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 in \texttt{float32} and \texttt{int16} format. To support both of these activities the \texttt{scikit-dsp-comm} Python modules \texttt{fir_design_helper.py} and \texttt{coeff2header.py} are available. Documentation explaining the use of \texttt{fir_design_helper} can be found on the GitHub pages associated with \texttt{scikit-dsp-comm} at \texttt{FIR and IIR design helpers}, and also in part included in this notebook. The details on using \texttt{coeff2header.py} is presently only found in this notebook, but may soon be integrated into the GitHub pages.

- Windowed (Kaiser window) and Equal-Ripple FIR Filter Design

The module \texttt{fir_design_helper.py} contains custom filter design code build on top of functions found in \texttt{scipy.signal}. Functions are available for winowed FIR design using a Kaiser window function and equal-ripple FIR design, both type have linear phase.
Fig. A.1: General amplitude response requirements for the lowpass, highpass, bandpass, and bandstop filter types.
Table A.1: FIR filter design functions in fir_design_helper.py.

<table>
<thead>
<tr>
<th>Type</th>
<th>FIR Filter Design Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kaiser Window</strong></td>
<td></td>
</tr>
<tr>
<td>Lowpass</td>
<td>h_FIR = fir_win_kaiser_lpf(f_pass, f_stop, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Highpass</td>
<td>h_FIR = fir_win_kaiser_hpf(f_stop, f_pass, d_stop, fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Bandpass</td>
<td>h_FIR = fir_win_kaiser_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop,</td>
</tr>
<tr>
<td></td>
<td>fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td>Bandstop</td>
<td>h_FIR = fir_win_kaiser_bsf(f_stop1, f_pass1, f_pass2, f_stop2, d_stop,</td>
</tr>
<tr>
<td></td>
<td>fs = 1.0, N_bump=0)</td>
</tr>
<tr>
<td><strong>Equiripple Approximation</strong></td>
<td></td>
</tr>
<tr>
<td>Lowpass</td>
<td>h_FIR = fir_remez_lpf(f_pass, f_stop, d_pass, d_stop, fs = 1.0,</td>
</tr>
<tr>
<td></td>
<td>N_bump=5)</td>
</tr>
<tr>
<td>Highpass</td>
<td>h_FIR = fir_remez_hpf(f_stop, f_pass, d_pass, d_stop, fs = 1.0,</td>
</tr>
<tr>
<td></td>
<td>N_bump=5)</td>
</tr>
<tr>
<td>Bandpass</td>
<td>h_FIR = fir_remez_bpf(f_stop1, f_pass1, f_pass2, f_stop2, d_pass,</td>
</tr>
<tr>
<td></td>
<td>d_stop, fs = 1.0, N_bump=5)</td>
</tr>
<tr>
<td>Bandstop</td>
<td>h_FIR = fir_remez_bsf(f_pass1, f_stop1, f_pass2, f_pass2, d_pass,</td>
</tr>
<tr>
<td></td>
<td>d_stop, fs = 1.0, N_bump=5)</td>
</tr>
</tbody>
</table>

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).

- **Example: Lowpass with** $f_s = 1$ Hz

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

- **Example: Lowpass with** $f_s = 1$ Hz

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

```python
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.
```python
def 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 Hz')
legend((r'Kaiser: %d taps' % len(b_k), r'Remez: %d taps' % len(b_r)), loc='best')
grid();
```

---

```
b_k_hp = fir_d.firwin_kaiser_hpf(1/8, 1/6, 50, 1.0)
b_r_hp = fir_d.fir_remez_hpf(1/8, 1/6, 0.2, 50, 1.0)
```

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

---

```
def fir_d.freqz_resp_list([b_k_hp, b_r_hp], [[1], [1]], 'dB', fs=1)
ylim([-80, 5])
title(r'Kaiser vs Equal Ripple Lowpass')
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in Hz')
legend((r'Kaiser: %d taps' % len(b_k), r'Remez: %d taps' % len(b_r)), loc='best')
grid();
```
```
fs = 48000
b_k_bp = fir_d.firwin_kaiser_bpf(7000,8000,14000,15000,50,fs)
b_r_bp = fir_d.fir_remez_bpf(7000,8000,14000,15000,0.2,50,fs)

Kaiser Win filter taps = 142.
Remez filter taps = 106.

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();
```
Exporting Coefficients to Header Files

Once a filter design is complete it can be exported as a C header file using `FIR_header()` for floating-point design and `FIR_fix_header()` for 16-bit fixed-point designs.

- **Float Header Export**

  ```python
  def FIR_header(fname_out,h):
      """
      Write FIR Filter Header Files
      
      Mark Wickert February 2015
      """
  ```

- **16 Bit Signed Integer Header Export**

  ```python
  def FIR_fix_header(fname_out,h):
      """
      Write FIR Fixed-Point Filter Header Files
      
      Mark Wickert February 2015
      """
  ```

These functions are available in `coeff2header.py`. 

---

Assignment 4 Notebook Sample
# Write a C header file
c2h.FIR_header('remez_8_14_bpf_f32.h', b_r_bp)

- The header file, `remez_8_14_bpf_f32.h` written above takes the form:

```c
//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,*/
/* 0.003667323660, 0.001589634576, 0.000242520766,*/
/* 0.002386316353, -0.002699251419, -0.00692787152,*/
/* 0.002072374590, 0.006247819434, -0.000017122009,*/
/* 0.00544273776, 0.01224920394, -0.008238424843,*/
/* 0.00584603175, 0.00968130613, 0.007237935594,*/
/* 0.003554185785, 0.00423864572, -0.002894644665,*/
/* 0.013460012489, 0.002388684318, 0.019352295029,*/
/* 0.002144732872, -0.009232278407, 0.000146728997,*/
/* -0.010111394762, -0.013491956960, 0.020872121644,*/
/* 0.025104278030, -0.013643042233, -0.015018451283,*/
/* -0.00068299117, -0.019644863999, 0.000002861510,*/
/* 0.052822261169, 0.015289946639, -0.049012297911,*/
/* -0.01664274836, -0.0001646469072, -0.032121234463,*/
/* 0.05995731027, 0.133383985599, -0.078819553619,*/
/* -0.23981117665, 0.036017541207, 0.285529343906,*/
/* 0.036017541207, -0.23981117665, -0.078819553619,*/
/* 0.133383985599, 0.05995731027, -0.0231234463,*/
/* -0.00164469072, -0.01664274836, -0.049012297911,*/
/* 0.015289946639, 0.052822261169, 0.000002861510,*/
/* -0.019644863999, -0.00068299117, -0.015018451283,*/
/* -0.013643042233, 0.025104278030, 0.020872121644,*/
/* -0.013491956960, -0.010111394762, 0.000146728997,*/
/* -0.009232278407, 0.02144732872, 0.019352295029,*/
/* 0.02388684318, -0.013460012489, -0.002894644665,*/
/* 0.000423864572, -0.003554185785, 0.007237935594,*/
/* 0.00968130613, -0.00584603175, -0.002388684318,*/
/* 0.001224920394, 0.000544273776, -0.000017122009,*/
/* 0.06247819434, -0.002388684318, -0.00692787152,*/
/* -0.002699251419, 0.002388684318, 0.000242520766,*/
/* 0.01589634576, 0.003667323660, -0.008238424843,*/
/* -0.006176846780, 0.001254178712, 0.004771062558,*/
/* 0.000735580994, -0.001475936747};
/************************************************************************/

This file can be included in the main module of an FM4 project
A Prototype FIR Filtering Function

Prototype a simple FIR filtering routine to run on the host. Here the compiler I have used the GNU compiler tools (gcc and g++). To make it easy to move the code to Keil compatible data types are defined.

```c
// simple_FIR.c
#include <stdio.h>
#include <math.h>
#define PI 3.141592653589793
#define NSAMPLES 50

int main(void)
{
    FILE *fp;
    int i, n;
    float x[NSAMPLES], y[NSAMPLES];
    float state[5] = {0,0,0,0,0};
    float h_FIR[5] = {1,1,1,1,1};
    float state[M_FIR];

    int main(void)
    {
        FILE *fp;
        int i, n;
        //int M_FIR;
        float x[NSAMPLES], y[NSAMPLES];
        //float h_FIR[5] = {1,1,1,1,1};
        //float state[5] = {0,0,0,0,0};

        float state[M_FIR];
        float _accum;
```
// Define an input sequence
for (i = 0; i < NSAMPLES; i++) {
    if (i == 0) {
        x[i] = 1.0;
    } else {
        x[i] = 1.0;
    }
    //x[i] = 1.0*cosf(2*PI*i/20);
    //printf("n = %-d, x = %6.4f\n", i, x[i]);
}

// Initialize filter states to zero
for (i = 0; i < M_FIR; i++) {
    state[i] = 0.0;
}

// Filter (sum of products) the input
//M_FIR = 5;
for (n = 0; n < NSAMPLES; n++) {
    accum = 0.0;
    state[0] = x[n];
    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];
    }
    printf("n = %-d, y = %6.4f\n", n, y[n]);
}

// Output results to a file
fp = fopen("FIR_test.txt", "w");
for (n = 0; n < NSAMPLES; n++) {
    if (n == 0) {
        fprintf(fp, "Filter I/O data as n, x, y\n");
        fprintf(fp, "%d, %6.4f, %6.4f\n", n, x[n], y[n]);
    } else if (n == NSAMPLES-1) {
        fprintf(fp, "%d, %6.4f, %6.4f", n, x[n], y[n]);
    } else {
        fprintf(fp, "%d, %6.4f, %6.4f\n", n, x[n], y[n]);
    }
} fclose(fp);

return 0;
Hilbert Transforming FIR Filter Design

Consider the design a Hilbert transforming digital filter. The `scipy.signal` module is capable of such designs. The second design below is a bandpass Hilbert that attempts to obtain a very flat passband over a narrow range of frequencies. The phase is linear in all cases with fixed phase shift of $-\pi/2$.

The wideband filters are designed assuming $f_s = 1$ Hz, but later when the frequency response is plotted the responses are frequency scaled to $f_s = 48$ kHz.

```python
fs = 48 # kHz for the BPF filter only, while the others effectively have fs = 1 Hz
b31 = signal.remez(31, array([0, .05,.1,.9,0.95,1])/2,[.1,1,.1],
    fs=1,type='hilbert')
b31_bpf = signal.remez(31,[0,10,15,18,23,fs/2],[0,1,0],weight=[10,100,10],
    fs=fs,type='hilbert')
b63 = signal.remez(63,array([0, .01,.05,.95,0.99,1])/2,[.01,1,.01],
    Hz=1,type='hilbert')
b121 = signal.remez(121,[0, .01,.05,.95,0.99,1])/2,[.01,1,.01],
    fs=1,type='hilbert')
stem(b31)
title(r'31-Tap Hilbert Transform FIR Impulse Response')
ylabel(r'Amplitude')
xlabel(r'Sample Index ($n$)')
grid();
```

![31-Tap Hilbert Transform FIR Impulse Response](image.png)
The bandpass design is preferred for this problem as the modulation is narrowband, centered at say 17 kHz.
Python Simulation of the Hilbert-Based Envelope Detector

In Python we implement all of the blocks less the FM4 A/D and D/A. We simulate the signal generator using a discrete-time signal equivalent.

```python
# Generate the AM test signal:
fs = 48000
Tsim = 0.01 #s
fc = 17000
fm = 1000
n = arange(0,int(fs*Tsim))
a_mod_index = 0.8
x = (1 + a_mod_index*cos(2*pi*fm/fs*n))*cos(2*pi*fc/fs*n)
t = n/fs*1000 #v make units ms

plot(t,x)
title(r'AM Signal')
ylabel('Amplitude')
xlabel(r'Time (ms)')
```

Phase Imparted to a Sinusoid (wideband design)
# Problem 1

FIR lowpass with various testing options.

```python
# Build up the simulation
# You will need to use signal.lfilter(b,a,x) for linear filtering

# Form xd[n] as a pure delayed by 15 samples version of x[n]
xd = 

# Form hat{xd}[n] as the approximate Hilbert transform via a 31-tap Hilbert FIR
xd_hat = 

# Find the envelope of xd[n] + j*xd_hat[n]
r = 

# Implement the DC block first-order IIR filter
r_blk = signal.lfilter()
# or set up a loop to evaluate the difference equation directly
```
Problem 2

High-order (greater than 200 taps) FIR implemented using DMA and frames-based processing to meet real-time.

```python
b = fir_d.fir_remez_bpf(4500, 5000, 7000, 7500, 1, 60, 48000, N_bump=3)

Remez filter taps = 185.
```

```python
fir_d.freqz_resp_list([b], [1], 'dB', 48000, Npts=2048)
ylim([-80, 5])
title(r'Frequency Response Magnitude: $f_s = 48$ kHz')
grid()
```

![Frequency Response: $f_s = 48$ kHz](image)

Repurpose the design for 8000 Hz Sampling Rate

```python
# When the design is repurposed with fs = 8000 Hz
fir_d.freqz_resp_list([b], [1], 'dB', 8000, Npts=2048)
ylim([-80, 5])
title(r'Frequency Response Magnitude: $f_s = 8$ kHz')
grid()
```
Problem 3

Hilbert FIR to form analytic signal from real signal, then envelope detector using the `sqrt()` of the complex envelope.