EECS 152B / CSE 135B

DSP Design & Laboratory

Assignment 4

Frequency Domain Filtering using TI c6713 DSK

Winter 2017

Diego Bravo

Hector Solano

Paul Dao

Lab Session:

Lab Dates: 03/1/2017, 03/8/2017, and 03/15/2017

Introduction

This lab begins with giving us C code for ready implementation of a three-band equalizer band which we must modify for proper frequency-domain filtering; the three bands are a lowpass, bandpass, and a highpass filter. The first part of the lab begins by making the three-band equalizer into a four-band equalizer so that summation of the filters in the time-domain results in a unity gain filter in the frequency-domain. We do so by adding an additional bandpass called “mid2”. Once “mid2” bandpass is accounted for, filtering in the frequency-domain will be implemented using Overlap-Add and Overlap-Save methods for the second and third part of the lab. Below are the preprocessor directives (code before the main() function) that will be used for every part of this assignment. The filter coefficients are determined using MATLAB fir1() functions of order 60 and the code is included in the questions section of the report. The FIR filter coefficients that are included in the "GraphicEQcoeff.h" header file are as follows:

```
float lpcoeff[61] =
{-0.00084949,-0.00064097,8.7828e-19,0.00085525,0.0014674,0.0012761,-1.63
71e-18,-0.0019407,-0.0033622,-0.002892,3.0234e-18,0.0041838,0.0070411,0.
0058879,-4.7977e-18,-0.0081092,-0.013381,-0.011014,6.653e-18,0.014889,0.
024529,0.020282,-8.2686e-18,-0.028377,-0.048429,-0.042275,9.3652e-18,0.0
73395,0.15768,0.22469,0.2502,0.22469,0.15768,0.073395,9.3652e-18,-0.0422
75,-0.048429,-0.028377,-8.2686e-18,0.020282,0.024529,0.014889,6.653e-18,
-0.011014,-0.013381,-0.0081092,-4.7977e-18,0.0058879,0.0070411,0.0041838
,3.0234e-18,-0.002892,-0.0033622,-0.0019407,-1.6371e-18,0.0012761,0.0014
674,0.00085525,8.7828e-19,-0.00064097,-0.00084949};

float bp1coeff[61] =
{0.00085108,0.0015503,-2.6398e-18,-0.0020686,-0.0014702,0.00052957,-1.64
01e-18,-0.00080535,0.0033686,0.0069951,-9.0874e-18,-0.01012,-0.0070543,0
.0024434,-4.8067e-18,-0.0033652,0.013406,0.02664,-1.9997e-17,-0.036013,-
0.024575,0.008417,-8.2842e-18,-0.011776,0.04852,0.10225,-2.8148e-17,-0.1
7752,-0.15797,0.093243,0.25066,0.093243,-0.15797,-0.17752,-2.8148e-17,0.
10225,0.04852,-0.011776,-8.2842e-18,0.008417,-0.024575,-0.036013,-1.9997
e-17,0.02664,0.013406,-0.0033652,-4.8067e-18,0.0024434,-0.0070543,-0.010
12,-9.0874e-18,0.0069951,0.0033686,-0.00080535,-1.6401e-18,0.00052957,-0
.0014702,-0.0020686,-2.6398e-18,0.0015503,0.00085108};

float bp2coeff[61] =
{0.00085108,-0.0015503,7.5296e-19,0.0020686,-0.0014702,-0.00052957,-1.64
01e-18,0.00080535,0.0033686,-0.0069951,3.2721e-17,0.01012,-0.0070543,-0.
0024434,-4.8067e-18,0.0033652,0.013406,-0.02664,3.3328e-17,0.036013,-0.0
24575,-0.008417,-8.2842e-18,0.011776,0.04852,-0.10225,4.6914e-17,0.17752
,-0.15797,-0.093243,0.25066,-0.093243,-0.15797,0.17752,4.6914e-17,-0.102
25,0.04852,0.011776,-8.2842e-18,-0.008417,-0.024575,0.036013,3.3328e-17,
-0.02664,0.013406,0.0033652,-4.8067e-18,-0.0024434,-0.0070543,0.01012,3.
2721e-17,-0.0069951,0.0033686,0.00080535,-1.6401e-18,-0.00052957,-0.0014
702,0.0020686,7.5296e-19,-0.0015503,0.00085108};
```

```
float hpcoeff[61] =
{-0.00084949,0.00064097,-2.5081e-18,-0.00085525,0.0014674,-0.0012761,-1.
6371e-18,0.0019407,-0.0033622,0.002892,-3.8706e-17,-0.0041838,0.0070411,
-0.0058879,-4.7977e-18,0.0081092,-0.013381,0.011014,-4.6571e-17,-0.01488
9,0.024529,-0.020282,-8.2686e-18,0.028377,-0.048429,0.042275,-6.5556e-17
,-0.073395,0.15768,-0.22469,0.2502,-0.22469,0.15768,-0.073395,-6.5556e-1
7,0.042275,-0.048429,0.028377,-8.2686e-18,-0.020282,0.024529,-0.014889,-
4.6571e-17,0.011014,-0.013381,0.0081092,-4.7977e-18,-0.0058879,0.0070411
,-0.0041838,-3.8706e-17,0.002892,-0.0033622,0.0019407,-1.6371e-18,-0.001
2761,0.0014674,-0.00085525,-2.5081e-18,0.00064097,-0.00084949};
```

PREPROCESSOR DIRECTIVES

All code in red has been modified from the original code or has been added. The preprocessor directives will be the same for Problem 1, Problem 2, & Problem 3.

```
#include "dsk6713_aic23.h"
#include "dsk6713_led.h"
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINEIN 0x0011
Uint32 fs = DSK6713_AIC23_FREQ_16KHZ;                    // 1
Uint16 inputsource = DSK6713_AIC23_INPUT_LINEIN;         // 0x011
#include <math.h>
#define PI 3.14159265358979
#define PTS 256                                          // number of FFT points
#define RADIX 2                                          // Division radix, FFT related
#define DELTA (2*PI)/PTS                                 // FFT frequency resolution
#define S 196                                            // L or block size
#define OVERLAP_LENGTH 61                                // length to be added/saved

typedef struct Complex_tag {float real,imag;} COMPLEX;   // structure to represent
                                                         // complex numbers
#pragma DATA_ALIGN(W,sizeof(COMPLEX))    // For FFT assembly code to work properly
                                         // the involved parameters should
                                         // be aligned to 8-byte memory locations.
                                         // This tells compiler to exactly do that
#pragma DATA_ALIGN(outWin,sizeof(COMPLEX))

#pragma DATA_ALIGN(h,sizeof(COMPLEX))
#pragma DATA_ALIGN(bass,sizeof(COMPLEX))
#pragma DATA_ALIGN(mid1,sizeof(COMPLEX))
```

```
#pragma DATA_ALIGN(mid2,sizeof(COMPLEX)) // aligns memory location for new array of
complex numbers, whose real components will represent the coefficients of the 2nd band
pass filter
#pragma DATA_ALIGN(treble,sizeof(COMPLEX))
COMPLEX W[PTS/RADIX] ;                          // twiddle array, used in FFT
COMPLEX outWin[PTS];                            // output window
COMPLEX h[PTS];                                 // freq domain filter

#include "GraphicEQcoeff.h"                     // time-domain FIR coefficients
COMPLEX bass[PTS], mid1[PTS], mid2[PTS], treble[PTS];      // individual equalizer filters,
                                                // frequency domain
float bass_gain = 1.0;                          // Gains to be applied to each filter.
                                                //  Adjustable by DIP switches later
float mid1_gain = 1.0;
float mid2_gain = 1.0;
float treble_gain = 1.0;
float iobuffer[S];                      // primary input/output buffer //S used to be
PTS/2
float overlap[OVERLAP_LENGTH];          // Vector to hold tail
short i;                                // general index variable
float a, b;                             // Temp variables for complex multiply
short NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);   // # of filter coefficients in filters
short iTwid[PTS/2]={0} ;                // Twiddle index array. Used in FFT

#include "digitrev_index.h"             // definition of digitrev_index function,
                                        // used for FFT
#include "LED_fun.h"                    // function to check dip switch status, turn
                                        // LEDs on and off, and set filter gains
```

Problem 1

OBJECTIVE & PROBLEM

Set up the DIP switches on the DSP board for frequency domain filtering by modifying the given code. More specifically, implementing the second bandpass to the code, called "mid2", in the preprocessor directives, the main function, and the "LED_fun.h" header file.

BRIEF EXPLANATION OF THE CODE

Before starting the polling process, we store all of the FIR coefficients generated in Matlab in arrays called "lpcoeff","bp1coeff","bp2coeff","hpcoeff" for the lowpass, two bandpass, and the highpass filters. We produce these coefficient vectors by calling the fir1 function and giving each order of 60 (i.e. length 61)**.** The following C code begins with filling the Twiddle array, which is used for FFT, with 0s. After assigning 0.0s to the real and imaginary attributes of the COMPLEX items in the "bass," "mid1," "mid2," and "treble" arrays with 0s, we assign all the coefficients from "lpcoeff," "bp1coeff," "bp2coeff," and "hpcoeff" to the first NUMCOEFFS items' real components of the "bass," "mid1," "mid2," and "treble" arrays. We initiate the Overlap-Add method by converting the time-domain values in "bass," "mid1," "mid2," and "treble" to frequency-domain values by calling "cfftr2_dit." Before entering the while(1) loop, we start the polling process by calling the comm_poll() function from the Vectors_poll.asm assembly code file. In the while(1) loop, we output all samples in the "iobuffer" and replace them with new, incoming samples. The new samples in "iobuffer" are copied over to the real attributes of the first PTS/2 COMPLEX items in "outWin," and the real attributes of the last PTS/2 items are filled with 0s. All of the imaginary parts of the items in "outWin" are set to 0.0. After changing the time-domain values in "outWin" to frequency-domain values by calling "cfftr_dit" on "outWin," we multiply the "bass_gain," "mid1_gain," "mid2_gain," and "treble_gain" with their respective polynomials, which is represented by the coefficients stored in the "bass," "mid1," "mid2," and "treble" arrays. We add the polynomials together after applying their respective gains and store the new coefficients in the "h" array, which represents our impulse response. Then, we multiply the FFT values of h and outWin together to get the output FFT values (i.e. X[k]*H[k]=Y[k]) and store the real and imaginary components of 256 COMPLEX items in outWin. We revert the frequency domain values of outWin to time-domain values by calling "icfftr_div" on outWin. The first PTS/2 = 256/2=128 COMPLEX numbers in outWin are added with "overlap" values at their respective positions to be outputted at the top of the while(1) loop. The while(1) loop ends with copying the last PTS/2 = 128 COMPLEX values into "overlap."

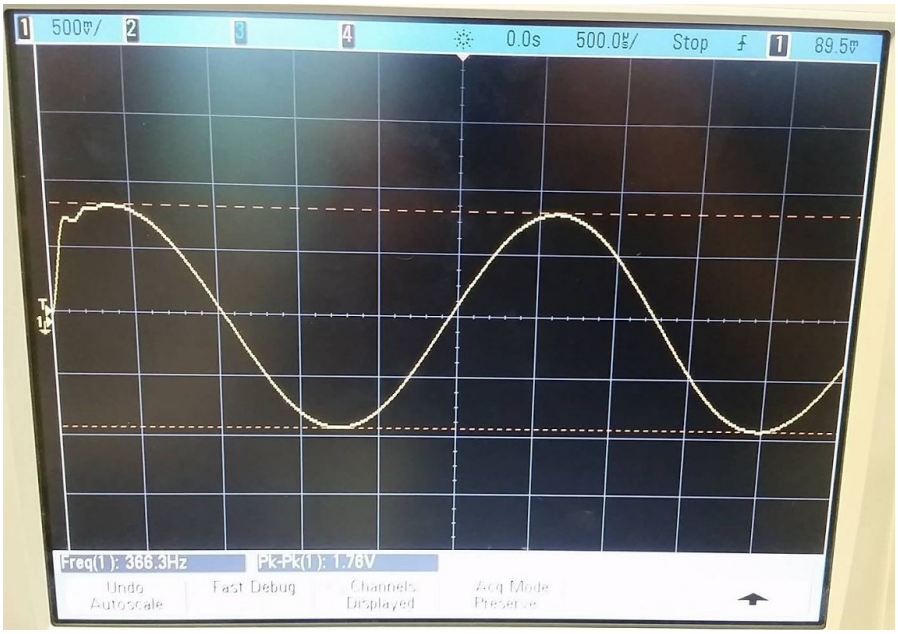
CODE

```
void main()
{
        for(i=0;i<PTS/2;i++){iTwid[i]=0;}

        digitrev_index(iTwid, PTS/RADIX, RADIX);
        for( i = 0; i < PTS/RADIX; i++ ) {
           W[i].real = cos(DELTA*i);
           W[i].imag = sin(DELTA*i);
        }
        bitrev(W, iTwid, PTS/RADIX);
        for (i=0 ; i<PTS ; i++) {
           bass[i].real = 0.0;      bass[i].imag = 0.0;
           mid1[i].real = 0.0;      mid1[i].imag = 0.0;
           mid2[i].real = 0.0;      mid2[i].imag = 0.0;    //initialize all COMPLEX components to
be 0.0
           treble[i].real = 0.0;      treble[i].imag = 0.0;
        }
        for (i=0; i<NUMCOEFFS; i++)
        {
           bass[i].real = lpcoeff[i];
           mid1[i].real = bp1coeff[i];
           mid2[i].real = bp2coeff[i];    //Store the 2nd band pass' coefficients into the real
components of the 1st NUMCOEFFS items of COMPLEX type
           treble[i].real = hpcoeff[i];
        }
        cfftr2_dit(bass,W,PTS);
        cfftr2_dit(mid1,W,PTS);
        cfftr2_dit(mid2,W,PTS);    //Execute FFT on mid2
        cfftr2_dit(treble,W,PTS);

        DSK6713_LED_init();
        DSK6713_DIP_init();
        //comm_intr();
        comm_poll();


        while(1) {
           output_sample(input_sample());
```

```
            LED_fun();
            for(i=0;i<PTS/2;i++){
               output_left_sample((short)(iobuffer[i]));
               iobuffer[i] = (float)(input_left_sample());
            }
         // follwoing two lines are used for interrupt. Remember to comment the for loop above
         //while (flag == 0);       //wait for iobuffer to fill up
         //         flag = 0;

            for (i=0 ; i<PTS/2 ; i++)   {outWin[i].real = iobuffer[i];}
            for (i=PTS/2 ; i<PTS ; i++) {outWin[i].real = 0;}
            for (i=0 ; i<PTS ; i++)         {outWin[i].imag = 0.0;}
            cfftr2_dit(  outWin,W,PTS);
            for (i=0 ; i<PTS ; i++) {
               h[i].real =           bass[i].real*bass_gain  +  mid1[i].real*mid1_gain
                           +  mid2[i].real*mid2_gain + treble[i].real*treble_gain;
               h[i].imag =           bass[i].imag*bass_gain  +  mid1[i].imag*mid1_gain
                           +  mid2[i].imag*mid2_gain  + treble[i].imag*treble_gain;
            }    //Multiply the real and imaginary parts of each COMPLEX item in the bass, mid1,
                 // mid2, and treble arrays with their respective gains. Then, sum those products
                 // together to get the coefficient for the impulse response
            for (i=0; i<PTS; i++) {
               a = outWin[i].real;
               b = outWin[i].imag;
               outWin[i].real = h[i].real*a - h[i].imag*b;
               outWin[i].imag = h[i].real*b + h[i].imag*a;
            }
            icfftr2_dif(outWin,W,PTS);
            for (i=0 ; i<PTS ; i++) {outWin[i].real /= PTS;}

            for(i=0;i<PTS/2;i++){
               iobuffer[i]=outWin[i].real+overlap[i];
               overlap[i]=outWin[i+PTS/2].real;
            }
      }
}
```

RESULTS & CONCLUSION

Using the oscilloscope and DIP switches to control what filters are active, we find that no matter what filter or filters we use, if we feed a signal to the DSP board that the filter should accept, we get a distorted signal. This implies that the Overlap-Add method originally given is implemented incorrectly, and we have to tweak it, as we did in Problem 2.



Testing LPF with DIP Switches

Problem 2

OBJECTIVE & PROBLEM

Implement Overlap-Add method for filtering and using the DIP switches on the DSP board to select the type of filtering to be carried out.

BRIEF EXPLANATION OF THE CODE

Before starting the polling process, we store all of the FIR coefficients generated in Matlab in arrays called “lpcoeff”,”bp1coeff”,”bp2coeff”,”hpcoeff” for the lowpass, two bandpass, and the highpass filter. We produce these coefficient vectors by calling the fir1 function and giving each order of 60 (i.e. length 61)**.**

The following C code roughly follows the code in Problem 1 but with some tweaks. In addition to filling the iTwid array with 0s, we also fill our overlap array with OVERLAP_LENGTH = 61 0’s. Everything before the while(1) loop is otherwise the same as in Problem 1. Then, in the while(1) loop, we copy the values in iobuffer into the real attribute of the first S=196 COMPLEX items in the outWin array. The last

61 COMPLEX items' attributes in outWin are set to 0, just like the first 196 COMPLEX items' imaginary attribute are set to 0. After taking the product of the FFT values of the input and the impulse response and converting the result back into time-domain values, we loop through the overlap array as many times as its length to apply the overlap to the first 61 samples of the input block. We next store the result in the first OVERLAP_LENGTH slots of the iobuffer and store the last OVERLAP_LENGTH samples of the input block into overlap for the next iteration of the while(1) loop. Everything inside the while(1) loop that has not been mentioned previously is the same as in Problem 1.

CODE

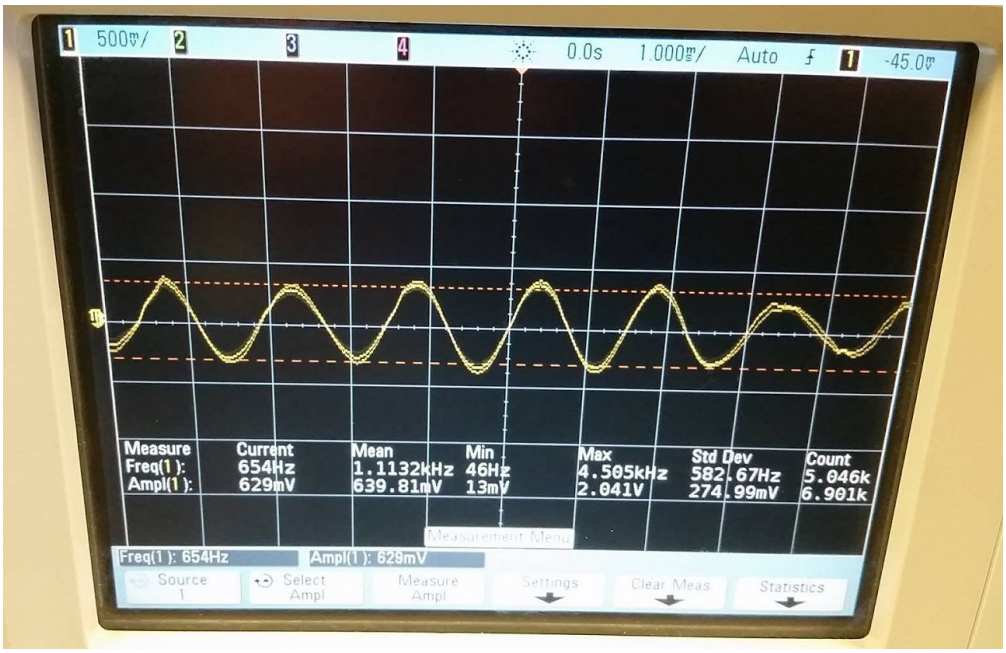
```
//Overlap-Add
void main()
{
        for(i=0;i<PTS/2;i++){iTwid[i]=0;}
        for(i=0;i<OVERLAP_LENGTH;++i){overlap[i]=0;}    //initialize overlap array to carry all
0s
        digitrev_index(iTwid, PTS/RADIX, RADIX);
        for( i = 0; i < PTS/RADIX; i++ ) {
           W[i].real = cos(DELTA*i);
           W[i].imag = sin(DELTA*i);
        }
        bitrev(W, iTwid, PTS/RADIX);
        for (i=0 ; i<PTS ; i++) {
           bass[i].real = 0.0;       bass[i].imag = 0.0;
           mid1[i].real = 0.0;       mid1[i].imag = 0.0;
           mid2[i].real = 0.0;       mid2[i].imag = 0.0;    //initialize all COMPLEX components to
be 0.0
           treble[i].real = 0.0;      treble[i].imag = 0.0;
        }
        for (i=0; i<NUMCOEFFS; i++)
        {
           bass[i].real = lpcoeff[i];
           mid1[i].real = bp1coeff[i];
           mid2[i].real = bp2coeff[i];    //Store the 2nd band pass' coefficients into the real
components of the 1st NUMCOEFFS items of COMPLEX type
           treble[i].real = hpcoeff[i];
        }
        cfftr2_dit(bass,W,PTS);
        cfftr2_dit(mid1,W,PTS);
        cfftr2_dit(mid2,W,PTS);    //Execute FFT on mid2
        cfftr2_dit(treble,W,PTS);
```

```
        DSK6713_LED_init();
        DSK6713_DIP_init();
        //comm_intr();
        comm_poll();

        while(1) {
           LED_fun();
           for(i=0;i<S;i++){    //Instead of making a block of length half of PTS, make the size of the block, or "iobuffer," equal to PTS - NUMCOEFFS + 1 = 256 - 61 + 1 = S = 196. Then, output all samples stored in the iobuffer and replace them with new incoming samples.
              output_left_sample((short)(iobuffer[i]));
              iobuffer[i] = (float)(input_left_sample());
           }
        // follwoing two lines are used for interrupt. Remember to comment the for loop above
        //while (flag == 0);       //wait for iobuffer to fill up
        //         flag = 0;

           for (i=0 ; i<S ; i++)   {outWin[i].real = iobuffer[i];}    //Assign all S number of input samples to outWin
           for (i=S ; i<PTS ; i++) {outWin[i].real = 0;}    //Assign 0s to the last PTS - S + 1 slots
           for (i=0 ; i<PTS ; i++)         {outWin[i].imag = 0.0;}
           cfftr2_dit(  outWin,W,PTS);
           for (i=0 ; i<PTS ; i++) {
              h[i].real =          bass[i].real*bass_gain  +  mid1[i].real*mid1_gain
                        +  mid2[i].real*mid2_gain + treble[i].real*treble_gain;
              h[i].imag =          bass[i].imag*bass_gain  +  mid1[i].imag*mid1_gain
                        +  mid2[i].imag*mid2_gain  + treble[i].imag*treble_gain;
           }    //Multiply the real and imaginary parts of each COMPLEX item in the bass, mid1, mid2, and treble arrays with their respective gains. Then, sum those products together to get the coefficient for the impulse response
           for (i=0; i<PTS; i++) {
              a = outWin[i].real;
              b = outWin[i].imag;
              outWin[i].real = h[i].real*a - h[i].imag*b;
              outWin[i].imag = h[i].real*b + h[i].imag*a;
           }
           icfftr2_dif(outWin,W,PTS);
           for (i=0 ; i<PTS ; i++) {outWin[i].real /= PTS;}
        // Overlap Add Method
           for(i=0;i<OVERLAP_LENGTH;i++){
              iobuffer[i]=outWin[i].real+overlap[i];
              overlap[i]=outWin[i + S].real;
```

```
            }    //Loop through overlap array as many times as its length to apply the overlap to the first OVERLAP_LENGTH = 61 samples of the input block, and store the result in the first OVERLAP_LENGTH slots of the iobuffer, which will be outputted in the next iteration of the while(1) loop. Store the last OVERLAP_LENGTH samples of the input block into overlap for the next iteration of the while(1) loop
            for(i=OVERLAP_LENGTH;i<S;++i)
                iobuffer[i] = outWin[i].real;    //Store the samples from index OVERLAP_LENGTH to index S - 1 in the last S - OVERLAP_LENGTH = 196 - 61 = 135 slots of the iobuffer array
        }
}
```

RESULTS & CONCLUSION

Using the oscilloscope and DIP switches to control what filters are active, we found that the low-pass filter allowed signals whose frequencies ranged from about 0 to 2000 Hz give or take; the 1st band-pass filter allowed signals whose frequencies ranged from about 2000 to 4000 Hz give or take; the 2nd band-pass filter allowed signals whose frequencies ranged from about 4000 to 6000 Hz give or take; and the high-pass filter allowed signals whose frequencies ranged from about 6000 to 8000 Hz give or take. We know this because if we turned on certain filter(s), signals within those filter(s)' frequency ranges were displayed on the oscilloscope as clear, yet fluctuating, sinusoids, and signals outside of those filter(s)' frequency ranges were shown as noise.



Lowpass Filter is tested at 654Hz for Overlap Add Filtering

Problem 3

OBJECTIVE & PROBLEM

Implement Overlap-Save method for filtering and using the DIP switches on the DSP board to select the type of filtering to be carried out.

BRIEF EXPLANATION OF THE CODE

The following C code roughly follows the code in Problem 2 but with some tweaks, primarily in the while(1) loop. We start the while(1) loop by applying the overlap samples to the real values of the first 61 COMPLEX objects in the outWin array. Then, we copy the values in the iobuffer into the real values of the last 196 COMPLEX values of the outWin array. The code up to the 2nd to last for loop in the while(1) loop is the same as in Problem 2. After taking the product of the FFT of the outWin and the FFT of the impulse response, represented by the h array, and applying the IFFT on that product, we save the last 61 values in the iobuffer and store them in the overlap array for use in the next iteration of the while(1) loop. Finally, we save the last 196 real values of the last 61 COMPLEX values in outWin in the iobuffer to be outputted in the next iteration of the while(1) loop.

CODE

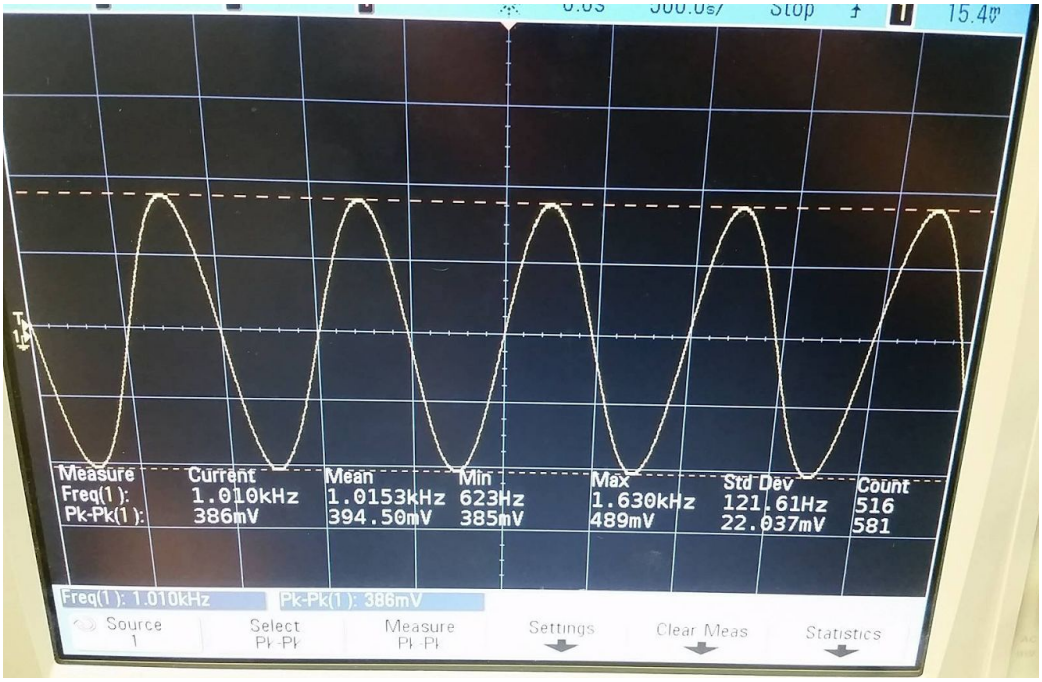
```
// Overlap Save
void main()
{
        for(i=0;i<PTS/2;i++){iTwid[i]=0;}
        for(i=0;i<OVERLAP_LENGTH;++i){overlap[i]=0;}    //initialize overlap array to carry all
0s
        digitrev_index(iTwid, PTS/RADIX, RADIX);
        for( i = 0; i < PTS/RADIX; i++ ) {
           W[i].real = cos(DELTA*i);
           W[i].imag = sin(DELTA*i);
        }
        bitrev(W, iTwid, PTS/RADIX);
        for (i=0 ; i<PTS ; i++) {
           bass[i].real = 0.0;       bass[i].imag = 0.0;
           mid1[i].real = 0.0;       mid1[i].imag = 0.0;
           mid2[i].real = 0.0;       mid2[i].imag = 0.0;    //initialize all COMPLEX components to
be 0.0
           treble[i].real = 0.0;      treble[i].imag = 0.0;
```

```
        }
        for (i=0; i<NUMCOEFFS; i++)
        {
           bass[i].real = lpcoeff[i];
           mid1[i].real = bp1coeff[i];
           mid2[i].real = bp2coeff[i];    //Store the 2nd band pass' coefficients into the real
components of the 1st NUMCOEFFS items of COMPLEX type
           treble[i].real = hpcoeff[i];
        }
        cfftr2_dit(bass,W,PTS);
        cfftr2_dit(mid1,W,PTS);
        cfftr2_dit(mid2,W,PTS);    // Execute FFT on mid2
        cfftr2_dit(treble,W,PTS);

        DSK6713_LED_init();
        DSK6713_DIP_init();
        //comm_intr();
        comm_poll();

        while(1) {
           LED_fun();
           for(i=0;i<S;i++){
              output_left_sample((short)(iobuffer[i]));
              iobuffer[i] = (float)(input_left_sample());
           }    //Instead of making a block of length half of PTS, make the size of the block, or
"iobuffer," equal to PTS - NUMCOEFFS + 1 = 256 - 61 + 1 = S = 196. Then, output all
samples stored in the iobuffer and replace them with new incoming samples.
           // follwoing two lines are used for interrupt. Remember to comment the for loop
above
           //while (flag == 0);   //wait for iobuffer to fill up
           //      flag = 0;
           for(i = 0; i< OVERLAP_LENGTH; i++) {outWin[i].real = overlap[i];}    //Pad the
outWin array with the overlap samples
           for (i=OVERLAP_LENGTH ; i<PTS ; i++)   {outWin[i].real =
iobuffer[i-OVERLAP_LENGTH];}    //After padding outWin with overlap samples, copy all the
input samples into the real attributes of the last 196 COMPLEX items in outWin
//         for (i=S ; i<PTS ; i++) {outWin[i].real = 0;}
           for (i=0 ; i<PTS ; i++)         {outWin[i].imag = 0.0;}
           cfftr2_dit(outWin,W,PTS);
           for (i=0 ; i<PTS ; i++) {
              h[i].real =          bass[i].real*bass_gain  +  mid1[i].real*mid1_gain
                        +  mid2[i].real*mid2_gain + treble[i].real*treble_gain;
```

```
                h[i].imag =          bass[i].imag*bass_gain  +  mid1[i].imag*mid1_gain
                            +  mid2[i].imag*mid2_gain + treble[i].imag*treble_gain;
            }    //Multiply the real and imaginary parts of each COMPLEX item in the bass,
mid1, mid2, and treble arrays with their respective gains. Then, sum those products together
to get the coefficient for the impulse response
            for (i=0; i<S; i++) {
                a = outWin[i].real;
                b = outWin[i].imag;
                outWin[i].real = h[i].real*a - h[i].imag*b;
                outWin[i].imag = h[i].real*b + h[i].imag*a;
            }
            icfftr2_dif(outWin,W,PTS);
            for (i=0 ; i<PTS ; i++) outWin[i].real /= PTS;

            for (i=0; i < OVERLAP_LENGTH; ++i)
                overlap[i] = iobuffer[S-OVERLAP_LENGTH+i];    //Copy the last 61 samples in
iobuffer to overlap, which will be used in the next iteration of the while(1) loop
            for(i = 0; i < S; ++i)
                iobuffer[i] = outWin[i+OVERLAP_LENGTH].real;    //Copy the last 196 samples'
real values of outWin to iobuffer, which will be outputted in the next iteration of the while(1)
loop
        }
}
```

RESULTS & CONCLUSION

Using the oscilloscope and DIP switches to control what filters are active, we found that the low-pass filter allowed signals whose frequencies ranged from about 0 to 2000 Hz give or take; the 1st band-pass filter allowed signals whose frequencies ranged from about 2000 to 4000 Hz give or take; the 2nd band-pass filter allowed signals whose frequencies ranged from about 4000 to 6000 Hz give or take; and the high-pass filter allowed signals whose frequencies ranged from about 6000 to 8000 Hz give or take. We know this because if we turned on certain filter(s), signals within those filter(s)' frequency ranges were displayed on the oscilloscope as clear, yet fluctuating, sinusoids, and signals outside of those filter(s)' frequency ranges were shown as noise.



Lowpass Filter is tested at 1010Hz for Overlap Save Filtering

Questions

Question 1: Include the modified C code for each of the three parts. Be sure to comment & annotate where you have made your changes. [[10]]

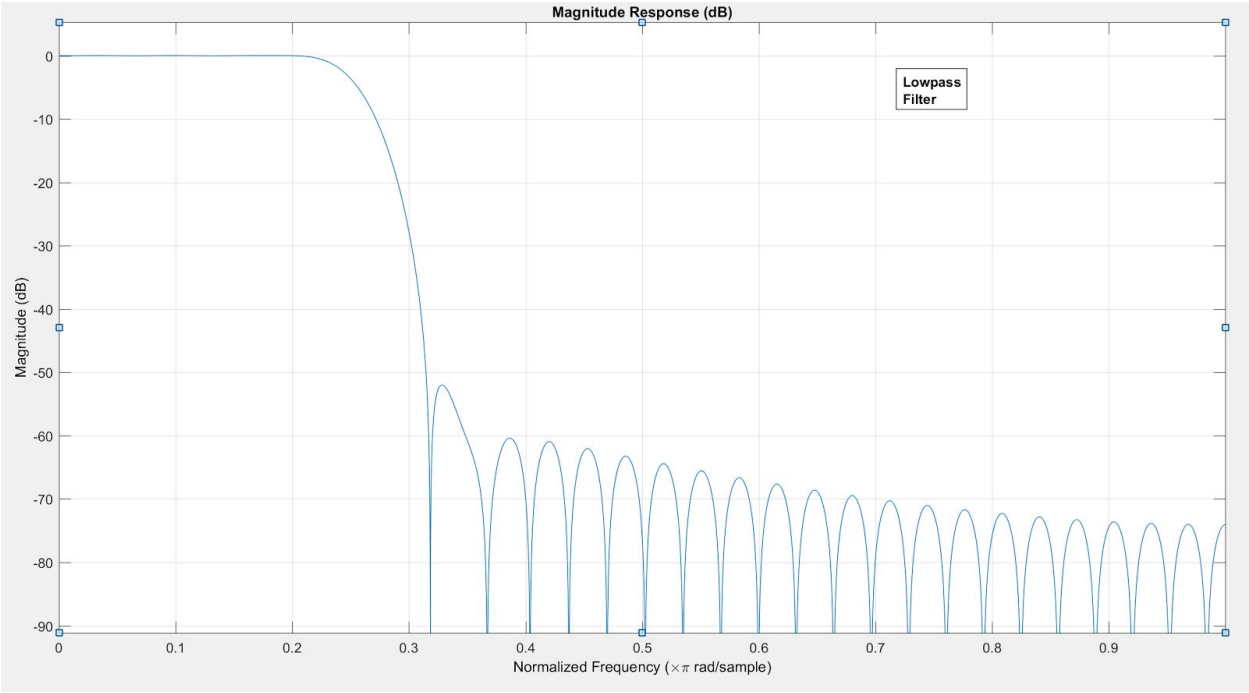
A: The modified code has been extensively commented in the Problems above.

Question 2: Using MATLAB, document the frequency response of each individual filter, as well as their combined response. Verify these findings with the oscilloscope. [[30]]

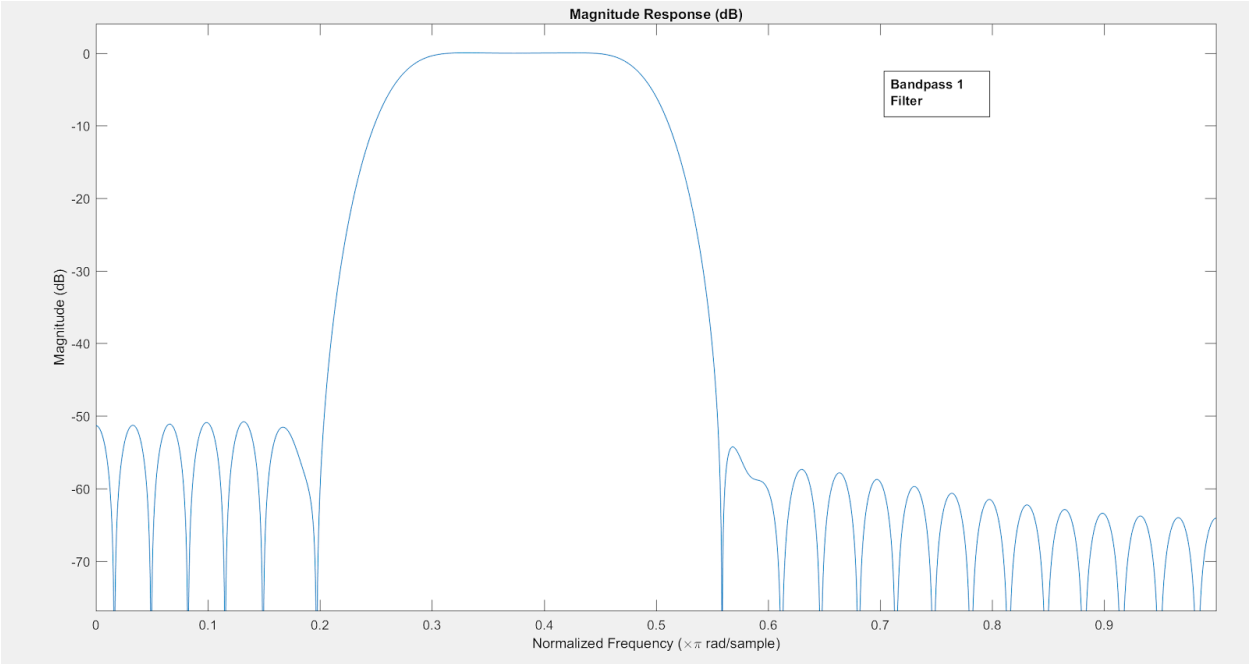
A: The following MATLAB code generates the coefficients for the FIR filters which is then imported to the C code used for filtering.

MATLAB CODE

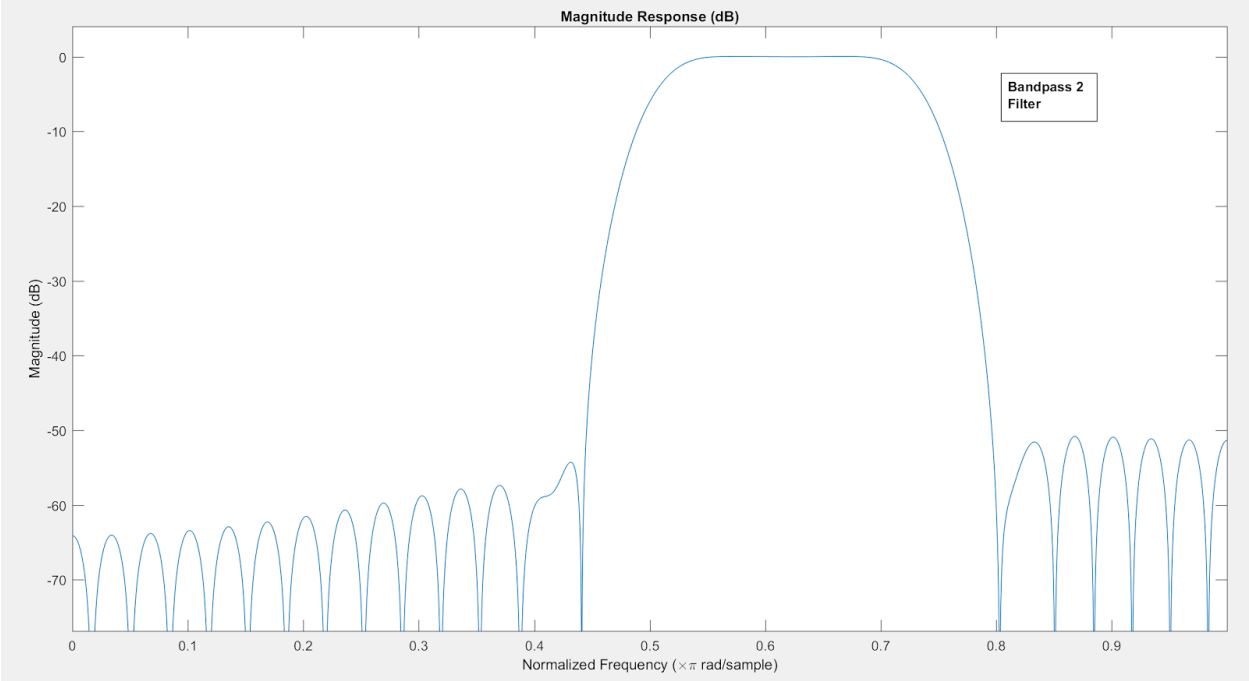
```
order=60;
lp=fir1(order,0.26,'low');
bp1=fir1(order,[.26 .5]);
bp2=fir1(order,[.5 .74]);
hp=fir1(order,0.74,'high');
h=lp+bp1+bp2+hp;
fvtool(lp); fvtool(bp1); fvtool(bp2); fvtool(hp);
fvtool(h)
```



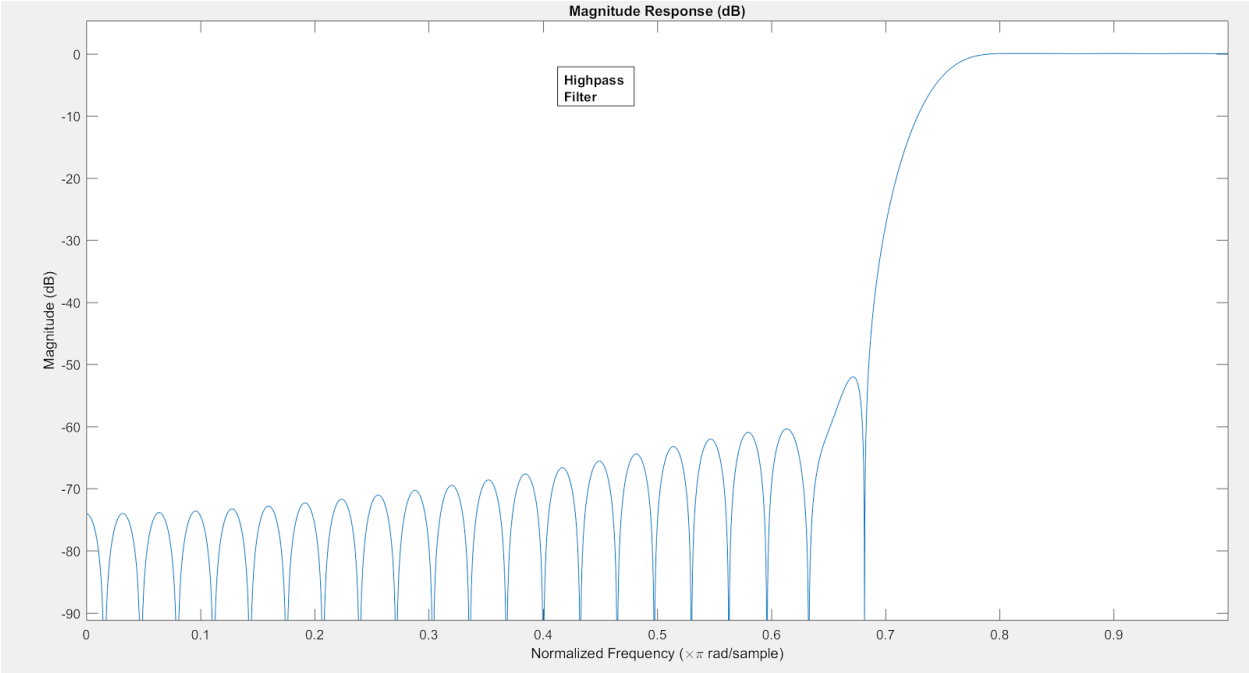
Lowpass Frequency Response H_lp(w)



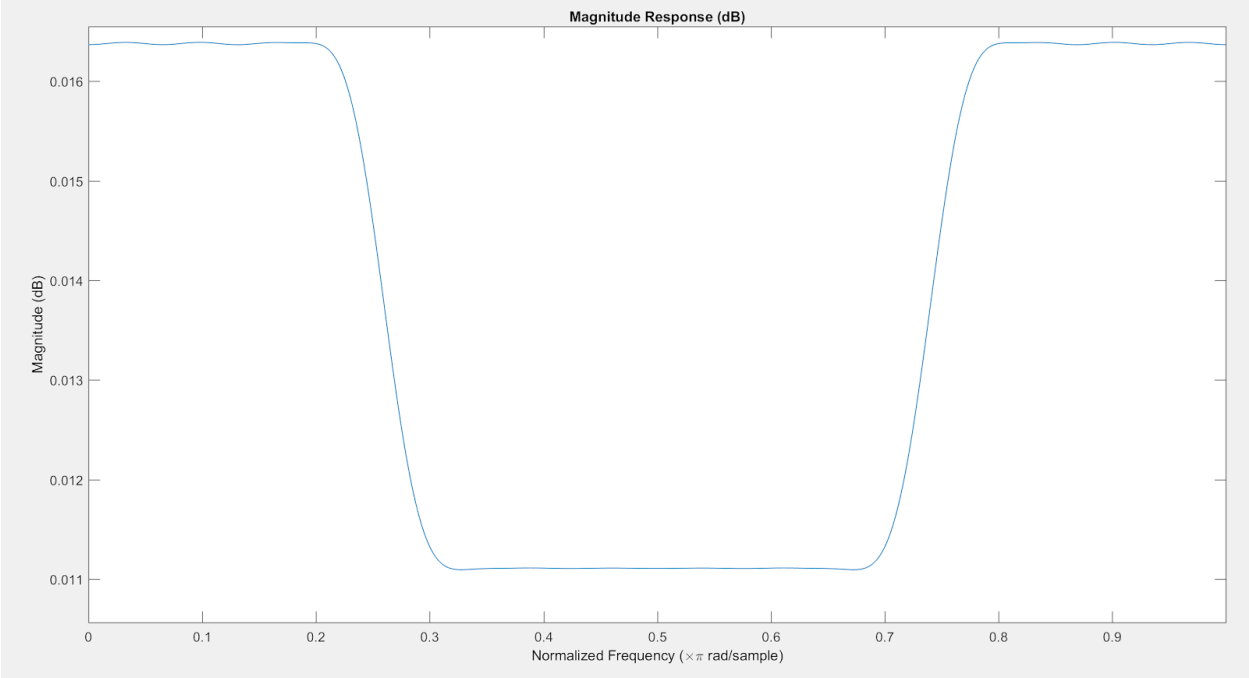
“Mid1” Bandpass Filter H_bp1(w)



“Mid2” Bandpass Filter H_bp2(w)



Highpass Filter H_hp(w)



Combined Response H(w)

The combined response fits well within the 0.02dB maximum ripple spec, so the filters meet the specs for filtering on the DSP board.

Question 3: What is the purpose of using the FFT in this assignment? [[15]]
A: The Fast Fourier Transform (FFT) is an efficient way of calculating the Discrete-Fourier Transform (DFT). It is necessary because the DSP board has limited memory and the FFT saves such memory and does it in *Nlog(N)* time rather than N^2 time.

Question 4: Explain the limitations on the input frame size for the Overlap-Add & Overlap-Save methods. [[15]]
A: Two conditions have to be met for FFT to be done efficiently in both time and space(i.e. memory) without aliasing. The first is the input frame size has to be less than or equal to the number of samples that results from the convolution of each block with the impulse response subtracted by the length of the filter plus 1 (i.e. N - P + 1, where N is the size of the output after convolving each block with the impulse response, and P is the size of the filter/impulse response). The second is the amount of computations involved in the FFT has to be minimized. With N and P given, in order to minimize the amount of computations done by FFT, L has to be as big as possible. Thus, in order to avoid aliasing, the size of each block should be equal to N - P + 1, which is why our input frame size is 196 = 256 - 61 + 1.

Question 5: Of the two methods investigated (Overlap-Add and Overlap-Save), which one is more computationally intensive? Explain your reasoning. [[30]]
A: The more computationally intensive method is Overlap-Add because there involves addition, whereas Overlap-Save simply replaces the parts that would be added in Overlap-Add.