ECE 4680 DSP Laboratory 4: FIR Digital Filters

Due Date: ________________

Introduction

Finite Impulse Response Filter Basics

Chapter 3 of the course text deals with FIR digital filters. In ECE 2610 considerable time was spent with this class of filter. Recall that the difference equation for filtering input $x[n]$ with filter coefficient set $b[n], n = 0, \ldots, N$, is

$$y[n] = \sum_{k=0}^{N} h[k]x[n-k]. \tag{1}$$

The number of filter coefficients (*taps*) is $N+1$ (later denoted `M_FIR`) and the filter order is N. The coefficients are typically obtained using filter design functions in `scipy.signal`, e.g. the custom module of function in `fir_design_helper.py`, also utilized in the Jupyter notebook `FIR Filter Design and C Headers.ipynb` or MATLAB's filter design function, `fdatool()`.

Notice that the calculation behind the FIR filter can be viewed as a sum-of-products or as a dot product between arrays/vectors containing the filter coefficients and the present and N past samples of the input. For the Nth filter the computational burden is $N+1$ multiplications and N additions.

The filter impulse response is obtained by setting $x[n] = \delta[n]$ as assuming zero initial conditions (the filter state array $[x[n], x[n-1], \ldots, x[n-N]]$ contains zeros)

$$h[n] = \sum_{k=0}^{N} h[k]\delta[n-k] \tag{2}$$

The filter frequency response is obtained by Fourier transforming (DTFT) the impulse response

$$H(e^{j\omega}) = \sum_{k=0}^{N} h[k]e^{-jk\omega} \tag{3}$$

and the filter system function (z-domain representation) is the z-transform of $h[n]$

$$H(z) = \sum_{k=0}^{N} h[k]z^{-1} = h[0] + h[1]z^{-1} + \cdots + h[N]z^{-N}. \tag{4}$$

From the system function it is clear why the filter order is N. The highest negative power of z is N.

Once a set of filter coefficients is available an FIR filter can make use of them. In Python/MATLAB code this is easy since we have the `lfilter()`/`filter()` function available. In Python suppose `x` is an `ndarray` (1D array) input signal values that we want to filter and vector `h` contains the FIR coefficients. We can obtain the filtered output array `y` via

```
>> y = lfilter(h,1,x);
```

In this lab we move beyond the use of Python (`lfilter(b,a,x)`)/MATLAB (`filter(b,a,x)`) for filtering signals, and consider the real-time implementation of a sample-by-sample filter algorithm in C/C++. The Reay text [1], Chapter 3, is devoted FIR filters and implementation in C. Under the `src` folder of the ZIP package for Lab 4 you find the code module `FIR_filters.c/FIR_filters.h` for implementing both floating-point and fixed-point FIR filters. Thirdly, FIR filters are also available in the ARM CMSIS-DSP library (http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html). Recall CMSIS is the ARM Cortex-M Software Interface Standard, freely available for use with the Cortex-M family of microcontrollers. In this lab you will get a chance to check out the last two approaches, plus write simple filtering code right inside the ISR routine.

Real-Time FIR Filtering in C

In this section several FIR filter implementations are considered. All of the approaches fall back on the same sum-of-products concept, but in slightly different ways. At the end of this section the ARM `CMSIS-DSP` functions are considered, but only at the interface level. The internal details are not explored.

Sum-of-Products In-place Filtering

A simple sum-of-products FIR filtering routine is given in the Jupyter notebook '`FIR Filter Design and C Headers.ipynb`'. This code was prototyped on the host PC using the GNU compilers (`gcc` and `g++`). The core filter code, that here runs inside a simulation loop over `NSAMPLES` is:

```
#define NSAMPLES 50
float32_t x[NSAMPLES], y[NSAMPLES];
float32_t state[M_FIR];
float32_t accum;
...
// Filter Input array x and output array y
for (n = 0; n < NSAMPLES; n++) {
    accum = 0.0;
    state[0] = x[n];
    // Filter (sum of products)
    for (i = 0; i < M_FIR; i++) {
        accum += h_FIR[i] * state[i];
    }
    y[n] = accum;
    // Update the states
    for (i = M_FIR-1; i > 0; i--) {
        state[i] = state[i - 1];
    }
}
```

where `M_FIR` is a constant corresponding to the tap count, as declared via a `#define` in a header file where the coefficients are listed. You will see later how this header file is obtained using a Python function that transfers digital filter coefficients from a Jupyter notebook to a `.h` file.

Brute Force Filtering for a Few Taps

Implementing, say a four tap moving average filter, can be done in a less formal way. Below x and y are `float32_t` values inside the FM4 ISR with `x_old`, essentially the filter state, defined globally as

```
float32_t x_old[4] = {0, 0, 0, 0};

...
// Brute force 4-tap solution
x_old[0] = x;
y = 0.25f*x_old[0] + 0.25f*x_old[1] + 0.25f*x_old[2] + 0.25f*x_old[3];
x_old[3] = x_old[2];
x_old[2] = x_old[1];
x_old[1] = x_old[0];
...
```

Using a Portable Filter Module

The code module `FIR_filters.c/FIR_filters.h`, found in the `src` folder for Lab 4, implements both floating-point and fixed-point FIR filtering routines using portable ANSI C. Here we will only explore the `float32_t` functions. 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 FIR_struct_float32
{
   int16_t M_taps;
   float32_t *state;
   float32_t *b_coeff;
};
```

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 `FIR_init_float32()` and `FIR_filt_float32()` which interact with the FIR filter data structure to initialize and then filter signal samples, respectively. The four steps to FIR filtering using this module are:

1. Create an instance of the data structure:

   ```
   struct FIR_struct_float32 FIR1;
   ```

   where now `FIR1` is essentially a filter object to be manipulated in code.

2. Have on hand two `float32_t` arrays of length `M_FIR` to hold the filter coefficients, e.g., `h_fir[]`, and the filter state, `state[]`. Note actual filter state is really just `M_FIR-1`, but one element is used to hold the present input.

3. Initialize `FIR1` in `main()` using:

   ```
   FIR_init_float32(&FIR1,state101,h_FIR,M_FIR);
   ```

   where here `state101` is the address to a `float32_t` array of length 101 elements used to hold the filter states and `h_FIR` is the address to a `float32_t` array holding 101 filter coefficients. The filter state array should be declared as a global. Often `h_FIR` will be filled using a header file, e.g.,

```
#include "remez_8_14_bpf_f32.h" // a 101 tap FIR
```

is a 101 tap FIR bandpass filter with passband from 8 to 14 kHz generated in a Jupyter notebook.

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

```
FIR_filt_float32(&FIR1,&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 argument 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 returned to the DAC.

The code behind `FIR_filt_float32()`, inside `FIR_filters.c`, is:

```
  //Process each sample of the frame with this loop
  for (iframe = 0; iframe < Nframe; iframe++)
  {
    // Clear the accumulator/output before filtering
    accum = 0;
    // Place new input sample as first element in the filter state array
    FIR->state[0] = x_in[iframe];
    //Direct form filter each sample using a sum of products
     for (iFIR = 0; iFIR < FIR->M_taps; iFIR++)
    {
       accum += FIR->state[iFIR]*FIR->b_coeff[iFIR];
    }
    x_out[iframe] = accum;
    // Shift filter states to right or use circular buffer
    for (iFIR = FIR->M_taps-1; iFIR > 0; iFIR--)
    {
      FIR->state[iFIR] = FIR->state[iFIR-1];
    }
  }
}
```

All working variables are `float32_t`. The outer `for` loop processes each sample within the frame. The first of the inner `for` loops is in fact the sum-of-products represented by (1). The array `FIR->state[]` holds $x[n-k]$ for $k = 0, \ldots, N$ (here `M_taps` is equivalent to $N+1$ in (1)). The second of the inner `for` loops updates the filter history by discarding the oldest input, $x[n-N]$, and sliding all the remaining samples to the left one position. The most recent input, $x[n]$, ends up in `FIR->state[1]` to make room for the new input being placed into `FIR->state[0]` on the next call of the ISR. A more efficient means of keep the state array updated is using *circular buffer*. More on that later.

A complete FIR filter example, `fm4_FIR_intr_GUI.c`, with the GUI configured can be found in the `src` folder of the Lab4 Keil project. This project is configured to load a 101 tap bandpass filter.

ARM CMSIS-DSP FIR Algorithms

An important part of the Arm® Cortex®-M processor family is the Cortex microcontroller software interface standard (CMSIS) `http://www.keil.com/pack/doc/CMSIS/General/html/index.html` of Figure 1. Of particular interest is the DSP library CMSIS-DSP of Figure 2. Fast and

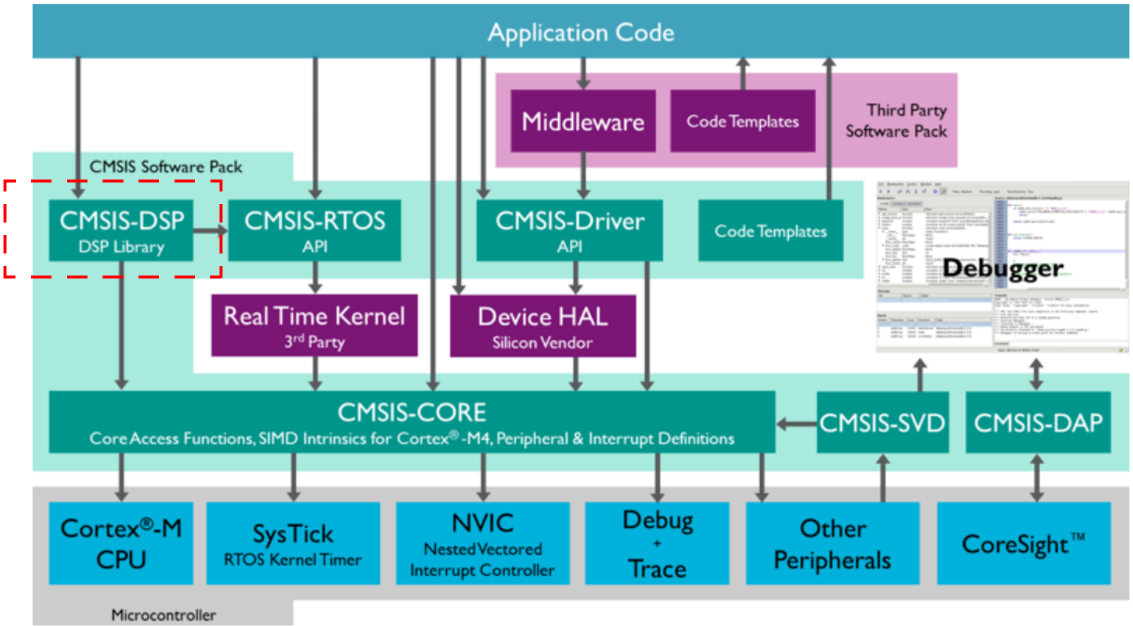
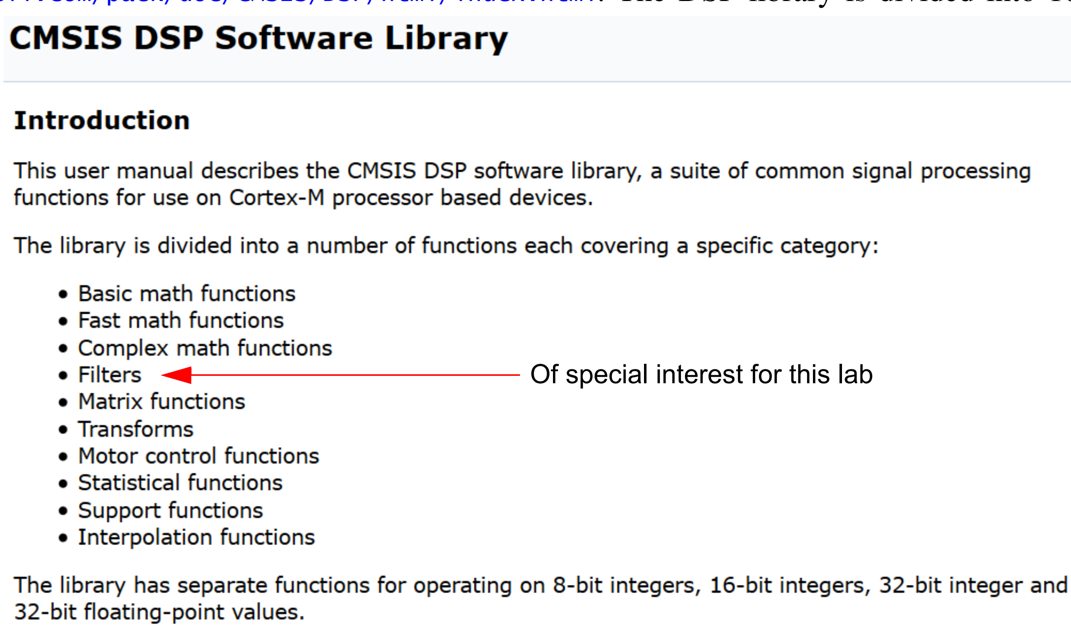


Figure 1: ARM® CMSIS the big picture showing where CMSIS-DSP resides.

efficient C data structure-based FIR filter algorithms are available in the library: `http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html`. The DSP library is divided into 10 major



CMSIS DSP Software Library

Introduction

This user manual describes the CMSIS DSP software library, a suite of common signal processing functions for use on Cortex-M processor based devices.

The library is divided into a number of functions each covering a specific category:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters ← Of special interest for this lab
- Matrix functions
- Transforms
- Motor control functions
- Statistical functions
- Support functions
- Interpolation functions

The library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values.

Figure 2: The CMSIS-DSP library top level description.

groups of algorithms. Other parts of CMSIS are at work in the FM4, for example CMSIS-DAP defines the Debug Access Port interface.

The documentation for the *Filter Functions* group com be expanded by clicking on *Reference* disclosure triangle in the left navigation pane of the CMSIS-DSP Web Site as shown in Figure 3. A good introduction to the inner workings of CMSIS-DSP can be found in Yiu [2]. Within the filtering subheading you find FIR filters, and finally `float32_t` implementations that run on the M4 with the floating-point hardware present. The functions `arm_fir_init_f32()` and `arm_fit_f32()`

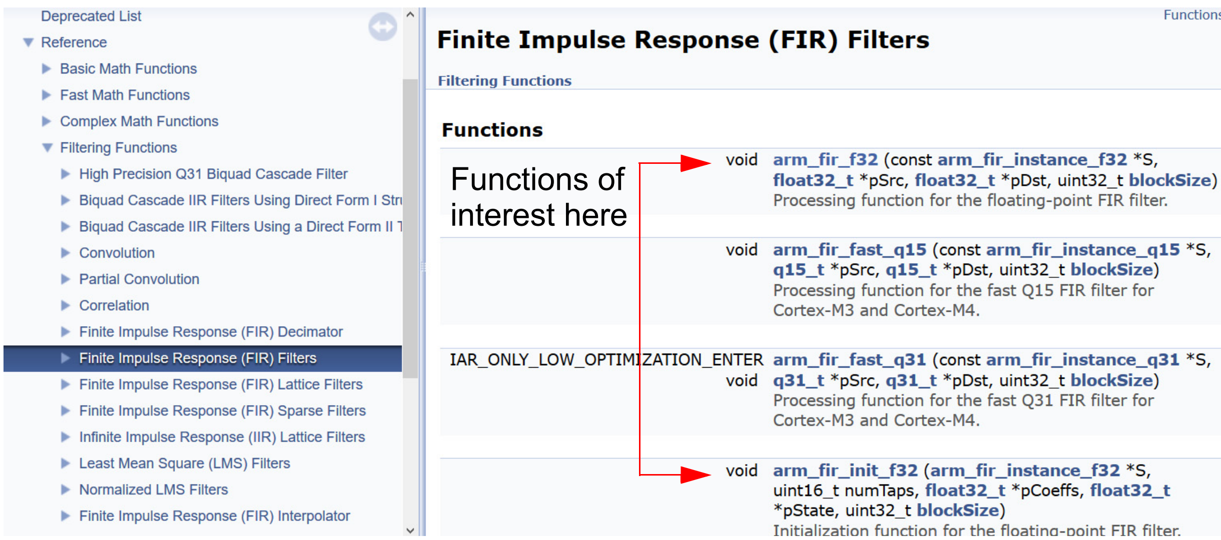


Figure 3: CMSIS-DSP FIR `float32_t` filter detail.

perform functions similar to `FIR_init_float32()` and `FIR_filt_float32()` respectively, of the previous subsection. The function calls are slightly different, but in the end take the same arguments, except in CMSIS-DSP the `init` function takes the frame length. The four steps to FIR filtering using this module are:

1. Create an instance of the data structure :
   ```
   #include "FIR_filters.h" //<<< don't forget this include
   arm_fir_instance_f32 FIR1;
   ```
   where now `FIR1` is essentially a filter object to be manipulated in code.

2. Again you need to have on hand two `float32_t` arrays of length `M_FIR` to hold the filter coefficients, e.g., `h_fir[]`, and the filter state, `state[]`. Note the actual filter state is really just `M_FIR-1`, but as before one element is used to hold the present input.

3. Initialize `FIR1` in `main()` using:
   ```
   FIR_init_float32(&FIR1,state101,h_FIR,M_FIR);
   ```
   where as before `state101` is the address to a `float32_t` array of length 101 elements used to hold the filter states and `h_FIR` is the address to a `float32_t` array holding 101 filter coefficients. The final argument sets frame length (ARM terminology `blocksize`). Here it is just one for sample-by-sample processing.

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

```
arm_fir_f32(&FIR1, &x, &y, 1);
```

where as before `x` is the input sample and `y` is the output sample. The argument 1 is the frame length, which for sample-by-sample processing is just one. No added include file needed for the ARM CMSIS-DSP library.

A complete FIR filter example, `fm4_FIR_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 `FIR_filters.c` module code. To use the ARM code simply comment out the `FIR_filters.c` statements and uncomment the ARM code. You will be doing this in Problem 3

Designing Filters Using Python

The scipy.signal package by itself does not have real strong FIR filter design support, but with some work design capability is available in the module `fir_design_helper.py`. The focus of this module is adding the ability to design linear phase FIR filters from amplitude response requirements. The MATLAB signal processing toolbox has a powerful set of design functions, but for the purposes of this lab the functions described below, which make use of `scipy.signal`, are adequate.

Most digital filter design is motivated by the desire to approach an ideal filter. Recall an ideal filter will pass signals of a certain of frequencies and block others. For both analog and digital filters the designer can choose from a variety of approximation techniques. For digital filters the approximation techniques fall into the categories of IIR or FIR. In this lab you design and implement FIR filters. In Lab 5 you design and implement IIR filters. In the design of FIR filters two popular techniques are truncating the ideal filter impulse response and applying a window, the *windowing method* and *optimum equiripple approximations* [3]. *Frequency sampling* based approaches are also popular, but will not be considered here, even though `scipy.signal` supports all three. Filter design generally begins with a specification of the desired frequency response. The filter frequency response may be stated in several ways, but amplitude response is the most common, e.g., state how $|H_c(j\Omega)|$ or $|H(e^{j\omega})| = |H(e^{j2\pi f/f_s})|$ should behave. A completed design consists of the number of coefficients (taps) required and the coefficients themselves (double precision float or `float64` in Numpy and `float64_t` in C). Figure 4 shows amplitude response requirements in terms of filter gain and critical frequencies for lowpass, highpass, bandpass, and bandstop filters. The critical frequencies are given here in terms of analog requirements in Hz. The sampling frequency is assumed to be f_s Hz. The passband ripple and stopband attenuation values are in dB. Note in dB terms attenuation is the negative of gain, e.g., -60 of stopband gain is equivalent to 60 dB of stopband attenuation.

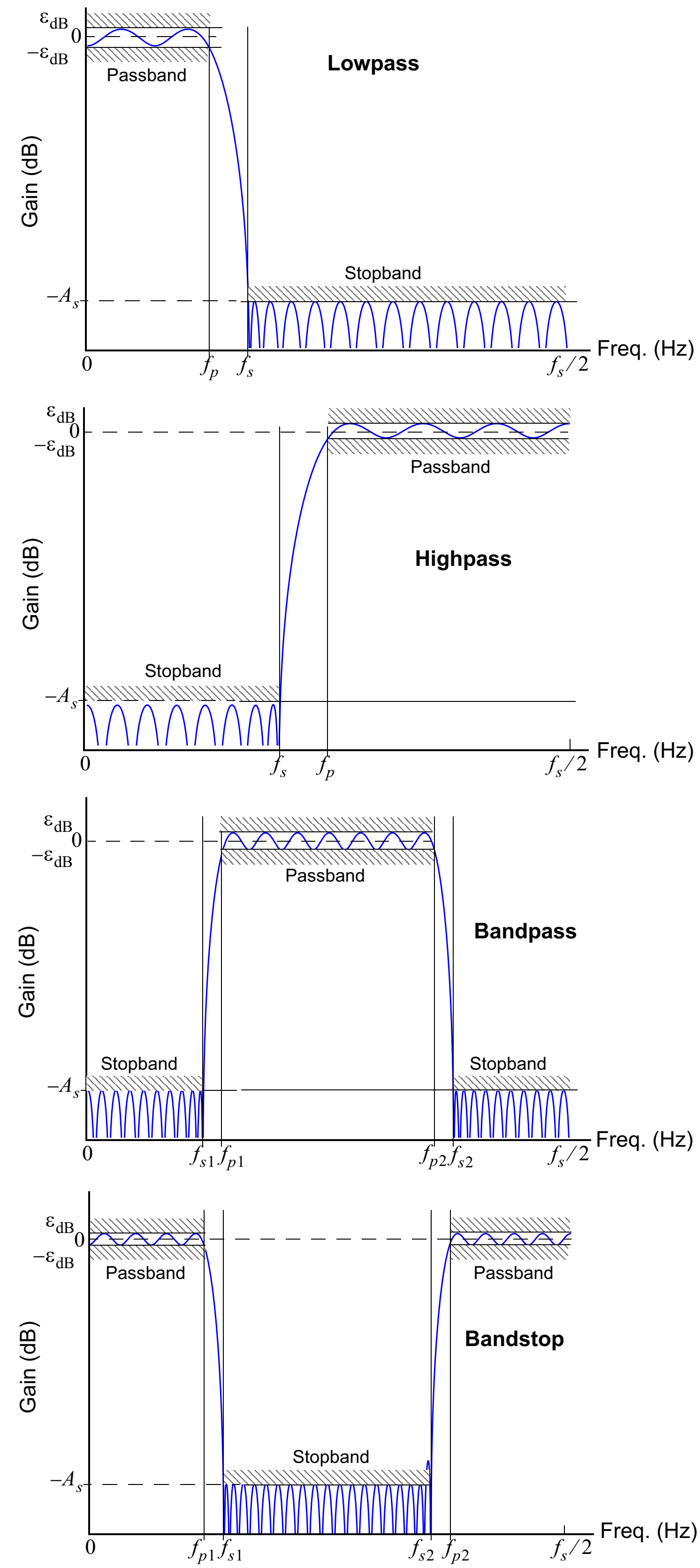


Figure 4: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop filter types.

There are 10 filter design functions and one plotting function available in `fir_design_helper.py`. Four functions for designing Kaiser window based FIR filters and four functions for designing equiripple based FIR filters. Of the eight just described, they all take in amplitude response requirements and return a coefficients array. Two filter functions are simply wrappers around the `scipy.signal` function `signal.firwin()` for designing filters of a specific order with one (lowpass) or two (bandpass) critical frequencies are given. The wrapper functions fix the window type to the `firwin` default of hann (hanning). The plotting function provides an easy means to compare the resulting frequency response of one or more designs on a single plot. Display modes allow gain in dB, phase in radians, group delay in samples, and group delay in seconds for a given sampling rate. This function, `freq_resp_list()`, works for both FIR and IIR designs. Table 1 provides the interface details to the eight design functions where `d_stop` and

Table 1: FIR filter design functions in `fir_design_helper.py`.

Type	FIR Filter Design Functions
Kasier Window	
Lowpass	`h_FIR = firwin_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, n_bump=0)`
Highpass	`h_FIR = firwin_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, n_bump=0)`
Bandpass	`h_FIR = firwin_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, n_bump=0)`
Bandstop	`h_FIR = firwin_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop, fs = 1.0, n_bump=0)`
Equiripple Approximation	
Lowpass	`h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0, n_bump=5)`
Highpass	`h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0, n_bump=5)`
Bandpass	`h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass, d_stop, fs = 1.0, n_bump=5)`
Bandstop	`h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_stop2, f_pass2, d_pass, d_stop, fs = 1.0, n_bump=5)`

The optional `N_bump` argument allows the filter order to be bumped up or down by an integer value in order to fine tune the design. Making changes to the stopband gain main also be helpful in fine tuning. Note also that the Kaiser bandstop filter order is constrained to be even (an odd number of taps).

`d_pass` are positive dB values and the critical frequencies have the same unit as the sampling frequency `f_s`. These functions do not create perfect results so some tuning of of the design parameters may be needed, in addition to bumping the filter order up or down via `N_bump`.

The frequency response plotting function, `freqz_resp_list`, interface is given below:

```
def freqz_resp_list(b,a=np.array([1]),mode = 'dB',fs=1.0,Npts = 1024,fsize=(6,4)):
    """
    A method for displaying digital filter frequency response magnitude,
    phase, and group delay. A plot is produced using matplotlib

    freq_resp(self,mode = 'dB',Npts = 1024)

    A method for displaying the filter frequency response magnitude,
    phase, and group delay. A plot is produced using matplotlib

    freqz_resp(b,a=[1],mode = 'dB',Npts = 1024,fsize=(6,4))

        b = ndarray of numerator coefficients
        a = ndarray of denominator coefficents
     mode = display mode: 'dB' magnitude, 'phase' in radians, or
            'groupdelay_s' in samples and 'groupdelay_t' in sec,
            all versus frequency in Hz
     Npts = number of points to plot; default is 1024
    fsize = figure size; defult is (6,4) inches
```

To see how all of this works a couple of examples extracted from the sample Jupyter notebook are provided below.

<u>*Lowpass Design*</u>

```
b_k = fir_d.firwin_kaiser_lpf(1/8,1/6,50,1.0)
b_r = fir_d.fir_remez_lpf(1/8,1/6,0.2,50,1.0)
```

```
Kaiser Win filter taps = 72.
Remez filter taps = 53.
```

```
fir_d.freqz_resp_list([b_k,b_r],[[1],[1]],'dB',fs=1)
ylim([-80,5])
title(r'Kaiser vs Equal Ripple Lowpass')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Kaiser: %d taps' % len(b_k),r'Remez: %d taps' % len(b_r)),loc='best')
grid();
```

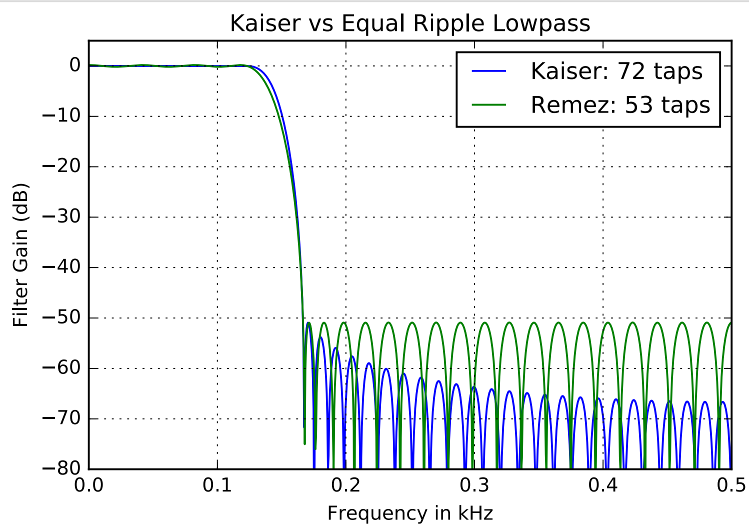


Figure 5: Lowpass design example; Kaiser vs optimal equal ripple.

Bandpass Design

```
b_k_bp = fir_d.firwin_kaiser_bpf(7000,8000,14000,15000,50,48000)
b_r_bp = fir_d.fir_remez_bpf(7000,8000,14000,15000,0.2,50,48000)
```

```
Kaiser Win filter taps = 142.
Remez filter taps = 101.
```

```
fir_d.freqz_resp_list([b_k_bp,b_r_bp],[[1],[1]],'dB',fs=48)
ylim([-80,5])
title(r'Kaiser vs Equal Ripple Bandpass')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Kaiser: %d taps' % len(b_k_bp),
        r'Remez: %d taps' % len(b_r_bp)),
       loc='lower right')
grid();
```

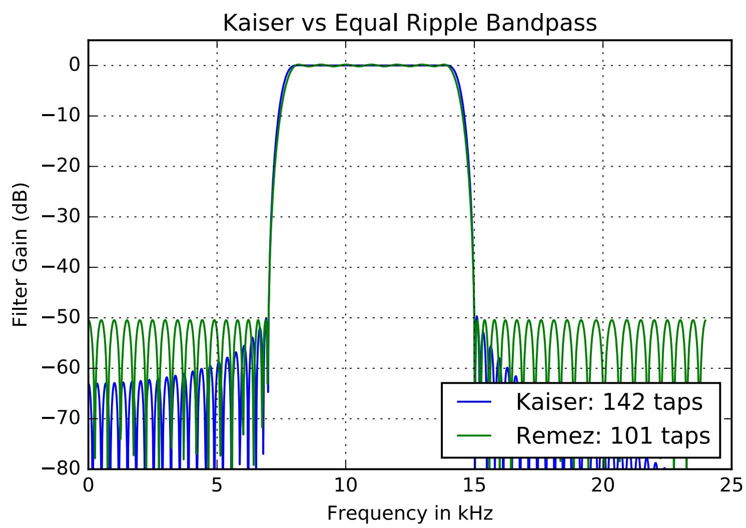


Figure 6: Bandpass design example; Kaiser vs optimal equal ripple.

- Note: In the first cell of the notebook you find
  `import fir_design_helper as fir_d`
- Integrated into this example is the use of `freq_resp_list()`

You may wonder why the filter order is much lower in the equiripple design compared with the Kaiser window. The equiripple design is optimal in the sense of meeting the amplitude response requirements with the lowest order filter, at the expense of an equal ripple response across the entire passband and stopband. The Kaiser window method is inefficient in terms of minimum order, but does asymptotically approach zero gain in the stopband. Another facet in the design process is that the pass band and stopband ripple are linked in the Kaiser window method, while in the equiripple (remez) the ripple is independently specified. By linked I mean by setting the stopband gain you also set the passband ripple.

Writing Filter Designs to Header Files

The final step in getting your filter design to run on the FM4 is to load the filter coefficients `h_FIR` into the 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 `float32_t`, `int16_t` fixed point, and IIR filters implemented as a cascade of second-order sections using `float32_t`. The IIR conversion function will be used in Lab 5. In the sample Jupyter notebook this is done for a bandpass design.

Writing a Coefficient Header File

```
# Write a C header file
c2h.FIR_header('remez_8_14_bpf_f32.h',b_r_bp)
```

- 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_FIR_intr_GUI.c`

The resulting header file is given below:

```
//define a FIR coefficient Array

#include <stdint.h>

#ifndef M_FIR
#define M_FIR 101
#endif
/************************************************************************/
/*                         FIR Filter Coefficients                      */
float32_t h_FIR[M_FIR] = {-0.001475936747, 0.000735580994, 0.004771062558,
                           0.001254178712,-0.006176846780,-0.001755945520,
                           0.003667323660, 0.001589634576, 0.000242520766,
                           0.002386316353,-0.002699251419,-0.006927087152,
                           0.002072374590, 0.006247819434,-0.000017122009,
                           0.000544273776, 0.001224920394,-0.008238424843,
                          -0.005846603175, 0.009688130613, 0.007237935594,
                          -0.003554185785, 0.000423864572,-0.002894644665,
                          -0.013460012489, 0.002388684318, 0.019352295029,
                           0.002144732872,-0.009232278407, 0.000146728997,
                          -0.010111394762,-0.013491956909, 0.020872121644,
                           0.025104278030,-0.013643042233,-0.015018451283,
                          -0.000068299117,-0.019644863999, 0.000002861510,
                           0.052822261169, 0.015289946639,-0.049012297911,
                          -0.016642744836,-0.000164469072,-0.032121234463,
                           0.059953731027, 0.133383985599,-0.078819553619,
                          -0.239811117665, 0.036017541207, 0.285529343096,
                           0.036017541207,-0.239811117665,-0.078819553619,
                           0.133383985599, 0.059953731027,-0.032121234463,
                          -0.000164469072,-0.016642744836,-0.049012297911,
                           0.015289946639, 0.052822261169, 0.000002861510,
                          -0.019644863999,-0.000068299117,-0.015018451283,
                          -0.013643042233, 0.025104278030, 0.020872121644,
                          -0.013491956909,-0.010111394762, 0.000146728997,
                          -0.009232278407, 0.002144732872, 0.019352295029,
```

```
                                 0.002388684318,-0.013460012489,-0.002894644665,
                                 0.000423864572,-0.003554185785, 0.007237935594,
                                 0.009688130613,-0.005846603175,-0.008238424843,
                                 0.001224920394, 0.000544273776,-0.000017122009,
                                 0.006247819434, 0.002072374590,-0.006927087152,
                                -0.002699251419, 0.002386316353, 0.000242520766,
                                 0.001589634576, 0.003667323660,-0.001755945520,
                                -0.006176846780, 0.001254178712, 0.004771062558,
                                 0.000735580994,-0.001475936747};
/***************************************************************************/
```

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

Measuring a 4-Tap FIR Frequency Response Using the Network Analyzer

1. The Comm/DSP lab has test equipment that be used to measure the frequency response of an *analog* filter. In particular, we can use this equipment to characterize the end-to-end response of a *digital* filter that sits inside of an A/D-$H(z)$-D/A processor, such as the FM4 Pioneer Kit. The instructor will demonstrate how to properly use the Agilent 4395A vector/spectrum analyzer for taking frequency response measurements.

   Turning to the vector network analyzer consider the block diagram of Figure 2. You use the vector network analyzer to measure the frequency response magnitude in dB as the ratio of the analog output over the analog input (say using analyzer ports B/R). The phase difference formed as the output phase minus the input phase, the phase response, can also be measured by the instrument, although that is not of interest presently. The frequency range you sweep over should be consistent with the sampling frequency. If you are sampling at 48 ksps what should the maximum frequency of interest be?

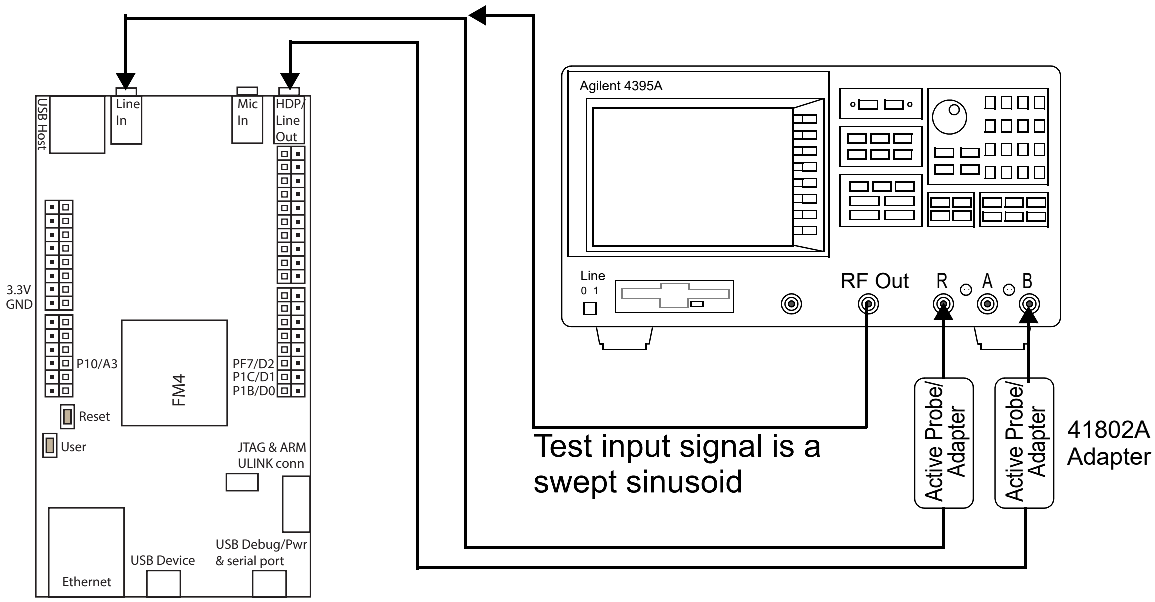


Figure 7: Agilent 4395A vector network analyzer interface to the FM4 audio codec.

Plots from the 4395A can be obtained in several ways. The recommended approach is to use the instruments capability to save an Excel data file or a tiff image file to the 3.5in floppy drive on the Agilent 4395A. Note portable floppy drives can be found in the lab for connecting to the PC's, and hence provides a means to transfer data from the network analyzer to the PC via *sneaker-net*.

a) Implement a 4-tap FIR design using the *Brute Force* technique described earlier in this document. Connect the tap coefficients to GUI slider parameters set to range from -0.5 to +0.5 in 0.01 increment steps. Set the default slider values of all four controls to 0.25 both in the GUI and the FM4 code module.

b) Obtain the frequency response in dB versus frequency for slider settings of [0.25, 0.25, 0.25, 0.25]. Note these coefficients insure that the DC gain of the filter is unity, thus no overload potential at the DAC output. Compare expected/theoretical results to the network analyzer results. I used the Analog Discovery network analyzer and a noise measurement technique you will use later to obtain the following:

```
f_AD, Mag_AD, Phase_AD = loadtxt('MA_4tap_48k.csv', delimiter=',',
                                 skiprows=6, unpack=True)
```

```
fs, x_wav = ss.from_wav('FIR_4tap_MA.wav')
# Choose channel 0 or 1 as appropriate
Px_wav, f_wav = ss.my_psd(x_wav[:,0],2**10,fs/1e3)
```

```
b = [0.25,0.25,0.25,0.25]
f = arange(0.001,0.5,1/1024)
w,H = signal.freqz(b,1,2*pi*f)
plot(f*48, 20*log10(abs(H)))
plot(f_AD/1e3, Mag_AD)
plot(f_wav, 10*log10(Px_wav/Px_wav[10]))
ylim([-60,0])
title(r'Four Tap Moving Average FIR with $f_s = 48$')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (kHz)')
legend((r'Theory',r'AD Measured',r'Noise Capture'),shadow=True)
grid();
savefig('FIR4tap_MA.pdf')
```

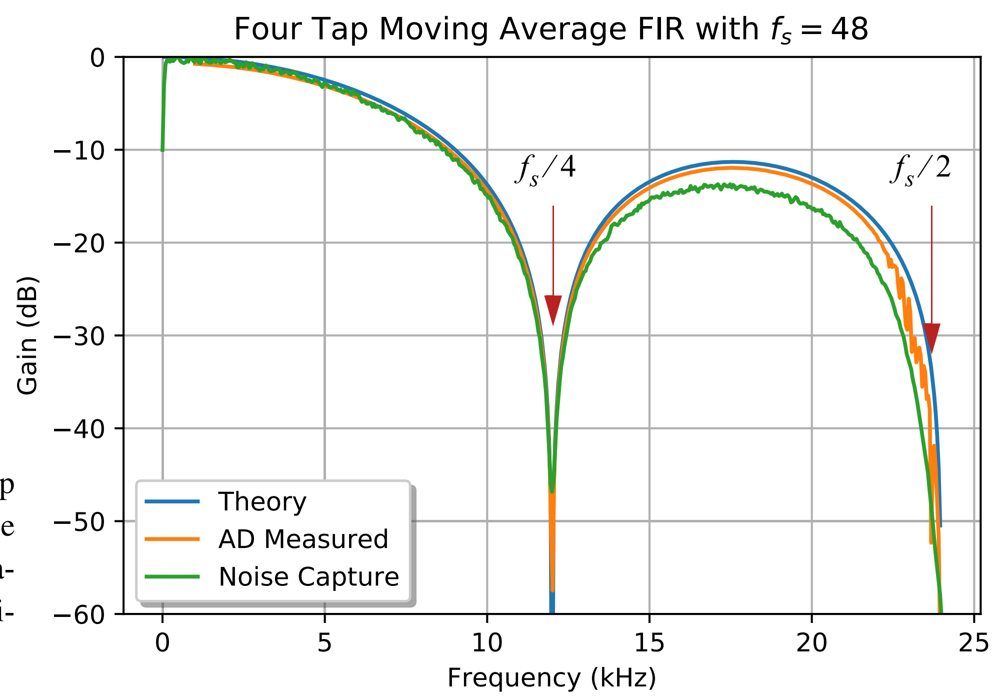


Figure 8: 4-tap moving average theory vs measured comparison.

c) Repeat part (b) except now set the coefficients to [0.25, -0.25, 0.25, -0.25]. Again use the analyzer to find the frequency response. Theoretically there is a connection be the part (b) and part (c) coefficients, namely the use of $(-1)^n = e^{j\pi n}$. Explain the connection between the two frequency responses using the DTFT theorem corresponding to multiplication by $e^{j\pi n}$.

Implementing an FIR Design Originating in `fir_design_helper.py`

2. Now its time to design and implement your own FIR filter using the filter design tools of `fir_design_helper.py`.

   The assignment here is to complete a design using a sampling rate of 48 kHz having an *equiripple* FIR lowpass lowpass response with 1dB cutoff frequency at 5 kHz, a passband ripple of 1dB, and stopband attenuation of 60 dB starting at 6.5 kHz. See Figure 10 for a graphical depiction of these amplitude response requirements.

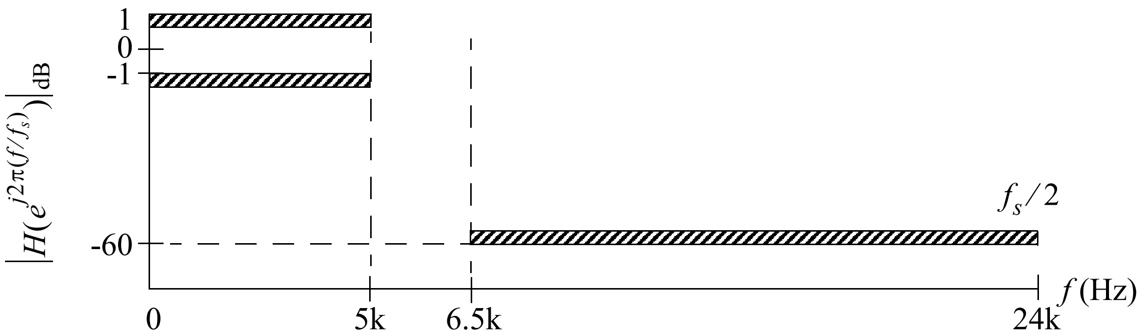


Figure 9: Equiripple lowpass filter amplitude response design requirements.

When the times comes to get your filter running you just need to move your h-file into the project and then include it in the main module, e.g.,

```
#include "remez_8_14_bpf_f32.h" // a 101 tap FIR
```

as in the bandpass example, and re-build the project. Make special note of the fact that the algorithm in the ISR only passes the left channel codec signal through your FIR filter design. The right channel goes straight through from input to output. If you should sweep this channel by accident it will result in a lowpass response, but the cutoff frequency is fixed at $f_s/2$ Hz (in this case 24 kHz).

As an example, results of the optimal FIR bandpass filter of Figure 6 is captured using the Analog Discovery and compared with theory in the Jupyter notebook in the following:

```
f_AD,Mag_AD, Phase_AD = loadtxt('BPF_8_14_101tap_48k.csv',
                                delimiter=',',skiprows=6,unpack=True)
```

```
fir_d.freqz_resp_list([b_r_bp],[[1]],'dB',fs=48)
ylim([-80,5])
plot(f_AD/1e3,Mag_AD+.5)
title(r'Kaiser vs Equal Ripple Bandpass')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz')
legend((r'Equiripple Theory: %d taps' % len(b_r_bp),
        r'AD Measured (0.5dB correct)'),loc='lower right',fontsize='medium')
grid();
```

- Note a small gain adjustment of 0.5 dB is applied to correct for gain differences in the signal chain through the FM4.

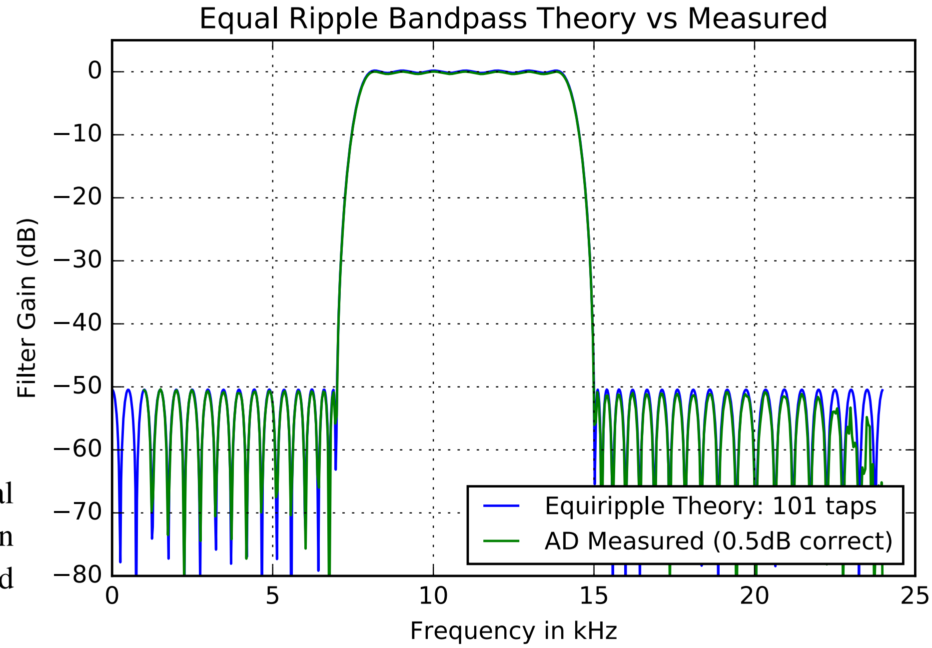


Figure 10: Optimal FIR bandpass design theory vs measured comparison.

Furthermore this example of a 101-tap FIR, makes it clear that a lot of real-time resources are consumed as evidenced from the logic analyzer measurement below:

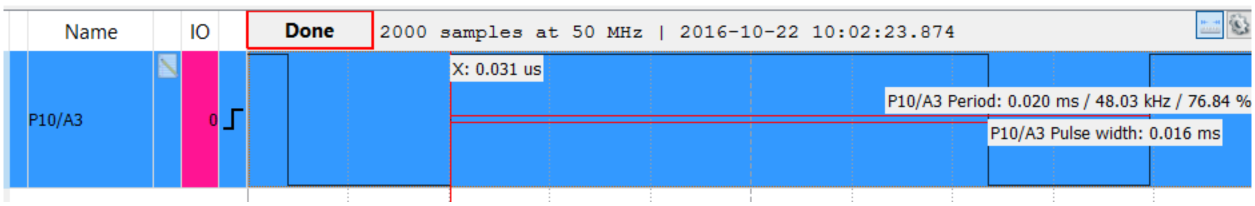


Figure 11: Logical analyzer showing filter loading of the ISR time interval.

a) Using the network analyzer obtain the analog frequency response of your filter design and compare it with your theoretical expectations from the design functions in `fir_design_helper.py`. Check the filter gain at the passband and stopband critical frequencies to see how well they match the theoretical design expectations. By expectations I mean the original amplitude response requirements.

b) Measure the time spent in the ISR when running the FIR filter of part (a) when using the normal -O3 optimization. Recall your experiences with the digital GPIO pin in Lab 3. How much total time is spent in the ISR, $T_{\text{alg}} + T_{\text{overhead}}$, when sampling at 48 kHz?

c) Repeat part (b) replacing the use of `FIR_filt_float32(&FIR1,&x,&y,1)` with the CMSIS-DSP function `arm_fir_f32(&FIR1, &x, &y, 1)`, and the corresponding creation and initialization code. Note the speed improvement offered by CMSIS-DSP.

d) What is the maximum sampling rate you can operate your filter at and still meet real-time operation? Find T_{overhead} by bypassing the filter and letting `y = x` when using the CMSIS-DSP functions.

e) Estimate the maximum number of FIR coefficients the present algorithm can support with $f_s = 48$ kHz and still meet real-time. Show your work. You can assume that T_{alg} grows linearly with the number of coefficients, `M_FIR`. Again assume you are using CMSIS-DSP.

Measuring Frequency Response Using White Noise Excitation

3. Rather than using the network analyzer as in Problems 1–2, this time you will use the PC digital audio system to capture a finite record of DAC output as shown in Figure 12. Use the

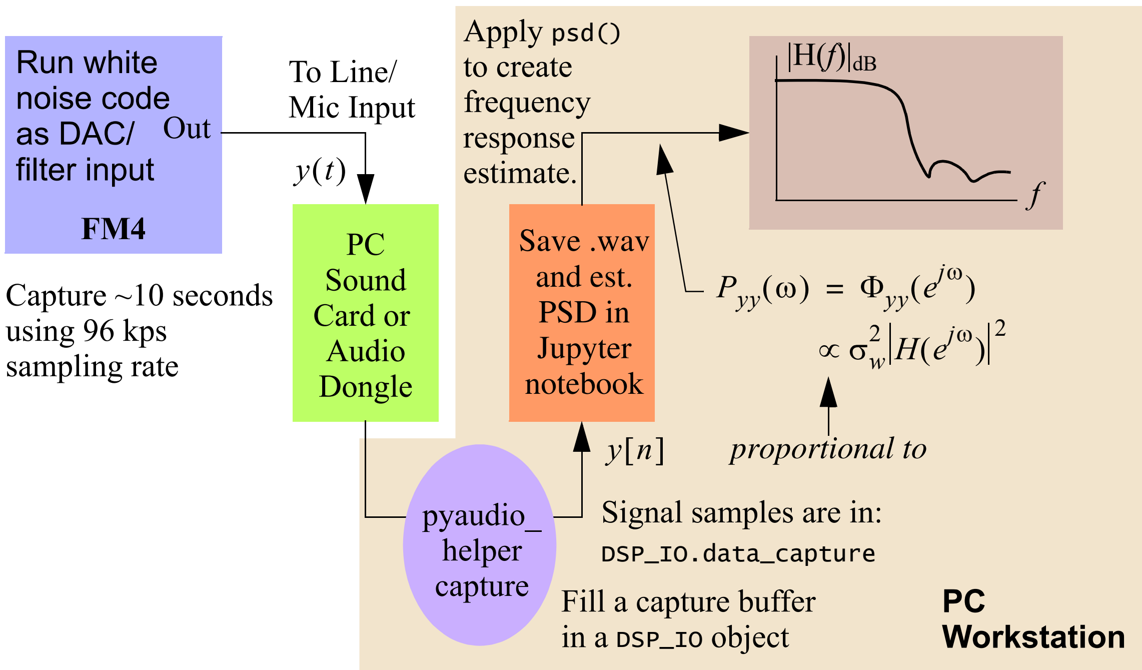


Figure 12: Waveform capture using a PC audio dongle and `pyaudio_helper`.

same FIR filter as used in Problem 2. The capture tool follows from Lab 2 using `pyaudio_helper`. A 10 second capture at 96 kHz seems to work well. Since the sampling rate is 482 ksps, the 96 kHz provides a 2x oversampling to see the full response of the filter in cascade with the D/A (sigma-delta) reconstruction filter frequency response. The Lab 4 Jupyter notebook, `FIR Filter Design and C Headers.ipynb`, contains audio capture details specific for the needs of this lab. Sample capture `.wav` files and plots are included.

To get this setup you first need to add a function to your code so that you can digitally generate a noise source as the input to your filtering algorithm. Add the following uniform random number generator function to the ISR code module:

```
// Uniformly distributed noise generator
int32_t rand_int32(void)
{
  static int32_t a_start = 100001;

  a_start = (a_start*125) % 2796203;
  return a_start;
}
```

Note the code is already in place in the project you extract from the Lab 4 zip file. Now you will drive your filter algorithm with white noise generated via the function `rand_int32()`. In your filter code you will replace the read from the audio code with something like to following:

```
# Replace ADC with internal noise scaled
//x = (float32_t) sample.uint16bit[LEFT];
x = (float32_t) (rand_int32()>>4);
```

The procedure is to use `pyaudio_helper` to obtain a capture buffer of noise samples of about 10s duration at 96 ksps. Ultimately you will have a .wav file archive that you can use for spectral analysis. The objective is to estimate the power spectral density of the filter output, denoted as $y[n]$ in Figure 12. Note $y[n]$ includes the frequency response of both the D/A and the USB audio system. To estimate the power spectral density we use the matplotlib function `psd()`, wrapped to provide more convenient output arrays using

```
Py_wav, f_wav = ss.my_psd(y, NFFT, fs)
```

The spectral analysis function implements Welch's method of averaged periodograms. Suppose that the `.wav` file is saved as `FIR_4tap_MA.wav`, then a plot of the frequency response can be created as follows:

```
fs,y_wav = ss.from_wav('FIR_4tap_MA.wav')
# Choose channel 0 or 1 as appropriate
Py_wav, f_wav = ss.my_psd(y_wav[:,0],2**10,fs/1e3)
# Normalize using a reasonable value close to f = 0, but not zero
plot(f_wav,10*log10(Py_wav/Py_wav[10]))
```

One condition to watch out for is overloading of either the PC sound card line/mic input of if using the Sabrent USB sound card, the mic input. Figure 11 describes how to manage signals levels from the filter output in the FM4 IRQ into the PC sound system internal level.

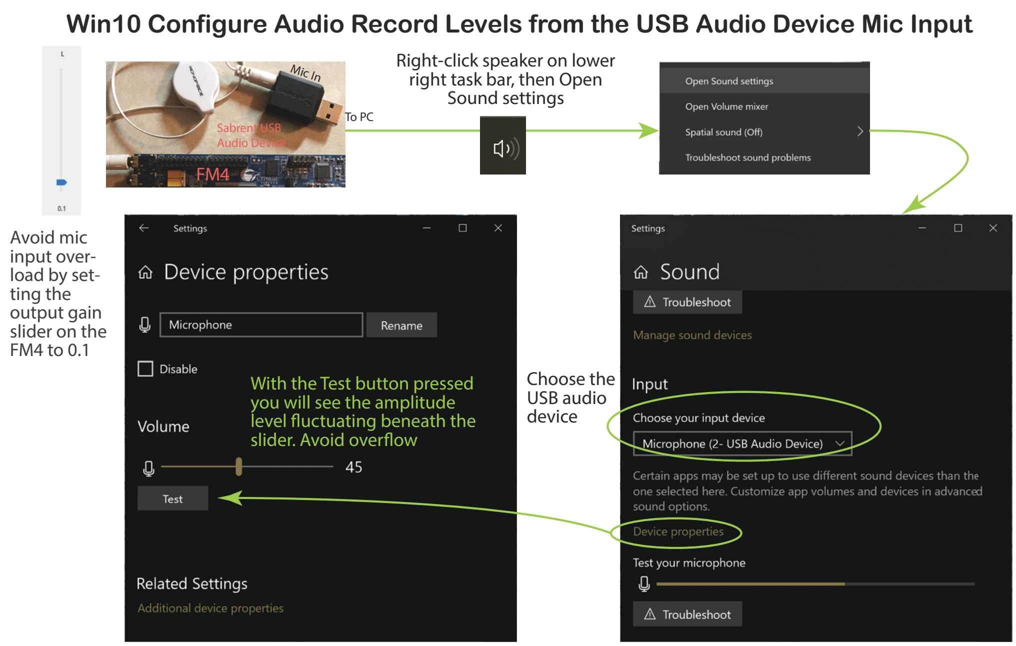


Figure 11: Controlling the input level to USB sound card.

In summary, using the procedure described above, obtain a Python plot of the frequency response magnitude in dB versus frequency of the FM4 DAC output channel. Normalize the filter gain so that it is unity at its peak frequency response magnitude. Overlay the theoretical frequency response estimate using the original Python generated filter coefficients.

Variable Time Delay Filter Using a Circular Buffer

4. **Circular Buffer Background**: In the development of the FIR filtering sum-of-products formula the present output, y, is formed as

$$y = \sum_{k=0}^{M_{\text{FIR}}-1} \text{state}[k] h_{\text{FIR}}[k], \tag{5}$$

in response to the present input x and $M_{\text{FIR}} - 1$ past inputs using the *linear* array $\text{state}[k]$, $0 \le k \le M_{\text{FIR}} - 1$. This array is updated by shifting all array entries to the right and placing the new *present* input on the left end at index zero. The circular buffer avoids this reshuffling by keeping a pointer to the oldest value [4]. The newest value is written over the oldest value as each new input, x, arrives to be filtered. All values remain static in the array as the pointer does all of the work. A graphical depiction of the linear array and the circular buffer approach is shown Figure 13. To access past values of the input modulo index arithmetic is

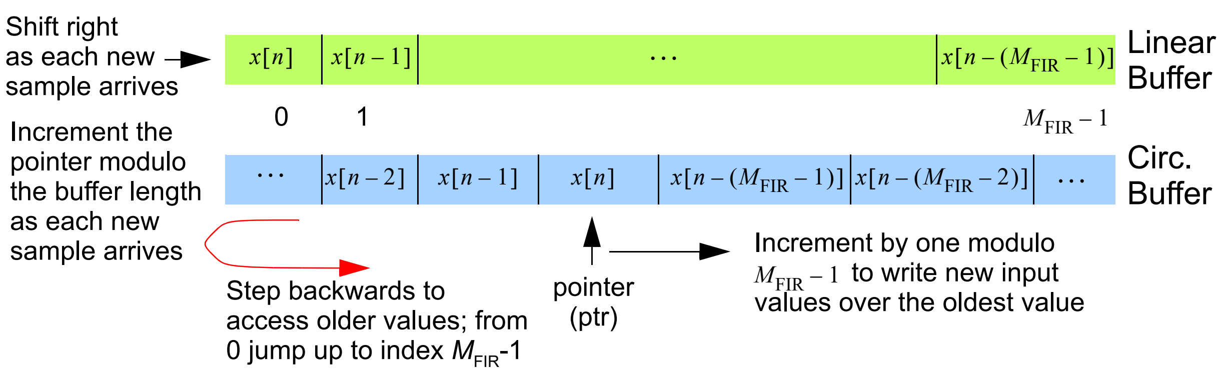


Figure 13: Linear array hold dynamic values versus the circular buffer holding static values.

needed to step backwards through the array, modulo the array length. The C language has the operator `%` for modulo math, but it does not work properly when a negative index occurs. The function

```
// A mod function that takes negative inputs
int16_t pmod(int16_t a, int16_t b)
{
   int16_t ret = a % b;
   if(ret < 0)
     ret += b;
   return ret;
}
```

solves this problem by returning nonnegative indices. A downside of using the circular buffer is that that both `%` and `pmod` are not single cycle operations. Reshuffling a linear array takes multiple clock cycles as well. Which requires fewer total clock cycles depends on the architecture factors. In dedicated DSP microprocessors hardware for circular buffer pointer control is built-in. On the Cortex-M family this is not the case [2].

Variable Delay Implementation: When implementing a pure delay filter modulo addressing inefficiency is not a major concern, as a pure delay filter of n_d samples has all zero taps except for a single unity tap, e.g.,

$$y[n] = x[n - n_d] \tag{6}$$

where $0 \le n_d \le N_{\text{FIR}} - 1$. Ignoring global variable initialization, the variable delay filter takes the form

```
// Begin buffer processing
// Write new input over oldest buffer value
circbuf[ptr] = x;
// Calculate delay index working backwards
delay = (int16_t) FM4_GUI.P_vals[2];
y = circbuf[pmod(ptr - delay,N_buff)];
// Update ptr to write over the oldest value next time
 ptr = (ptr + 1) % N_buff;
```

a) Fill in the rest of details in the above code snippets to implement a variable delay that is adjustable from 0 to 10 ms when $f_s = 48\,\text{kHz}$. The GUI slider control should take integer steps. Notice that since the buffer also holds the present sample value, the length of the buffer has to be sample longer than you might think to get the desired time delay.

b) Verify that the delay is adjustable over the expected 10 ms range using the oscilloscope. Show your results to the lab instructor. To make the testing clear input a 50 Hz square wave turned pulse train by setting the duty cycle to about 5%. A sample display from the Analog Discovery is shown in Figure 14.

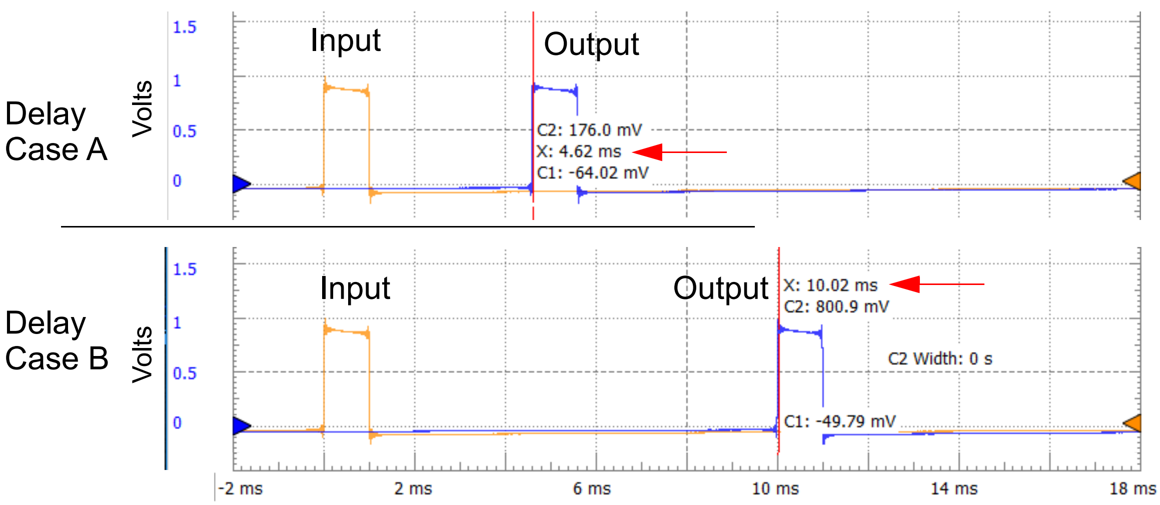


Figure 14: AD output from the fully implemented 0–10ms variable delay.

c) Record the ISR timing to see how efficient the time delay is, in spite of how long the circular buffer is. Nice?

d) In a later lab (likely Lab 6) you will use this variable delay along with a low frequency sinusoidal signal generator to implement an audio special effect known as *flanging*[1] [4]. To get a test of this under manual control (slider control), input a 1 kHz sinusoid from a bench function generator. Using speakers of earphones listen to the time delayed output moving the time delay slider control up and down. By changing the time in real time you compressing and then expanding the time axis, which should make the tone you hear waver in frequency (like the Doppler effect). Demonstrate this to your lab instructor.

e) What else? Not complete at this time is an experiment with a simple two element microphone array combined with the adjustable time delay to provide beam steering.

References

[1] Donald Reay, *Digital Signal Processing Using the ARM Cortex-M4*, Wiley, 2016

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

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

[4] Thad B. Welch, Cameron H.G. Wright, and Michael Morrow, *Real-Time Digital Signal Processing from MATLAB to C with the TMS320C6x DSPs*, second edition, CRC Press, 2012.

Appendix A: Designing FIR Filters from Amplitude Response Requirements

The Python module `fir_design_helper.py` supports the design of windowed (Kaiser window) and equal-ripple (remez) linear phase FIR filters starting amplitude response requirements. The filter order (or equivalently order + 1 = number of taps) is determined as part of the design process. Design functions are available for lowpass, highpass, and bandpass filters.

For details see the Read-the-Docs helps pages for scikit-dsp-comm at: http://scikit-dsp-comm.readthedocs.io/en/latest/?badge=latest

Appendix B: Writing FIR Coefficient Header Files

The Python module `coef2header.py` supports the writing of C style header files for use in the filter function of FIR_Filter.c/FIR_Filter.h and the ARM functions.

For details see the Read-the-Docs helps pages for scikit-dsp-comm at: http://scikit-dsp-

1. https://en.wikipedia.org/wiki/Flanging

comm.readthedocs.io/en/latest/?badge=latest