EE 5350 - Program Assignment 5

James Grisham
1000792179

August 7, 2014

List of Figures

Listings

This program assignment involved IIR filter design. A 4th order IIR Butterworth filter was designed using the bilinear transform. This was implemented using C++. The user specifies the cutoff frequencies and the C++ code designs the filter and applies it to the input.

The poles of the Butterworth filter are

$$s_k = e^{j\theta[k]}$$

where

$$\theta[k] = \frac{\pi}{2} + \frac{\pi}{2N} + \frac{(k-1)\pi}{N}$$

For a 4th order digital filter, we start with a second order prototype lowpass filter and transform it to a bandpass filter. The transfer function for the prototype lowpass filter is

$$H_1(s) = \frac{1}{\displaystyle\prod_{k=1}^{N/2}(s^2 - 2\cos(\theta[k]) + 1)}$$

This analog LP filter is transformed to an analog BP filter by the following transformation:

$$H_a(s) = H_1(s)\Big|_{s = \dfrac{s^2 + \Omega_0^2}{sBw}}$$

where

$$\Omega_0^2 = \Omega_{c1}\Omega_{c2} \quad \text{and} \quad Bw = \Omega_{c2} - \Omega_{c1}$$

The analog cutoff frequencies are determined by prewarping the digital cutoff frequencies. this is done as follows:

$$\Omega_{c1} = \frac{2}{T}\tan\left(\frac{\omega_{d1}}{2}\right)$$
$$\Omega_{c2} = \frac{2}{T}\tan\left(\frac{\omega_{d2}}{2}\right)$$

where T is assumed to be unity. Next, the analog transfer function is transformed to the z-domain using the bilinear transform.

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}$$

All of this algebra was accomplished using Mathematica. After finding $H(z)$, a recursive difference equation is found. This difference equation is then used to apply the filter. The Mathematica notebook that contains all the symbolic manipulations is included at the end of this document.

This filter was applied using cutoffs given by

$$\omega_{d1} = 1 \text{ rad} \quad \text{and} \quad \omega_{d2} = 2 \text{ rad}$$

The amplitude response of the filter was calculated in two different ways, namely, applying the filter to the Kronecker delta to get the impulse response and using the amp function of program assignment 3, and secondly, by substituting $z = e^{j\omega}$ into the transfer function in the z domain. This is valid because general equations for the poles of the filter were derived. If the poles are outside the unit circle, the IIR filter is unstable. This is checked in the C++ code as a part of the design procedure. As long as the filter is stable, then the region of convergence of the z transform includes the unit circle, which in turn allows for the frequency response to be evaluated from the transfer function.

Figure 1 shows the amplitude response from the first approach and Figure 2 shows the amplitude response using the second approach. They are identical.

1 Figures

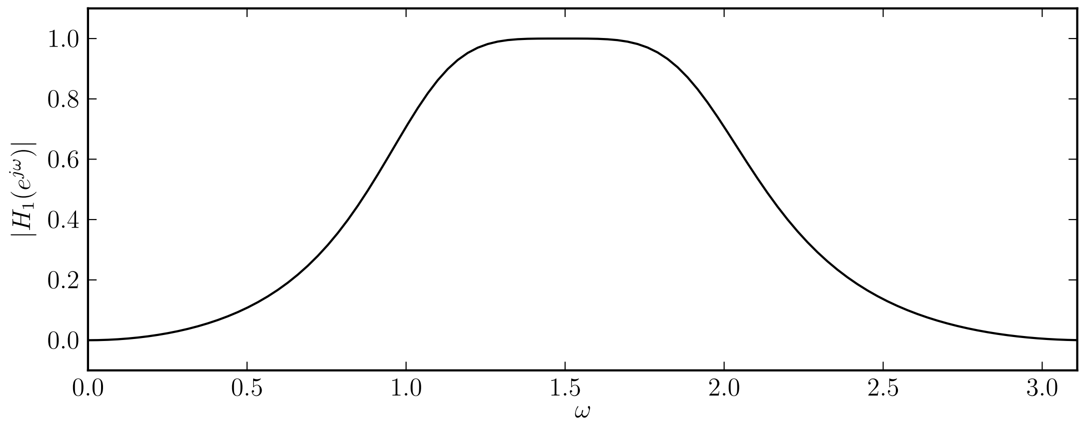


Figure 1: Amplitude response of filter (from program 3 amp function).

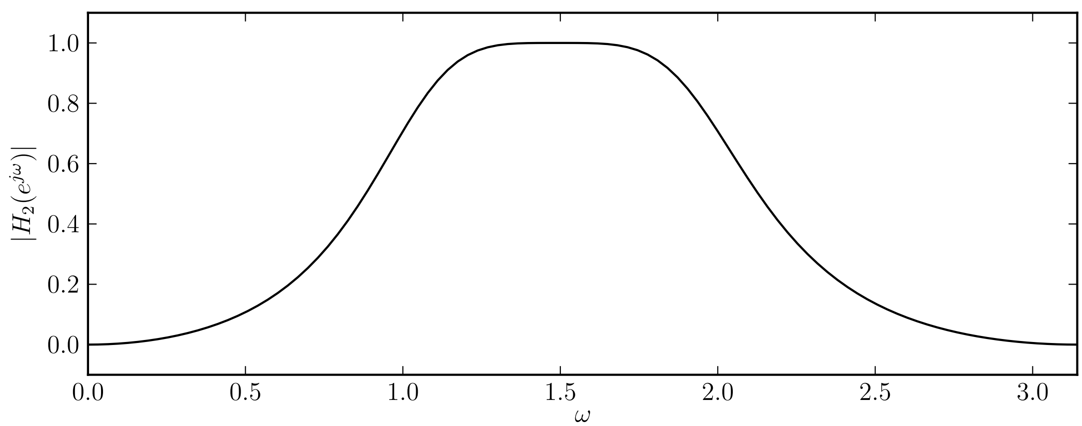


Figure 2: Amplitude response of filter (calculated using $H(z)$).

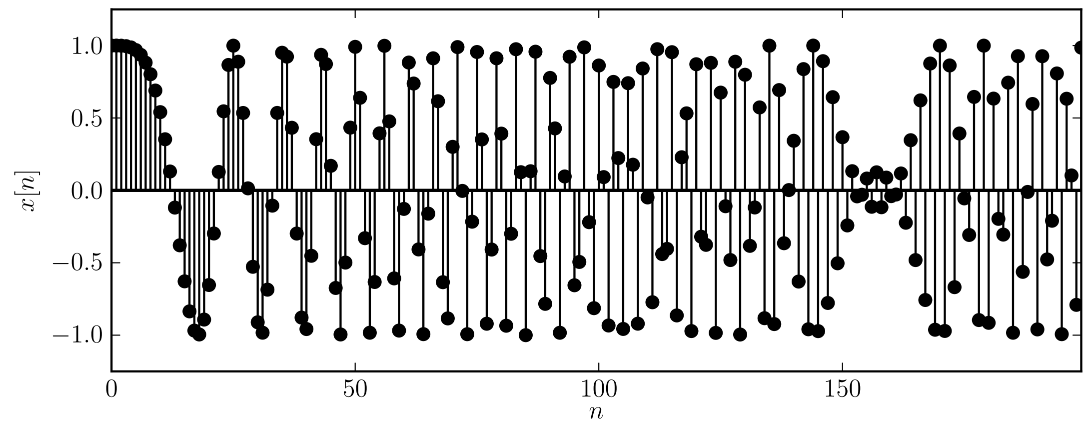


Figure 3: Plot of input signal.

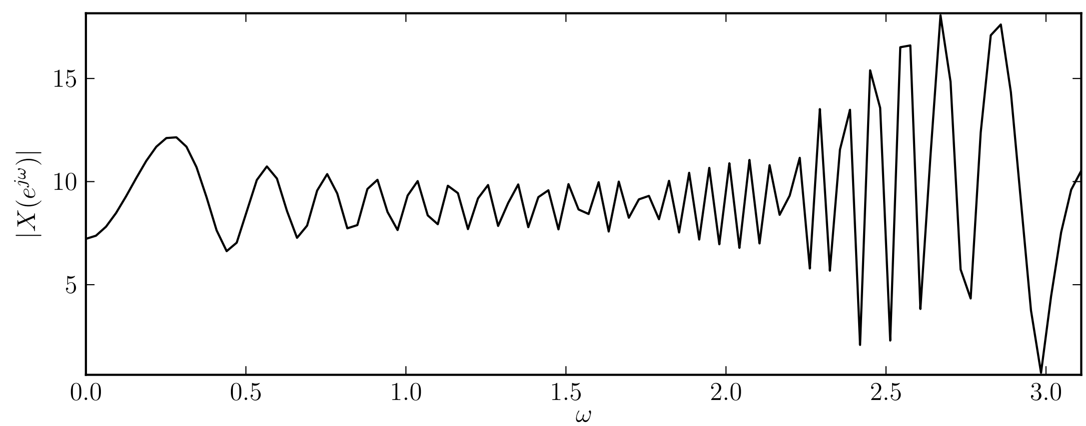


Figure 4: Amplitude response of input signal.

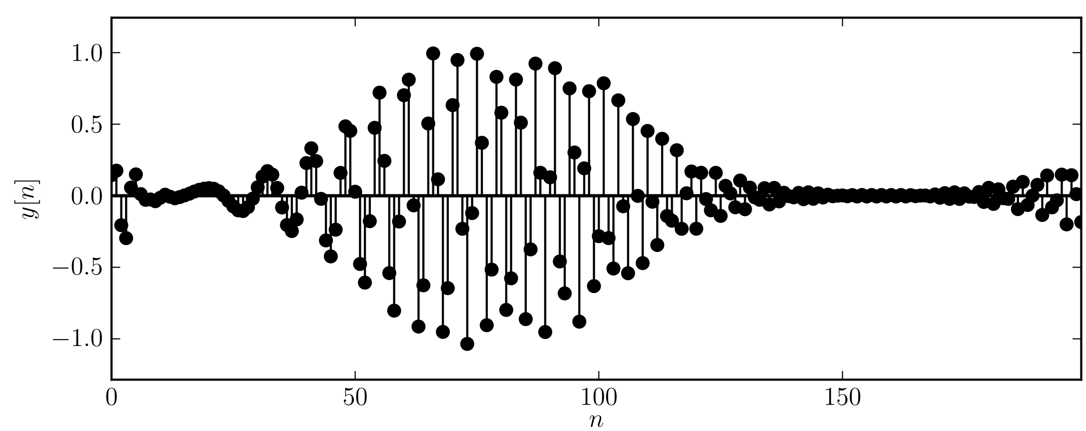


Figure 5: Plot of filtered signal.

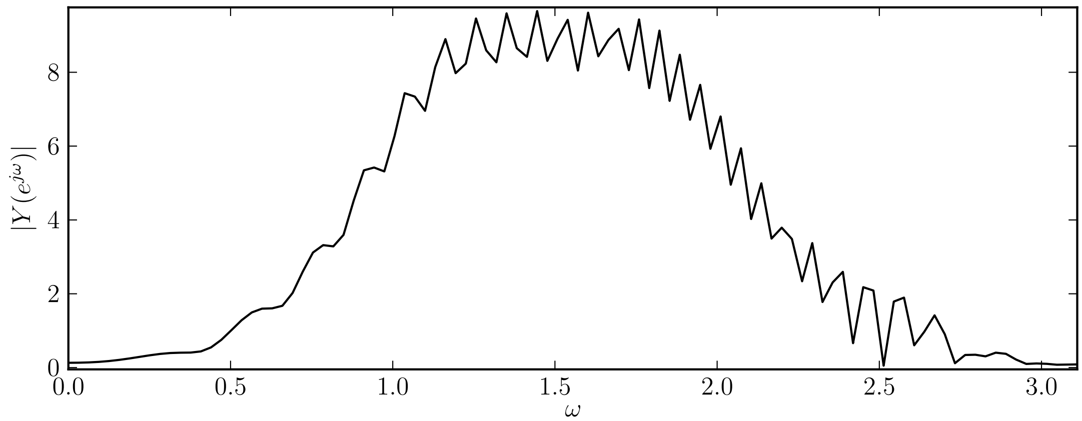


Figure 6: Amplitude response of filtered signal.

2 Code Listing

Listing 1: Program 6 Main C++ file

```cpp
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <cstdlib>
#include "dsptools.hpp"

using namespace std;

/*
 *
 * prog6.cpp is a C++ implementation of 4th order bandpass
 * Butterworth filter that was designed using the bilinear
 * transform.
 *
 * Author       : James Grisham
 * Date         : 08/06/2014
 * Revision Date:
 *
 */

// Prototype for function that writes amplitude spectra
void write_amp(string file_name, vector<double> w, vector<double> Amp);

//=============================================================
// Main
//=============================================================

int main() {

    // Instantiating new object
    dsptools dsp;
```

```
    // Declaring variables
    int Nx=200;
    int Nm=100;
    double wc1, wc2;  // these are the cutoff frequencies
    vector<double> x (Nx);
  vector<double> y (Nx);
    vector<double> delta (Nx);
    vector<double> h (Nx);
    vector<double> omega_x;
    vector<double> X;
    vector<double> omega_y;
    vector<double> Y;
    vector<double> omega_amp1;
    vector<double> H_amp1;
    vector<double> omega_amp2; // memory allocated in amp2
    vector<double> H_amp2;     // memory allocated in amp2

    // Setting cutoff frequencies
    wc1 = 1.0;
    wc2 = 2.0;

    // Generating the signal
    for (int n=0; n<Nx; n++) {
        x[n] = cos(0.01*((double) n)*((double) n));
    }

    // Writing signal to file
    string filename = "./DataFiles/prog6_x.dat";
    dsp.writeSignal(filename,x);

    // Designing the filter
    dsp.dsine2(wc1,wc2);

    // Applying the filter
    dsp.filt(x,y);

    // Writing filtered signal to file
    filename = "./DataFiles/prog6_y.dat";
    dsp.writeSignal(filename,y);

    // Computing the amplitude response of x
    dsp.amp(Nm, x, Nx, X, omega_x);
    filename = "./DataFiles/prog6_X.dat";
    write_amp(filename,omega_x,X);

    //Computing the amplitude response of y
    dsp.amp(Nm, y, Nx, Y, omega_y);
    filename = "./DataFiles/prog6_Y.dat";
    write_amp(filename,omega_y,Y);

    // Computing the amplitude response of the filter
    dsp.amp2(Nm,omega_amp2,H_amp2);
    filename = "./DataFiles/prog6_Hamp2.dat";
    write_amp(filename,omega_amp2,H_amp2);

    // Checking to make sure that amplitude is correct
    delta[0] = 1.0;
    dsp.filt(delta,h);
```

```
    filename = "./DataFiles/prog6_h.dat";
    dsp.writeSignal(filename,h);
    dsp.amp(Nm,h,Nx,H_amp1,omega_amp1);
    filename = "./DataFiles/prog6_Hamp1.dat";
    write_amp(filename,omega_amp1,H_amp1);

    return 0;

}

// Function to write data file
void write_amp(string file_name, vector<double> w, vector<double> Amp) {

    ofstream fp;
    fp.open(file_name.c_str());
    for (int n=0; n<w.size(); n++) {
        fp << w[n] << " " << Amp[n] << "\n";
    }
    fp.close();

}
```

Listing 2: DSP Tools

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <complex>
#include "dsptools.hpp"

using namespace std;

//================================================================
// Method for writing signal to file
//================================================================

void dsptools::writeSignal(string filename, const vector<double>& x) const {

    // Opening file
    FILE *fp;
    fp = fopen(filename.c_str(),"w");

    // Checking to make sure that file is open
    if (fp == NULL) {
        fprintf(stderr,"\nERROR: writeSignal() can’t open file.\n\n");
        exit(1);
    }

    // Writing data
    for (int n=0; n<x.size(); n++) {
        fprintf(fp,"%d  %4.4f\n",n,x[n]);
    }

    // Closing file
    fclose(fp);
```

```

}

//============================================================
// Method for discrete convolution -- Standard Offline
//============================================================

void dsptools::CONV(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h, int& Ny, vector<double>& y) const {

    // Declaring variables
    int m;

    // Computing Ny
    Ny = Nh + Nx;

    // Convolving the two signals (x and h) and storing output to y
    for (int n=0; n< Ny; n++) {

        // Setting n-th term equal to zero initially
        y.push_back(0.0);

        for (int k=0; k<Nh; k++) {

            m = n-k;
            if ((m>=0)&&(m<=Nx)) {
                y[n] = y[n] + h[k]*x[n-k];
            }

        }

    }

}

//============================================================
// Method for discrete convolution -- Alternate Offline
//============================================================

void dsptools::CONV2(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h, int& Ny, vector<double>& y) const {

    // Declaring variables
    int n;

    // Computing Ny
    Ny = Nh + Nx;

    // Setting y to zeros
    for (n=0; n<=Ny; n++) {
        y.push_back(0.0);
    }

    // Convolving the two signals
    for (int k=0; k<=Nh; k++) {
        for (int m=0; m<=Nx; m++) {
            n = k+m;
            y[n] = y[n] + h[k]*x[m];
        }
```

```
    }

}

//=============================================================
// Method for discrete convolution -- Real-time
//=============================================================

void dsptools::CONV3(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h, int& Ny, vector<double>& y) const {

    // Declaring variables
    double XX;
    double Y;

    // Calculating Ny
    Ny = Nh + Nx;

    // Initializing X vector to zeros
    vector<double> X;
    for (int n=0; n<=Nh; n++) {
        X.push_back(0.0);
    }

    // Convolving signals
    for (int n=0; n<=Ny; n++) {

        // Getting current signal value (simulating A/D)
        // This is a fix because x is not defined for n>Nx
        // This loop calls values of x up to Ny which is
        // greater than Nx.
        if (n>x.size()) {
            XX = 0.0;
        }
        else {
            XX = x[n];
        }

        // Calling update
        Y = update(XX,X,h,Nh);

        // Storing value
        y.push_back(Y);

    }

}

//=============================================================
// Method for updating state vector for real time convolution
//=============================================================

double dsptools::update(double XX, vector<double>& X, const vector<double>& h, const int Nh) const {

    X[0] = XX;
    double Y = 0.0;

    // Convolving signals
```

```
    for (int k=0; k<=Nh; k++) {
        Y += h[k]*X[k];
    }

    // Updating X
    for (int n=Nh; n>=1; n--) {
        X[n] = X[n-1];
    }
    X[0] = XX;

    return Y;

}

//==============================================================
// Method for computing amplitude response of a signal
//==============================================================

void dsptools::amp(int Nm, const vector<double>& x, int Nx, vector<double>& Am, vector<
    double>& omega) const {

    // Declaring variables
    complex<double> Z;
    complex<double> J(0.0,1.0);
    complex<double> sum;

    for (int k=0; k<Nm; k++) {

        // Computing omega and Z
        omega.push_back(M_PI*(double) k/(double) Nm);
        Z = exp(J*omega[k]);

        // Computing the amplitude response
        sum=0.0;
        for (int n=0; n<Nx; n++) {
            sum += x[n]*pow(Z,n);
        }
        Am.push_back(abs(sum));

    }

}

//==============================================================
// Method for moving average filter
//==============================================================

void dsptools::dsine(double omega_c, const vector<double>& x, vector<double>& y, int k)
    const {

    // Declaring variables
    double Nd;
    int N;

    // Determining value for N using omega_c
    Nd = ceil(2.0*M_PI*((double) k)/omega_c);
    N = (int) Nd;
    cout << "\nWindow size for cutoff frequency of "<< omega_c << " is " << N << ". \n\n";

```

```
    // Computing filtered signal
    for (int n=0; n<x.size(); n++) {

        // Checking for indices < 0
        if (n==0) {
            y.push_back(x[n]/((double) N));
        }
        else if ((n-N)<0) {
            y.push_back(y[n-1] + x[n]/((double) N));
        }
        else {
            y.push_back(y[n-1] + (x[n] - x[n-N])/((double) N));
        }

    }

}

//================================================================
// Method for signal reconstruction
//================================================================

void dsptools::recon(const vector<double>& xx, vector<double>& xr, int N1, int Nh) const {

    // Declaring variables
    vector<double> h (Nh);
    vector<double> y;

    // Generating a lowpass filter
    for (int n=0; n<Nh; n++) {
        if (((n-Nh/2))==0) {
            h[n] = 1.0;
        }
        else {
            h[n] = sin(M_PI*((double) n - ((double) Nh)/2.0)/((double) N1))/(M_PI*((double)
     n - ((double) Nh)/2.0)/((double) N1));
        }
    }

    // Convolving the signals
    int Ny;
    int Nx = (int) xx.size();
    CONV2(Nx,Nh,xx,h,Ny,y);

    // Shifting the output
    for (int n=0; n<xx.size(); n++) {
        xr.push_back(y[n+1+Nh/2]);
    }

}

//================================================================
// Method designing a 4th order IIR bandpass filter
//================================================================

void dsptools::dsine2(const double omega_d1, const double omega_d2) {

    // Declaring some variables
    double A,B,C,D,E,F,G,H;
```

```

    // Order of the analog Butterworth filter
    int N = 2;

    // Prewarping cutoff frequencies
    Omega_c1 = 2.0*tan(omega_d1/2.0);
    Omega_c2 = 2.0*tan(omega_d2/2.0);

    // Determining bandwidth and other parameters
    Omega_0 = sqrt(Omega_c1*Omega_c2);
    Bw = Omega_c2 - Omega_c1;
    theta = M_PI/2.0 + M_PI/(2.0*((double) N));

    // Computing coefficients for difference equation
    A = 16.0 + 4.0*pow(Bw,2) + 16.0*Bw*cos(theta) + 8.0*pow(Omega_0,2) + 4.0*Bw*cos(theta)*pow(Omega_0,2) + pow(Omega_0,4);
    B = -64.0 - 32.0*Bw*cos(theta) + 8.0*Bw*cos(theta)*pow(Omega_0,2) + 4.0*pow(Omega_0,4);
    C = 96.0 - 8.0*pow(Bw,2) - 16.0*pow(Omega_0,2) + 6.0*pow(Omega_0,4);
    D = -64.0 + 32.0*Bw*cos(theta) - 8.0*Bw*cos(theta)*pow(Omega_0,2) + 4.0*pow(Omega_0,4);
    E = 16.0 + 4.0*pow(Bw,2) - 16.0*Bw*cos(theta) + 8.0*pow(Omega_0,2) - 4.0*Bw*cos(theta)*pow(Omega_0,2) + pow(Omega_0,4);
    F = 4.0*pow(Bw,2);
    G = -8.0*pow(Bw,2);
    H = 4.0*pow(Bw,2);

    // Normalizing coefficients
    A1 = F/E;
    A2 = G/E;
    A3 = H/E;
    A4 = A/E;
    A5 = B/E;
    A6 = C/E;
    A7 = D/E;

    // Checking the stability of the filter (the poles must be within the unit circle)
    complex<double> j(0.0,1.0);
    complex<double> z1;
    complex<double> z2;
    complex<double> z3;
    complex<double> z4;
    complex<double> disc1;
    complex<double> disc2;

    // Computing the values of the poles
    disc1 = sqrt(exp(-j*theta)*(pow(Bw,2) - 4.0*exp(2.0*j*theta)*pow(Omega_0,2)));
    disc2 = sqrt(pow(Bw,2)*exp(j*theta) - 4.0*exp(-j*theta)*pow(Omega_0,2));
    z1 = (-exp(j*theta)*(pow(Omega_0,2) - 4.0) + 2.0*exp(j*theta/2.0)*disc1)/(-2.0*Bw+exp(j*theta)*(4.0 + pow(Omega_0,2)));
    z2 = exp(j*theta/2.0)*(exp(j*theta/2.0)*(-4.0 + pow(Omega_0,2)) + 2.0*disc1)/(2*Bw - exp(j*theta)*(4.0 + pow(Omega_0,2)));
    z3 = (4.0 - pow(Omega_0,2) + 2.0*exp(j*theta/2.0)*disc2)/(4.0 - 2.0*Bw*exp(j*theta) + pow(Omega_0,2));
    z4 = (-4.0 + pow(Omega_0,2) + 2.0*exp(j*theta/2.0)*disc2)/(-4.0 + 2.0*Bw*exp(j*theta) - pow(Omega_0,2));

    // Checking the magnitude of the poles to make sure they are inside the unit circle.
    if ((abs(z1)>=1.0) || (abs(z2)>=1.0) || (abs(z3)>=1.0) || (abs(z4)>=1.0)) {
        cerr << "\nERROR: Resulting IIR filter is unstable." << endl;
        cerr << "Poles are " << z1 << " " << z2 << " " << z3 << " " << z4 << endl;
```

```
        cerr << "Ending program.\n\n";
        exit(1);
    }

    // Printing some information
    cout << "\nThe IIR filter is stable." << endl;
    cout << "Poles are " << z1 << " " << z2 << " " << z3 << " " << z4 << "\n" << endl;

}

//==============================================================
// Method that actually applies the filter
//==============================================================

void dsptools::filt(const vector<double>& x, vector<double>& y) const {

    // Initializing values
    y[0] = A1*x[0];
    y[1] = A1*x[1] - A7*y[0];
    y[2] = A1*x[2] + A2*x[0] - A6*y[0] - A7*y[1];
    y[3] = A1*x[3] + A2*x[1] - A5*y[0] - A6*y[1] - A7*y[2];

    // Iterating through the rest of the vector
    for (int n=4; n<x.size(); n++) {
        y[n] = A1*x[n] + A2*x[n-2] + A3*x[n-4] - A4*y[n-4] - A5*y[n-3] - A6*y[n-2] - A7*y[n
    -1];
    }

}

//==============================================================
// Method to compute the amplitude response of the filter
//==============================================================

void dsptools::amp2(int Nm, vector<double>& omega, vector<double>& amp) const {

    // The amplitude response of the filter is determined by
    // substituting z = e^{j\omega}

    // Declaring variables
    complex<double> j(0.0,1.0);
    complex<double> z;

    // Setting up omega vector
    double domega = M_PI/(((double) Nm) - 1.0);
    for (int n=0; n<Nm; n++) {
        omega.push_back(((double) n)*domega);
    }

    // Determining amplitude response
    for (int k=0; k<Nm; k++) {
        z = exp(j*omega[k]);
        amp.push_back(abs((A1 + A2*pow(z,-2) + A3*pow(z,-4))/(A4*pow(z,-4) + A5*pow(z,-3) +
     A6*pow(z,-2) + A7*pow(z,-1) + 1.0)));
    }

}
```

Listing 3: DSP Tools header

```
#ifndef SIGTOOLSHEADER
#define SIGTOOLSHEADER

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <cmath>

using namespace std;

/*
================================================================
| This class holds methods developed in the EE 5350 Digital    |
| Signal Processing class in the summer of 2014.               |
|                                                              |
| Methods:                                                     |
|    - writeSignal() method that writes a signal to a data file.|
|    - CONV() method for discrete convolution using a standard  |
|      offline method.                                         |
|    - CONV2() method for discrete convolution using an         |
|      alternative offline method.                             |
|    - CONV3() method for real time discrete convolution.       |
|    - update() method that updates the state vector for real   |
|      time discrete convolution.                              |
|    - amp() computes the amplitude response of a signal.       |
|    - dsine() is a method for a moving average filter.         |
|    - recon() is a method that uses bandlimited interpolation  |
|      to reconstruct a signal.                                |
|                                                              |
| Author       : James Grisham                                 |
| Date         : 06/03/2014                                    |
| Revision Date: 07/12/2014                                    |
================================================================
*/

class dsptools
{

private:

// These are private members used in the dsine2 method that
// designs a 4th order Butterworth bandpass filter
double A1, A2, A3, A4, A5, A6, A7;
double Bw, Omega_c1, Omega_c2, theta, Omega_0;

public:

// Method for writing signal to file
void writeSignal(string filename, const vector<double>& x) const;

// Method for discrete convolution -- Standard offline method
void CONV(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h, int
    & Ny, vector<double>& y) const;

// Method for discrete convolution -- Alternate offline method
void CONV2(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h,
```

```
    int& Ny, vector<double>& y) const;

// Method for discrete convolution -- Real-time version
void CONV3(const int Nx, const int Nh, const vector<double>& x, const vector<double>& h, int& Ny, vector<double>& y) const;

// Method for updating state vector for real time convolution
double update(double XX, vector<double>& x, const vector<double>& h, const int Nh) const;

// Method for computing amplitude response
void amp(int Nm, const vector<double>& x, int Nx, vector<double>& Am, vector<double>& omega ) const;

// Method for moving average filter
void dsine(double omega_c, const vector<double>& x, vector<double>& y, int k) const;

// Method for signal reconstruction
void recon(const vector<double>& xx, vector<double>& xr, int N1, int Nh) const;

// Method for designing 4th-order Butterworth bandpass filter
void dsine2(const double omega_d1, const double omega_d2);

// Method for applying the 4th order Butterworth filter
void filt(const vector<double>& x, vector<double>& y) const;

// Method to compute amplitude response of filter
void amp2(int Nm, vector<double>& omega, vector<double>& amp) const;

};

#endif
```

Listing 4: Makefile

```
CC=g++
CCFLAGS= -O
LIBS=
INCLUDE=

all : prog1 prog2 prog3 prog4 prog6

dsptools.o : dsptools.cpp dsptools.hpp
	$(CC) -c $(CCFLAGS) dsptools.cpp

prog1.o : prog1.cpp dsptools.hpp
	$(CC) -c $(CCFLAGS) prog1.cpp

prog1 : dsptools.o prog1.o
	$(CC) $(CCFLAGS) -o prog1 dsptools.o prog1.o

prog2.o : prog2.cpp dsptools.hpp
	$(CC) -c $(CCFlAGS) prog2.cpp

prog2 : dsptools.o prog2.o
	$(CC) $(CCFlAGS) -o prog2 dsptools.o prog2.o

prog3.o : prog3.cpp dsptools.hpp
	$(CC) -c $(CCFlAGS) prog3.cpp

```

```
prog3 : dsptools.o prog3.o
    $(CC) $(CCFlAGS) -o prog3 dsptools.o prog3.o

prog4.o : prog4.cpp dsptools.hpp
    $(CC) -c $(CCFLAGS) prog4.cpp

prog4 : dsptools.o prog4.o
    $(CC) $(CCFLAGS) -o prog4 dsptools.o prog4.o

prog6.o : prog6.cpp dsptools.hpp
    $(CC) -c $(CCFLAGS) prog6.cpp

prog6 : dsptools.o prog6.o
    $(CC) $(CCFLAGS) -o prog6 dsptools.o prog6.o

clean :
    rm -rf *.o
```

Listing 5: Python script

```
#!/usr/bin/env python

"""

This script imports data files that were written by the C++ program and
creates stem plots.

Author       : James Grisham
Date         : 08/07/2014
Revision Date:

"""

import sys
import os
import numpy as np
import matplotlib.pyplot as plt

#================================================================
# Class for importing and plotting data
#================================================================


class Signal(object):
    """
    This is a class for importing and plotting discrete signals.
    """

    # Default constructor
    def __init__(self, file_name, data_name):
        self.file_name = file_name
        self.data_name = data_name
        self.n = []
        self.x = []
        self.omega = []
        self.H = []

    # Method for assigning data
    def assign_data(self, xdata, ydata):
```

```
        self.n = xdata
        self.x = ydata

    # Method for importing data
    def import_data(self):

        # Check to see if file exists
        if not os.path.isfile(self.file_name):
            print("\nERROR: {} does not exist.".format(self.file_name))
            sys.exit(1)

        # Importing data
        data = np.genfromtxt(self.file_name)
        self.n = data[:, 0]
        self.x = data[:, 1]

    # Method for importing and plotting amplitude spectra
    def plot_amp(self, plot_width, plot_height, save_name):

        # Checking to see if file exists
        if not os.path.isfile(self.file_name):
            print("\nERROR: {} does not exist.".format(self.file_name))
            sys.exit(1)

        # Importing data
        d = np.genfromtxt(self.file_name)
        self.omega = d[:, 0]
        self.H = d[:, 1]

        # Plotting
        fig = plt.figure(figsize=(plot_width, plot_height))
        plt.plot(self.omega, self.H, "-k")
        plt.xlabel(r"$\omega$")
        plt.ylabel(r"$|"+self.data_name+"(e^{j\omega})|$")
        plt.gca().set_ylim([min(self.H) - 0.1, max(self.H) + 0.1])
        plt.gca().set_xlim([min(self.omega), max(self.omega)])
        plt.tight_layout()
        if save_name is not None:
            fig.savefig(save_name)

    # Method for plotting the data
    def plot_data(self, plot_width, plot_height, save_name):
        fig = plt.figure(figsize=(plot_width, plot_height))
        markerline, stemlines, baseline = plt.stem(self.n, self.x, "-k")
        plt.xlabel(r"$n$")
        plt.ylabel(r"${}[n]$".format(self.data_name))
        plt.gca().set_ylim([min(self.x) - 0.25, max(self.x) + 0.25])
        plt.gca().set_xlim([min(self.n), max(self.n)])
        plt.tight_layout()
        plt.setp(markerline, "markerfacecolor", "k")
        plt.setp(baseline, "color", "k", "linewidth", 1.5)
        if save_name is not None:
            fig.savefig(save_name)

#==============================================================
# Importing and plotting data
#==============================================================

# Setting some defaults
```

```python
fontsize = 14
font = {"size": fontsize}
plt.rc("text", usetex=True)
plt.rc("font", **{"family": "serif", "serif": ["Computer Modern"]})
plt_width = 7.25
plt_height = 3.0

# Directory in which images will be saved
imgDir = "/Users/user/Desktop/School/Courses/UTA/" + \
         "EE 5350 - Digital Signal Processing/Images/"
if sys.platform == "linux2":
    imgDir = "/home/james/Desktop/School/Courses/EE 5350 - Digital Signal Processing/Images/"

# Making directory if it doesn't exist
if not os.path.isdir(imgDir):
    os.mkdir(imgDir)

# File names
x_file = "./DataFiles/prog6_x.dat"
X_file = "./DataFiles/prog6_X.dat"
y_file = "./DataFiles/prog6_y.dat"
Y_file = "./DataFiles/prog6_Y.dat"
h_file = "./DataFiles/prog6_h.dat"
H1_file = "./DataFiles/prog6_Hamp1.dat"
H2_file = "./DataFiles/prog6_Hamp2.dat"
files_list = [x_file, y_file, h_file]
data_names = ["x", "y", "h"]
amp_list = [X_file, Y_file, H1_file, H2_file]
amp_dnames = ["X", "Y", "H_1", "H_2"]
amp_fnames = [imgDir+"prog6_X.pdf", imgDir+"prog6_Y.pdf", imgDir+"prog6_H1.pdf", imgDir+"prog6_H2.pdf"]
save_names = [imgDir+"prog6_x.pdf", imgDir+"prog6_y.pdf", imgDir+"prog6_h.pdf"]

# Instantiating new signal objects
signal_object = []
for n in range(0, len(files_list)):
    signal_object.append(Signal(files_list[n], data_names[n]))

# Iterating through the objects, importing and plotting
n = 0
for obj in signal_object:
    obj.import_data()
    obj.plot_data(plt_width, plt_height, save_names[n])
    n += 1

# Importing and plotting amplitude spectra
amp_obj = []
for f, d in zip(amp_list, amp_dnames):
    amp_obj.append(Signal(f, d))

# Iterating through the objects and plotting
n = 0
for obj in amp_obj:
    obj.plot_amp(plt_width, plt_height, amp_fnames[n])
    n += 1

# Showing plots
plt.show()
```

Program assignment 6

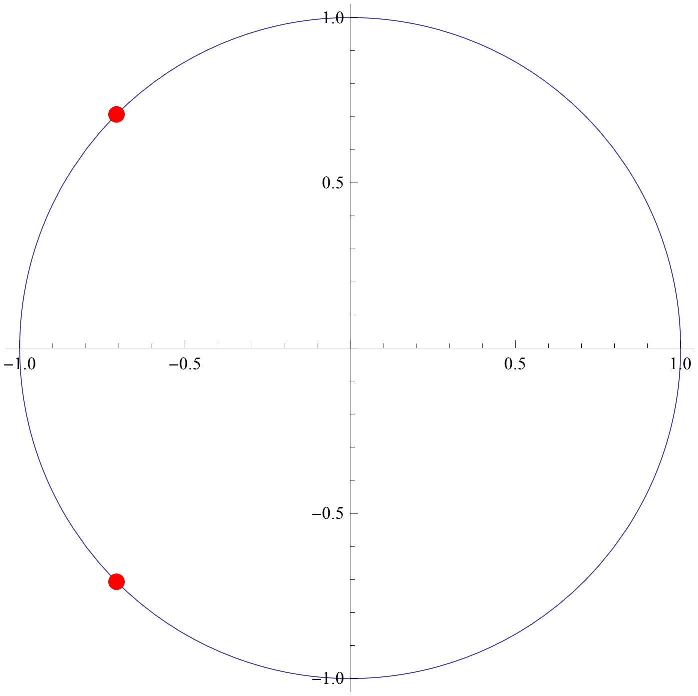
James Grisham

```
In[43]:= ClearAll["Global`*"]
```

Analog prototype filter

```
In[44]:= N1 = 2; (* This is the order of the prototype LP filter *)

In[45]:= θ[k_] := Pi / 2.0 + Pi / (2 * N1) + (k - 1) * Pi / N1

In[46]:= poles = Table[Exp[I * θ[k]], {k, 2}];

In[47]:= ph1 = Plot[Sqrt[1 - x^2], {x, -1, 1}];

In[48]:= ph2 = Plot[-Sqrt[1 - x^2], {x, -1, 1}];

In[49]:= marker = Graphics[{Red, Disk[]}];

In[50]:= data = Table[{Re[poles[[i]]], Im[poles[[i]]]}, {i, 2}];

In[51]:= ph3 = ListPlot[data, PlotMarkers → {marker, 0.025}];

In[52]:= Show[ph1, ph2, ph3, PlotRange → Automatic, AspectRatio → Automatic]
```

Out[52]=




```
In[53]:= (* Transfer function for analog prototype filter *)
```

In[54]:= `H1 = 1 / (s1^2 - 2 * Cos[θ] * s1 + 1)`

Out[54]= $$\frac{1}{1 + s1^2 - 2\, s1 \operatorname{Cos}[\theta]}$$

Transforming from analog LP to analog BP

In[55]:= `T = 1;`

In[56]:= `Ha = H1 /. s1 → (s^2 + Ω0^2) / (s * Bw) // Apart`

Out[56]= $$\frac{Bw^2\, s^2}{Bw^2\, s^2 + s^4 - 2\, Bw\, s^3 \operatorname{Cos}[\theta] + 2\, s^2\, \Omega_0^2 - 2\, Bw\, s \operatorname{Cos}[\theta]\, \Omega_0^2 + \Omega_0^4}$$

Transforming from $H_a(s)$ to $H(z)$

In[57]:= `Hz = Ha /. s → 2 / T * (1 - z1) / (1 + z1) // Apart`

Out[57]= $$\left(4\, Bw^2\, (-1 + z1)^2\, (1 + z1)^2\right) / \left(16 + 4\, Bw^2 - 64\, z1 + 96\, z1^2 - 8\, Bw^2\, z1^2 - 64\, z1^3 + 16\, z1^4 + 4\, Bw^2\, z1^4 - 16\, Bw \operatorname{Cos}[\theta] + 32\, Bw\, z1 \operatorname{Cos}[\theta] - 32\, Bw\, z1^3 \operatorname{Cos}[\theta] + 16\, Bw\, z1^4 \operatorname{Cos}[\theta] + 8\, \Omega_0^2 - 16\, z1^2\, \Omega_0^2 + 8\, z1^4\, \Omega_0^2 - 4\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 - 8\, Bw\, z1 \operatorname{Cos}[\theta]\, \Omega_0^2 + 8\, Bw\, z1^3 \operatorname{Cos}[\theta]\, \Omega_0^2 + 4\, Bw\, z1^4 \operatorname{Cos}[\theta]\, \Omega_0^2 + \Omega_0^4 + 4\, z1\, \Omega_0^4 + 6\, z1^2\, \Omega_0^4 + 4\, z1^3\, \Omega_0^4 + z1^4\, \Omega_0^4\right)$$

In[58]:= `Numerator[Hz] // Expand`

Out[58]= $4\, Bw^2 - 8\, Bw^2\, z1^2 + 4\, Bw^2\, z1^4$

In[65]:= `Collect[Denominator[Hz], z1]`

Out[65]= $$16 + 4\, Bw^2 - 16\, Bw \operatorname{Cos}[\theta] + 8\, \Omega_0^2 - 4\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 + \Omega_0^4 + z1^4 \left(16 + 4\, Bw^2 + 16\, Bw \operatorname{Cos}[\theta] + 8\, \Omega_0^2 + 4\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 + \Omega_0^4\right) + z1 \left(-64 + 32\, Bw \operatorname{Cos}[\theta] - 8\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 + 4\, \Omega_0^4\right) + z1^3 \left(-64 - 32\, Bw \operatorname{Cos}[\theta] + 8\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 + 4\, \Omega_0^4\right) + z1^2 \left(96 - 8\, Bw^2 - 16\, \Omega_0^2 + 6\, \Omega_0^4\right)$$

The poles of H(z) must be within the unit circle for the filter to be stable.

In[66]:= `p = Denominator[Hz] /. z1 → z^(-1)`

Out[66]= $$16 + 4\, Bw^2 + \frac{16}{z^4} + \frac{4\, Bw^2}{z^4} - \frac{64}{z^3} + \frac{96}{z^2} - \frac{8\, Bw^2}{z^2} - \frac{64}{z} - 16\, Bw \operatorname{Cos}[\theta] + \frac{16\, Bw \operatorname{Cos}[\theta]}{z^4} - \frac{32\, Bw \operatorname{Cos}[\theta]}{z^3} + \frac{32\, Bw \operatorname{Cos}[\theta]}{z} + 8\, \Omega_0^2 + \frac{8\, \Omega_0^2}{z^4} - \frac{16\, \Omega_0^2}{z^2} - 4\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2 + \frac{4\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2}{z^4} + \frac{8\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2}{z^3} - \frac{8\, Bw \operatorname{Cos}[\theta]\, \Omega_0^2}{z} + \Omega_0^4 + \frac{\Omega_0^4}{z^4} + \frac{4\, \Omega_0^4}{z^3} + \frac{6\, \Omega_0^4}{z^2} + \frac{4\, \Omega_0^4}{z}$$

In[68]:= **Solve[p == 0, z] // FullSimplify**

Out[68]=
$$\left\{\left\{z \to \frac{-e^{i\theta}\left(-4+\Omega_0^2\right)+2\,e^{\frac{i\theta}{2}}\sqrt{e^{-i\theta}\left(Bw^2-4\,e^{2i\theta}\,\Omega_0^2\right)}}{-2\,Bw+e^{i\theta}\left(4+\Omega_0^2\right)}\right\},\right.$$
$$\left\{z \to \frac{e^{\frac{i\theta}{2}}\left(e^{\frac{i\theta}{2}}\left(-4+\Omega_0^2\right)+2\sqrt{e^{-i\theta}\left(Bw^2-4\,e^{2i\theta}\,\Omega_0^2\right)}\right)}{2\,Bw-e^{i\theta}\left(4+\Omega_0^2\right)}\right\},$$
$$\left.\left\{z \to \frac{4-\Omega_0^2+2\,e^{\frac{i\theta}{2}}\sqrt{Bw^2\,e^{i\theta}-4\,e^{-i\theta}\,\Omega_0^2}}{4-2\,Bw\,e^{i\theta}+\Omega_0^2}\right\},\left\{z \to \frac{-4+\Omega_0^2+2\,e^{\frac{i\theta}{2}}\sqrt{Bw^2\,e^{i\theta}-4\,e^{-i\theta}\,\Omega_0^2}}{-4+2\,Bw\,e^{i\theta}-\Omega_0^2}\right\}\right\}$$