering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 2(25):143–152, April 1977.

[5] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Prentice Hall, Uper Saddle River, NJ, 1996.

[6] Steven A. Tretter. *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711*. Kluwer Academic/Plenum Publishers, New York, 2003.