# ECE 4680 DSP Laboratory 5: IIR Digital Filters

**Due Date: _____**

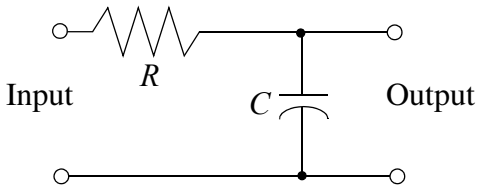
## Introduction and Background

### Infinite Impulse Response Basics

Chapter 4 of the Reay [1] deals with infinite impulse response (IIR) digital filters. IIR filters often result from a desire to represent a traditional analog filter (Butterworth, Chebyshev, elliptical, or Bessel[1]) in discrete-time form. Characteristics of these classical analog filter types are:

- The Butterworth filter is optimum1 in the sense that it provides the best Taylor series approximation to an ideal lowpass filter magnitude at both $f = 0$ and $\infty$
- A Chebyshev design achieves a more rapid rolloff rate near the cutoff frequency than the Butterworth by allowing ripple in the passband (type I) or stopband (type II). Monotonicity of the stopband or passband is still maintained respectively.
- Allows both passband and stopband ripple to obtain a narrow transition band. The elliptic (Cauer) filter is optimum in the sense that no other filter of the same order can provide a narrower transition band.

As a simple motivational example, consider the RC lowpass filter shown in Figure 1. We may




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

$$h_c(t) = \frac{1}{RC} e^{-t/(RC)} u(t)$$

**Figure 1:** RC lowpass filter.

choose to implement this in the discrete-time domain using either the *impulse invariant* approach or via a *bilinear transformation*. For details on these two approaches consult a DSP text, such as Oppenheim & Schafer [2]. In general we desire approaches to convert any $H_c(s)$ into $H(z)$. Let us first recall the general form of a discrete-time IIR system and consider filter topologies that will lead to efficient real-time implementation.

### General IIR Form

When a general IIR filter is obtained from say MATLAB or scipy.signal, the result is a coefficient set that starts out in direct form, that is we have

1. Testing reveals that `bessel` design does not work in the most recent version (0.18) of scipy.signal.

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

where $N$ is the filter order. The corresponding difference equation which must be implemented in real-time, for a direct-form based filter topology, is

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k] \tag{2}$$

## Direct Form I

An IIR filter has feedback, thus from DSP theory we recall the general form of an $N$-order IIR filter is

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{r=0}^{M} b_r x[n-r]\,. \tag{3}$$

By $z$-transforming both sides of (3) and using the fact that $H(z) = Y(z)/X(z)$, we can write

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}} \tag{4}$$

IIR filters can be implemented in a variety of topologies, the most common ones, direct form I, II, cascade, and parallel, will be reviewed below.

## Direct Form I

Direct implementation of (3) leads to the following structure of Figure 2. The calculation of $y[n]$

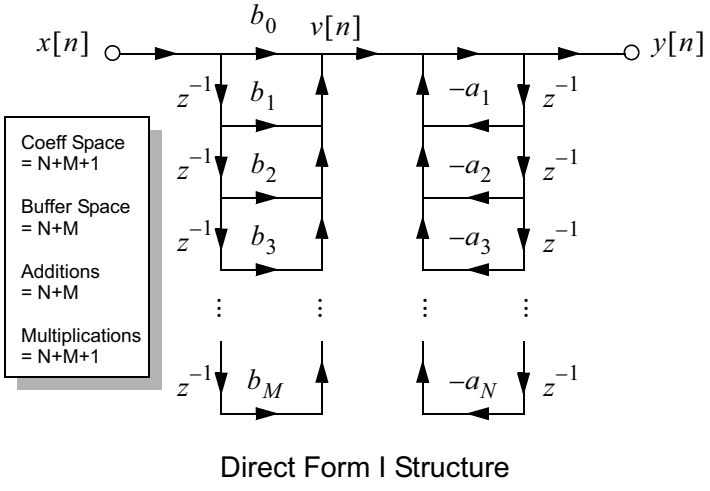



**Figure 2:** Direct form I structure for IIR filter implementation.

for each new $x[n]$ requires the ordered solution of two difference equations

$$v[n] = \sum_{k=0}^{M} b_k x[n-k]$$
$$y[n] = v[n] - \sum_{k=1}^{N} a_k y[n-k] \tag{5}$$

## Direct Form II

A more efficient direct form structure can be realized by placing the feedback section first, followed by the feedforward section. The first step in this rearrangement is that of Figure 3.

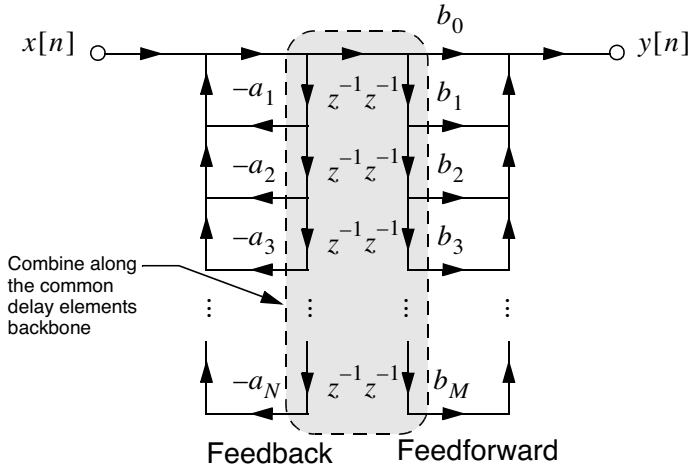



**Figure 3:** Rearrange the direct form I structure to place the feedback terms first.

The final direct form II structure is shown below in Figure 4.

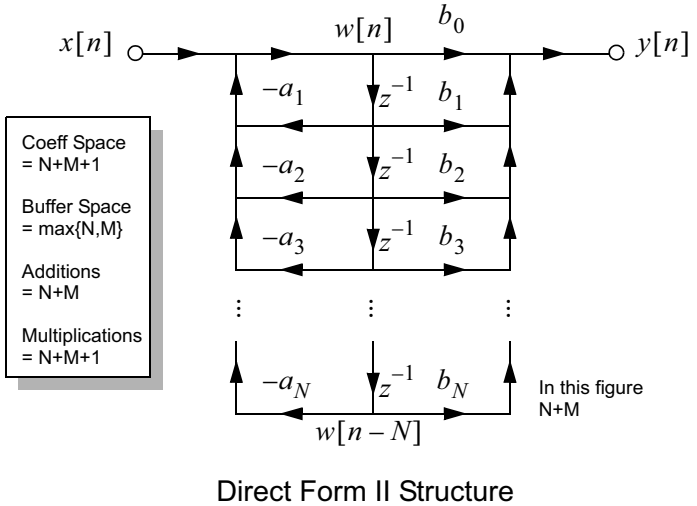



**Figure 4:** Direct form II structure.

The ordered pair of difference equations needed to obtain $y[n]$ from $x[n]$ is

$$\begin{aligned} w[n] &= x[n] - \sum_{k=1}^{N} a_k w[n-k] \\ y[n] &= \sum_{k=0}^{M} b_k w[n-k] \end{aligned} \tag{6}$$

## Cascade Form

Since the system function, $H(z)$, is a ratio of polynomials, it is possible to factor the numerator and denominator polynomials in a variety of ways. The most popular factoring scheme is as a product of second-order polynomials, which at the very least insures that conjugate pole and zeros pairs can be realized with real coefficients

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} = \prod_{k=1}^{N_s} H_k(z), \tag{7}$$

where $N_s = \lfloor (N+1)/2 \rfloor$ is the largest integer in $(N+1)/2$. A product of system functions corresponds to a cascade of *biquad* system blocks is shown in Figure 5. The *k*th biquad can be imple-

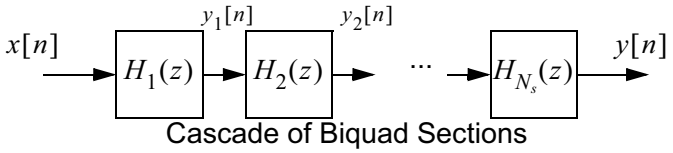



**Figure 5:** Cascade of biquadratic sections to implement a $N > 2$ IIR filter.

mented using a direct form structure (typically direct form II), as shown in Figure 6. The

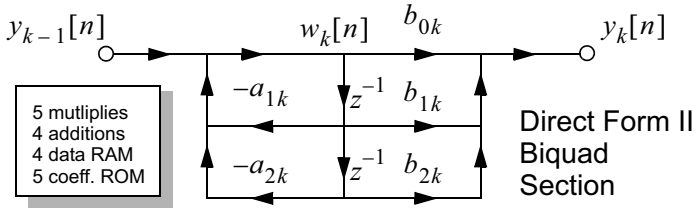



**Figure 6:** Detailed view of the direct form II biquadratic section.

corresponding biquad difference equations are

$$w_k[n] = y_{k-1}[n] - a_{1k}w_k[n-1] - a_{2k}w_k[n-2] \tag{8}$$

$$y_k[n] = b_{0k}w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2]. \tag{9}$$

The cascade of biquads is very popular in real-time DSP, is supported by scipy.signal via the `sos` which takes the form of 2D `ndarray` (a matrix in numpy). The `sos` 2D array will converted to a C header file and ultimately placed in an FM4 project. The organization of `sos` follows from (7)

```
sos =  [[b01, b11, b21, a01, a11, a21],
        [b02, b12, b22, a02, a12, a22],
         ...
        [b0k, b1k, b2k, a0k, a1k, a2k],
        ...
        [b0Ns, b1Ns, b2Ns, a0Ns, a1Ns, a2Ns]]
```

# Filter s-Domain to z-Domain Transformation

## Impulse Invariant Method

A simple and natural way to obtain a discrete-time implementation of an analog filter is to sample the impulse response $h_c(t)$, i.e., let

$$h[n] = h_c(nT). \tag{10}$$

In the frequency domain the analog frequency response, $H_a(j\Omega) = H_a(\Omega) = H_c(j\Omega)$, becomes the discrete-time frequency response, $H(e^{j\omega})$, via the ideal sampling theory result

$$H(e^{j\omega}) = \sum_{k=-\infty}^{\infty} H_a\left(\frac{\omega}{T} + \frac{2\pi k}{T}\right). \tag{11}$$

From Figure 7 we see that if $H_a(\Omega)$ is not bandlimited, aliasing shows up in $H(e^{j\omega})$

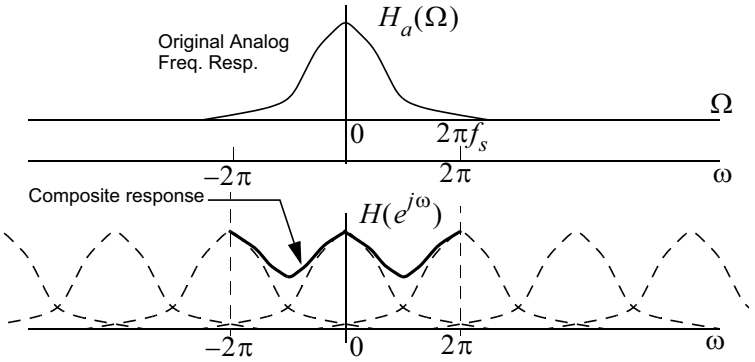



**Figure 7:** Frequency response of an impulse invariant filter is likely to have some aliasing when compared to the analog prototype.

Applying this to the RC lowpass filer example introduced earlier, we have

$$\begin{aligned} h[n] &= Th_c(nT) \\ &= \frac{T}{RC}e^{-nT/(RC)}u[n] \\ &= \frac{T}{RC}(e^{-T/(RC)})^n u[n] \end{aligned} \tag{12}$$

In difference equation form we now have

$$y[n] = e^{-\frac{T}{RC}}y[n-1] + \frac{T}{RC}x[n] \tag{13}$$

and the pole-zero plot of Figure 8.

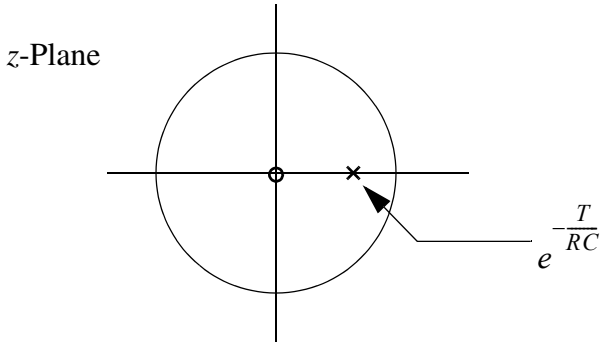



**Figure 8:** Pole-zero plot of RC low-pass filter following impulse invariant transformation.

In the $z$-domain the filter takes the form

$$H(z) = \frac{T/(RC)}{1 - e^{-T/(RC)}z^{-1}}, \text{ ROC: } |z| > e^{-T/(RC)}. \tag{14}$$

## Bilinear Transformation Method

A drawback of the impulse invariant approach is that aliasing of the analog filter's frequency response can occur unless certain filter roll-off conditions are met. Basically, any portion of the analog filter frequency response that extends above the folding frequency, $f_s/2$, folds into the principle alias band that runs over $[0, f_s/2]$. The bilinear transform approach avoids through a frequency warping transformation which makes the analog frequency axis $0 \le f < \infty$ is first mapped to the frequency interval $0 \le f < f_s/2$, before being converted to the discrete-time domain.

It turns out that the impulse invariant technique used the many-to-one mapping

$$z = e^{sT}. \tag{15}$$

To correct the aliasing problem we first employ a one-to-one mapping,

$$s' = \frac{2}{T}\tanh^{-1}\left(\frac{sT}{2}\right), \tag{16}$$

which compresses the entire *s*-plane into a strip as shown in Figure 9.

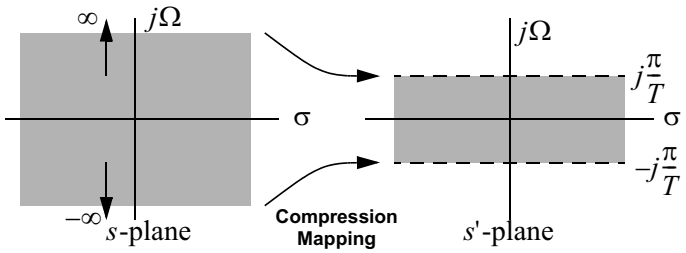



**Figure 9:** Remapping the s-plane to avoid aliasing.

Following the compression mapping we convert to the *z*-plane as before, except this time there is nothing that can alias. The complete mapping from *s* to *z* is

$$s = \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) \tag{17}$$

or in reverse

$$z = \frac{1+\frac{T}{2}s}{1-\frac{T}{2}s} \tag{18}$$

The frequency axis mapping is of the form

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right) \tag{19}$$

or

$$\omega = 2\tan^{-1}(\Omega T/2) \tag{20}$$

The basic filter design equation is

$$H(z) = H_a(s)\Big|_{s=\frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)} \tag{21}$$

In a practical design in order to preserve desired discrete-time critical frequencies, such as the passband and stopband cutoff frequencies, we use frequency *prewarping*

$$\Omega_i = \frac{2}{T}\tan\left(\frac{\omega_i}{2}\right) \tag{22}$$

where $\omega_i$ is a discrete-time critical frequency that must be used in the design of an analog prototype with corresponding continuous-time critical frequency $\Omega_i$. The frequency axis compression imposed by the bilinear transformation can make the transition ratio in the discrete-time domain smaller than in the continuous-time domain, thus resulting in a lower order analog prototype than if the design was implemented purely as an analog filter. Given a ratio of polynomials in the *s*-domain, or an amplitude response specification, we can proceed to find $H(z)$. The resulting trans-

formation is know as the bilinear transform and takes the form

$$s = \frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}. \tag{23}$$

For a given s-domain filter prototype, $H_c(s)$, the transformation produces $z$-domain system function of the form

$$H(z) = H_c\left(\frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}\right). \tag{24}$$

In the case of the RC lowpass filter example, we have

$$\begin{aligned} H(z) &= \frac{1}{1 + \frac{2RC}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right)} = \frac{T}{2RC} \cdot \frac{1 + z^{-1}}{1 + \frac{T}{2RC} - z^{-1}} \\ &= \frac{T/(2RC)}{1 + T/(2RC)} \cdot \frac{1 + z^{-1}}{1 - \left(1 + \frac{T}{2RC}\right)z^{-1}} \end{aligned} \tag{25}$$

The difference equation is

$$y[n] = \frac{2RC}{2RC + T} y[n-1] + \frac{T}{2RC + T}\{x[n] + x[n-1]\} \tag{26}$$

The pole-zero plot is shown in Figure 10.

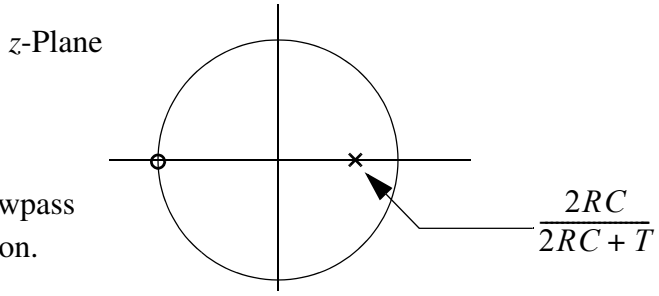



**Figure 10:** Pole-zero plot of RC lowpass filter following bilinear transformation.

# Bilinear Filter Design in Python

The `scipy.signal` package fully supports the design of IIR digital filters from analog prototypes. IIR filters like FIR filters, are typically designed with amplitude response requirements in mind. A collection of design functions are available, including `signal.iirdesign()`. To make the design of lowpass, highpass, bandpass, and bandstop filters consistent with the module `fir_design_helper.py` the module `iir_design_helper.py` was written to support this lab. Examples of general lowpass, highpass, bandpass, and bandstop amplitude design requirements are shown in Figure 11.

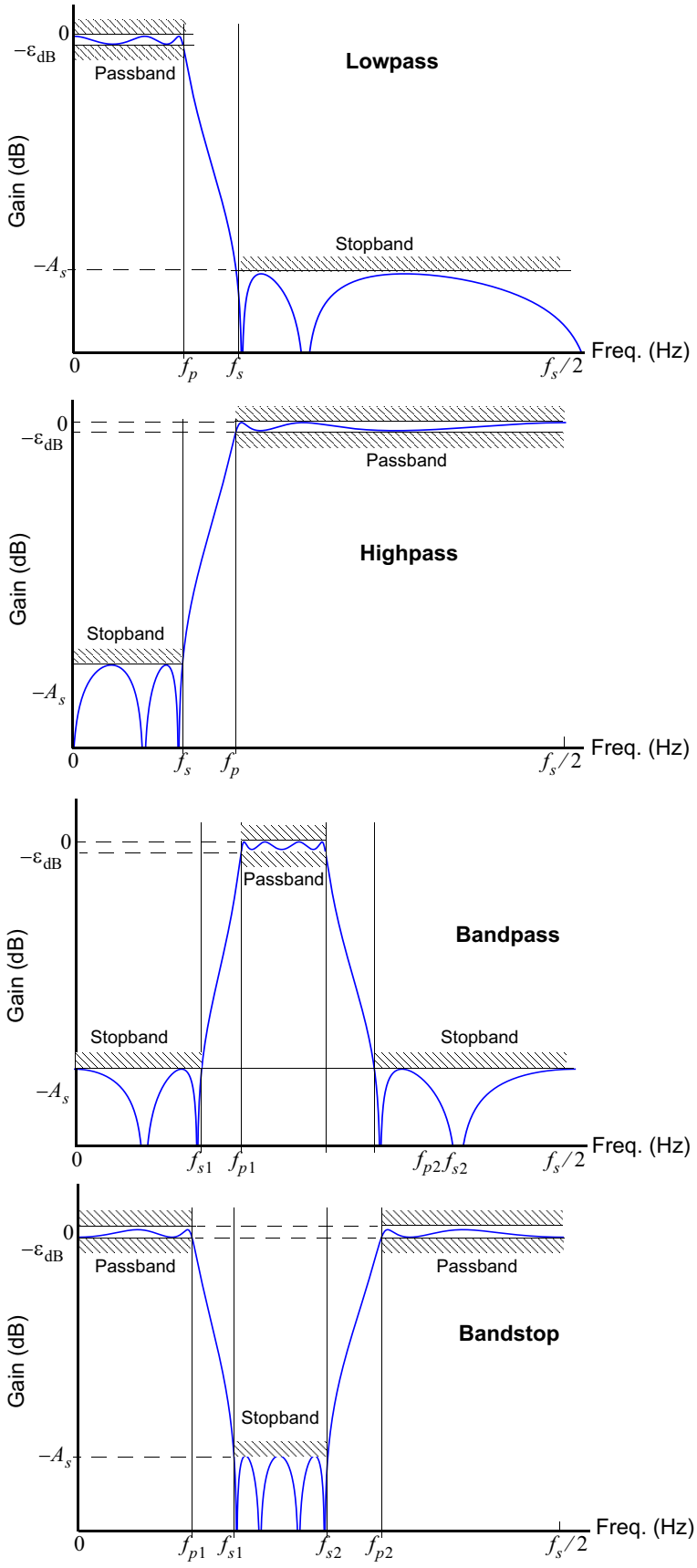



**Figure 11:** General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop IIR filter types.

Within `iir_design_helper.py` there are four filter design functions and a collection of supporting functions available. The four functions are used for designing lowpass, highpass, bandpass, and bandstop filters according to Butterworth, Chebshev type 1, Chebyshev type 2, elliptical and Bessel analog filter prototypes [2] are described in Table 1. The filter functions output the

**Table 1:** IIR filter design functions in `iir_design_helper.py` and key support functions.

| Type | IIR Filter Design Functions* |
|---|---|
| Transfer Function (b,a) and SOS | |
| Lowpass (bilinear) | `b, a, sos = IIR_lpf(f_pass, f_stop, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')`<br>`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel` |
| Highpass (bilinear) | `b, a, sos = IIR_hpf(f_stop, f_pass, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')`<br>`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel` |
| Bandpass (bilinear) | `b, a, sos = IIR_bpf(f_stop1, f_pass1, f_pass2, f_stop2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')`<br>`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel` |
| Bandstop (bilinear) | `b, a, sos = IIR_bsf(f_pass1, f_stop1, f_stop2, f_pass2, Ripple_pass, Atten_stop, fs = 1.00, ftype = 'butter')`<br>`ftype may be 'butter', 'cheby1', 'cheby2', 'elliptic' or bessel` |
| Support Functions | |
| SOS list plot | `freqz_resp_cas_list(sos,mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4))` |
| SOS `freqz` | `w, Hcas = freqz_cas(sos,w)` |
| SOS plot pole-zero | `sos_zplane(sos,auto_scale=True,size=2,tol = 0.001)`<br>More accurate root factoring results in more accurate pole-zero plot. |
| Cascade SOS | `sos = sos_cascade(sos1,sos2)` |

*These functions wrap `scipy.signal.iirdesign()` to provide an interface more consistent with the FIR design functions found in the module `fir_design_helper.py`. The function `unique_cpx_roots()` is used to mark repeated poles and zeros in `sos_zplane`. Note: All critical frequencies given in increasing order.

design in two formats:

1. Traditional transfer function form using numerator (`b`) and denominator (`a`) coefficients arrays, and
2. Cascade of biquadratic sections form using the previously introduced `sos` 2D array or matrix.

Both are provided to allow further analysis with either a *direct form* topology or the `sos` form. The underlying `signal.iirdesign()` function also provides a third option: a list of pole-zeros. IN the real-time implementation yo will be making exclusive use of the `sos` form.

Of the remaining support functions four are also described in Table 1. The most significant functions are `freqz_resp_cas_list`, available for graphically comparing the frequency response over several designs, and a function for plotting the pole-zero pattern (`sos_zplane`), both operate directly from the `sos` matrix. A transfer function form for frequency response plotting, `freqz_resp_list`, is also present in the `iir_design_helper.py` module. Thi function was first introduced on Lab 4. The frequency response function plotting offers modes for gain in dB, phase in radians, group delay in samples, and group delay in seconds, all for a given sampling rate, $f_s$ in Hz. The pole-zero plotting function locates pole and zeros more accurately than `ssd.zplane`, as the numpy function `roots()` is only solving quadratic polynomials. Also, repeated roots can be displayed as theoretically expected, and also so noted in the graphical display by superscripts next to the pole and zero markers.

*Lowpass Design Example and Comparison*

Consider the design of a lowpass filter having:

- $f_{\text{pass}} = 5\,\text{kHz}$
- $f_{\text{stop}} = 8\,\text{kHz}$
- $\varepsilon_{\text{dB}} = 0.5\,\text{dB}$
- $A_s = 60\,\text{dB}$

```
fs = 48000
f_pass = 5000
f_stop = 8000
b_but,a_but,sos_but = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'butter')
b_cheb1,a_cheb1,sos_cheb1 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby1')
b_cheb2,a_cheb2,sos_cheb2 = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'cheby2')
b_elli,a_elli,sos_elli = iir_d.IIR_lpf(f_pass,f_stop,0.5,60,fs,'ellip')
```

```
IIR butter order = 15.
IIR cheby1 order = 8.
IIR cheby2 order = 8.
IIR ellip order = 6.
```

```
iir_d.freqz_resp_cas_list([sos_but,sos_cheb1,sos_cheb2,sos_elli],'dB',fs=48)
ylim([-80,5])
title(r'IIR Lowpass Compare')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Butter order: %d' % (len(a_lpf_but)-1),
        r'Cheby1 order: %d' % (len(a_lpf_cheb1)-1),
        r'Cheby2 order: %d' % (len(a_lpf_cheb2)-1),
        r'Elliptic order: %d' % (len(a_lpf_elli)-1)),loc='best')
grid();
```

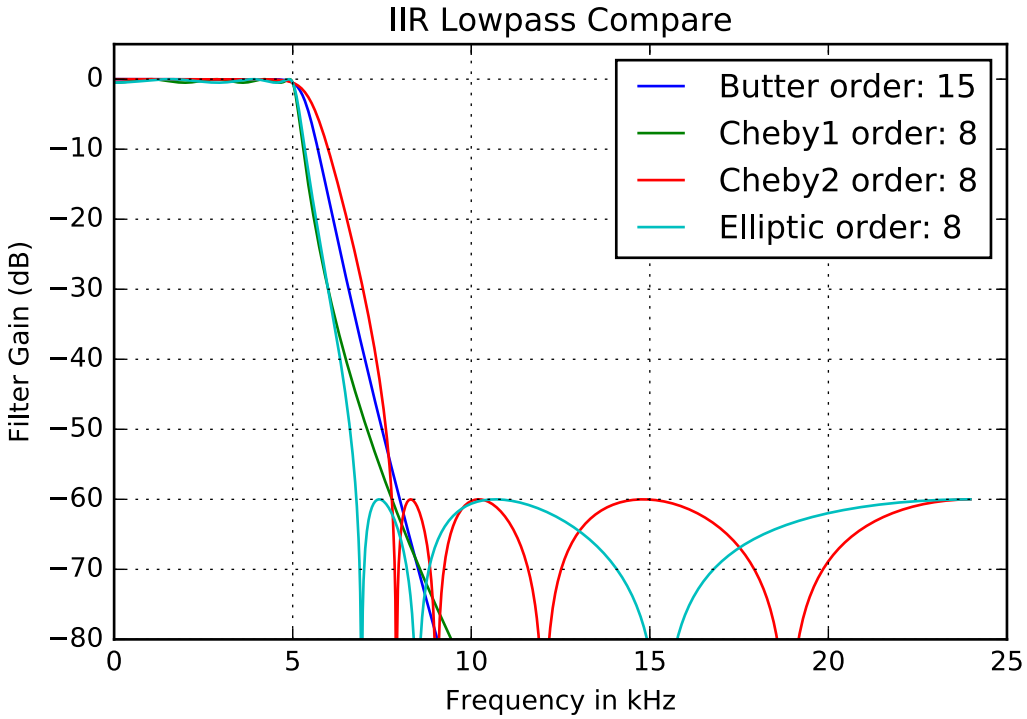



**Figure 12:** Lowpass comparison plot obtained using `freqz_resp_cas_list`.

Note the same plot could be obtain using the transfer function form via `freqz_resp_list()`, as double precision coefficients are being used. For the 15th-order Butterworth the bilinear transformation maps the expected 15 zeros at infinity to $\omega = \pi$, or $f = f_s/2$. If you use just ssd.zplane()

```
iir_d.sos_zplane(sos_but)
```

```
(15, 15)
```

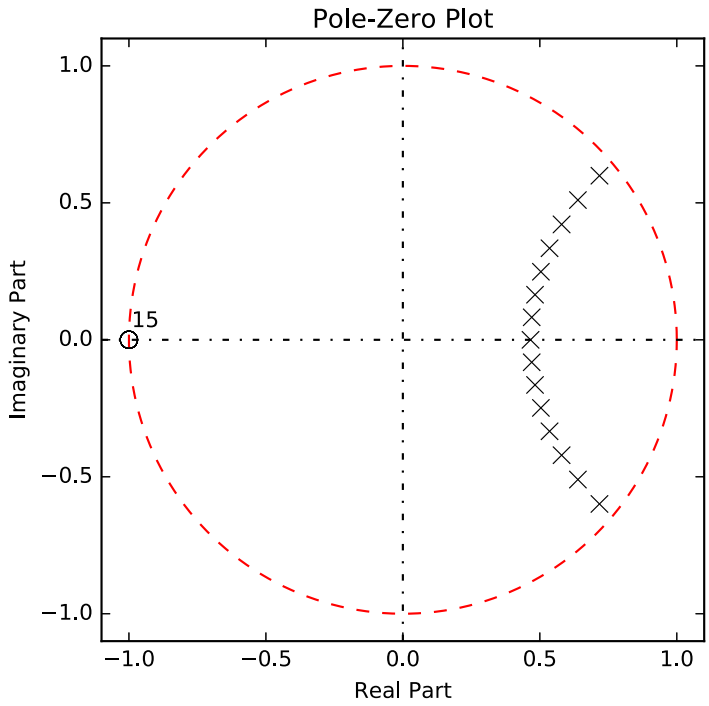



**Figure 13:** Pole-zero plot of the 15th-order Butterworth design.

you will find that the 15 poles at $z = -1$ are in a tight circle around $z = -1$, indicating polynomial rooting errors. The `sos` matrix resolves this as the underlying `signal.iirdesign()` function works with poles and zeros from the start.

# Writing C Coefficient Files

The final step in getting your filter design to run on the FM4 is to load the `sos` filter coefficients into C code. It is convenient to store the filter coefficients in a C header file and just `#include` them in code. The Python module `coeff2header.py` takes care of this for IIR filters implemented as a cascade of second-order sections using `float32_t`. In the sample Jupyter notebook this is done for a lowpass design.

*Writing a Coefficient Header File*

```
# Write a C header file
c2h.IIR_sos_header('IIR_elliptic_lpf_5_8.h',sos_elli)
```

- Note: In the first cell of the notebook you find

  ```
  import coeff2header as c2h
  ```

- A complete FM4 design example using this filter is found in the `src` folder when using the main module `fm4_IIR_intr_GUI.c`

The resulting header file for a sixth-order design is three `sos` stages as given below:

```
//define a IIR SOS CMSIS-DSP coefficient array

#include <stdint.h>

#ifndef STAGES
#define STAGES 3
#endif
/*************************************************************/
/*                     IIR SOS Filter Coefficients           */
float32_t ba_coeff[15] = { //b0,b1,b2,a1,a2,... by stage
    +3.405534e-03, +2.879634e-03, +3.405534e-03,
    +1.555966e+00, -6.373943e-01,
    +1.000000e+00, -8.884023e-01, +1.000000e+00,
    +1.525341e+00, -7.895790e-01,
    +1.000000e+00, -1.232239e+00, +1.000000e+00,
    +1.530379e+00, -9.375340e-01
};
/*************************************************************/
```

To utilize this 1-D array of filter coefficients you need a corresponding filter algorithm. Equations (8) and (9) implement the second-order section using five coefficients per section.

```
ba_coeff[] = {b01, b11, b21, a11, a21, // 1st section in linear array
              b02, b12, b22, a12, a22, // 2nd section (use index stride 5*1
              b03, b13, b23, a13, a23, // 3nd section (use index stride 5*2
              ... };
```

The actual C-code description is described in the next section.

## Cascade of Biquad Sections (SOS) C Code Implementation

Similar to Lab 4 where FIR C code implementations were studied, in this lab you can write your own code or choose between IIR_filters.c/IIR_filters.h or use CMSIS-DSP. In the lab exercises that follow near the end of this document, you will have a chance to try all three.

*Using a Portable Filter Module*

The code module `IIR_filters.c/IIR_filters.h`, found in the `src` folder for Lab 5, implements a floating-point `sos` algorithm using portable ANSI C. The functions allow for sample-by-sample processing as well as *frame-based* processing, where a block of length `Nframe` samples are processed in one function call. The data structure shown below is used to organize the filter details:

```
struct IIR_struct_float32
{
   int16_t N_stages;    //number of biquad stages = ceil(N_order/2)
   float32_t *state;    //two per stage w1,w2,... so 2*N_stages total
   float32_t *ba_coeff; //five coefficients per stage b0,b1,b2,a1,a2,...
                        //so 5*N_stages total
};
```

Pointers are used to manage all of the arrays, and ultimately the data structure itself, to insure that function calls are fast and efficient. Recall in particular that in C an array name is actually the address to the first element of the array. This property is used by the functions `IIR_sos_init_float32()` and `IIR_sos_filt_float32()` which interact with the IIR filter data structure to initialize and then filter signal samples, respectively. The four steps to IIR `sos` filtering using this module are:

1. Create an instance of the data structure:
   ```
   struct IIR_struct_float32 IIR1;
   ```
   where now `IIR1` is essentially a filter object to be manipulated in code.

2. Have on hand two `float32_t` arrays. One of length `2*N_STAGES` to hold the filter states (two states per stage). The second of length `5*N_STAGES` to hold the filter coefficients in an array (five coefficients per stage: three feed forward and two feedback).

3. Initialize `IIR1` in `main()` using:
   ```
   IIR_sos_init_float32(&IIR1, STAGES, ba_coeff, IIRstate);
   ```
   where here `IIRstate` is the address to a `float32_t` array of length `2*N_STAGES` elements used to hold the filter states and `ba_coeff` is the address to a `float32_t` array holding `5*N_STAGES` filter coefficients. The filter state array should be declared as a global. Typically `ba_coeff` will be filled using a header file, e.g.,
   ```
   #include "test_cheby1.h" // 4 stage sos
   ```
   is a 4-stage (IIR lowpass filter with $f_{pass} = 5$ kHz and $f_{stop} = 8$ kHz) generated in a Jupyter notebook. Exceptions to this are when you want a custom `sos` array where perhaps some of the coefficients are under the control of the GUI slider.

4. With the structure initialized, we can now filter signal samples in the ISR using
   ```
   IIR_sos_filt_float32(&IIR1, &x, &y, 1);
   ```

where `x` is the input sample and `y` is the output sample. Notice again passing by address in the event a frame of data is being filtered. The final argument of `1` is the frame length, which for sample-by-sample processing is just one. By sample-by-sample I mean that each time the ISR runs a new sample has arrived at the ADC and a new filtered sample must be returned to the DAC.

The code behind `IIR_sos_filt_float32()`, inside `IIR_filters.c`, is:

```
//Process each sample of the frame with this loop
for (iframe = 0; iframe < Nframe; iframe++)
{
  input = x_in[iframe];
  //Biquad section filtering stage-by-stage using coefficients
  //and biquad states each stored in 1D arrays.
  //A float accumulator is used.
  wn = 0; // Clear the accumulator
  for (iIIR = 0; iIIR < IIR->N_stages; iIIR++)
  {
    icstride = 5*iIIR; //filter coefficient stride into linear array
    isstride = 2*iIIR; //filter state stride into linear array
    //biquad 2nd-order LCCDE code from Lab 5 eqn 8 & 9
     //Note: The sign of the a[1] and a[2] coefficients is flipped to match
     //the format used by CMSIS-DSP. Thus there is a sign change in the
     //difference equation below for ba_coeff[icstride+3] & ba_coeff[icstride+4]
    wn = input
       + IIR->ba_coeff[icstride+3] * IIR->state[isstride]
       + IIR->ba_coeff[icstride+4] * IIR->state[isstride+1];
    result = IIR->ba_coeff[icstride]*wn
           + IIR->ba_coeff[icstride+1]*IIR->state[isstride]
           + IIR->ba_coeff[icstride+2]*IIR->state[isstride+1];
    //Update filter state for stage i
    IIR->state[isstride+1] = IIR->state[isstride];
    IIR->state[isstride] = wn;
    input = result; //the output is the next stage input
  }
  x_out[iframe] = result;
}
```

All working variables are `float32_t`. The outer `for` loop processes each sample within the frame. The inner `for` loop implements equations (8) and (9) on each pass, which are the feedback and feed forward difference equations associated with each bi-quadratic section, respectively. This is the cascade of biquads in a serial pipeline. The array `IIR->state[]` holds $w_k[n-1]$ and $w_k[n-2]$ for $k = 1$ to $k = N_{\text{stages}}$, thus as a linear array containing two states per section. Note the working variable `isstride` manages access to the state variables. The working variable `icstride` manage access to the filter coefficients, which are stored in the linear array `ba_coeff[]`. The five elements per stride are as stated earlier `b0k, b1k, b2k, a1k, a2k`. The filter states are updated in two lines of code, the first updates the oldest value, per stride, and the second line the second oldest value. The variable `result` holds the per state output, which at the end of the inner for loop is finally loaded into the output frame buffer, `x_out[]`.

A complete IIR filter example, `fm4_IIR_intr_GUI.c`, with the GUI configured can be found in the `src` folder of the Lab5 Keil project. This project is configured to load a 101 tap bandpass filter.

*ARM CMSIS-DSP IIR Algorithms*

In Lab 4 you were introduced to the CMSIS-DSP library for FIR filtering. This library also supports a variety of IIR filter topologies. One is the cascade of Direct Form II transposed structure as highlighted in the screen capture of Figure 14. The functions `arm_biquad_cascade_df2T_init_f32()` and `arm_biquad_cascade_df2T_f32()` perform functions similar to `IIR_sos_init_float32()` and `IIR_sos_filt_float32()` respectively, of the previous subsection. You will learn in testing however that the CMSIS-DSP IIR `sos` function executes faster. The latter however are portable and the code easy to follow, as described in the previous section. In contrast to the FIR functions of Lab 4, the IIR function calls are identical. The four steps to IIR filtering using the CMSIS-DSP library are:


**Figure 14:** CMSIS-DSP IIR cascade of second-order sections `float32_t` filter detail.

1. Create an instance of the data structure:

   ```
   arm_biquad_cascade_df2T_instance_f32 IIR1;
   ```

   where once again `IIR1` is essentially a filter object to be manipulated in code.

2. Have on hand two `float32_t` arrays. One of length `2*N_STAGES` to hold the filter states (two states per stage). The second of length `5*N_STAGES` to hold the filter coefficients in an array (five coefficients per stage: three feed forward and two feedback).

3. Initialize `IIR1` in `main()` using:

   ```
   arm_biquad_cascade_df2T_init_f32(&IIR1, STAGES, ba_coeff, IIRstate);
   ```

   where again `IIRstate` is the address to a `float32_t` array of length `2*N_STAGES` elements used to hold the filter states and `ba_coeff` is the address to a `float32_t` array holding `5*N_STAGES` filter coefficients.

4. With the structure initialized, we can now filter signal samples in the ISR using

```
arm_biquad_cascade_df2T_f32(&IIR1,&x,&y,1);
```

where as before `x` is the input sample and `y` is the output sample. Notice again passing by address in the event a frame of data is being filtered. The final argument of `1` is the frame length, which for sample-by-sample processing is just one.

A complete IIR filter example, `fm4_IIR_intr_GUI.c`, with the GUI configured can be found in the `src` folder of the Lab4 Keil project. The ARM code is commented out next to the corresponding `IIR_filters.c` module code. To use the ARM code simply comment out the `IIR_filters.c` statements and uncomment the ARM code.

*Elliptic Design and Test*

Earlier a design example for an IIR with $f_{\text{pass}} = 5\,\text{kHz}$ and $f_{\text{stop}} = 8\,\text{kHz}$ was presented. The `sos` coefficient are written to `IIR_cheby1_lpf_5_8.h` in the included "`Jupyter notebook IIR Filter Design and C Headers.ipynb`". The sample file `fm4_IIR_intr_GUI.c` included in the Lab5 ZIP is configured to import the elliptic coefficient header. The network analyzer capability of the Analog Discovery is used to collect data to CSV file and used back in the Jupyter notebook for comparison. The results this comparison is shown in Figure 15. The measurements compare favorably.

```
f_AD,Mag_AD, Phase_AD = loadtxt('IIR_elliptic_5_8_48k.csv',
                                delimiter=',',skiprows=6,unpack=True)
```

```
iir_d.freqz_resp_cas_list([sos_elli],'dB',fs=48)
ylim([-80,5])
plot(f_AD/1e3,Mag_AD)
title(r'Third-Order Elliptic Lowpass: $f_{pass}=5$kHz and $f_{stop}=8$kHz')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Theory',r'AD Measured'),loc='upper right')
grid();
```

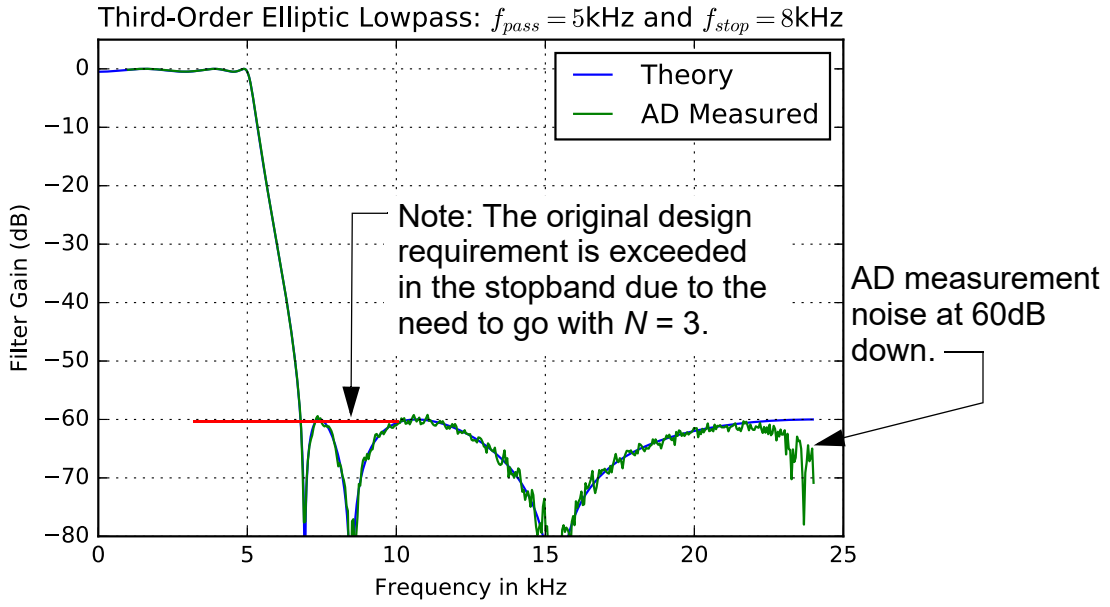



**Figure 15:** Theory versus measured for the elliptic design example.

# Expectations

When completed, submit a lab report which documents code you have written and a summary of your results. Screen shots from the scope and any other instruments and software tools should be included as well. I expect lab demos of certain experiments to confirm that you are obtaining the expected results and knowledge of the tools and instruments.

# Problems

1. In this first problem you will implement a specific IIR design to meet certain amplitude response requirements. The filter topology will be a cascade of second-order sections that follows from the example given earlier in the this lab document. Test your design using the network analyzer *and* the white noise/PC sound card test approach. Here you will use the function `sos_C_header()` to create a cascade of biquad sections C header file and then use the program `ISRs_sos_iir_float.c` (both in the Lab 5 ZIP package) to implement the filter in real-time. Design your filter to meet specific amplitude response requirements given in Figure 16, using an *elliptic lowpass filter prototype*.

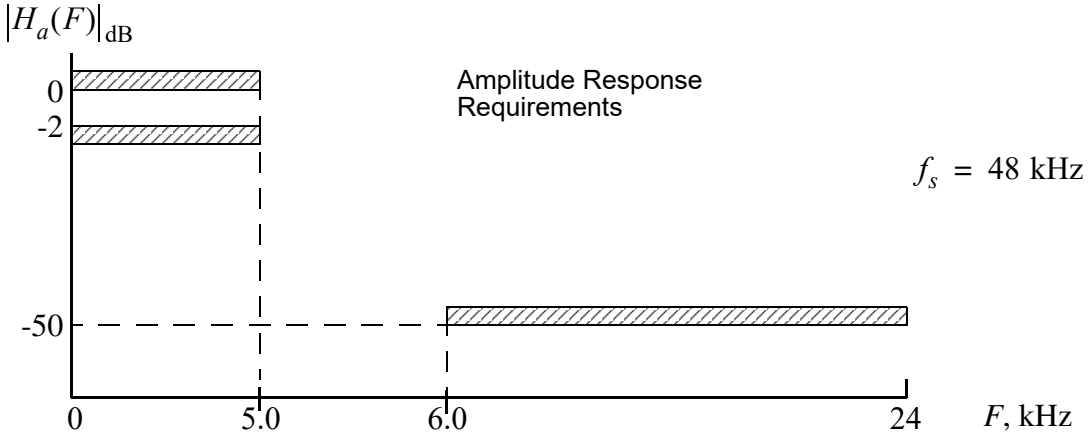



**Figure 16:** Lowpass filter design requirements.

a) Design the filter using the Jupyter notebook, "IIR Filter Design and C Headers.ipynb", found in the Python folder of the Keil project ZIP for Lab 5. Consider Butterworth, Chebyshev type 1 and type 2, and elliptic analog prototypes. Comment the resulting filter orders required. Ultimately carry the elliptic design forward in the remainder of this problem.

b) Compare the theoretical magnitude response in dB versus frequency in Hz using `freqz_resp_cas_list()`. By compare it means all four designs. This means in $H(e^{j\omega})$ you make the substitution $\omega = 2\pi(f/f_s)$. Overlay the amplitude response design requirements.

c) The lowest order filter is not always the best choice. Along with the amplitude response a filter also has a phase response and *group delay*, $T_g(e^{j\omega})$ ,response, where

$$T_g(e^{j\omega}) = -\frac{d}{d\omega}\angle H(e^{j\omega}) \text{ (samples)} \tag{27}$$

Group delay characterizes the time delay that a cluster or group of frequencies near some $\omega$ value receives as they pass through the filter. Constant group delay is desirable, as it means all spectral components of a signal passing through the filter receive the same time delay, and hence there is no signal *dispersion*. The filtering of digital communications signals is particularly adverse to non-constant group delay filtering. In high quality audio can change the timbre of a musical instrument as the phase shift of overtones will no longer be natural. Of course since a filter will pass some signals and block others, the idea of constant group delay typically only applies to the filter passband. The FIR filters of Lab 4 were constrained to be linear phase and hence constant group delay over all frequencies. The functions `freqz_resp_list()` and `freqz_resp_cas_list()` both have an option for plotting group delay in either samples or in seconds relative to the specified sampling frequency in Hz. For the frequency response plots of part (b) plot the corresponding group delay in samples and ms. Hint: For a ms scaled time axis enter $f_s$ in kHz. Comment on the relative flatness of the group delay for the four filters. Order the responses from best to worse. Any surprises?

d) Plot the pole-zero pattern for the elliptic design and comment on how the geometry of the poles and zeros define the passband and the stopband locations.

e) Obtain the frequency response magnitude in dB from the network analyzer. Compare critical frequencies of the measured response to the theoretical response in the Jupyter notebook.

f) Obtain the frequency response magnitude in dB using the white noise approach described in Lab 4. By normalizing the passband gain to 0 dB you should be able to overlay the theoretical response directly in the Jupyter notebook.

2. Repeat Problem 1 for the ultimate design of a *type II chebyshev bandpass filter* having amplitude response requirements as shown in Figure 17. Repeat parts (a) and (b) and either

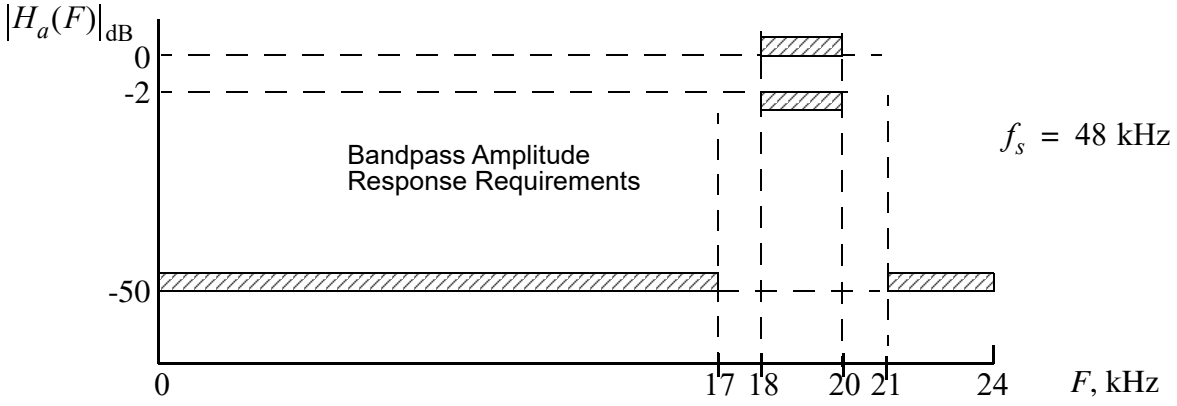



**Figure 17:** Bandpass filter design requirements.

part (c) *or* (d). A comparison in the Jupyter notebook is expected whether you use (c) or (d).

3. In this problem you will design and implement a second-order *tunable* frequency bandpass filter. The filter has system function

$$H_{\text{BP}}(z) = \frac{1-\alpha}{2} \cdot \frac{1-z^{-2}}{1-\beta(1+\alpha)z^{-1}+\alpha z^{-2}} \tag{28}$$

where

$$\begin{aligned} \beta &= \cos(\omega_0) = \cos\left(2\pi\frac{f_0}{f_s}\right) \\ \alpha &= \frac{1-\sin\left(2\pi\frac{\Delta f}{f_s}\right)}{\cos\left(2\pi\frac{\Delta f}{f_s}\right)}, \end{aligned} \tag{29}$$

with $f_0$ the center frequency, $\Delta f$ the 3 dB bandwidth, and $f_s$ the sampling rate in Hz.

a) Show that under the constraint $a_0 = 1$ you can write

$$H_{BP}(z) = \frac{b_0 + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{30}$$

with

$$\begin{aligned} b_0 &= \frac{1-\alpha}{2}, \qquad b_1 = 0, \qquad b_2 = -\frac{1-\alpha}{2} = -b_0 \\ a_0 &= 1, \qquad a_1 = -\beta(1+\alpha), \qquad a_2 = \alpha \end{aligned} \tag{31}$$

b) Verify using Python that the theoretical frequency response magnitude is as shown in Figure 18. Also verify that the pole-zero plot of the second-order filter, for the case $f_0 = 4\,\text{kHz}$, $\Delta f = 1\,\text{kHz}$ and $f_s = 32\,\text{kHz}$, is as shown in Figure 19.

```
b1, a1, sos1 = tunable_bpf(4000,1000,48000)
b2, a2, sos2 = tunable_bpf(12000,1000,48000)
b3, a3, sos3 = tunable_bpf(12000,2000,48000)
iir_d.freqz_resp_cas_list([sos1,sos2,sos3],'dB',fs=48)
ylim([-40,5])
title(r'Tunable IIR Bandpass Compare with $f_s = 48$ ksps')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'$f_0 = 4$k, $\Delta f = 1$k',
        r'$f_0 = 12$k, $\Delta f = 1$k',
        r'$f_0 = 12$k, $\Delta f = 2$k'),loc='best')
grid();
```

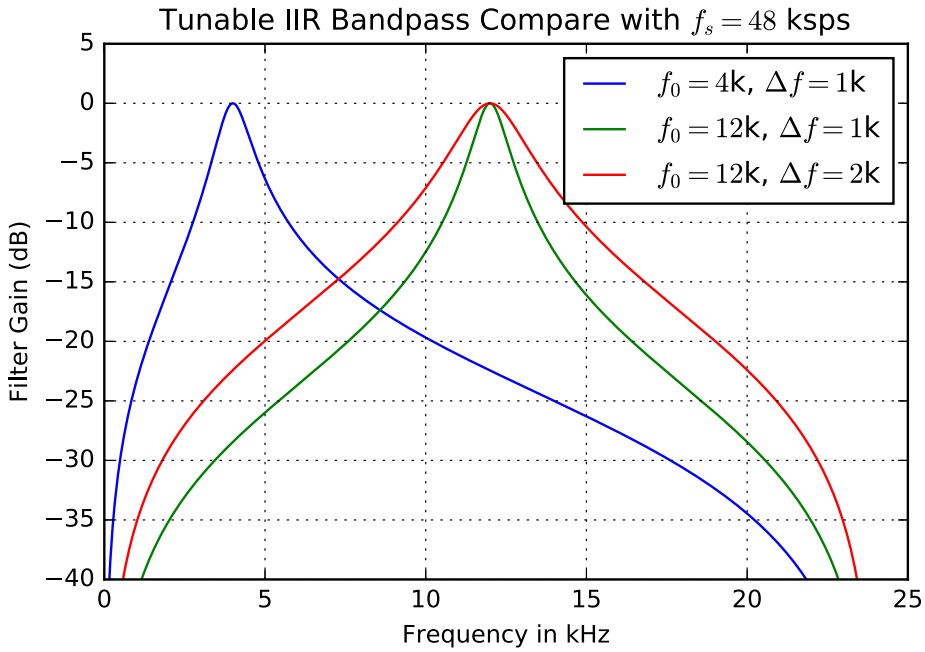



**Figure 18:** Second-order BPF frequency response magnitude in dB.

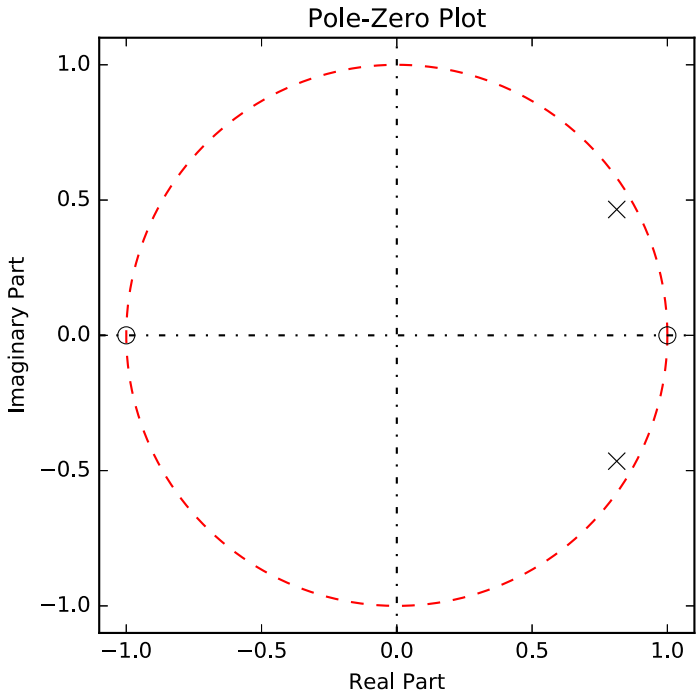



**Figure 19:** Second-order BPF pole-zero plot for $f_0/f_s = 0.125$ and $\Delta f/f_s = 1/32$ .

c) Implement the design on the FM4 board with $f_s$ = 48 kHz and with the filter tuning parameters $f_0$ and $\Delta f$ interfaced to GUI slider controls. Obtain the experimental frequency response in dB for one of the cases shown in Figure 18. Use the Jupyter notebook for your comparison (choose network analyzer or noise analysis).

The sin() and cos() functions are available for float arguments via the C math library, which is already included the Keil project by virtue of having CMSIS-DSP capability. The C math library caters to double precision math, which for the Cortex M4 you want to use single precision, thus the trig function calls are `sinf()` and `cosf()` respectively. In particular for evaluating $\cos(2\pi f_0/f_s)$ consider

```
//TWO_PI_OVER_FS = 2*pi/48000
#define TWO_PI_OVER_FS 0.0001308996938995747

...
// Inside ISR
beta = cosf(TWO_PI_OVER_FS*f0); // f0 in Hz
// etc
...
```

Alternatively CMSIS-DSP contains fast trig functions via:

```
float32_t arm_cos_f32 ( float32_t  x)
float32_t arm_sin_f32 ( float32_t  x)
```

The cascade of second-order sections is not strictly needed for this implementation, but will work with a single section. You can also just implement the difference equation directly as a direct from structure, i.e.,

$$\begin{aligned} y[n] &= b_0(x[n]-x[n-2]) - a_1 y[n-1] - a_2 y[n-2] \\ y &= b_0(x - x_{\text{old2}}) - a_1 y_{\text{old1}} - a_2 y_{\text{old2}} \end{aligned} \tag{32}$$

where you will need to maintain state variables, that is past values of the input and output, such as `xold1`, `xold2`, `yold1`, and `yold2`. They will need to be updated at the end of the filtering algorithm before leaving the ISR, i.e.,

```
// Inside the ISR with xold1, xold2, yold1, yold2 global
y = b0*(x - xold2) -a1*yold1 - a2*yold2;
xold2 = xold1;
xold1 = x;
yold2 = yold1;
yold1 = y;
```

A few comments/hints are in order as to where in code the filter coefficients should be updated. You do not want to burden the ISR lots of extra calculations. In this application you can likely get away with because the filter order is so low. Updating of the filter coefficients directly in the main `while` loop also is wasteful. A very efficient approach is to check to see when a GUI slider parameter has changed and just change to filter coefficients that need to be changed. A general structure for doing this is shown below:

```
while(1){
```

```
    // Update slider parameters
    update_slider_parameters(&FM4_GUI);
    if (FM4_GUI.idx_P_rec == 2) {
       //Do some calculations related to only parameter 2 changed
    }
    if (FM4_GUI.idx_P_rec == 3) {
       //Do some calculations related to only parameter 3 changed

    }
    if ((FM4_GUI.idx_P_rec == 2) || (FM4_GUI.idx_P_rec == 3)) {
       //Do some calculations related to parameter 2 or 3 changed
    }
}
```

Record the ISR time for your implementation. Expectations are 0.72 µ s. The frequency response should be similar to that of Figure 20.

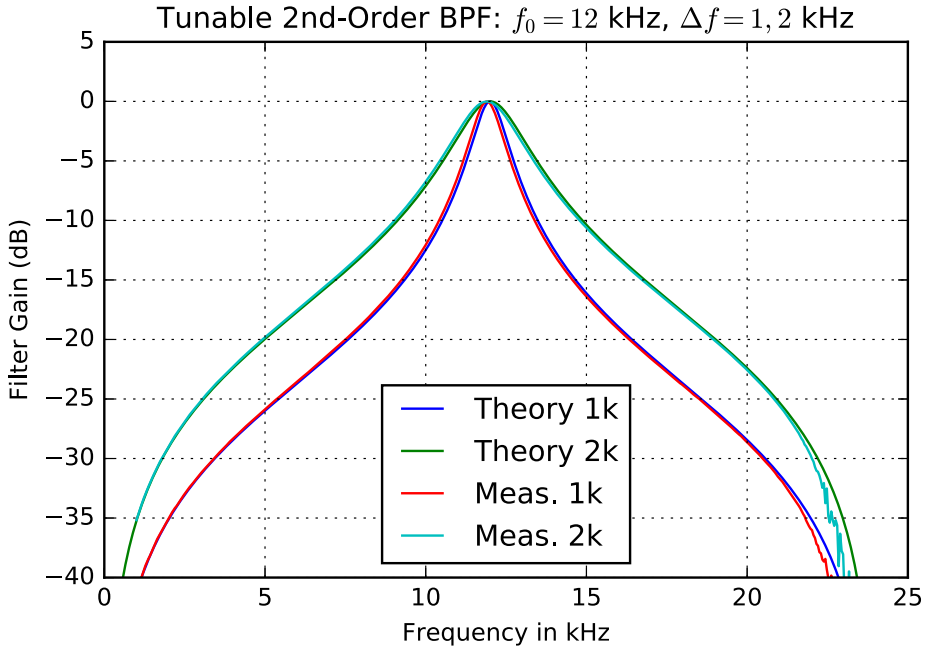



**Figure 20:** Expected theory versus measured for the tunable bandpass filter app.

Demo your working FM4 tunable bandpass filter to the lab instructor. You may want to play music through the filter as an application example.

# References

[1] Donald Reay, *Digital Signal Processing Using the ARM Cortex-M*4, Wiley, 2016

[2] Alan V. Oppenheim and Ronald W. Schafer, Discrete-Time Signal Processing, third edition, Prentice Hall, New Jersey, 2010.

[3] Joseph Yiu, T*he Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, third edition, Newnes, 2014.

# Code Listings

```
//IIR_filters.h
// IIR Filters header
// Mark Wickert April 2015

#define ARM_MATH_CM4
#include <s6e2cc.h>
#include "arm_math.h"

/*
Structures to hold IIR filter state information. Two direct form FIR
types are implemented at present: (1) float32_t and in the future (2)
int16_t. Each type requires both an initialization function and a
filtering function. The functions feature both sample-based and
frame-based capability.
*/

struct IIR_struct_float32
{
  int16_t N_stages;    //number of biquad stages = ceil(N_order/2)
  float32_t *state;    //two per stage w1,w2,... so 2*N_stages total
  float32_t *ba_coeff; //five per stage b0,b1,b2,a1,a2,...
                       //so 5*N_stages total
};


void IIR_sos_init_float32(struct IIR_struct_float32 *IIR,
                          int16_t N_stages,
                          float32_t *ba_coeff,
                          float32_t *state);
void IIR_sos_filt_float32(struct IIR_struct_float32 *IIR,
                          float32_t *x_in,
                          float32_t *x_out,
                          int16_t Nframe);
```

```
//IIR_filters.c
// IIR SOS Filter Implementation
// Mark Wickert April 2015

#include "IIR_filters.h"

void IIR_sos_init_float32(struct IIR_struct_float32 *IIR,
                          int16_t N_stages,
                          float32_t *ba_coeff,
                          float32_t *state) {
  // Load the filter coefficients and initialize the filter state.
  int16_t iIIR;
  IIR->N_stages = N_stages;
  IIR->ba_coeff = ba_coeff;
  IIR->state = state;
  for (iIIR = 0; iIIR < 2*IIR->N_stages; iIIR++)
  {
    IIR->state[iIIR] = 0;
  }
}

void IIR_sos_filt_float32(struct IIR_struct_float32 *IIR,
                          float32_t *x_in, float32_t *x_out,
```

```
                          int16_t Nframe) {
       int16_t iframe;
       int16_t iIIR;
       int16_t icstride;
       int16_t isstride;
       float32_t input;
       float32_t result;
       float32_t wn;

  //Process each sample of the frame with this loop
       for (iframe = 0; iframe < Nframe; iframe++)
  {
    input = x_in[iframe];
    //Biquad section filtering stage-by-stage using coefficients
    //and biquad states stored in a 1D array.
    //A float accumulator is used.
    wn = 0; // Clear the accumulator
    for (iIIR = 0; iIIR < IIR->N_stages; iIIR++)
    {
      icstride = 5*iIIR; //filter coefficient stride into linear array
      isstride = 2*iIIR; //filter state stride into linear array
      //biquad 2nd-order LCCDE code from notes eqn 7.8 & 7.9
                    //Note: The sign of the a[1] and a[2] coefficients is flipped to
match
                    //the format used by CMSIS-DSP. Thus there is a sign change in the
                    //difference equation below for ba_coeff[icstride+3] &
ba_coeff[icstride+4]
      wn = input
         + IIR->ba_coeff[icstride+3] * IIR->state[isstride]
         + IIR->ba_coeff[icstride+4] * IIR->state[isstride+1];
      result = IIR->ba_coeff[icstride]*wn
             + IIR->ba_coeff[icstride+1]*IIR->state[isstride]
             + IIR->ba_coeff[icstride+2]*IIR->state[isstride+1];
      //Update filter state for stage i
      IIR->state[isstride+1] = IIR->state[isstride];
      IIR->state[isstride] = wn;
      input = result; //the output is the next stage input
    }
    x_out[iframe] = result;
  }
}

//IIR_filters.c
// IIR SOS Filter Implementation
// Mark Wickert April 2015

#include "IIR_filters.h"

void IIR_sos_init_float32(struct IIR_struct_float32 *IIR,
                          int16_t N_stages,
                          float32_t *ba_coeff,
                          float32_t *state) {
  // Load the filter coefficients and initialize the filter state.
  int16_t iIIR;
  IIR->N_stages = N_stages;
  IIR->ba_coeff = ba_coeff;
  IIR->state = state;
  for (iIIR = 0; iIIR < 2*IIR->N_stages; iIIR++)
```

```
  {
    IIR->state[iIIR] = 0;
  }
}

void IIR_sos_filt_float32(struct IIR_struct_float32 *IIR,
                          float32_t *x_in, float32_t *x_out,
                          int16_t Nframe) {
       int16_t iframe;
       int16_t iIIR;
       int16_t icstride;
       int16_t isstride;
       float32_t input;
       float32_t result;
       float32_t wn;

  //Process each sample of the frame with this loop
       for (iframe = 0; iframe < Nframe; iframe++)
  {
    input = x_in[iframe];
    //Biquad section filtering stage-by-stage using coefficients
    //and biquad states each stored in 1D arrays.
    //A float accumulator is used.
    wn = 0; // Clear the accumulator
    for (iIIR = 0; iIIR < IIR->N_stages; iIIR++)
    {
      icstride = 5*iIIR; //filter coefficient stride into linear array
      isstride = 2*iIIR; //filter state stride into linear array
      //biquad 2nd-order LCCDE code from Lab 5 eqn 8 & 9
                    //Note: The sign of the a[1] and a[2] coefficients is flipped to
match
                    //the format used by CMSIS-DSP. Thus there is a sign change in the
                    //difference equation below for ba_coeff[icstride+3] &
ba_coeff[icstride+4]
      wn = input
         + IIR->ba_coeff[icstride+3] * IIR->state[isstride]
         + IIR->ba_coeff[icstride+4] * IIR->state[isstride+1];
      result = IIR->ba_coeff[icstride]*wn
             + IIR->ba_coeff[icstride+1]*IIR->state[isstride]
             + IIR->ba_coeff[icstride+2]*IIR->state[isstride+1];
      //Update filter state for stage i
      IIR->state[isstride+1] = IIR->state[isstride];
      IIR->state[isstride] = wn;
      input = result; //the output is the next stage input
    }
    x_out[iframe] = result;
  }
}
```