EC581 DSP Projects: Lab Project #2

Dept. of Electrical and Computer Engineering
Rose-Hulman Institute of Technology

FIR Filter Implementation in MATLAB and in C

Adapted for the 6711 DSK by Mark A. Yoder, 3/27/03

Last Modified on 3/8/2002 (KEH)

I. FIR Digital Filter Design and Implementation in MATLAB

The MATLAB function, or "M-file", shown in Listing 1 on Page 5 provides an example of how to design and use finite impulse response (FIR) filters using the FIR filter design function **fir1(N,Wn),** the digital filter response calculating function **freqz(A,B)**, and the digital filtering function **filter(B,A,X)** which are provided in the "*MATLAB Signal Processing Toolbox*", which is described at

(http://www.mathworks.com/access/helpdesk/help/toolbox/signal/signal.shtml).

Make sure that you completely understand this M-file. MATLAB provides "help" documentation for these three routines: (For example, you may obtain information on the FIR1 function when running MATLAB by typing "helpwin FIR1" at the MATLAB prompt):

Note that in the example M-file of Listing 1, we must set the pole coefficient array, A, equal to 1, which implies that a1 = 1, and all the other coefficients, a2, a3, a4, ... = 0, since an FIR filter has no poles.

A. Download the M-file of Listing 1, called **lab02.m**, from `G:\ee\yoder\ece581\Labs\Lab02`

   Start MATLAB and execute this M-file. Verify that this function does what you expect.

B. Now modify this M-file to obtain the sixteen FIR filter coefficients that correspond to a
   1. 16^{th}-order band-pass FIR filter with a pass-band between 4800 Hz and 14.4 kHz, and a sampling frequency of 48 kHz.
   2. 16^{th}-order high-pass FIR filter with unity-gain passband above 12.0 kHz, and a sampling frequency of 48 kHz.

Include a printout of each of these sets of FIR coefficients as ***Attachment A*** in your lab report.

II. Real-Time, Floating Point FIR Digital Filter Implementation

Study the digital filtering program shown in Listing 2 on Page 6. This FIR filtering program was written to be easily understood, and is therefore not very efficient. For example, the "for loop" in the ISR that updates the input sample storage array, x[ ], by shifting the newly converted sample into x[0], what was in x[0] goes into x[1], what was in x[1] goes into x[2], etc., is certainly not efficient, though it makes the convolution calculation very straightforward. The use of a circular input buffer to hold the last N input samples leads to much more efficient code, though the subscript variable management becomes trickier. In this case, a reference pointer (that points to the most recent input sample in the buffer) is simply rotated around the circular buffer, allowing the previous inputs in the buffer remain in their original position within the buffer.

This real-time FIR filtering program implements the digital band-stop filter that corresponds to the MATLAB M-file of Listing 1. Recall that this was a stop-band filter with a stop band between 9.6 kHz and 14.4 kHz at a sampling frequency of 48 kHz.

A. Create a new project called Lab02 based on your Lab01. Download **lab02a.c** from the previously cited AFS network class directory. Use the contents of **lab02a.c** to replace your **processBuffer()** routine in Lab02.c. Place this file in a subdirectory, along with the other relevant linker command, assembly, and library files, and follow the steps outlined in Lab 1 to build the project.

   Run the program. Display the *CPU Load Graph* to be sure your CPU has time enough to do all the processing. Mine used only 25% of the CPU without optimization. If yours is running too slow, see this footnote[1] for suggestions. Verify, using a function generator and an oscilloscope, that this filter attenuates audio signals roughly between 9.6 kHz and 14.4 kHz, as expected. Replace the oscilloscope with a loudspeaker. Note that the attenuation in the stop band is not as apparent with the loudspeaker, since your hearing responds logarithmically rather than linearly.

B. Now insert the seventeen filter coefficients you obtained in Part I (B.1) above for the bandpass filter. Run the program. Listen to the output of this filter when you speak into the microphone. Your voice should have that characteristic band-limited "telephone sound"!

C. Use the Microsoft EXCEL spreadsheet to plot the observed filter magnitude response of this band-pass filter in decibels versus frequency, where

$$AvdB = 20\log(Voutpeak/Vinpeak)$$

   Vary the frequency using a function generator over a range of 100 Hz – 3.7 kHz. Be sure that the amplitude of the function generator is not turned up too high, since we do not want the filter output to become distorted at any frequency within the specified range. Use an oscilloscope to measure the gain at (at least) eleven evenly-spaced frequencies of 100 Hz, 500 Hz, 900 Hz, 1.3 kHz, 1.7 kHz, 2.1 kHz, 2.5 kHz, 2.9 kHz, 3.3 kHz, 3.7 kHz. You may want to take additional measurements at frequencies where the response of the filter is changing dramatically. You may have trouble estimating the amplitude of the output sinusoid at the higher frequencies, since there are fewer samples per cycle, although the CODEC data sheet claims to provide the proper reconstruction low-pass filtering, with a break frequency of Fs/2. Compare the observed frequency response with that predicted by MATLAB for this band-pass filter. Include both of these plots in your report memo as Attachment B. The experimentally measured frequency response plot of gain (in dB) vs. frequency should have the same shape as the MATLAB-predicted plot, however, it may differ by an additive constant, since the gain of the mixer box is not known (unless the mixer box is bypassed.)

---

[1] To make sure that the C67x C compiler produces well-optimized code, make sure that compiler optimization is enabled compiler optimization in Code Composer Studio 2 by clicking on Project – Build Options – Basic. Make sure that "Speed Most Critical" option is selected in the "Opt Speed vs. Size" box. Also, in the "Opt Level" box, choose "Register –o0". The "Generate Debug Info" box should normally have "Full Symbolic Debug" selected, when you are in the process of debugging your code, though I have found that program execution is substantially speeded up by changing this to "No Debug" after you code is debugged, allowing you to go to a higher sampling rate.

D. Now change the coefficients to those of Part I (B.2) above (for the high-pass filter). Hint: Create a different include file. (*Isn't it amazing that changing the coefficient values can so thoroughly change the "personality" of the filter!*) Then repeat Parts B and C. Include both the observed ***and*** the MATLAB predicted frequency response curves plotted on the same axis for this high-pass filter in your report memo as ***Attachment C***.

III. Fixed-Point FIR Filter Implementation

Modify the floating point FIR band-pass filtering program that you obtained in Part I (B) so it can be run on the (cheaper and faster) C62x DSP chip, which executes the (fixed-point) subset of the C67 instruction set. The C62x DSP has the same instructions and architecture (even the same pinout!) as the C67x DSP chip. Now the "rules of the game" have changed: you may no longer use any floating point variables or operations in your C program, since floating point operations cannot be efficiently executed by the C62.

Obviously, you will have to translate your filter coefficients into integer form. In order to take advantage of the full dynamic range of the C62's 32-bit integer representation, you should multiply these coefficients by 0x7FFF. This will convert (scale up) these original floating point coefficients into signed 16-bit integer quantities (of type "short int"), since the FIR coefficients of a unity gain filter range between (-1.0, 1.0). These (short int) coefficient values should be entered into your new program, and declared to be of the "short int" data type. In your filtering program, you should multiply by the signed 16-bit input sample in the lower half of the 32-bit integer coming in from the CODEC, corresponding to the right CODEC channel. You should use a "C compiler intrinsic function" which forces the use of a specific C62x/67x machine instruction that has no direct counterpart in the C language. The intrinsic C function that is "just what the doctor ordered" in this particular case is "_mpy" (multiply), and it may be invoked by a line of C code similar to this:

```
temp = _mpy(sample,coeff);
```

This line of C code forces the in-line insertion of the "MPY" C62x/C67x DSP chip's "16 X 16" integer multiply instruction. This instruction multiplies the 16 LSBs of the first (32-bit) integer argument (sample) by the 16 LSBs of second (32-bit) integer argument (coeff), and returns a 32-bit signed integer product to the integer variable "temp".

The upper 16 bits of this 32-bit variable "temp" holds the16-bit signed output sample. Note that taking this upper 16-bit portion of the 32-bit product in "temp" as the result corresponds to dividing the 32-bit result by 0x10000, since it lies in the upper 16 bits of the 32-bit integer. However, we want to divide by 0x8000, which almost corrects for the multiplication of the coefficients by 0x7FFF that we had to perform earlier, in order to scale the coefficients for use in this fixed-point (integer) version of the FIR filtering program. Therefore, one final right shift must be applied to our result to make the lower 16-bits of "temp" to be equivalent to the result divided by 0x8000.

```
temp = temp >> 15;
```

The 16-bit result is now in the proper position (lower-most 16 bits) to go out to the right channel of the CODEC. However, before sending the result (temp) out to the CODEC, the top 16 bits of temp should be masked out, to ensure that no residual noise will go out the CODEC on the left channel.

Demonstrate that your fixed-point FIR band-pass filter program passes frequencies with least attenuation in the range 800 Hz – 2.4 kHz . Obtain the instructor's validating signature on the program listing. Include the listing of your modified "fixed-point" FIR filter program listing as ***Attachment D***.

IV. More Efficient FIR Filter Implementation

When it comes to DSP speed is important. For this last part you are to either write a more efficient FIR routine, or use one from TI's library.

Write your own

Write, and then demonstrate, the proper operation of a more efficient FIR floating-point band-pass filtering program. As suggested earlier, a circular buffer should be used to replace the input sample storage array x[ ]. This modification would eliminate the need for the time-wasting "for loop" that updates x[ ] each time the interrupt service routine. (Of course, the most efficient FIR implementations would require use of C6x assembly-language coding.)

Use IT's

Look in G:\ee\yoder\ece581\TI library. Here you will find two zip files, *FixedPointLibrary.zip* and *FloatingPointLibrary.zip*. First decide whether you want to use the fixed-point or the floating-point version. Then copy of corresponding zip file to your directory. Unzip it and run the *exe* file. This will create a directory with a *docs* folder in it. Read the documentation in the *docs* folder and use the appropriate routine.

Either Way

Include the listing of your more efficient filter implementation as **Attachment E**, and on this listing, include comments on the CPU load for all of the parts of the lab.

Listing 1. Example Matlab m-file illustrating FIR filter design evaluation.

```
% Finite Impulse Response filter design example
% found in the MATLAB Signal Processing Toolbox
% using the MATLAB FIR1 function (M-file)

Fs=48e3;  %Specify Sampling Frequency
Ts=1/Fs; %Sampling period.
Ns=512;  %Nr of time samples to be plotted.

t=[0:Ts:Ts*(Ns-1)];  %Make time array that contains Ns elements
                     %t = [0, Ts, 2Ts, 3Ts,..., (Ns-1)Ts]
f1=500*6;
f2=1800*6;
f3=2000*6;
f4=3200*6;

x1=sin(2*pi*f1*t); %create sampled sinusoids at different frequencies
x2=sin(2*pi*f2*t);
x3=sin(2*pi*f3*t);
x4=sin(2*pi*f4*t);

x=x1+x2+x3+x4;  %Calculate samples for a 4-tone input signal
grid on;
N=16;            %FIR1 requires filter order (N) to be EVEN
                 %when gain = 1 at Fs/2.
W=[0.4 0.8];     %Specify Bandstop filter with stop band between
                 %0.4*(Fs/2) and 0.6*(Fs/2)

B=FIR1(N,W,'DC-1'); %Design FIR Filter using default (Hamming window.
B               %Leaving off semi-colon causes contents of
                %B (the FIR coefficients) to be displayed.

A=1;            %FIR filters have no poles, only zeros.

figure(1)
freqz(B,A,Ns,Fs);        %Plot frequency response - both amp and phase response.

soundsc(x, Fs);

% pause;        %User must hit any key on PC keyboard to go on.
figure(2);       %Create a new figure window, so previous one isn't lost.
subplot(2,1,1); %Two subplots will go on this figure window.
Npts=200;
plot(t(1:Npts),x(1:Npts)) %Plot first Npts of this 4-tone input signal
title('Time Plots of Input and Output');
xlabel('time (s)');
ylabel('Input Sig');
   %Now apply this filter to our 4-tone test sequence

y = filter(B,A,x);

subplot(2,1,2);  %Now go to bottom subplot.
plot(t(1:Npts),y(1:Npts)); %Plot first Npts of filtered signal.
xlabel('time (s)');
ylabel('Filtered Sig');
```

```
soundsc(y, Fs);

% pause;

figure(3);  %Create a new figure window, so previous one isn't lost.
subplot(2,1,1);
xfftmag=(abs(fft(x,Ns)));   %Compute spectrum of input signal.
xfftmagh=xfftmag(1:length(xfftmag)/2);
   %Plot only the first half of FFT, since second half is mirror imag
   %the first half represents the useful range of frequencies from
   %0 to Fs/2, the Nyquist sampling limit.
f=[1:1:length(xfftmagh)]*Fs/Ns;   %Make freq array that varies from
                                  %0 Hz to Fs/2 Hz.
plot(f,xfftmagh);       %Plot frequency spectrum of input signal
title('Input and Output Spectra');
xlabel('freq (Hz)');
ylabel('Input Spectrum');
subplot(2,1,2);
yfftmag=(abs(fft(y,Ns)));
yfftmagh=yfftmag(1:length(yfftmag)/2);
   %Plot only the first half of FFT, since second half is mirror image
   %the first half represents the useful range of frequencies from
   %0 to Fs/2, the Nyquist sampling limit.
plot(f,yfftmagh);       %Plot frequency spectrum of input signal
xlabel('freq (Hz)');
ylabel('Filt Sig Spectrum');
```

Listing 2. C-Language FIR Real-time Digital Filtering Program (lab02.c)

```
#include "coeff.h"
....
void processBuffer(void) {          // 6-16
      int *gBufferRcv, *gBufferXmt;       // Points to the ping or pong buffer.
      Uint32 pingPong = SWI_getmbox();    // 6-16

      if(pingPong == PING) {  // Make sure we're pointing to the right buffer.
            gBufferRcv = gBufferRcvPing;
            gBufferXmt = gBufferXmtPing;
      } else {
            gBufferRcv = gBufferRcvPong;
            gBufferXmt = gBufferXmtPong;
      }

      if(DIP_get(DIP_1) == 0)
            addSine(gBufferRcv, BUFFSIZE);
      if(DIP_get(DIP_3) == 0)
            FIRfilter(gBufferRcv, gBufferXmt, coeff, BUFFSIZE, NUMCOEFFS-1);
      else
            copyData(gBufferRcv, gBufferXmt, BUFFSIZE);
}

/*
      The following code is run by the period timer.  It checks the
      MSB (not the sign bit).  If it is 1, it turns the LED on.
      This is a way to see if there is clipping.
*/
void blinkLeds(void) {
```

```
      int i;
      short lmax=0, ltmp;
      short rmax=0, rtmp;

      for(i=0; i<BUFFSIZE; i++) {
            ltmp = (short) _abs(gBufferRcvPing[i]>>16);
            rtmp = (short) _abs(gBufferRcvPing[i]);
            lmax = lmax<ltmp ? ltmp: lmax;
            rmax = rmax<rtmp ? rtmp: rmax;
      }
      LOG_printf(&logTrace, "blinkLeds:  max = %d, %d", lmax, rmax);
      // Turn LED on if MSB (not sign bit) is on.
      if(lmax & 0x4000) LED_on(LED_1); else LED_off(LED_1);
      if(rmax & 0x4000) LED_on(LED_3); else LED_off(LED_3);
}

/*
      This routine convolves the present and the Norder previous input
      samples with the Norder+1 FIR coefficients  h(0) through h(Norder):

            y(n) = h(0)*x(n) + h(1)*x(n-1) + ... + h(Norder)*x(n-Norder)
*/

void FIRfilter(int *inbuf, int *outbuf, float *coeff, int buffSize,
                        int Norder) {
    int intsamp,i,j;
    float floatsamp,sum;
    static float x[NUMCOEFFS];

for(j=0; j<buffSize; j++) {

    intsamp=inbuf[j];                       /* read the input */
    intsamp = intsamp & 0xffff;             /* Mask right channel data (bottom 16 bits)
*/
    if(intsamp & 0x8000)
        intsamp = intsamp | 0xffff0000;     /* Sign extend right channel data */
    floatsamp = (float) intsamp;            /* Convert right channel to floating point */

/*
      Update past input sample array, where x[0] holds present sample, x[1] holds
      sample from 1 sample period ago, x[N] holds sample from N sampling periods ago.

      This time-consuming loop could be eliminated if circular buffering were
      to be employed.
*/
    for(i=Norder; i >= 0;i--)
            x[i]=x[i-1];
    x[0] = floatsamp;
    sum = 0.0;
    for(i=0; i<=Norder; i++)                /* Perform FIR filtering (convolution) */
            sum=sum+x[i]*coeff[i];

    intsamp = ((int) sum) & 0xffff;         /* Convert result back to integer form.   */
    outbuf[j] = intsamp << 16 | intsamp;    /* Send to buffer (both channels )  */
      }
}
```

Listing 3. Include file with filter coefficients (coeff.h)

```
/* Band-Stop Filter Coefficients Cut From MATLAB
    B =
  Columns 1 through 7
   -0.0038    0.0000    0.0218    0.0000   -0.0821    0.0000    0.1625
  Columns 8 through 14
    0.0000    0.8031    0.0000    0.1625    0.0000   -0.0821    0.0000
  Columns 15 through 17
    0.0218    0.0000   -0.0038
  */

/* Here are the 17 coefficients corresponding   */
/* to a 16th-order FIR band reject FIR filter   */
/* FIR filter obtained using MATLAB FIR1        */
/* Fs = 48 kHz, rejects 9.6k to 14.4KHz               */

#define NUMCOEFFS 17
float coeff[NUMCOEFFS] = {
   -0.0038,  0.0000,  0.0218,  0.0000, -0.0821,  0.0000,  0.1625,
    0.0000,  0.8031,  0.0000,  0.1625,  0.0000, -0.0821,  0.0000,
    0.0218,  0.0000, -0.0038
};
```