ECEN4002/5002 Digital Signal Processing Laboratory

Spring 2002

Laboratory Exercise #4

IIR Filters and Equalizers

Introduction

Previously (in Lab #2) you worked with the design and implementation of an FIR filter. Then in Lab #3 you experimented with some elementary IIR structures for recirculating delays and comb reverberators. In this exercise you will use the tools and skills from the previous labs to design, implement, and test some more complicated IIR digital filters. The first exercise will be to implement and test a high-order IIR filter made from a cascade of second-order sections. MATLAB is used to select the filter coefficients, which are then included in the 56300 filter implementation. The second exercise is to implement a bandpass parametric EQ filter, and to test and verify its behavior.

Design and implementation of IIR filters

IIR filters have transfer functions of the form

$$H(z)=\frac{\sum_{k=0}^{M-1} b_k z^{-k}}{1+\sum_{k=1}^{N-1} a_k z^{-k}},$$

which is a ratio of two polynomials, and it is assumed that M≤N. Unlike FIR filters, IIR filters can have both zeros and poles. This means that IIR filters are not guaranteed to be stable: the poles must be inside the unit circle in the *z*-plane for stability. However, we can more easily approximate a desired spectrum with an IIR filter than with a FIR filter due to the transfer function flexibility of having both poles and zeros to manipulate. Consequently, the computation load for an IIR filter tends to be smaller than with an equivalent FIR filter when implementing an arbitrary frequency response function.

Like the FIR filter, there are many ways to design an IIR filter. However, due to the rational structure of the IIR filter expression, we can easily approximate the design from an analog prototype filter. This is the method that we will use for this class. Given a set of filter specifications, we obtain a continuous-time filter approximation, $H_d(s)$, to the desired spectrum. The analog design $H_d(s)$ is then transformed to obtain the digital filter H(z). There are several methods for doing this transformation, and we will consider the *bilinear transform* method for this experiment.

The essential idea behind the bilinear transform method is to construct a digital filter H(z) for which $H\left(e^{j\omega}\right)=H_d\left(j\Omega(\omega)\right)$. The function Ω(ω) is a nonlinear frequency *warping* function that is used to map the discrete-time frequency ω (-π≤ω≤π), to the entire continuous-time frequency range -∞ to ∞. Specifically, if Ω( ) is chosen to increase monotonically from -∞ to ∞ as ω increases from -π to π then the mapping will warp the entire jω axis in the *s*-plane to the unit circle in the *z*-plane, and the entire left half of the *s*-plane to the inside of the unit circle in the *z*-plane. Such a warping function ensures that a stable discrete-time filter maps to and from a stable continuous-time filter. This allows us to start with the desired digital filter specifications, use the bilinear transform to map the design specs to the analog domain, choose an analog filter that meets the specs, then transform the design back into a digital transfer function.

The desired warping properties can be achieved by a transformation *z*=ψ(*s*), where

$$z = \psi(s) = \frac{1+s}{1-s}.$$

Note in particular that this transformation maps the origin of the *s*-plane to *z*=1, and as jω→∞, *z*→ -1.

The inverse transformation is

$$s = \psi^{-1}(z) = \frac{z-1}{z+1}.$$

The frequency warping function is obtained by setting $z=e^{j\omega}$, and is

$$j\Omega = j\frac{\sin(\omega/2)}{\cos(\omega/2)} = j\tan\left(\frac{\omega}{2}\right).$$

For this lab you will use IIR design tools from the MATLAB Signal Processing Toolbox, so it will not be necessary to manipulate the bilinear transform directly. Nevertheless, it is useful to keep in mind how the frequency warping process will map digital filter characteristics between the continuous and discrete domains.

The usual engineering design method is to specify the filter first, and then select the analog filter type that will best satisfy our requirements. Next we choose the corresponding tool in MATLAB and calculate the filter's poles and zeros. This leads to the final question of this section: how should we implement IIR filters?

We will begin by assuming that our IIR filter is only a second order filter, which can be stated as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}.$$

The 2nd-order expression is sometimes called a *biquadratic* formula, or just a "biquad" for short. Using the linear system property that Y(*z*)=H(*z*)X(*z*) we can write a difference equation using the inverse *z*-transform in terms of the filter coefficients and data,

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] \quad - a_1 y[n-1] - a_2 y[n-2].$$

The difference equation leads directly to the block diagram shown in Figure 1.

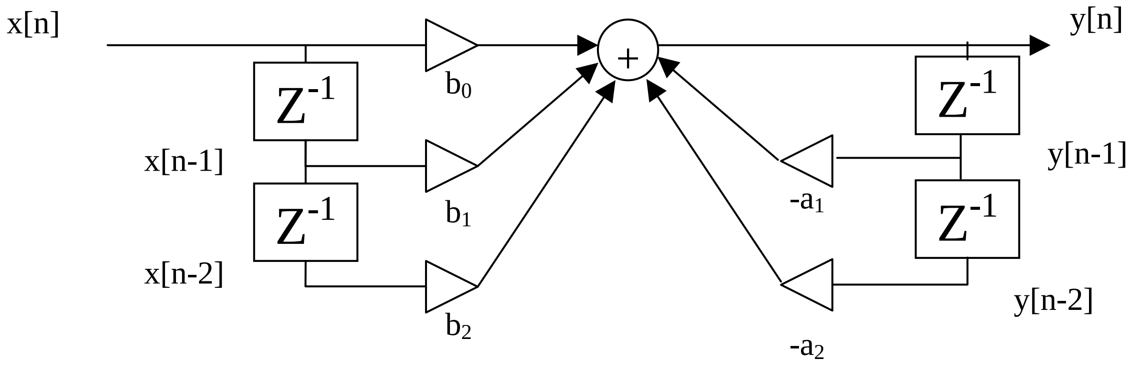

Figure 1: *Implementation of 2nd-order difference equation (Direct Form I).*

By breaking the central summation into two separate summers, one for the feed-forward terms and the other for the feedback terms, exchanging the order of the summation blocks—which is OK since the blocks are linear and time-invariant—and then combining the now adjacent matching delay elements, an alternative structure can be formed, as shown in Figure 2. There are numerous other structures for IIR implementations, but those will be left for a filter theory course.

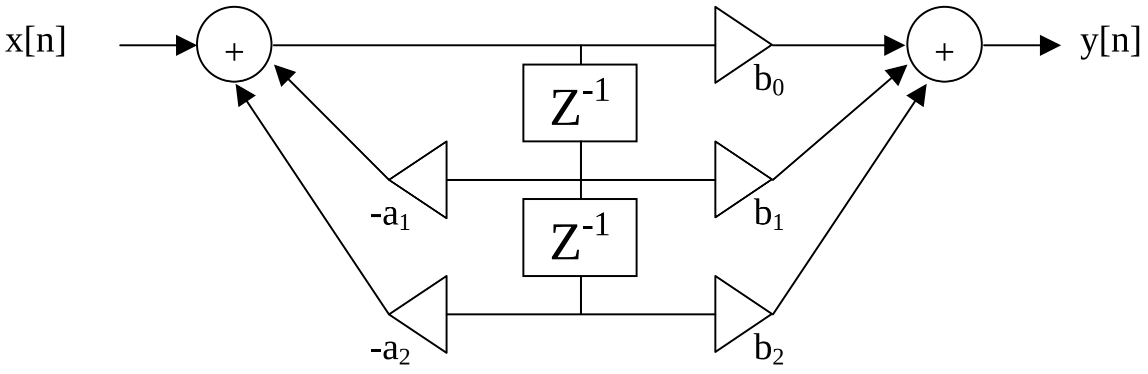

Figure 2: *Direct Form II implementation of 2nd-order difference equation.*

The sensitivity of a digital filter to coefficient approximation is similar to the need to deal with component tolerance in analog filters, and this sensitivity can be reduced by separating a high-order polynomial expression into a sum or cascade of lower-order sections. For this reason the implementation of IIR filters is almost always done with second order sections.

When the required filter order is greater than two, we factor the high-order numerator and denominator polynomials into second order polynomials, which is easily accomplished with knowledge of the filter's poles and zeros (look at the MATLAB `roots` and `poly` functions). Naturally, if the filter has even order, then we can build the filter with an integer number of cascaded second order sections. If the filter has odd order, then we need to add one more first order section making use of the relation

$$y[n] = b_0 x[n] + b_1 x[n-1] \quad - a_1 y[n-1] .$$

⇨Exercise A: IIR filters

For this exercise you will use some MATLAB functions to select a set of IIR digital filter coefficients, then use the coefficients in a filter implemented on the 5630x. We will use the `butter` function (Butterworth analog prototype) to design the filter, the `scope` function to examine the poles and zeros, and then a function `iirtable` to create an assembler-compatible text file (like you did with `firtable` in Lab #2). The `butter` function is part of the Toolbox, and the `iirtable` and `scope` functions are on the course web site along with several ancillary `.m` files. Remember, be smart about the way you develop your code. Start with just the 2nd order building block, debug everything, then extend to the higher-order version.

Part 1: IIR Butterworth lowpass filter, 8th order, 2.5kHz cutoff frequency

```
fs=48000; fc=2500;
[b,a]=butter(8,2*fc/fs);
scope(b,a,fs)
iirtable(b,a,’z:iirp1’)
```

The MATLAB function `scope` produces two figures that may be used to verify the design and implementation of the filter. The function `iirtable` generates a highly-structured assembler version of the IIR filter description. The file produced by `iirtable` includes coefficients that have been scaled to reduce the likelihood of overflow in the internal state of the filter. Review the structure of the output file carefully: your IIR filter implementation on the 5630x must use this data format directly! Use a Direct Form I implementation.

Since the coefficients of the digital filter must be between –1 and +1 in order to be compatible with the 56300 fractional representation, we often have a problem with the linear term of the 2nd-order polynomial, which is greater than 1. The `iirtable` function takes care of the numerator by scaling down each of the coefficients by the same factor, and the denominator is handled by dividing each of the coefficients by 2. Note that the output of `iirtable` stores $a_2/2$ and $a_1/2$. So, after doing the multiply-accumulate for the denominator coefficients, be sure to do a left arithmetic shift to compensate for the coefficients being divided by two. In other words, the coefficients are divided by 2, so the mac result must be multiplied by 2 to get the correct denominator value.

Incidentally, the 56300 has a feature that allows a program to use the coefficients divided by 2, then automatically scale up the product (left shift) in the hardware when the result is read out of the accumulator. This uses the scaling mode bits in the "mode" portion (MR) of the status register. Look at the *56300 Family Documentation* (chapter 5) to see how this is done, if you are interested.

Use the non-real time file i/o method to verify that your filter implements the desired response. Apply a unit pulse, record the output, and compare to the expected response shown by the scope program. Try applying a sinusoidal input with an interesting frequency. Note any overflow or any other unexpected behavior. Reduce the amplitude of your input signal if necessary.

Now run your filter in real time with mono input/output. Carefully measure the frequency response on a uniform logarithmic scale (20, 50, 100, 200, 500, 1000, …) and plot the results in Bode form (gain in dB on vertical axis, frequency in log scale on horizontal axis).

Finally, now that everything is working, consider how you could optimize your IIR filter (fewer instructions, more parallel moves, less memory, etc.) Try some of your optimizations (save a copy of your working file first!) and re-verify the results.

Part 2: IIR Butterworth bandpass filter, 10th order, 3kHz to 8kHz passband.

Now use the `butter` function to design a 10th order Butterworth bandpass filter. Read the MATLAB help for `butter` to see how the bandpass is specified.

Obtain the unit sample response using the file I/O method, then run in real time and verify the results as in Part 1.

Additional exercise for Graduate Students:
Design another 8th-order lowpass filter, but this time use an elliptic filter with 2.5kHz cutoff frequency. See the MATLAB `ellip` function. You can allow 0.5dB of ripple in the passband, and the stopband attenuation should be 60dB. Compare the frequency response of this filter with the Butterworth design from Part 1, and discuss the differences.

A parametric EQ filter

A common feature of audio recording and reproduction systems is frequency equalization, or EQ for short. EQ systems can come in several different forms, ranging from simple "tone" controls (low frequency or high frequency boost/cut), to graphic equalizers with multiple bandpass filters, and to parametric EQs with independently adjustable center frequency, bandwidth, and boost/cut. In this section a particular type of bandpass parametric EQ filter is considered.

EQ filters are typically variable filters, meaning that the user will want a "knob" or the software equivalent to make subjective adjustments to the filter settings. It is generally impractical to make a variable filter by recalculating all the coefficients in an ordinary IIR filter structure. Instead, we would like to devise a structure in which we can tune specific filter characteristics by adjusting only a single parameter. In an analog filter the adjustable element is usually a resister, while in a digital filter it is one of the numerical coefficients in the structure.

One example of a simple digital bandpass parametric equalizer comes from an engineering report by Regalia and Mitra (*IEEE Trans. ASSP-35, no. 1, January, 1987*). The structure of the Regalia-Mitra filter is shown in Figure 3.

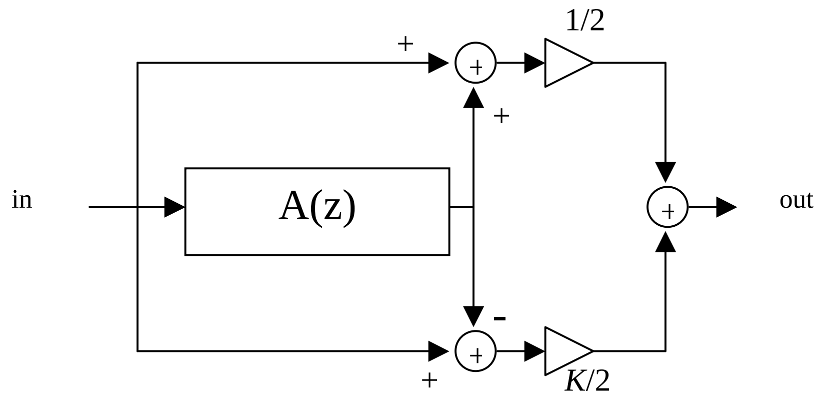

Figure 3: *Regalia-Mitra EQ filter structure.*

The embedded A(z) element is a second-order all-pass filter. Its transfer function is given by

$$A(z) = \frac{\alpha + \beta(1+\alpha)z^{-1} + z^{-2}}{1 + \beta(1+\alpha)z^{-1} + \alpha z^{-2}}$$

Note that A(z) is surrounded by several signal paths, gains, and summers. The overall transfer function of the entire bandpass equalizer is

$$F(z) = \frac{1}{2}\left[1 + A(z)\right] + \frac{K}{2}\left[1 - A(z)\right]$$

The structure has 3 gain elements: *K*, α, and β. *K* controls the amount of boost or cut of the bandpass EQ filter. The α parameter controls the bandwidth of the filter, and β controls the center frequency. The design equations can be derived to be:

$$\beta = -\cos\omega_0$$
$$K = \left|F\left(e^{j\omega_0}\right)\right| \qquad ,$$
$$\alpha = \frac{1 - \tan(BW/2)}{1 + \tan(BW/2)}$$

where ω_0 is the radian center frequency ($0 < \omega_0 < \pi$, corresponding to zero to $f_s/2$), and *BW* is roughly the -3dB bandwidth of the filter with *K* set to zero (notch filter). A few examples are shown in Figure 4. Note that the controls are pretty much independent: varying the bandwidth (α) parameter doesn't affect the gain or center frequency, and so forth.

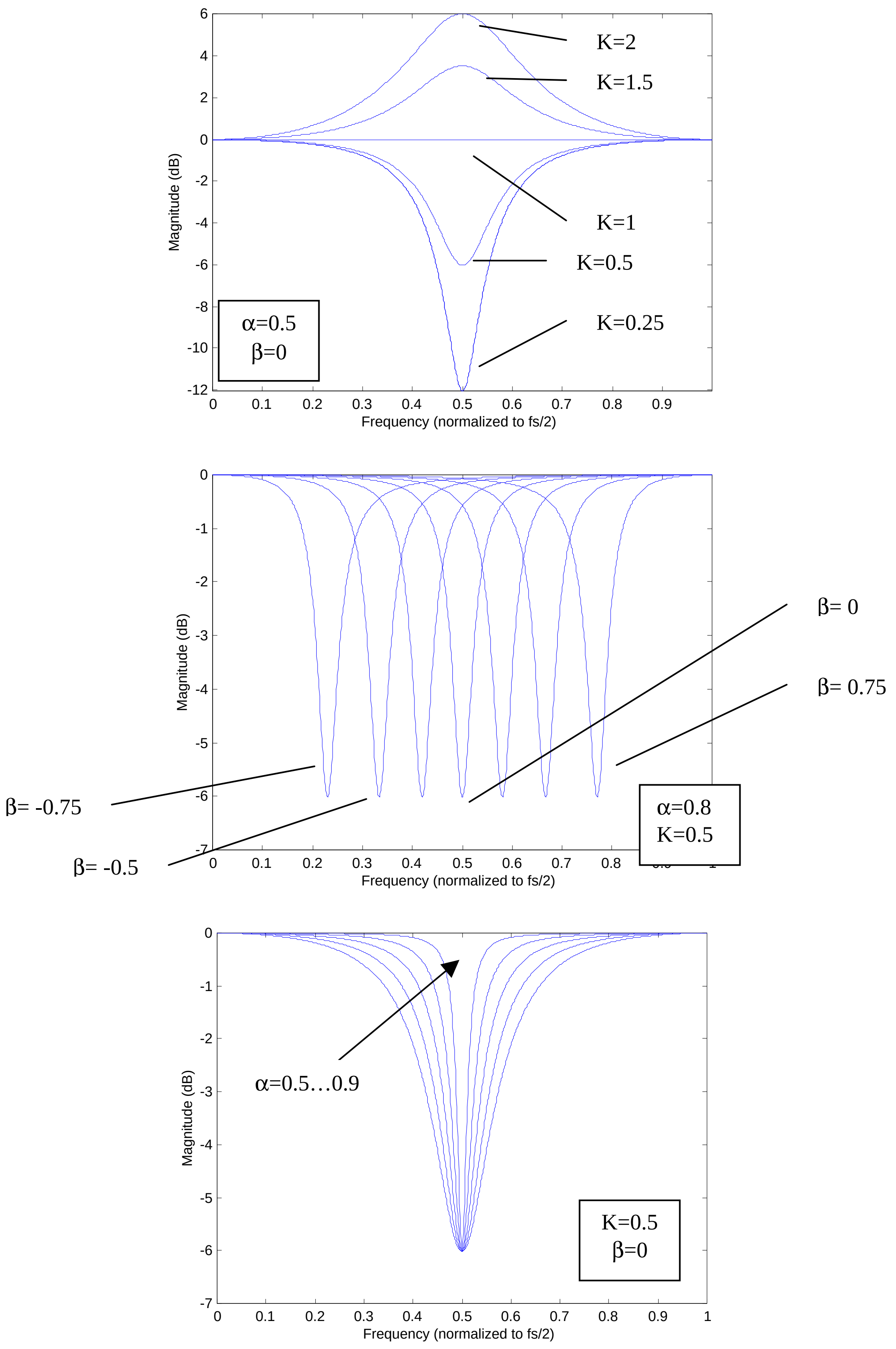

Figure 4: *EQ response for various values of K, α, and β.*

⇨Exercise B: EQ filter

Write assembly code to implement the Regalia-Mitra EQ filter. Test the filter for several settings of K, α, and β. Compare your 56300 results to predictions from MATLAB. Can you somehow handle values of K that are greater than 1, perhaps by doing a scale-down and scale-up algorithm?

Extra Credit

If you have time, construct a cascade of 3 independent EQ filters. Set the filters with different center frequencies and produce several different combinations of boost and cut with several different bandwidths. Show your results and comment on the behavior of your EQ system.

Report and Grading Checklist

A: IIR Filters

Code listing for IIR routine that uses `iirtable` coefficient file, with comments on scaling.
Part 1: results (MATLAB results, non-real time output, real time frequency response)
Part 2: results (MATLAB results, non-real time and real time results)
Measurements and data verifying design and implementation.
Comments on optimization, if any.

B: EQ Filters

Code listing with comments for Regalia-Mitra filter.
Results for several values of K, α, and β.
Verification and written comments.

Grading Guidelines (for each grade, you must also satisfy the requirements of all lower grades):

F	Anything less than what is necessary for a D.
D	Exercise A parts 1&2 MATLAB results.
C	Exercise A IIR implementation, meager results.
C+	Exercise A Parts 1&2 with verification results.
B	Exercise B EQ filter implementation.
A	Exercise B with validation and comments.

Note: grad student grading also requires the elliptic filter design for exercise A.