FIR Filter Design

Both floating point and fixed-point FIR filters are the objective here. we will also need a means to export
the filter coefficients to header files. Header export functions for float32 and int16 format are provided
below. The next step is to actually design some filters using functions found in scipy.signal.

Note: The MATLAB signal processing toolbox is extremely comprehensive in its support of digital filter
design. The use of Python is adequate for this, but do not ignore the power available in MATLAB.
Exporting Coefficients to Header Files

Float Header Export

In [5]: def FIR_header(fname_out,h):
    """
    Write FIR Filter Header Files
    
    Mark Wickert February 2015
    """
    M = len(h)
    N = 3 # Coefficients per line
    f = open(fname_out,'wt')
    f.write('//define a FIR coeffient Array\n\n')
    f.write('#include <stdint.h>

')
    f.write('#ifndef M_FIR
')
    f.write('#define M_FIR %d
' % M)
    f.write('#endif
')
    f.write('/************************************************************************/
')
    f.write('/* FIR Filter Coefficients */
')
    f.write('float32_t h_FIR[M_FIR] = {
')
    kk = 0;
    for k in range(M):
        #k_mod = k % M
        if (kk < N-1) and (k < M-1):
            f.write('%15.12f,' % h[k])
            kk += 1
        elif (kk == N-1) & (k < M-1):
            f.write('%15.12f,
' % h[k])
            if k < M:
                f.write(' ')
            kk = 0
        else:
            f.write('%15.12f % h[k])
    f.write('};
')
    f.write('/************************************************************************/
')
    f.close()

Signed 16-Bit Export

In [6]: def FIR_fix_header(fname_out,h):
    """
    Write FIR Fixed-Point Filter Header Files
    
    Mark Wickert February 2015
    """
    M = len(h)
    hq = int16(rint(h*2**15))
    N = 8 # Coefficients per line
    f = open(fname_out,'wt')
    f.write('//define a FIR coeffient Array\n\n')
    f.write('#include <stdint.h>

')
    f.write('#ifndef M_FIR
')
    f.write('#define M_FIR %d
' % M)
    f.write('#endif
')
    f.write('/************************************************************************/
')
    f.write('/* FIR Filter Coefficients */
')
    f.write('float32_t h_FIR[M_FIR] = {
')
    f.write('float32_t h_FIX_16
')
    f.write('for i in range(M):
        if (kk < N-1) and (k < M-1):
            f.write('%15.12f,' % h[k])
            kk += 1
        elif (kk == N-1) & (k < M-1):
            f.write('%15.12f,
' % h[k])
            if k < M:
                f.write(' ')
            kk = 0
        else:
            f.write('%15.12f % h[k])
    f.write('};
')
    f.write('/************************************************************************/
')
    f.close()
Example: Windowed FIR Filter Design using `firwin()`

The `scipy.signal` function `firwin()` designs windowed FIR filters as described in the Chapter 6 notes. As a specific example consider a lowpass filter of 31 taps and normalized digital cutoff frequency $f = F / F_s$, where $F_s$ is the sampling frequency in Hz and $F$ is the continuous-time frequency variable. Here, $f$, represents the normalized discrete-time frequency axis.

For this 31 tap filter we choose the cutoff frequency to be $F_c = F_s / 8$, or in normalized form $f_c = 1/8$.

In [36]: # N_taps, 2*Fc/Fs = 2*f_c
   ...: b = signal.firwin(31,2*1/8)
   ...: b

Out[36]: array([-1.20388000e-03, -2.05336094e-03, -2.07962901e-03,
   ...:               1.64050411e-18, 4.76490069e-03, 9.89603364e-03,
   ...:               9.97846427e-03, 9.89603364e-03, 4.76490069e-03,
   ...:               1.64050411e-18, -2.07962901e-03, -2.05336094e-03,
   ...:               -1.20388000e-03])

In [37]: f = arange(0,0.5,.001)
   ...: w,B = signal.freqz(b,1,2*pi*f)
   ...: plot(f*48,20*log10(abs(B)));
   ...: plot([48/8,48/8],[-80,0],'r')
   ...: ylim([-80,0])
   ...: title(r'FIR Filter Gain in dB vs F in (Hz) (assume $F_s = 48$kHz)')
   ...: ylabel(r'Gain (dB)')
   ...: xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
   ...: grid();
In this case the filter coefficients are cut-and-pasted into Keil MDK and reformatted. An equivalent set of fixed-point coefficients is obtained by scaling/rounding/quantizing the float values shown above.

```
In [38]: b = signal.firwin(31,2*1/8)
    :   int16(rint(b*2**15))
```

```
Out[38]: array([-39, -67, -68, 0, 156, 324, 327, 0, -621,
    :   -1189, -1139, 0, 2249, 5022, 7322, 8216, 7322, 5022,
    :   2249, 0, -1139, -1189, -621, 0, 327, 324, 156,
    :   0, -68, -67, -39], dtype=int16)
```

```
In [39]: sum(int16(rint(b*2**15)))
```

```
Out[39]:

32770
```

```
In [40]: fs_48,P31_float = loadtxt('spec_noise_b31_fs48.csv',delimiter=',',
    :   skiprows=1,usecols=(0,1),unpack=True)

In [41]: f = arange(0,1.0,.001)
    :   w,B = signal.freqz(b,1,2*pi*f)
    :   plot(f*48,20*log10(abs(B)));
    :   plot(fs_48[:340]/1000,P31_float[:340]-P31_float[0])
    :   ylim([-80,5])
    :   xlim([0,48/2])
    :   title(r'FIR float32_t Noise Capture and Theory Compared')
    :   ylabel(r'Gain (dB)')
```
Execution time for float32 is 6.48μs

In [42]: fs_48fix,P31_fix = loadtxt('spec_noise_b31_fs48fix.csv',delimiter=',',
                             skiprows=1,usecols=(0,1),unpack=True)

In [43]: f = arange(0,1.0,.001)
   w,B = signal.freqz(b,1,2*pi*f)
   plot(f*48,20*log10(abs(B))); 
   w,Bq = signal.freqz(rint(b*2**15)/2**15,1,2*pi*f)
   plot(f*48,20*log10(abs(Bq))); 
   plot(fs_48[:340]/1000,P31_fix[:340]-P31_fix[0]) 
   ylim([-80,5])
   xlim([0,48/2])
   title(r'FIR int16_t Noise Capture and Theory Compared')
   ylabel(r'Gain (dB)')
   xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
   legend((r'Theory',r'Theory Quantized',r'Measured'),loc='best')
   grid();
Execution time for `int16_t` is 5.22\(\mu s\).
Execution time using the CMSIS-DSP library function `arm_fir_f32()` results in 3.78\(\mu s\).

In [4]: # Load an equi-ripple design code module from GNU Radio
       #(in notebook ZIP package)
       import optfir

Equi-Ripple Design Using Remez

Scipy.signal does not have a direct means to design equal-ripple FIR filters from amplitude response specifications. Within GNU Radio there is a Python module named `optfir.py` which contains some support functions. A modified version of the `optfir.py` module, which comments out some other functions not needed, and not supported without another GNU Radio module, is included in the ZIP package for this notebook. Once this module is imported, you then have access to the function `optfir.remezord()`. The recipe for using this function along with `signal.remez()` is shown below. The amplitude response requirements are given in positive dB values. Note: The passband ripple in dB is the peak-to-peak value. This function should also work for bandpass designs, but the details have not been worked out yet.

In [9]:
   d_pass = 0.2
   d_stop = 60.0
   fs = 48000
   f_pass = 3500
   f_stop = 5000
   n, ff, aa, wts=optfir.remezord([f_pass,f_stop], [1,0],
                                  [1-10**(-d_pass/20),10**(-d_stop/20)],
                                  fsamp=48000)
   # Bump up the order by 5 to bring down the final d_pass & d_stop
   n_bump = n + 5
   b1 = signal.remez(n_bump, ff, aa[0::2], wts,Hz=2)
Note: The original amplitude response requirements have been changed. The passband ripple is now 0.2 db and the passband critical frequency is reduced from 4000 to 3500 Hz. This reduces the filter order.

Frequency Response Magnitude in dB

In [10]: f = arange(0,0.5,.001)
   w,B = signal.freqz(b1,1,2*pi*f)
   plot(f*fs/1e3,20*log10(abs(B)));
   title(r'Equiripple Lowpass: %d Taps' % n_bump)
   ylabel(r'Filter Gain (dB)')
   xlabel(r'Frequency in kHz ($f_s =$ %d kHz)' % (fs/1e3,))
   ylim([-70,0])
   xlim([0,fs/1e3/2])
   grid();

Pole Zero Plot

In [9]: ssd.zplane(b1,[1],auto_scale=False,size=1.2)

Out[9]: (77, 0)
In [11]: b1_fix = int16(rint(b1*2**15))
   b1_fix


In [13]: FIR_header('s4_p1_remez_f32.h',b1)

In [10]: FIR_fix_header('s4_p1_remez.h', b1)

Remez Bandpass Using optfir.remezord

In [12]: d_pass = 0.2
d_stop = 50.0
s = 48000
f_stop1 = 7000
\[ f_{\text{pass1}} = 8000 \]
\[ f_{\text{pass2}} = 14000 \]
\[ f_{\text{stop2}} = 15000 \]
\[ n, ff, aa, wts=\text{optfir.remezord}([f_{\text{stop1}}, f_{\text{pass1}}, f_{\text{pass2}}, f_{\text{stop2}}], [0, 1, 0], [10^{-(d_{\text{stop}}/20)}, 1-10^{-(d_{\text{pass}}/20)}, 10^{-(d_{\text{stop}}/20)}], fsamp=48000) \]

# Bump up the order by 5 to bring down the final \( d_{\text{pass}} \) & \( d_{\text{stop}} \)
\[ n_{\text{bump}} = n + 5 \]
\[ b2 = \text{signal.remez}(n_{\text{bump}}, ff, aa[0::2], wts, Hz=2) \]

```python
In [13]: f = arange(0,0.5,.001)
w, B = signal.freqz(b2, 1, 2*pi*f)
plot(f*fs/1e3, 20*log10(abs(B)));
title(r'Equiripple Lowpass: %d Taps' % n_bump)
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz ($f_s =$ %d kHz)' % (fs/1e3,))
ylim([-70, 0])
xlim([0, fs/1e3/2])
grid();
```

In [27]: FIR_header('remez_8_14_bpf_f32.h', b2)

In [28]: FIR_fix_header('remez_8_14_bpf_int16.h', b2)