Lab 4

IIR Filters

ECE 406: Real-Time Digital Signal Processing

March 31, 2015

Christopher Daffron

cdaffron@utk.edu

There were two objectives for this lab. The first one was to get hands-on experience with IIR filter design and the second one was to get practice with a circular buffering implementation of a direct form 1 IIR filter.

Task 0

Part A

All of the tasks that are part of this section of the assignment involve doing manual calculation of selected parameters using various methods of generating filter coefficients and then verifying those parameters using MATLAB functions. The first part was to determine the order of a given butterworth filter and the 3 dB cutoff frequency. To determine the order, the following equation was used:

$$\frac{1}{10} = \frac{1}{\sqrt{1 + 1\left(\frac{.65}{.6}\right)^{2n}}}$$

$$n = 28.704 \approx 29$$

There are two 3 db cutoff frequencies. They are determined using the following equation:

$$\frac{1}{1.414} = \frac{1}{\sqrt{1 + 1\left(\frac{\Omega_n}{.6}\right)^{2(29)}}}$$

$$\Omega_n = 0.59999$$

$$\frac{1}{1.414} = \frac{1}{\sqrt{1 + 1\left(\frac{\Omega_n}{0.35}\right)^{2(29)}}}$$

$$\Omega_n = 0.3499$$

Next, MATLAB was used to verify the calculations. First, the buttord() function was used to find the order and 3 dB frequency of the filter. The code that was used to create this can be found in the appendix. The order of the analog filter from this calculation was 22. If a digital filter is created directly from the function, the order is 16. There are several reasons for the difference from the hand calculations. The algorithm used by MATLAB is more accurate than the hand equations and this order is based on an analog filter instead of a digital filter. The 3 dB cutoff frequencies that were found by MATLAB were 0.3476 and 0.6042. These values are pretty close to the calculated values. Next, the butter() function was used to create the analog filter coefficients and the impinvar() function was used to create the digital coefficients. The plot of the frequency response can be found below.

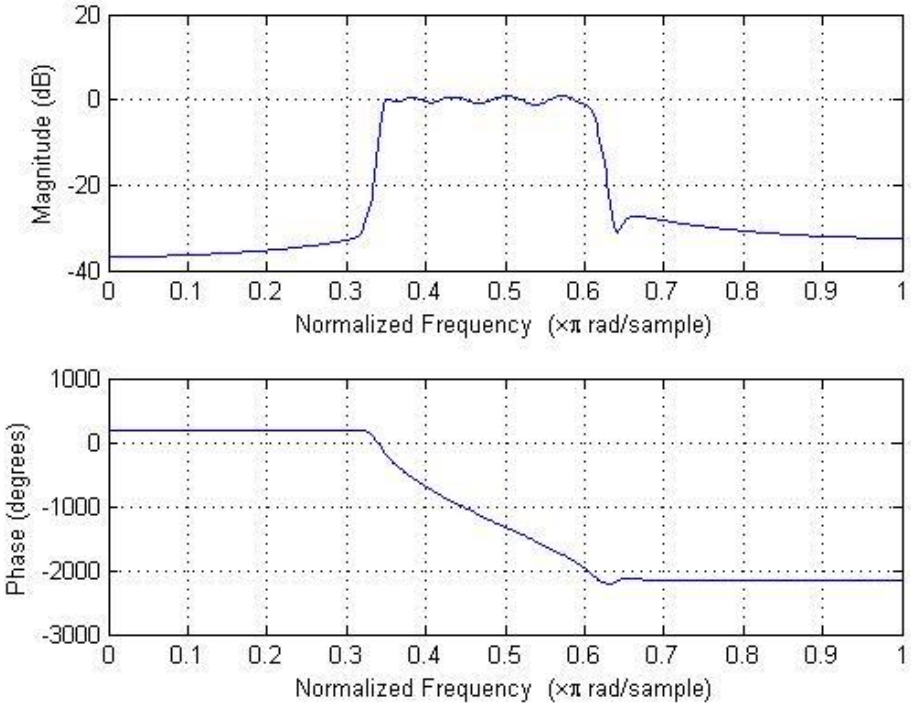

Figure 1: Impulse Invariance Frequency Response

The frequency response of the analog filter coefficients generated by the butter() function can be found below.

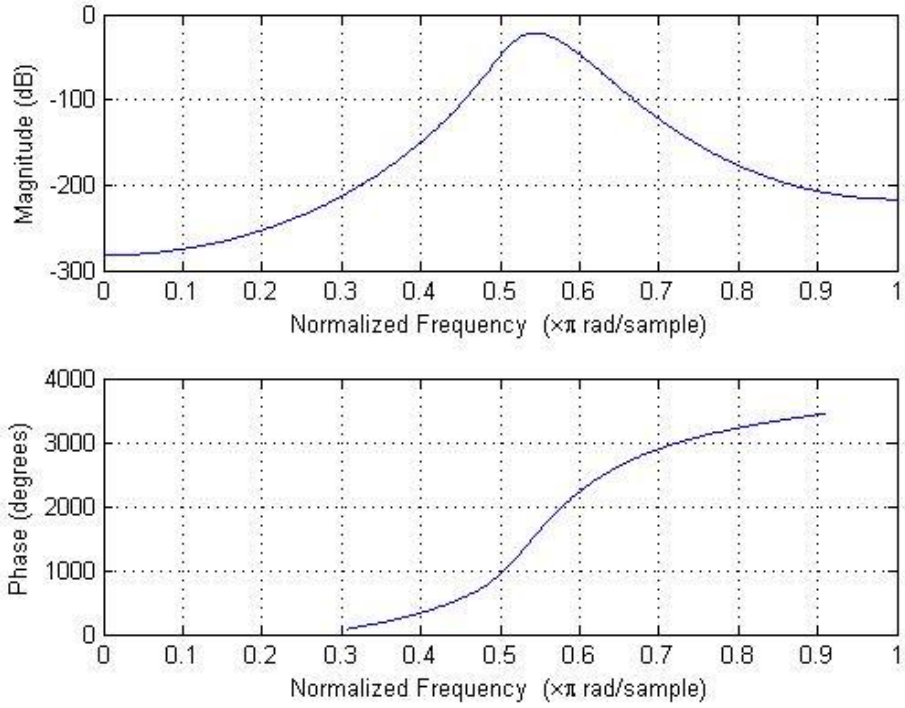

Figure 2: Analog Butterworth Response

This frequency response is much worse than the response of a digital filter created using the butter() function directly, as shown below.

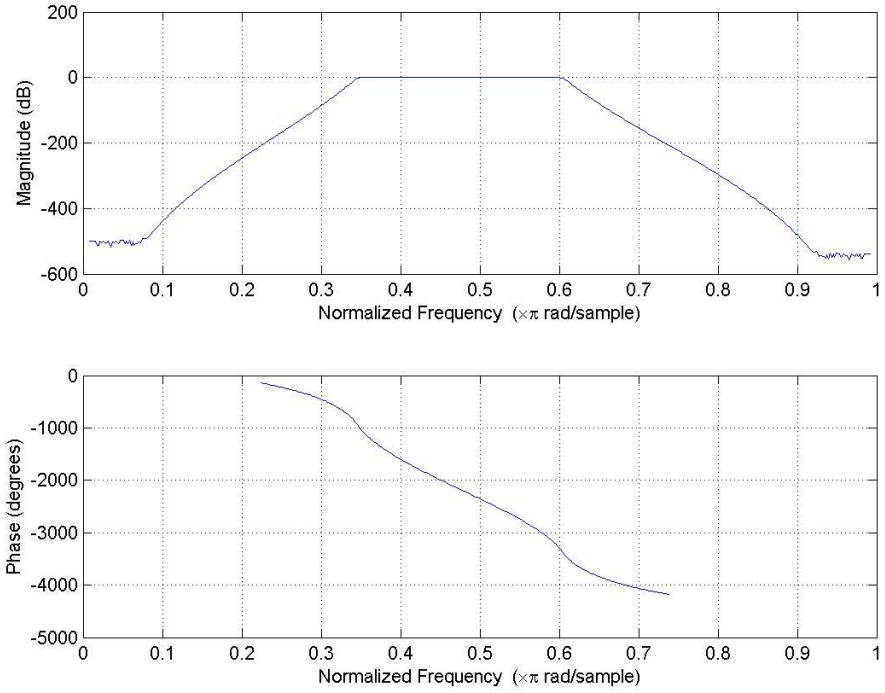

Figure 3: Digital Butterworth Response

After finding the frequency response and filter coefficients manually using the individual MATLAB functions, the instructions said to use the FDA Tool to find the coefficients, but the FDA Tool does not include the option to use impulse invariance, so I skipped this step.

Part B

The next part was to use the bilinear transformation. MATLAB functions were first used to determine the order and 3 dB cutoff frequency of the filter. These numbers and code are the same as the previous section. The code for this can be found in the appendix. Once those were found, the following frequency responses were generated using a bilinear transformation.

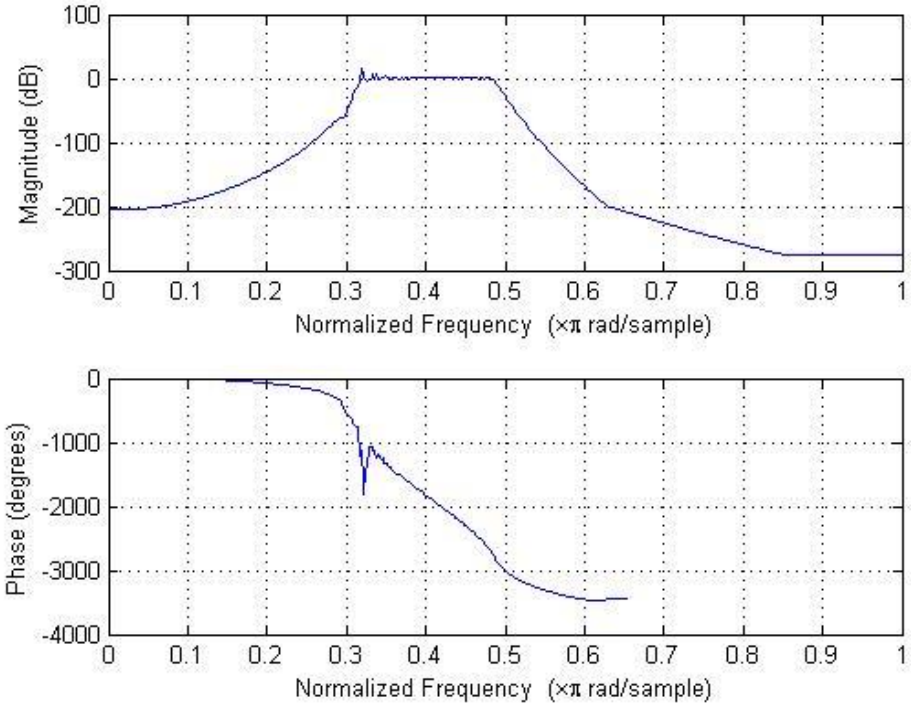

Figure 4: Bilinear Transformation Frequency Reponse

Again, this response is much worse than the digital frequency response shown in the previous section. Finally, MATLAB's FDA Tool was used to determine the coefficients. Though there is no option to select between impulse invariance and bilinear transformation in the FDA Tool, the documentation for the butter function says that it uses a bilinear transformation internally. A screenshot of the FDA Tool can be found below and the coefficients generated can be found in the appendix.

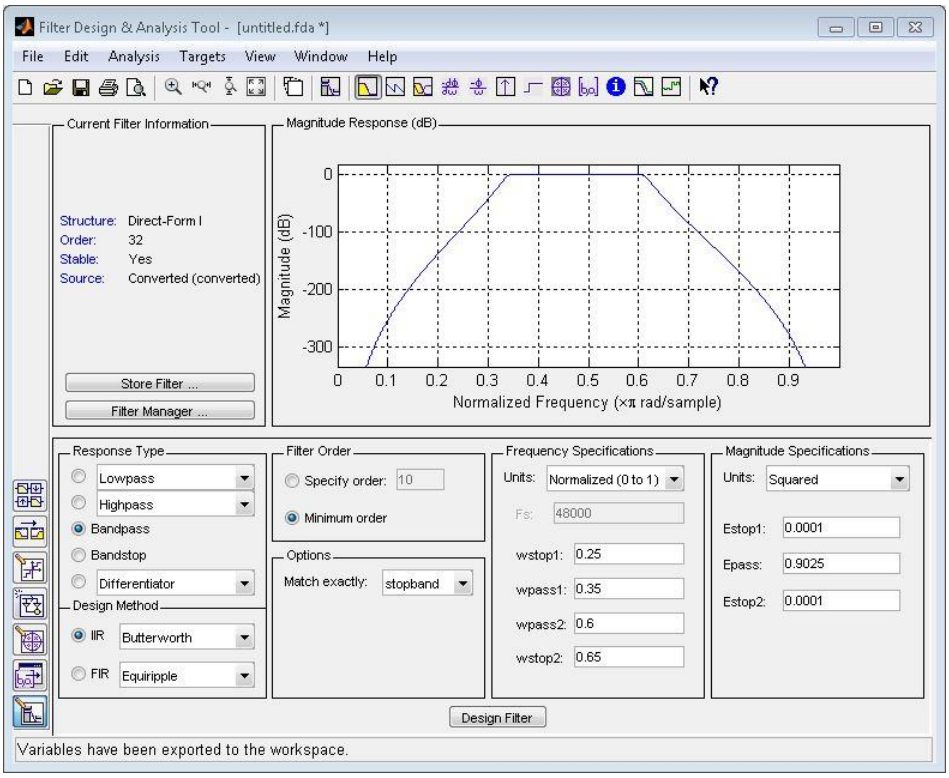

Figure 5: FDA Tool

Next, the appropriate MATLAB functions were used to design an elliptic filter. The code used to generate it can be found in the appendix. The follow frequency responses were generated using a bilinear transformation.

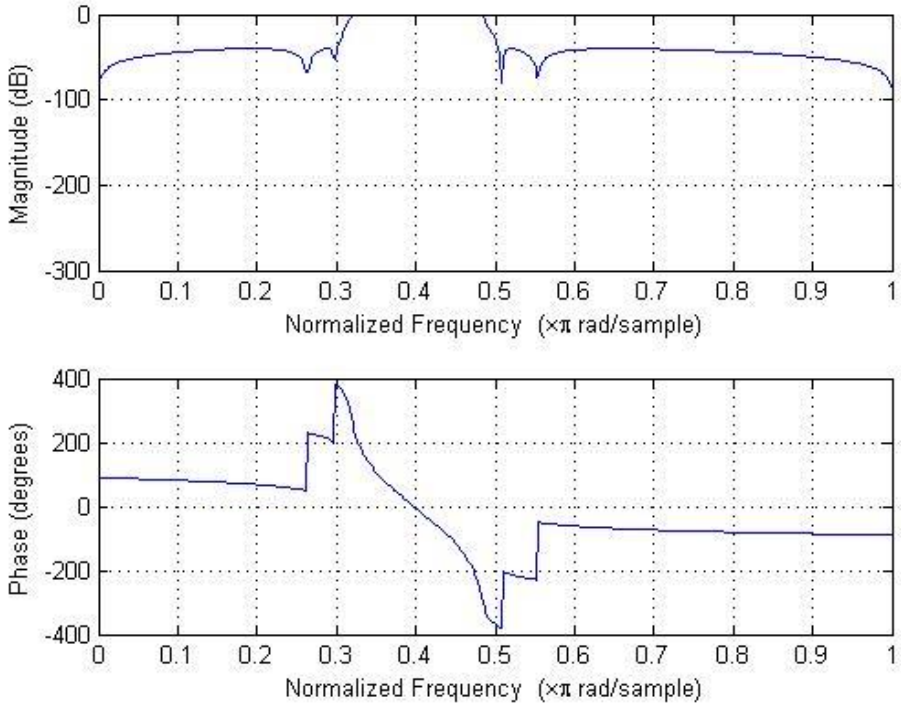

Figure 6: Elliptic Digital Filter Bilinear Transformation Response

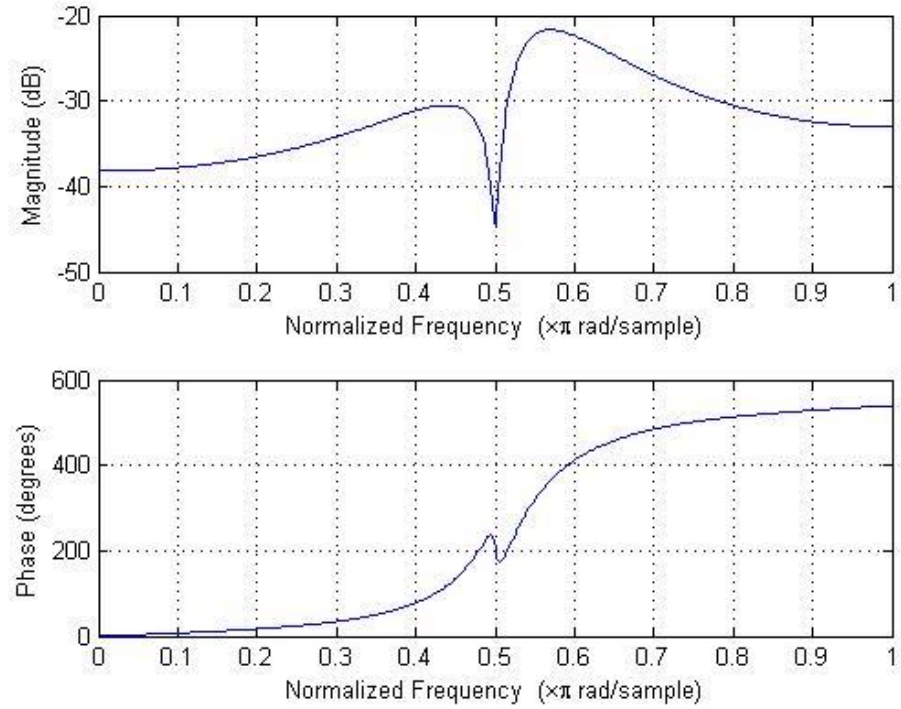

Figure 7: Elliptic Analog Filter Bilinear Transformation Response

Finally, the FDA Tool was used to design the filter and obtain the coefficients. A screenshot of the FDA Tool can be found below.

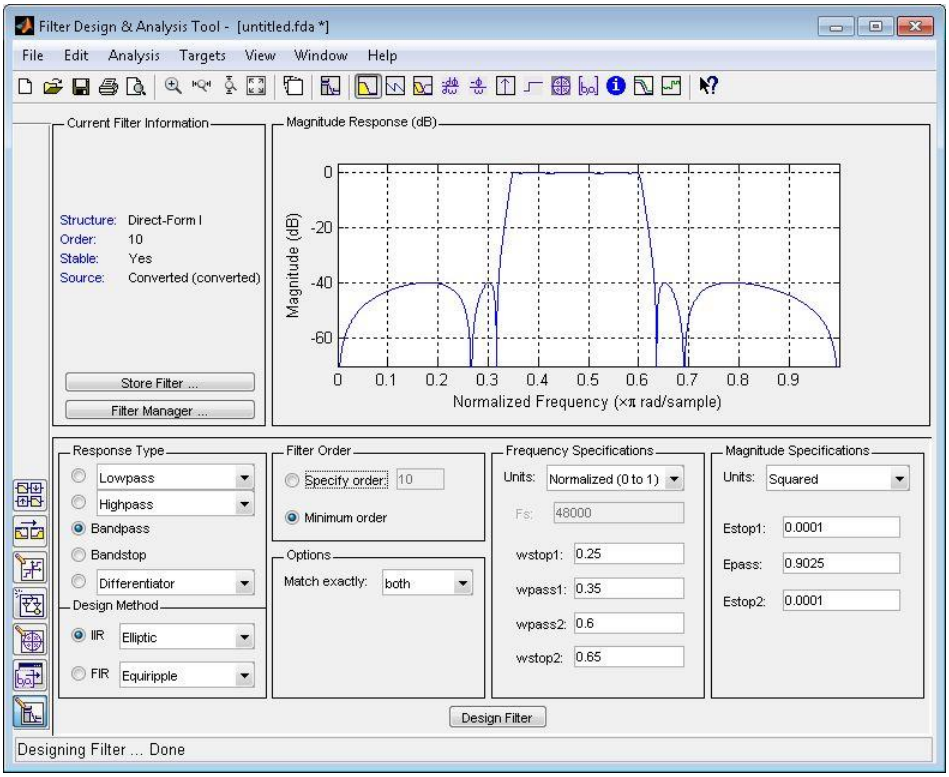

Figure 8: FDA Tool

Task 1

The first task was to measure and graph the theoretical frequency response of all of the generated filters. Those graphs can be found below.

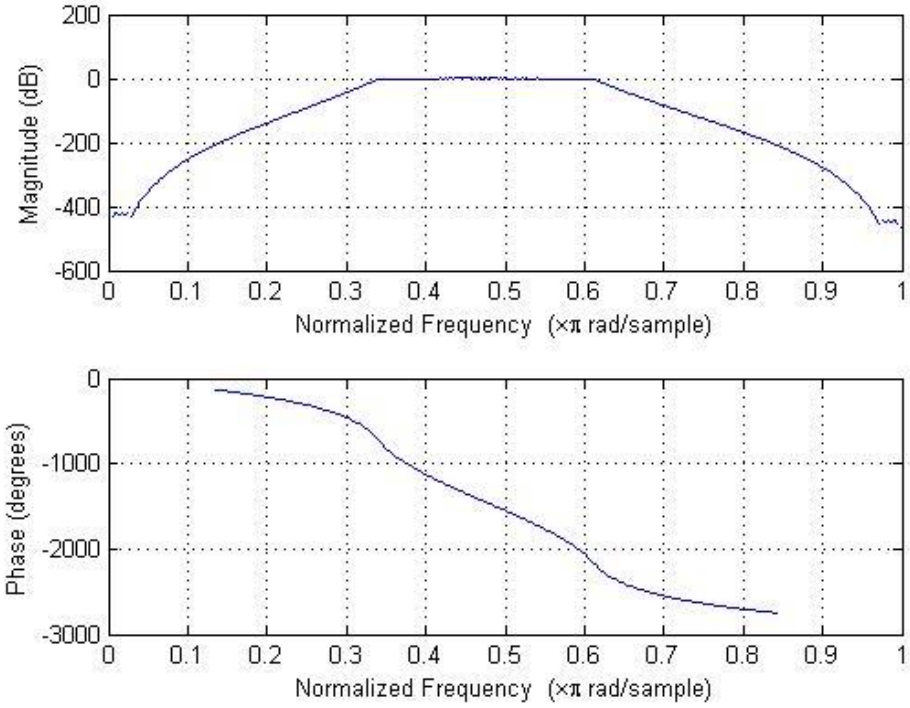

Figure 9: Theoretical Frequency Response – Butterworth Bilinear

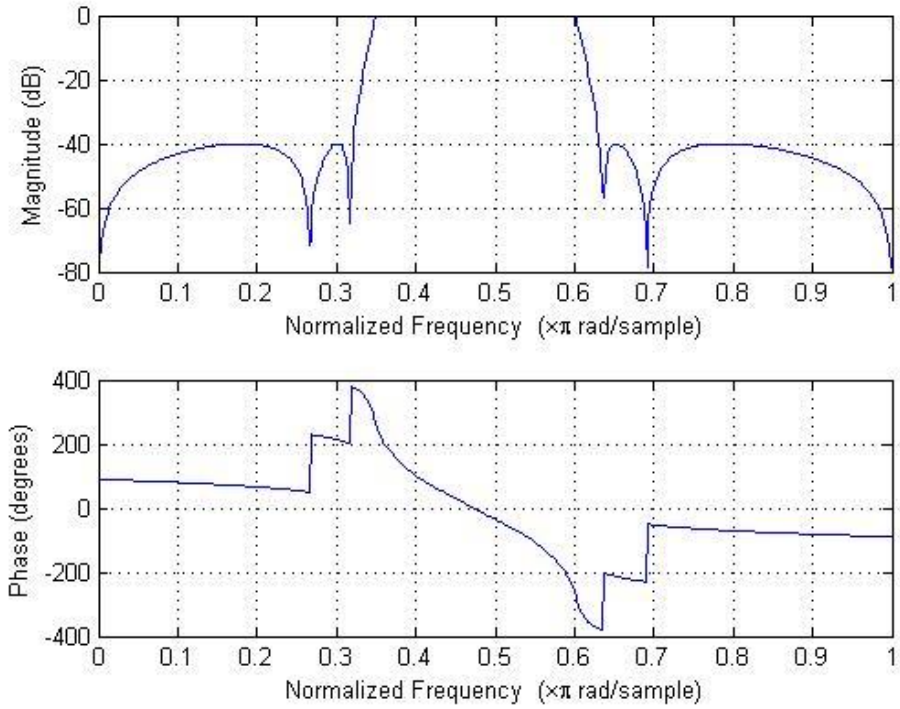

Figure 10: Theoretical Frequency Response – Elliptic Bilinear

Task 2

Next, an implementation of a Direct-Form I IIR filter using linear buffering was written in C. This code can be found in the appendix. This code was then run using the coefficients generated from the FDA Tool and the experimental magnitude response was measured and graphed for comparison against the theoretical magnitude response. The first filter that was run was the Butterworth Filter using Bilinear Transformation. In both of the comparisons, it is very noticeable that the actual attenuation in the stopbands is much greater than the theoretical attenuation. This is because the noise going into the oscilloscope becomes greater than the actua signal around -25 dB. The comparison of the experimental and theoretical frequency responses can be found below.

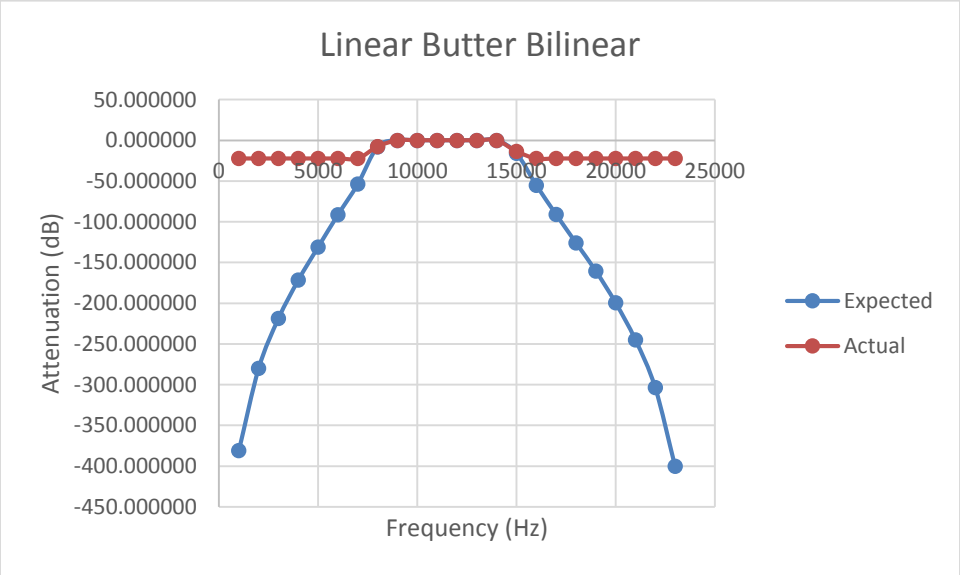

Figure 11: Frequency Response Comparison, Butterworth Bilinear

Next, the same was done for the Elliptic Filter using Bilinear Transformation. The comparison of the experimental and theoretical frequency responses is shown below.

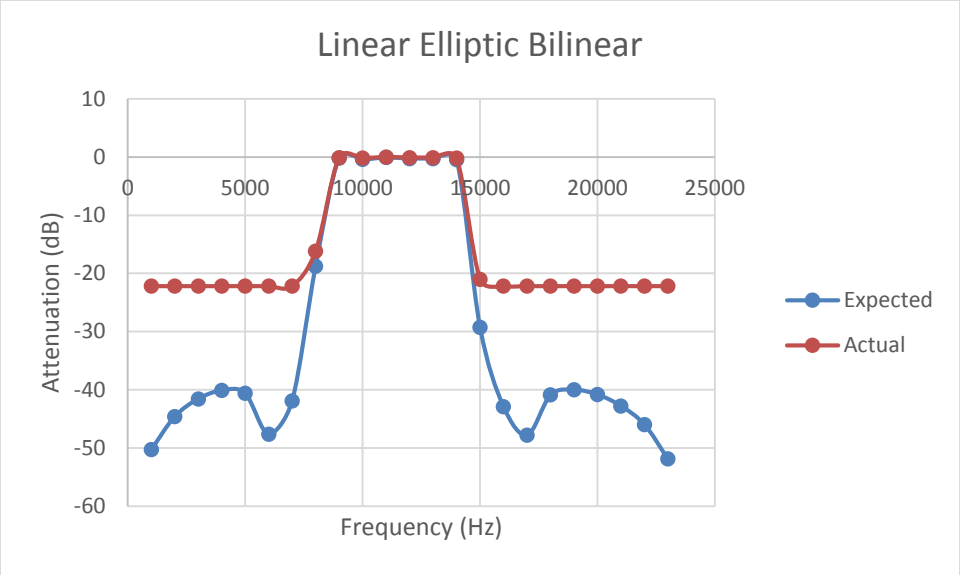

Figure 12: Frequency Response Comparison, Elliptic Bilinear

After completing this part, a second implementation of the Direct-Form IIR Filter was completed using circular buffering. The C code that was generated can be found in the appendix. This code was fully tested and produces the exact same output as the linear buffer, as it should. For this reason, a comparison of the frequency responses does not need to be done again.

Finally, task 3 was to compare the number of clock cycles taken up by the linear buffering implementation versus the circular buffering implementation. After running these tests, I found that the linear buffering implementation takes 3,671 clock cycles to service a single interrupt and the circular buffering implementation takes 2,891 clock cycles to service a single interrupt, an improvement of 26.9%.

Task 3

There are noticeable differences between the versions of the butterworth filter that use bilinear transformation vs the impulse invariance methods. When converted from an analog filter created using the butter function, the impulse invariance method looks much better, with a better defined passband that conforms to the specified parameters more closely. Conversely, the bilinear transformed one shows more ripple in the passband and the transition bands are wider. Based solely on this, I would say that the impulse invariance method shows better performance, but there is one more thing to consider. When the butter function is used to provide a digital filter directly, it uses a bilinear transformation internally (according to the documentation) and creates a filter that looks much better than the manually created results of either method. Because of this, I think that more investigation would be necessary to definitively determine which method is better for this given filter.

There are several difference between FIR and IIR implementations. First, the IIR implementation requires more memory because the recent outputs must be stored in addition to the recent inputs, which also must be stored with the FIR filter. Secondly, the order of the IIR filter seems to be less for a similar filter, with an Equiripple Filter designed using the Parks McClellan method having an order of 59 and an Elliptic Fitler designed using the Bilinear Transformation method having an order of 10. The number of taps required for the FIR filter can be cut in half by taking advantage of the symmetry of the coefficients, but it would still require 30 taps, which is much more than the number required for the Elliptic Filter. Because of this, I would say that the IIR filter is more efficient even though it does require slightly more memory.

The usage of the different buffering methods does not change the frequency response at all since the same values are being used in the equations, they are simply stored in a different location.

There are several difference when using different filters such as Butterworth and Elliptic. The butterworth filter has much better stopband attenuation, but requires a higher order. Conversely, the elliptic filter has a much lower order and can therefore have better performance.

Conclusion

In this lab, I got hands-on experience with IIR filter design and also got more practice using circular buffering with an IIR implementation. I also learned more about the calculations necessary to design these filters and the appropriate MATLAB functions to create them. I also feel like I have a better understanding of the algorithms that are used to implement digital IIR filters on the DSK.

Appendix

Part 1

Filter_code.m

```
Wp = [0.35*pi 0.6*pi];
Ws = [0.25*pi 0.65*pi];
Rp = -20 * log10( 0.95 );
Rs = -20 * log10( 0.01 );

% Uncomment only one of the following sections to perform that calculation

%% Butterworth Impulse Invariance Section
% [n,Wn] = buttord(Wp, Ws, Rp, Rs, 's');
% [b,a] = butter(n, Wn, 'bandpass', 's');
% freqz(b,a);
% figure()
% [bz, az] = impinvar( b, a, 1 );
% freqz(bz,az);
```

```
%% Butterworth Bilinear Transformation Section
[n,Wn] = buttord(Wp, Ws, Rp, Rs, 's');
[b,a] = butter(n, Wn, 'bandpass', 's');
freqz(b,a);
figure();
[bz, az] = bilinear( b, a, 1);
freqz(bz,az);

%% Elliptic Bilinear Transformation Section
% [n,Wn] = ellipord(Wp, Ws, Rp, Rs, 's');
% [b,a] = ellip(n, Rp, Rs, Wp, 's');
% freqz(b,a);
% figure();
% [bz,az] = bilinear( b, a, 1 );
% freqz(bz,az);
```

Task 2

IIRmono_ISRs.c – Linear Buffering

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
/////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "coeff.h"

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
       Uint32 UINT;
       Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
//#define N 20              // IIR filter order

/* ----- BUTTERWORTH BILINEAR ----- */
float x[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0};   // input value (buffered)
```

```
float y[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0};   // output values (buffered)

/* ----- ELLIPTIC BILINEAR ----- */
//float x[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
//float y[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

float yn = 0.0;

interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */


       if(CheckForOverrun())                                  // overrun error occurred
(i.e. halted DSP)
              return;                                                    // so serial
port is reset to recover

       CodecDataIn.UINT = ReadCodecData();             // get input data samples

       int i;
       yn = 0.0;
       x[0] = CodecDataIn.Channel[LEFT];
       yn += ((B[0]*x[0]));

       for(i = 1; i <= N; i++)
       {
              yn += (B[i]*x[i]);
              yn -= (A[i]*y[i]);
       }

       y[0] = yn;
       for(i = N; i > 0; i--)
       {
              x[i]=x[i-1];
              y[i]=y[i-1];
       }

       CodecDataOut.Channel[LEFT] = (Int16)yn;
       CodecDataOut.Channel[RIGHT] = (Int16)yn;

       WriteCodecData(CodecDataOut.UINT);       // send output data to  port

       return;
}
```

Task 3

IIRmono_ISRs.c – Circular

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
/////////////////////////////////////////////////////////////////////////

#include “DSP_Config.h”
#include “coeff.h”

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
       Uint32 UINT;
       Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* ----- BUTTERWORTH BILINEAR ----- */
float x[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0};   // input value (buffered)
float y[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0};   // output values (buffered)

/* ----- ELLIPTIC BILINEAR ----- */
//float x[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
//float y[N+1] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

float yn;
int sampleInd = 0;

interrupt void Codec_ISR()
/////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
```

```
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */


       if(CheckForOverrun())                                  // overrun error occurred
(i.e. halted DSP)
              return;                                                  // so serial
port is reset to recover

       CodecDataIn.UINT = ReadCodecData();             // get input data samples

        nti ;
       int indLeft;

       yn = 0.0f;
       indLeft = sampleInd – 1;

    if( indLeft < 0 )
        indLeft = N;

       x[sampleInd] = CodecDataIn.Channel[LEFT];

       yn += (B[0]*x[sampleInd]);

       for(i = 1; i <= N; i++)
       {
              yn += ((B[i]*x[indLeft]) – (A[i]*y[indLeft]));

              if(--indLeft < 0)
                     indLeft = N;
       }

       y[sampleInd] = yn;

       if(++sampleInd > N)
              sampleInd = 0;

       CodecDataOut.Channel[LEFT] = yn;
       CodecDataOut.Channel[RIGHT] = yn;

       WriteCodecData(CodecDataOut.UINT);       // send output data to  port

       return;
}
```

Shared By Parts

Coeff.h

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/* coeff.h                              */
```

```
/* IIR filter coefficients             */

/* ----- BUTTERWORTH BILINEAR ----- */
#define N 32

/* ----- ELLIPTIC BILINEAR ----- */
//#define N 10

extern float B[];
extern float A[];
```

coeff.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/* coeff.c                             */
/* FIR filter coefficients             */
/* exported by MATLAB using FIR_DUMP2C */


/* Equiripple FIR LPF with passband to */
/* 5 kHz assuming Fs=48 kHz            */


#include "coeff.h"

/* ----- BUTTERWORTH BILINEAR ----- */

float B[N+1] = {
 0.000000034667186407017829,
 0                         ,
-0.00000055467498251228526  ,
 0                           ,
 0.00000416006236884213970    ,
 0                             ,
-0.00001941362438792998610     ,
 0                               ,
 0.00006309427926077245500       ,
 0                                 ,
-0.00015142627022585387            ,
 0                                  ,
 0.00027761482874739875             ,
 0                                  ,
-0.00039659261249628392             ,
 0                                  ,
 0.00044616668905831937             ,
 0                                  ,
-0.00039659261249628392             ,
 0                                  ,
 0.00027761482874739875             ,
 0                                  ,
-0.00015142627022585387             ,
 0                                  ,
 0.00006309427926077245500          ,
 0                                  ,
-0.00001941362438792998610          ,
```

```
 0                                    ,
 0.0000041600623688421397             ,
 0                                    ,
-0.0000055467498251228526             ,
 0                                    ,
 0.0000000346671864070178 29          ,
};

float A[N+1] = {
   1           ,
  -1.9801853890905561,
   9.1649400704466846 ,
 -14.888835049793361   ,
  39.091650150313797    ,
 -54.053519038133658     ,
 103.7940178244709        ,
-124.78218130144654        ,
 192.54424577919335         ,
-203.86713917280235          ,
 264.81870122989505           ,
-248.81759164235612            ,
 279.2832130917119              ,
-233.71194760902372               ,
 230.21109514072145                ,
-171.64616409746702                 ,
 149.72630338342509                  ,
 -99.189355655056573                  ,
  76.98095076610899                   ,
 -45.023608184758089                  ,
  31.129471108367586                  ,
 -15.897784481372415                  ,
   9.7752574545111433                 ,
  -4.2821998764558771                 ,
   2.3303846019656183                 ,
  -0.85057993413808886                ,
   0.40625646626076067                ,
  -0.11746136096242073                ,
   0.048599889346246553               ,
  -0.010070168757280885               ,
   0.0035386230275000646              ,
  -0.00040345651517289648             ,
   0.00011677740101310004             ,
};

/* ----- ELLIPTIC BILINEAR ----- */

//float B[N+1] = {
//             0.022446109482960157               ,
//            -0.010573379073474012,
//             0.014402415206826613,
//            -0.0080366144026941533              ,
//             0.022544111354959838,
//             0.000000000000000003469446951953614 2,
//            -0.022544111354959838,
//             0.0080366144026941498              ,
//            -0.014402415206826613,
//             0.010573379073474012,
//            -0.022446109482960157,
```

```
//};
//
//float A[N+1] = {
//             1,
//            -0.70635031270725446,
//             3.5219499200390536,
//            -2.0200628628750592,
//             5.5383108441644708,
//            -2.4294345269044149,
//             4.6510805728204474,
//            -1.4124566248726653,
//             2.0783744842441458,
//            -0.3332994829583692,
//             0.39240786067834416 ,
//};
```