Assignment #5

Due Monday April 29, 2019

Make Note of the Following:

- If possible write your lab report in Jupyter notebook
- If you choose to use the spectrum/network analyzer to obtain tiff graphics, just import these graphics files into Jupyter notebook as well.

Problems: Real-Time IIR Digital Filters

1. Cascade of Biquads IIR Using Floating-Point Coefficients

   a.) Using a 48 kHz sampling rate and the main module `FM4_IIR_intr.c`, design an **elliptic** bandpass filter using Python (see examples in notes Chapter 7) that satisfies the amplitude response specifications shown below. Create a header filter file in SOS format as described in the Assignment 5 Jupyter notebook. Compare the elliptic design to the complexity of Butterworth and Cheby 1 designs.

   b.) Provide Python design information, including magnitude and phase response plots. Your plots should use a digital frequency axis scaled to the actual sampling frequency.

   c.) Verify the filter real-time frequency response by placing the coefficients in a working cascade of biquads IIR filtering program such as described in notes Chapter 7.

   d.) For verification all you need to obtain is the frequency response magnitude in dB. The best approach is probably to use the vector network analyzer, but optionally you can drive the filter routine with the software noise generator or the noise source in the Analog Discovery. Once you have experimental data in a file, import it into your Jupyter notebook (again see the Chapter 7 sample notebook) to overlay with the theoretical response.

   e.) Time your code with `-o3` optimization using the GPIO IRQ timing pulse. Estimate the maximum sampling rate possible without loss of real-time performance? Switch to the ARM SOS routine to see how much faster it is.
2. Using the GUI parameter slider control for the Cypress FM to implement a variable center frequency notch filter of the form

\[ H(e^{j\omega}) = \frac{1 - 2\cos\omega_0 z^{-1} + z^{-2}}{1 - 2r\cos\omega_0 z^{-1} + r^2 z^{-2}}, \quad 0 < \omega_0 < \pi \]

with \( r = 0.9 \). The parameter the GUI adjusts will be \( \omega_0 \) or have some relation to it. Use float32_t for your design. Test the filter with a sampling frequency of 48 kHz, and verify using the network analyzer the ability to tune the notch around the interval \( f_0 \in (0, f_s/2) \) Hz. What is the notch depth in dB when tuned to about 1 kHz? I will expect a sound demo of this in the lab. A nice test of this system is to sum an audio music source together with a single tone jammer from a function generator, and then see how well the tone can be suppressed from the music without coloration. With a second slider you can use \( r \) to adjust the filter bandwidth. As an audio source you can use an MP3 player/iPod, portable CD player, or Web radio.

**Note:** The notch filter above is really just a single biquad section. So create an SOS coefficients array as described in notes Chapter 7, such that you can change values in the array according to the needed \( \omega_0 \) and \( r \) values. Then you can use the filter functions found in IIR_filters.c/IIR_filters.h, i.e., IIR_sos_filt_float32() or use the CMSIS-DSP functions arm_biquad_cascade_df2T_f32()/arm_biquad_cascade_df2T_init_f32(). As globals include

```c
// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

// IIR notch filter related variables
float32_t fs = 48000;
float32_t r = 0.9;
float32_t f0 = 12000.0;
float32_t ba_coeff1[5]; // single SOS section in place of IIR SOS header include
float32_t x, y, IIRstate1[2];

struct IIR_struct_float32 IIR1;
arm_biquad_cascade_df2T_instance_f32 IIR2;
```

In main initialize the GUI slider accordingly and the one only section biquad as

```c
init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0, 0.0, 12000.0, 0.9);
...

//Initialize IIR notch to the f0 and r values
ba_coeff1[0] = 1.0;
b_a_coeff1[1] = -2.0*cosf(2*PI*f0/fs); // or use FM4_GUI.P_vals[4] for f0
ba_coeff1[2] = 1.0;
b_a_coeff1[3] = -r*ba_coeff1[1]; // or use FM4_GUI.P_vals[5] for r
ba_coeff1[4] = -r*r; // or use FM4_GUI.P_vals[5] for r
```

I recommend updating the filter coefficients only when the parameter slider of interest changes. That means putting an if() code block inside the main while() loop where fol-
Following `update_slider_parameters()`, ...

```c
while(1)
{
    // Update slider parameters
    update_slider_parameters(&FM4_GUI);
    // Reload notch parameters if P_idx is 4 or 5 (meaning f0 or r has changed)
    if((FM4_GUI.P_idx == 4) || (FM4_GUI.P_idx == 5))
    {
        ba_coeff1[1] = -2.0*cosf(2*PI*FM4_GUI.P_vals[4]/fs);
        ba_coeff1[3] = -P_vals[5]*ba_coeff1[1];
        ba_coeff1[4] = -P_vals[5]*P_vals[5];
    }
}
```

The expected results using the Analog Discovery network analyzer, take the form...

![Diagram of notch filter with parameters](image)

3. **Multi-band Cascade of Peaking Filters** Using a Single Biquad per Stage and an LUT for 1 dB Gain Step Coefficients (first introduced in notes Chapter 7 starting at page 7–53): IIR filtering can be used to gain equalize selected frequency bands. The objective here is to implement partially, a multi-band graphic equalizer using a filter coefficient LUT. A peaking filter is used to provide gain or loss (attenuation) at a specific center frequency $f_c$. The peaking filter has unity or 0 dB gain frequency response magnitude at frequencies far removed from the center frequency. At the center frequency $f_c$, the frequency response magnitude in dB is $G_{dB}$. At the heart of the peaking filter is a second-order IIR filter (single biquad section)

$$H_{pk}(z) = C_{pk} \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

which has coefficients...
where

\[ \mu = 10^{G_{\text{dB}}/20} \]

\( f_c \) is the center frequency in Hz relative to sampling rate \( f_s \) in Hz, \( G_{\text{dB}} \) is the peaking filter gain in dB, and \( Q \), typically around 3.5, is inversely proportional to the bandwidth, just as in an analog RLC tank circuit. Here we will be storing the coefficients in a LUT for gain values ranging from -20 dB to +20 dB, in steps of \( \Delta G_{\text{dB}} \) and the remaining parameters fixed. We would like to make \( \Delta G_{\text{dB}} \) as small as 1 dB. For creating biquad coefficient sets, we multiply \( C_{pk} \) through the numerator and establish new numerator coefficients: \( b_0' = C_{pk} \), \( b_1' = C_{pk} b_1 \), and \( b_2' = C_{pk} b_2 \). This problem is centered around creating a cascade of gain tunable peaking filters using an sos 2D array to store the coefficients. The Jupyter notebook Assignment5_sp2018.py contains the function below for this purpose. This function relies on the function `peaking(GdB, fc, Q, fs)` found in the scikit-dsp-comm module sigsys.py. When the sos 2D array is converted to a C header file via

```python
def pf_SOS_LUT(fc, Gmin, Gmax, Gstep = 1.0, Q = 3.5, fs = 48000):
    ""
    Create a peaking filter 2D array using the same format as an SOS array,
    except now each row of coefficients corresponds to a filter with a different gain value
    ""
    GdB = arange(Gmin,Gmax+Gstep,Gstep)
    sos = zeros((len(GdB),6))
    for k, GdBk in enumerate(GdB):
        b,a = ss.peaking(GdBk,fc,Q,fs)
        sos[k,None, :] = hstack((b,a))
    return sos
```

and imported into an FM4 project, you can manipulate the IIR_struct_float32 data structure field `ba_coeff` to point to a particular set of five coefficients that make up a biquad section corresponding to \( G_{\text{dB}} \) of interest. It is left to you to figure out the coding details that tie a particular `FM4_GUI.P_vals[]` float to set the proper pointer value (address) into the sos array.

Before getting into the specific FM4 task, first observe that the peaking filter is parameterized in terms of the peak gain, \( G_{\text{dB}} \), the center frequency \( f_c \), and a parameter \( Q \). Examples of the peaking filter frequency response can be found in Figures 1 and 2. The impact of changing the gain at the center frequency, \( f_c \), can be seen in Figure 2.
The impact of changing $Q$ can be seen in Figure 2, in particular. Peaking filters are generally placed in cascade to form a graphic equalizer. With the peaking filters in cascade, and the gain setting of each filter at 0 dB, the cascade frequency response is unity gain (0 dB) over all frequencies from 0 to $f_s$. In Figure 3 we see the individual response of six filters set at octave center frequencies according to the formula:

$$f_{ci} = 31.25 \times 2^i \text{Hz}, \quad i = 0, 1, \ldots, 9$$

Figure 1: Individual peaking filter magnitude responses in dB for $f_c$ fixed at 500 Hz, $f_s = 44.1$ kHz, $Q$ fixed at 2, and $G_{\text{dB}} = -20, -10, 5, 10, \text{ and } 20$.

The impact of changing $Q$ can be seen in Figure 2, in particular. Peaking filters are generally placed in cascade to form a graphic equalizer. With the peaking filters in cascade, and the gain setting of each filter at 0 dB, the cascade frequency response is unity gain (0 dB) over all frequencies from 0 to $f_s/2$. In Figure 3 we see the individual response of six filters set at octave center frequencies according to the formula:

$$f_{ci} = 31.25 \times 2^i \text{Hz}, \quad i = 0, 1, \ldots, 9$$

This corresponds to a ten octave graphic equalizer (*octave band equalizer*), where the octave band frequencies are spread from 31.25 Hz to 16 kHz. The idea here being a means to reasonably cover the 20–20 kHz audio spectrum. Note that human hearing begins to fall off severely above 15 kHz, so here the 16 kHz peaking filter may have little value. When the three filters described above are placed in cascade, the composite response of Figure 4 is obtained.

![Diagram of peaking filters in cascade](image)

Figure 3: Individual peaking filter magnitude responses in dB with \( Q = 2 \), \( G_{\text{db}} = \{10, -5, 5\} \) dB, \( f_{ci} = \{125, 4000, 8000\} \) Hz, and \( f_s = 48 \) kHz.

This corresponds to a ten octave graphic equalizer (*octave band equalizer*), where the octave band frequencies are spread from 31.25 Hz to 16 kHz. The idea here being a means to reasonably cover the 20–20 kHz audio spectrum. Note that human hearing begins to fall off severely above 15 kHz, so here the 16 kHz peaking filter may have little value. When the three filters described above are placed in cascade, the composite response of Figure 4 is obtained.

![Diagram of peaking filters in cascade](image)

Figure 4: Peaking filter cascade magnitude response in dB with \( Q = 2 \), \( G_{\text{db}} = \{10, -5, 5\} \) dB, \( f_{ci} = \{125, 4000, 8000\} \) Hz, and \( f_s = 48 \) kHz.
From Figure 4, in particular, we notice that when the peaking filters are placed in cascade, the responses do not mesh together perfectly. In fact, the gain flatness in a particular band of frequencies depends on how much gain, $G_{\text{dB}}$, you want to achieve at a particular center frequency, relative to the adjacent frequency bands.

**FM4 Implementation:** On to the implementation of a cascade of at least three single biquad peaking filters, with individual gain adjustment using the FM4 GUI slider control. Your choice on the three center frequencies, but the gain steps must be 1 dB over the range [-20,20] dB. Below is a network analyzer capture from the Analog Discovery that mimics Figure 4. I will expect a network analyzer demo showing that the sliders dynamically change the peaking across the audio spectrum. Second play music through the system and demo how the music bass, mid, and treble response can be varied. In my example above I effectively have one bass control and two treble controls. Again, the center frequencies are your choice. Time the code and estimate if 10 bands can be supported for a single audio channel. If you wanted stereo how many bands can be supported. Understand that for stereo the coefficients can be shared.

4. **Shelving Filters:** A good project idea. Not assigned.

5. **TBD**