

ECE 3640

Filter Programming Assignment

Objective and Background

In this programming assignment you will design both IIR and FIR filters. The IIR filter is “designed” by pole placement. The FIR filter is designed with the help of a MATLAB design tool. You will then implement the filters in C++. You will test the filter response by providing sinusoidal inputs at a variety of frequencies and comparing the actual magnitude response with the theoretical magnitude response.

The signals you are to deal with are sampled at a rate of $F_s = 8000$ samples/sec.

One of the algorithms you will use for the filter design uses the Matlab routine `remez`. (Note: most recent versions of MATLAB call this function `firpm`. However, since there may be older versions of MATLAB still in use, we will refer to the function as `remez`. You should check the `help` function in MATLAB.) The `remez` routine is an implementation of the algorithm known as the Parks-McClellan algorithm for digital filter design, after its inventors. (The name “remez” comes from a particular numerical algorithm, known as the “Remez exchange” which forms part of the Parks-McClellan algorithm.) The Matlab manual pages for the `remez` function are provided with this assignment. (For example, do `help remez` in MATLAB. To use the `remez` routine, you specify the lower and upper frequencies and the desired magnitude response at those frequencies. The frequencies are given as a fraction of the Nyquist frequency for the system (the maximum frequency representable in the sampled system). For example, if you are sampling at $F_s = 8000$ samples/sec, the Nyquist frequency is 4000 Hz. The continuous-time frequency of 4000 Hz. corresponds in the `remez` algorithm, to a digital frequency of 1.

Assignment

Design of the filters

1. Design a filter using the `remez` routine. (See the notes below.)
 - (a) Using the `remez` routine and the sampling rates described above, design a linear-phase FIR digital filter with the following characteristics:
 - 100 coefficients
 - Lowpass filter
 - Passband frequency of 1000 Hz with unit gain.
 - Stopband frequency of 1500 Hz with zero gain.
 - (b) Plot the desired magnitude response and the actual magnitude response.
 - (c) Make a plot of the phase response of the filter (see `freqz` in MATLAB). In your writeup, comment on whether the phase response is actually linear (as it should be). Note: the MATLAB command `unwrap` may be helpful here.
 - (d) Save your filter coefficients to a file that you can later read in to your C++ program.
2. Design a filter using pole-zero placement.
 - (a) Design a lowpass digital filter with the same passband and stopband as before using pole-zero placement. That is, place some poles and zeros, make a plot using `freqz`, and see if it works. You decide where and how many poles and zeros to place. This may take several iterations, and you will probably not obtain the same quality of design as you did using `remez`. Helpful functions to be aware of: `roots`, `poly`, `zplane`, `freqz`. Look up MATLAB help on these. Also, it would probably be very helpful to write a MATLAB script (i.e., program), so that all you need to do is change some numbers to have the results computed for you.
 - (b) Scale the result so the DC frequency response has magnitude 1.
 - (c) Make a plot of the desired magnitude response and the actual magnitude response of your filter.
 - (d) Make a plot of the phase response of the filter (see `freqz`).

- (e) Convert the pole-zero representation to a rational transfer function representation (the routine `zp2tf` might be helpful).
- (f) Save the filter coefficients to a file you can use.

Implementation Write a C++ class that implements digital filtering. The class might have the following form:

```
class filter {
    // private data
    double *numcoeff; // pointer to numerator coefficients
    double *dencoeff; // pointer to denominator coefficients
    int type;         // IIR or FIR?
public:              // public interface
    filter(int n, double *numcoeff); // FIR constructor
    filter(int n, double *numcoeff, int m, double *dencoeff); // IIR constructor
    ~filter();         // destructor -- make sure you de-allocate memory
    double filt(double input); // the actual filter function
    // whatever other functions you might want
};
```

Your constructor functions need to allocate space for the filter coefficients and the filter storage, copy the filter coefficients passed in into their appropriate storage arrays. Your filter function should do filtering as appropriate, whether IIR or FIR. (To be really fancy, you might want to use some sort of virtual function to select whether you are doing FIR or IIR filtering, but this level of sophistication is not really necessary.)

You could use this filter class in your program as follows:

```
#include "filter.h" // <-- you write this
#include <math.h>
... // other stuff

main( ... ) {
// ...

    int n = 100; // number of FIR filter coefficients
    // ... // read the FIR filter coefficients into an array called y
    // ...

    filter FIR(n,y); // instantiate the FIR filter
    // ...

    // Here is how to use it:
    x = ... // x = the input function
    y = FIR.filt(x); // "pass" x into the filter, then y is the output

    // read the IIR coefficients into a numerator array called num, and a
    // denominator array called den.
    int n = ... // number of numerator coefficients
    int m = ... // number of denominator coefficients
```

```

// ...

filter IIR(n,num,m,den); // instantiate the IIR filter

// Here is how to use it:
x = ...                // x = the input function
y = IIR.filt(x);       // "pass" x into the filter, then y is the output

// ...
}

```

Testing You will test your filter by generating sinusoids at different frequencies, comparing the output amplitude with the input amplitude. You should save these ratios into a file that can be loaded into MATLAB so that you can later make a plot. Then, get in MATLAB and **plot the desired magnitude response (obtained using freqz), overlaid with the actual frequency response as determined by your program.**

Remember that for each different frequency you choose, you must wait for a period of time for the transient to die down, before determining the output amplitude. The following might be helpful outline.

```

#include <math.h>      // use this so that cos works correctly
#include "filter.h"   // your filter stuff
// other stuff

main()
{
    double Fs = 8000;    // sample rate
    double fmin = 10;   // minimum frequency to pass through filter (Hz)
    double fmax = Fs/2; // maximum frequency to pass through filter (Hz)
    double fstep = 50;  // frequency step increment (Hz)
    double Tmax = 20;   // maximum time in seconds of signal to filter
                        // (you may need to adjust this!)
    double Ttransient=2; // amount of seconds to wait for transient to die down
    double f;           // current frequency value
    double t;           // current time value
    double T = 1/Fs;    // sample interval
    double F;           // discrete-time frequency
    double outamplitude; // amplitude of filtered signal
    double inamplitude = 1; // input amplitude;

    // Read in filter coefficients, instantiate filter objects, etc., as before
    // Say that we are dealing with an object called FIR.

    // open the output file
    // ...

    // Now get down to business:
    for(f = fmin; f <= fmax; f += fstep) { // step through the frequency values
        F = f*T;                          // compute digital frequency
        for(t = 0; t <= Tmax; t += T) { // time values
            x = inamplitude*cos(2*pi*f*t);
            y = FIR.filt(x); // pass the signal through the filter
            if(t >= Ttransient) { // if we have waited long enough

```

```

        // determine the amplitude of the output signal (think carefully!)
        // ...
        outamplitude = ...;
    }
} // end for t
ratio = outamplitude/inamplitude;
// write the frequency F and the ratio to a file
// ...
} // end for t

// close the output file
// ...
} // end main

```

To plot the results, you will need to get into MATLAB, plot the desired magnitude response with `freqz`, load in your actual results using the `load` command, then plot these on the same axes. For example, you might have the following:

```

% Let a and b represent the numerator and denominator polynomials
% of the designed filter
[h,f] = freqz(a,b);
clf; % clear current figure (remove existing plots)
subplot(2,1,1); % plot in part of the available window
plot(f/(2*pi),20*log10(abs(h))); % plot the magnitude response in dB
hold on; % make it so that the plot can be overlaid
xlabel('discrete frequency') % label the axes
ylabel('magnitude response, dB')

% read in your data, say into the arrays fa and ha
% ...

% Now plot your results on top of the theoretical results
plot(fa,20*log10(ha),' : '); % plot your computed response with
                             % dotted line

```

Turn In the following:

- Plots of filter magnitude and phase response (and the ideal magnitude response) for the FIR filter.
- Listings of your filter class code.
- Listings of your testing code.
- Plots comparing the actual magnitude response with the tested magnitude response (overlay the actual values with the design values on a single plot). For the FIR filter, also overlay the desired frequency response. Also include a discussion on whether the phase response of the FIR filter is linear.
- The transfer function for your IIR filter in both pole-zero and polynomial form. Also a plot of the pole and zero locations you used in your design
- Plot showing the actual IIR-filtered magnitude response for the given frequencies. Also a description of how you chose the normalizing factor.
- A discussion of what you learned and comparisons between FIR and IIR filters (e.g., which runs faster, and why, which does a better job of filtering, and why, etc.)

Notes on the `remez` function

The `remez` function is invoked as

```
h = remez(n,f,m)
```

(See the `help remez` in MATLAB.) The following information comes from the manual for *The Student Edition of Matlab, v. 4*.

- The filter computes the coefficients for an FIR filter with $n + 1$ coefficients.
- The argument `f` is a vector of pairs of frequency points expressed in normalized frequency: the frequency of 1 corresponds to half the sampling frequency, and 0 corresponds to 0 Hz (DC). The frequencies in `f` must be listed in increasing order.
- `m` is a vector containing the desired magnitude response at the points specified in `f`. The vectors `m` and `f` must be the same length.
- The desired magnitude function at frequencies between pairs of points `f(k), f(k+1)` for `k odd` is the line segment connecting the points `f(k), m(k)` and `f(k+1), m(k+1)`.
- The desired magnitude function at frequencies between pairs of points `f(k), f(k+1)` for `k even` is unspecified. These correspond to transition bands or “don’t care” regions.
- If you use the form

```
h = remez(n,f,m,w)
```

then `w` is a weight vector, which uses the weights to fit in each frequency band. The length of `w` is half the length of `f` or `m`, so there is exactly one weight per band.

Here is an example of a bandpass filter: We desire a filter to pass signals between $f = 0.4$ and $f = 0.6$ (normalized frequency) with an amplitude of 1, and to attenuate other signals. A plot of the desired frequency response is

```
f = [0 0.3 0.4 .6 0.7 1];
m = [0 0 1 1 0 0]; % bands: 0-0.3 is a stop band (at 0)
                    %      0.3-0.4 is a transition band
                    %      0.5-0.6 is the pass band
                    %      0.6-0.7 is a transition band
                    %      0.7-1 is a stop band (at 0)
clf;                % clear previous plot, if any
plot(f,m,':');      % a plot of the desired frequency response
h = remez(17,f,m);  % create a filter with 18 coefficients
[response,freq] = freqz(h,1,512); % compute the frequency response
hold on;           % overlay the plots
plot(freq/pi,abs(response)); % freq/pi converts to this normalized
                        % freq. scale
                        % abs(response) is the magnitude response
xlabel('f (normalized)');
ylabel('Magnitude response');
legend('Desired frequency response','Actual frequency response, 18 coefs.')
```

The plot below shows the results:

