# ECE444: HW6

Thomas Smallarz

November 15, 2020

## Contents

# Design

## Steps

From the information learned in Chapter 2 and Chapter 8 the general flow of designing this filter will be:

1. Compute Poles/Zeros for CT Lowpass Inverse Chebyshev Prototype filter (based on Ex. 2.17)
2. Compute pre-warped frequencies using simplified method (based on Ex. 8.7)
3. Compute Poles/Zeros for DT Bandpass Inverse Chebyshev filter H(z) (based on Ex. 8.7)
4. Sort DT Poles/Zeros so that Poles/Zeros are can be matched with their conjugate, and are ordered so pairs of Poles/Zeros that are nearest each other can be created into 2nd order systems
5. Combine conjugate pairs to create second order systems
6. Output coefficients to header file

This general structure will help keep us on track as we trek through this problem. Before we begin computing P/Z's for anything, we should first make sure we understand all the requirements. Some of them are pretty straightforward, such as the sampling frequency requirement, but let's look at the other ones.

- Selectable factor-of-2 filter order K

This is initially frightening to read, but if we look later in the assignment under implementation we see that we don't have to adjust the order K on the hardware, but be able to adjust it in our design calculations then export the coefficients we generate to our hardware.

- Selectable 2dB lower passband frequency $= 250Hz \leq f_{p1} \leq 2000Hz$. The upper 2dB passband frequency is determined as $f_{p2} = \frac{F_s}{2} - f_{p1}$

This isn't too bad. This is almost the same thing as the selectable filter order. We don't have to have some equation on our hardware calculating different passband frequencies. We only need to implement four different lower passband frequencies to our hardware.

- Inverse Chebyshev BPF with max stopband gain of 0.1 and design method of bilinear transform with pre-warping that preserves 2dB passband frequencies

This will dictate which sections of the book we need to read, and takes a lot of the decision making off our backs – which is nice.

- Output filter coefficients for realization as cascade of 2nd Order DFII filter stages.

Main takeaway from this is that our final realization should be a cascade of 2nd Order filter stages. Whether they are DFI, DFII, or transpose versions only matters when we get to programming. This also takes design decisions off our backs which is helpful.

## Computed Poles/Zeros for CT Lowpass Inverse Chebyshev Prototype filter

We know from our general structure listed above that the first step is to design a Continuous Time Lowpass Inverse Chebyshev Prototype Filter with $\omega_0 = 1\frac{rad}{sec}$. We know from Chapter 2 that we traditionally use filter transformations to create HP, BS, and BP filters from LP filters. We know from Chapter 8 that we can transform a CT to a DT filter with methods such as the bilinear transform. Combined with the design requirement of an Inverse Chebyshev filter type, we are led to start with Section *2.7.3 – Inverse Chebyshev Filters*.

From Sections 2.7.2, 2.7.3 and Example 2.17 in this chapter we read that we must first determine a Lowpass Prototype filter. Below are the equations used to do this based on the passband ripple, stopband ripple, and filter order.

Eq. 2.53: Stopband gain

$$\epsilon^2 = \frac{1}{10^{\alpha_s/10} - 1}$$

Eq. 2.47: Type I Chebyshev Poles

$$p_k = -\omega_p \sinh\left[\frac{1}{K}\sinh^{-1}\left(\frac{1}{\epsilon}\right)\right]\sin\left[\frac{\pi(2k-1)}{2K}\right] + j\omega_p \cosh\left[\frac{1}{K}\sinh^{-1}\left(\frac{1}{\epsilon}\right)\right]\cos\left[\frac{\pi(2k-1)}{2K}\right] \qquad k = 1, 2, \ldots, K$$

Eq. 2.55: Type II Chebyshev Poles, where $p'_k$ is the poles calculated in Eq. 2.47

$$p_k = \frac{\omega_p \omega_s}{p'_k}$$

Eq. 2.56: Type II Chebyshev Zeros

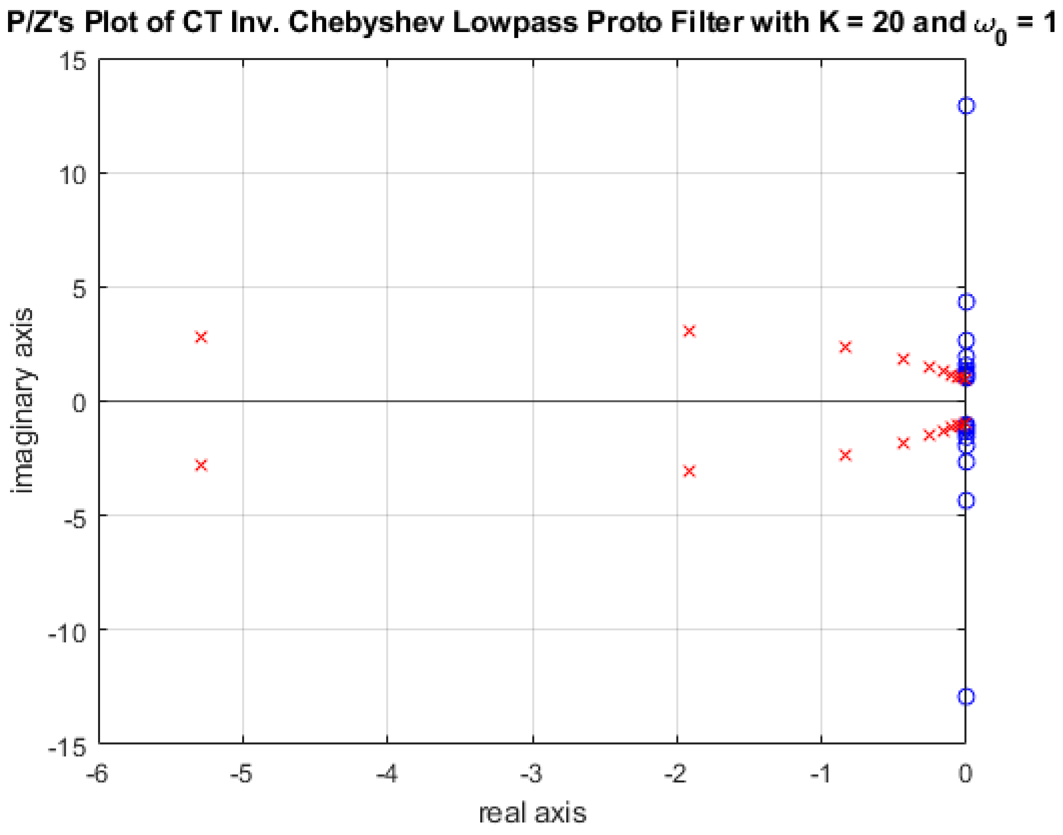
$$z_k = j\omega_s \sec\left(\frac{\pi(2k-1)}{2K}\right) \qquad k = 1, 2, \ldots, K$$

By copying and modifying the MATLAB code found on page 143 we can calculate the poles and zeros for our prototype filter based on the passband ripple, stopband ripple, and order of the filter.

```
%% Design Low-Pass Inverse Chebyshev Prototype Filter with normalized freq
% Based on Ex. 2.17
alphap = 2; % pass-band alpha of 2dB
alphas = 20; % stop-band alpha of 20dB
omegap = 1; % prototype filter cutoff freq of 1 rad/sec
% calculate stop-band frequency based on alpha's and omega
omegas = omegap*cosh(acosh(sqrt((10^(alphas/10)-1)/(10^(alphap/10)-1)))/K);
epsilon = 1/sqrt(10^(alphas/10)-1);
k = 1:K;
% calculate poles
pk = -omegap*sinh(asinh(1/epsilon)/K)*sin(pi*(2*k-1)/(2*K))+...
    1j*omegap*cosh(asinh(1/epsilon)/K)*cos(pi*(2*k-1)/(2*K));
pk = omegap*omegas./pk;
% calculate zeros
zk = 1j*omegas.*sec(pi*(2*k-1)/(2*K));
% calculate coefficients of expanded form based on poles/zeros
B = prod(pk./zk)*poly(zk); A = poly(pk);
```

This pk (Poles) and zk (Zeros) of this CT Inverse Chebyshev Lowpass Prototype filter are below:

| pk: | zk: |
|---|---|
| -0.0118 + 1.0050i | 0.0000 - 1.0164i |
| -0.0367 + 1.0292i | 0.0000 - 1.0421i |
| -0.0665 + 1.0805i | 0.0000 - 1.0968i |
| -0.1061 + 1.1656i | 0.0000 - 1.1884i |
| -0.1645 + 1.2969i | 0.0000 - 1.3326i |
| -0.2605 + 1.4976i | 0.0000 - 1.5602i |
| -0.4391 + 1.8114i | 0.0000 - 1.9393i |
| -0.8321 + 2.3201i | 0.0000 - 2.6479i |
| -1.9207 + 3.1040i | 0.0000 - 4.3406i |
| -5.2833 + 2.7990i | 0.0000 -12.9149i |
| -5.2833 - 2.7990i | 0.0000 +12.9149i |
| -1.9207 - 3.1040i | 0.0000 + 4.3406i |
| -0.8321 - 2.3201i | 0.0000 + 2.6479i |
| -0.4391 - 1.8114i | 0.0000 + 1.9393i |
| -0.2605 - 1.4976i | 0.0000 + 1.5602i |
| -0.1645 - 1.2969i | 0.0000 + 1.3326i |
| -0.1061 - 1.1656i | 0.0000 + 1.1884i |
| -0.0665 - 1.0805i | 0.0000 + 1.0968i |
| -0.0367 - 1.0292i | 0.0000 + 1.0421i |
| -0.0118 - 1.0050i | 0.0000 + 1.0164i |



Nice. We see that our poles and zeros all have conjugate pairs. Our zeros are all along the $s = j\omega$ axis which we know from the bilinear transform that this will map them all to the circle $|z| = 1$. Our poles are all in the left half plane, which means this system is stable, which is of course expected. Our poles do get close to the marginally stable region of $s = j\omega$.

## Compute Poles/Zeros for DT Bandpass Inverse Chebyshev filterer

Next, we need to convert these poles/zeros to a DT Bandpass version. Reading through Chapter 8, we see that in section 8.1.4 it goes through an example of designing a Digital Bandpass IIR Filter. This example describes about how we can do a filter transformation (Lowpass to Bandpass) and bilinear transform with pre-warping in one (ish) steps, instead of doing a CT transformation, then the bilinear transform separately.

There are two stages: first is to compute our $c_1$ and $c_2$ values from our pre-warped frequencies, second is computing poles/zeros for DT Bandpass Filter.

### *Compute $c_1$ and $c_2$ based on pre-warped frequencies*

From the book we know our pre-warped frequencies and our $c_1$ and $c_2$ values are:

Eq. 8.19:

$$\omega'_m = \tan\left(\frac{\omega_m T}{2}\right)$$

Eq. 8.24:

$$c_1 = \frac{\omega'_1\omega'_2 - 1}{\omega'_1\omega'_2 + 1} \quad \text{and} \quad c_2 = \frac{\omega'_2 - \omega'_1}{\omega'_1\omega'_2 + 1}$$

In MATLAB, this is:

```
F = [250 500 1000 2000];
K = 20;
Fs = 10000; T = 1 / Fs;

fp1 = F(k);
fp2 = Fs / 2 - fp1; % upper pass band frequencies (Hz)

wp1 = 2*pi.*fp1; wp2 = 2*pi.*fp2; % convert from Hz to Rad/Sec
wp1_w = tan(wp1.*T/2); % SIMPLIFIED procedure for pre-warped lower passband omegas
wp2_w = tan(wp2.*T/2); % SIMPLIFIED procedure for pre-warped upper passband omegas

c1 = (wp1_w*wp2_w - 1) / (wp1_w*wp2_w + 1);
c2 = (wp2_w - wp1_w) / (wp1_w*wp2_w + 1);
```

*note that in my MATLAB code, I have a for-loop that calculates all the necessary coefficients for each starting passband frequency in the variable F. That is why there is the $f_{p1} = F(k)$ line*

### *Compute Poles/Zeros and Gain for DT Bandpass Inverse Chebyshev filter H(z)*

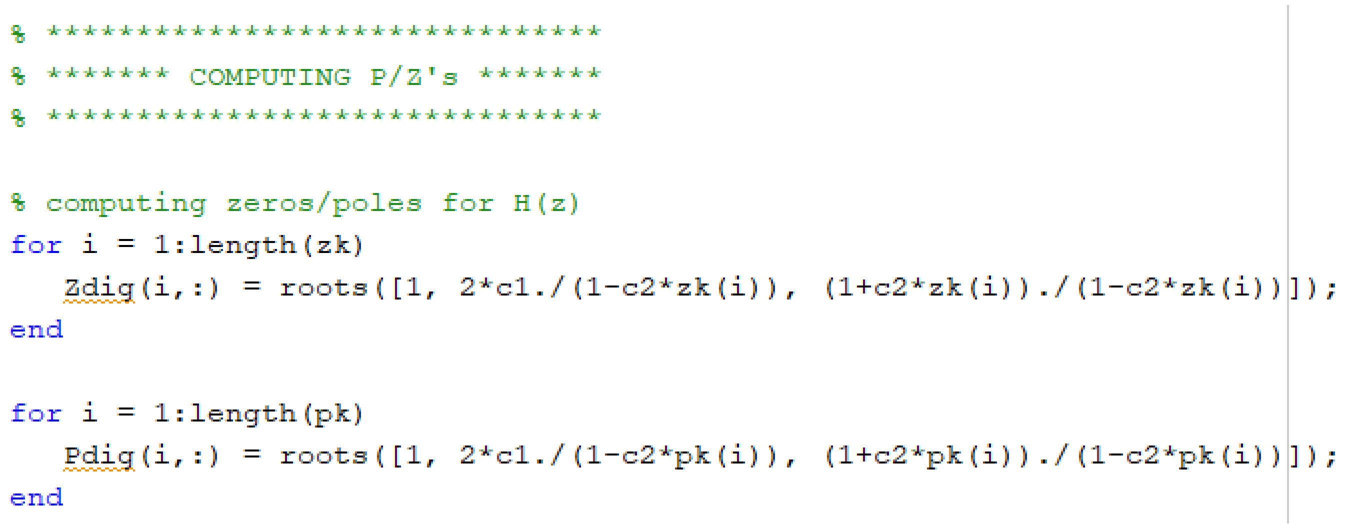
Nice. Now we have our $c_1$ and $c_2$ values. Now we can calculate our DT Poles/Zeros and Gain. In the book:

Eq. 8.25:

$$H(z) = \left( \frac{b_L \prod_{l=1}^{L} (\frac{1}{c_2} - z_l)}{a_K \prod_{k=1}^{K} (\frac{1}{c_2} - p_k)} \right) \frac{\prod_{l=1}^{L} \left( z^2 + \left( \frac{2c_1}{1-c_2 z_l} \right) z + \frac{1+c_2 z_l}{1-c_2 z_l} \right)}{\prod_{k=1}^{K} \left( z^2 + \left( \frac{2c_1}{1-c_2 p_k} \right) z + \frac{1+c_2 p_k}{1-c_2 p_k} \right)} \left( z^2 - 1 \right)^{K-L}$$

From this equation we can see that we will have a constant gain factor in front, and a bunch (K) of 2nd Order Pole/Zero equations. The last $(z^2 - 1)$ term will just be a one, since our K and L values are the same. Calculations in MATLAB, including using roots( ) command, will give us our Poles/Zeros and gain. For a K = 20 filter, with $f_{p1} = 1kHz$ we get:

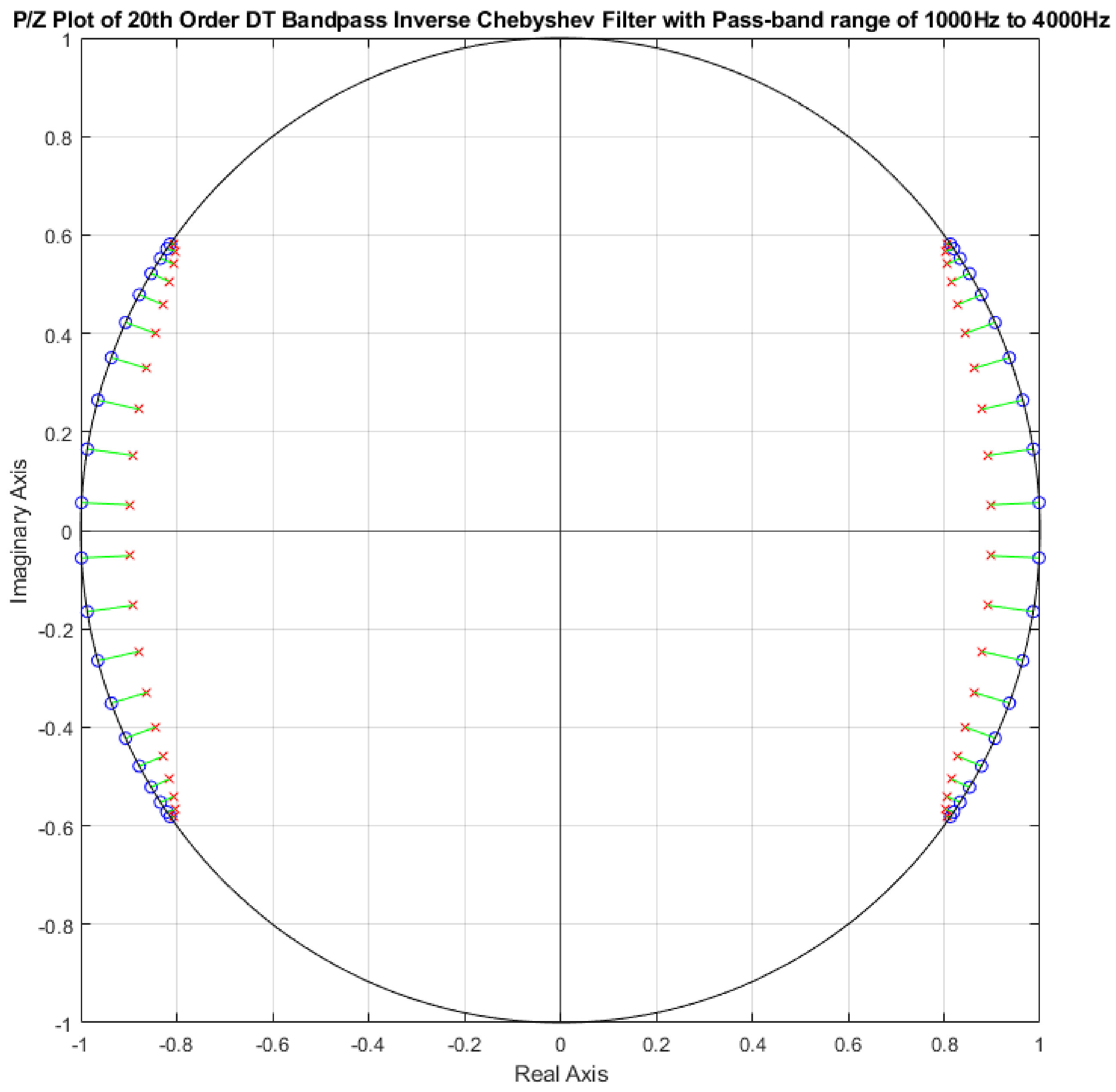
| Zeros | | Poles | |
|---|---|---|---|
| 0.8135 - 0.5815i | -0.8135 + 0.5815i | 0.8059 + 0.5826i | -0.8059 - 0.5826i |
| -0.8203 + 0.5719i | 0.8203 - 0.5719i | 0.8036 + 0.5668i | -0.8036 - 0.5668i |
| 0.8337 - 0.5523i | -0.8337 + 0.5523i | 0.8072 + 0.5414i | -0.8072 - 0.5414i |
| -0.8532 + 0.5216i | 0.8532 - 0.5216i | 0.8161 + 0.5057i | -0.8161 - 0.5057i |
| 0.8780 - 0.4787i | -0.8780 + 0.4787i | 0.8292 + 0.4589i | -0.8292 - 0.4589i |
| 0.9065 - 0.4221i | -0.9065 + 0.4221i | 0.8452 + 0.4002i | -0.8452 - 0.4002i |
| -0.9364 + 0.3508i | 0.9364 - 0.3508i | 0.8621 + 0.3290i | -0.8621 - 0.3290i |
| 0.9644 - 0.2646i | -0.9644 + 0.2646i | 0.8778 + 0.2458i | -0.8778 - 0.2458i |
| 0.9863 - 0.1651i | -0.9863 + 0.1651i | 0.8900 + 0.1523i | -0.8900 - 0.1523i |
| 0.9984 - 0.0562i | -0.9984 + 0.0562i | 0.8966 + 0.0516i | -0.8966 - 0.0516i |
| 0.9984 + 0.0562i | -0.9984 - 0.0562i | 0.8966 - 0.0516i | -0.8966 + 0.0516i |
| -0.9863 - 0.1651i | 0.9863 + 0.1651i | 0.8900 - 0.1523i | -0.8900 + 0.1523i |
| -0.9644 - 0.2646i | 0.9644 + 0.2646i | 0.8778 - 0.2458i | -0.8778 + 0.2458i |
| 0.9364 + 0.3508i | -0.9364 - 0.3508i | 0.8621 - 0.3290i | -0.8621 + 0.3290i |
| -0.9065 - 0.4221i | 0.9065 + 0.4221i | 0.8452 - 0.4002i | -0.8452 + 0.4002i |
| -0.8780 - 0.4787i | 0.8780 + 0.4787i | 0.8292 - 0.4589i | -0.8292 + 0.4589i |
| -0.8532 - 0.5216i | 0.8532 + 0.5216i | 0.8161 - 0.5057i | -0.8161 + 0.5057i |
| 0.8337 + 0.5523i | -0.8337 - 0.5523i | 0.8072 - 0.5414i | -0.8072 + 0.5414i |
| -0.8203 - 0.5719i | 0.8203 + 0.5719i | 0.8036 - 0.5668i | -0.8036 + 0.5668i |
| 0.8135 + 0.5815i | -0.8135 - 0.5815i | 0.8059 - 0.5826i | -0.8059 + 0.5826i |



```
% ******************************
% ******* COMPUTING P/Z's *******
% ******************************

% computing zeros/poles for H(z)
for i = 1:length(zk)
   Zdig(i,:) = roots([1, 2*c1./(1-c2*zk(i)), (1+c2*zk(i))./(1-c2*zk(i))]);
end

for i = 1:length(pk)
   Pdig(i,:) = roots([1, 2*c1./(1-c2*pk(i)), (1+c2*pk(i))./(1-c2*pk(i))]);
end
```

Computing Gain

```
G = B(1)/A(1)*prod(1/c2-zk)/prod(1/c2-pk);
```



## Sorting DT Poles/Zeros

Now we have all our Poles/Zeros and Gain. This is nice, but from eq. 8.25 they don't match up with their conjugate pairs to produce real 2nd order systems. We, at the end of the day, want a cascade of 2nd order system for our even number filter order.

This means we must find a way to sort these poles/zeros so they align nicely. From the book in section 8.1.5 we see that for a cascade of 2nd Order sections we want to follow some other steps, such as:

- Pair conjugates together to get real 2nd Order sections
- Pair conjugate pole pairs with conjugate zero pairs that are closest to each other

- Reverse order, where poles closest to the unit circle are the last section, is typically common
- Filter gain is either applied before, after, or distributed to each stage

This is a lot to take it. Let's start with sorting our poles/zeros and see if they happen to line up with each other like they should.

Just from looking at the plot of our poles/zeros, we know that every one has a conjugate pair somewhere. So if we can sort by their imaginary portion, we can just use the conj( ) function in MATLAB to calculate its conjuage.

```
% appending columns of zeros, then sorting by imag value
temp = -j.*Zdig;
temp2 = sort([temp(:,1);temp(:,2)],'ComparisonMethod','real');
Zdig_sort = j.*temp2;
clear temp temp2;

temp = -j.*Pdig;
temp3 = sort([temp(:,1);temp(:,2)],'ComparisonMethod','real');
Pdig_sort = j.*temp3;
clear temp temp2;

Ztemp = Zdig_sort([1:2:length(Zdig_sort)/2-1],:);
Ptemp = Pdig_sort([1:2:length(Pdig_sort)/2-1],:);

Ztemp2 = abs(real(Ztemp(:))) + j.* abs(imag(Ztemp(:)));
Ptemp2 = abs(real(Ptemp(:))) + j.* abs(imag(Ptemp(:)));

Ztemp3 = zeros(length(Ztemp2)*4,1);
Ptemp3 = zeros(length(Ptemp2)*4,1);

for i = 1:length(Ztemp2)
   Ztemp3(2*i-1) = Ztemp2(i);
   Ztemp3(2*i) = -real(Ztemp2(i)) + j*imag(Ztemp2(i));
   Ztemp3(length(Ztemp3)-2*i+2) = conj(Ztemp2(i));
   Ztemp3(length(Ztemp3)-2*i+1) = conj(-real(Ztemp2(i)) + j*imag(Ztemp2(i)));
end

for i = 1:length(Ptemp2)
   Ptemp3(2*i-1) = Ptemp2(i);
   Ptemp3(2*i) = -real(Ptemp2(i)) + j*imag(Ptemp2(i));
   Ptemp3(length(Ptemp3)-2*i+2) = conj(Ptemp2(i));
   Ptemp3(length(Ptemp3)-2*i+1) = conj(-real(Ptemp2(i)) + j*imag(Ptemp2(i)));
end

Zdig_sort = Ztemp3;
Pdig_sort = Ptemp3;
```

This gives us:

| Zeros: | Poles: |
|---|---|
| 0.8135 + 0.5815i | 0.8059 + 0.5826i |
| -0.8135 + 0.5815i | -0.8059 + 0.5826i |
| 0.8203 + 0.5719i | 0.8036 + 0.5668i |
| -0.8203 + 0.5719i | -0.8036 + 0.5668i |
| 0.8337 + 0.5523i | 0.8072 + 0.5414i |
| -0.8337 + 0.5523i | -0.8072 + 0.5414i |
| 0.8532 + 0.5216i | 0.8161 + 0.5057i |
| -0.8532 + 0.5216i | -0.8161 + 0.5057i |
| 0.8780 + 0.4787i | 0.8292 + 0.4589i |
| -0.8780 + 0.4787i | -0.8292 + 0.4589i |
| 0.9065 + 0.4221i | 0.8452 + 0.4002i |
| -0.9065 + 0.4221i | -0.8452 + 0.4002i |
| 0.9364 + 0.3508i | 0.8621 + 0.3290i |
| -0.9364 + 0.3508i | -0.8621 + 0.3290i |
| 0.9644 + 0.2646i | 0.8778 + 0.2458i |
| -0.9644 + 0.2646i | -0.8778 + 0.2458i |
| 0.9863 + 0.1651i | 0.8900 + 0.1523i |
| -0.9863 + 0.1651i | -0.8900 + 0.1523i |
| 0.9984 + 0.0562i | 0.8966 + 0.0516i |
| -0.9984 + 0.0562i | -0.8966 + 0.0516i |
| -0.9984 - 0.0562i | -0.8966 - 0.0516i |
| 0.9984 - 0.0562i | 0.8966 - 0.0516i |
| -0.9863 - 0.1651i | -0.8900 - 0.1523i |
| 0.9863 - 0.1651i | 0.8900 - 0.1523i |
| -0.9644 - 0.2646i | -0.8778 - 0.2458i |
| 0.9644 - 0.2646i | 0.8778 - 0.2458i |
| -0.9364 - 0.3508i | -0.8621 - 0.3290i |
| 0.9364 - 0.3508i | 0.8621 - 0.3290i |
| -0.9065 - 0.4221i | -0.8452 - 0.4002i |
| 0.9065 - 0.4221i | 0.8452 - 0.4002i |
| -0.8780 - 0.4787i | -0.8292 - 0.4589i |
| 0.8780 - 0.4787i | 0.8292 - 0.4589i |
| -0.8532 - 0.5216i | -0.8161 - 0.5057i |
| 0.8532 - 0.5216i | 0.8161 - 0.5057i |
| -0.8337 - 0.5523i | -0.8072 - 0.5414i |
| 0.8337 - 0.5523i | 0.8072 - 0.5414i |
| -0.8203 - 0.5719i | -0.8036 - 0.5668i |
| 0.8203 - 0.5719i | 0.8036 - 0.5668i |
| -0.8135 - 0.5815i | -0.8059 - 0.5826i |
| 0.8135 - 0.5815i | 0.8059 - 0.5826i |

If we look through this list we can see that some of the pairs don't match up vertically (highlighted one in red). But, the first top half are all unique numbers, with their conjugates placed *somewhere* on the bottom half.

Also, by looking through this list, we see that the poles line up with the zeros closest to them, but the poles that are closest to the unit circle are not ordered first. We can utilize the flipud( ) command in MATLAB to fix this.

```
Zdig_sort = flipud(Zdig_sort);
Pdig_sort = flipud(Pdig_sort);
```

## Combine conjugate pairs to create second order systems

Now, using MATLAB, we can combine these conjugate pairs to create second order coefficients.

```
% create 2nd order real systems to cascade
I = length(Zdig_sort);
for i = 1:I/2
    Zdig2(i,:) = poly([Zdig_sort(i,1),conj(Zdig_sort(i,1))]);
end

I = length(Pdig_sort);
for i = 1:I/2
    Pdig2(i,:) = poly([Pdig_sort(i,1),conj(Pdig_sort(i,1))]);
end
```

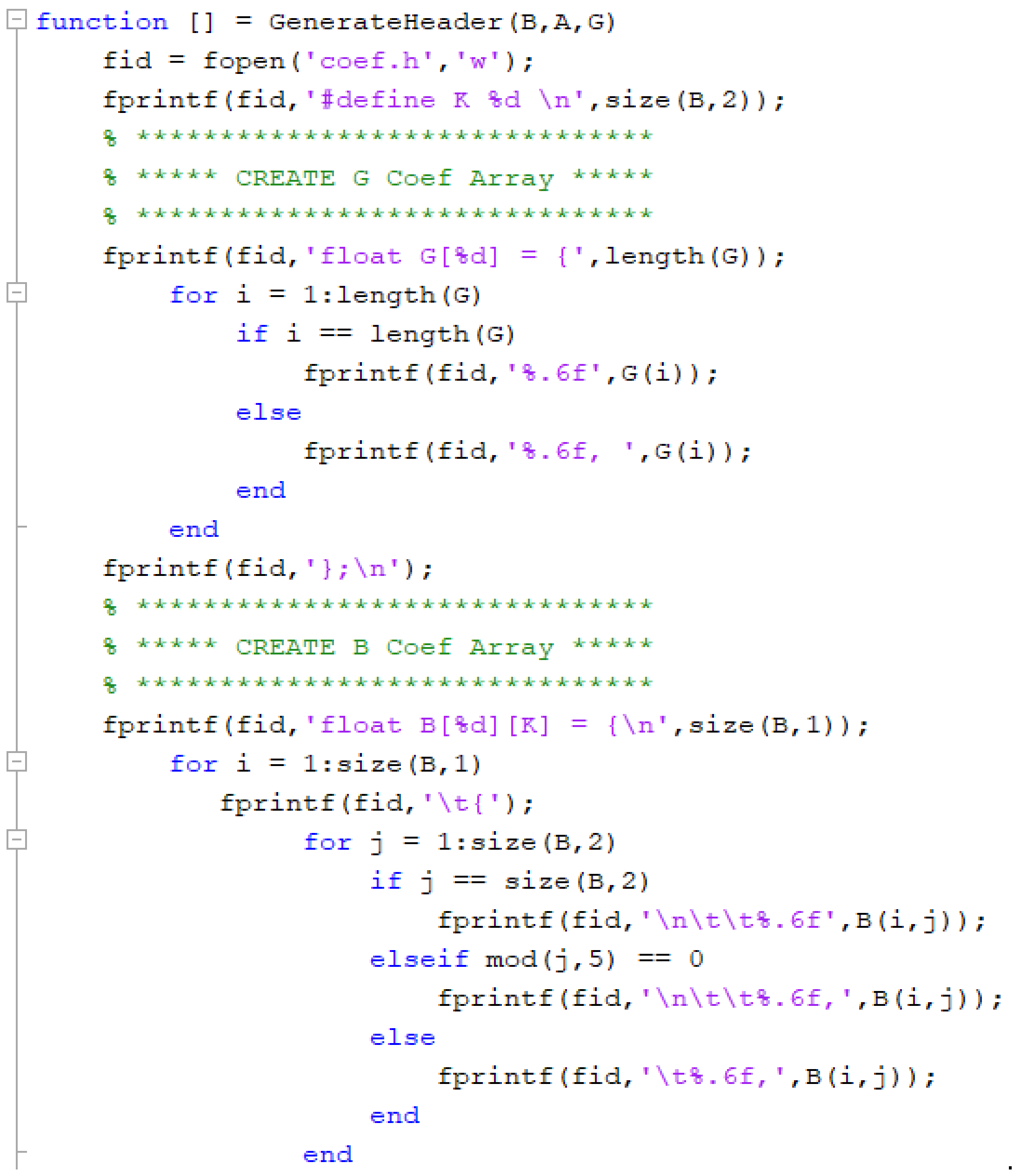
| Zeros | | | Poles | | |
|---|---|---|---|---|---|
| $z^2$ | z | 1 | $z^2$ | z | 1 |
| 1 | -1.6270 | 1 | 1 | -1.6120 | 0.9890 |
| 1 | 1.6270 | 1 | 1 | 1.6120 | 0.9890 |
| 1 | -1.6410 | 1 | 1 | -1.6070 | 0.9670 |
| 1 | 1.6410 | 1 | 1 | 1.6070 | 0.9670 |
| 1 | -1.6670 | 1 | 1 | -1.6140 | 0.9450 |
| 1 | 1.6670 | 1 | 1 | 1.6140 | 0.9450 |
| 1 | -1.7060 | 1 | 1 | -1.6320 | 0.9220 |
| 1 | 1.7060 | 1 | 1 | 1.6320 | 0.9220 |
| 1 | -1.7560 | 1 | 1 | -1.6580 | 0.8980 |
| 1 | 1.7560 | 1 | 1 | 1.6580 | 0.8980 |
| 1 | -1.8130 | 1 | 1 | -1.6900 | 0.8740 |
| 1 | 1.8130 | 1 | 1 | 1.6900 | 0.8740 |
| 1 | -1.8730 | 1 | 1 | -1.7240 | 0.8510 |
| 1 | 1.8730 | 1 | 1 | 1.7240 | 0.8510 |
| 1 | -1.9290 | 1 | 1 | -1.7560 | 0.8310 |
| 1 | 1.9290 | 1 | 1 | 1.7560 | 0.8310 |
| 1 | -1.9730 | 1 | 1 | -1.7800 | 0.8150 |
| 1 | 1.9730 | 1 | 1 | 1.7800 | 0.8150 |
| 1 | -1.9970 | 1 | 1 | -1.7930 | 0.8070 |
| 1 | 1.9970 | 1 | 1 | 1.7930 | 0.8070 |

Nice, our $b_0, b_2, and\ a_0$ terms are all 1. This makes hardware a little easier/faster. We can also see that the z terms for the poles and zeros flip from positive to negative. This could maybe help optimize code, but it would take a little while to implement so I'm not going to include that.

## Output coefficients to header file

Now that we have our coefficients, we need to generate a header file for our hardware implementation so that we don't have to copy/paste values every time we want to change our filter order, or starting passband frequency.

This is pretty simple, just a little time consuming to get right. I'm not going to paste full MATLAB code, but here is a snippet:



```matlab
function [] = GenerateHeader(B,A,G)
    fid = fopen('coef.h','w');
    fprintf(fid,'#define K %d \n',size(B,2));
    % ******************************
    % ***** CREATE G Coef Array *****
    % ******************************
    fprintf(fid,'float G[%d] = {',length(G));
        for i = 1:length(G)
            if i == length(G)
                fprintf(fid,'%.6f',G(i));
            else
                fprintf(fid,'%.6f, ',G(i));
            end
        end
    fprintf(fid,'};\n');
    % ******************************
    % ***** CREATE B Coef Array *****
    % ******************************
    fprintf(fid,'float B[%d][K] = {\n',size(B,1));
        for i = 1:size(B,1)
           fprintf(fid,'\t{');
                for j = 1:size(B,2)
                    if j == size(B,2)
                        fprintf(fid,'\n\t\t%.6f',B(i,j));
                    elseif mod(j,5) == 0
                        fprintf(fid,'\n\t\t%.6f,',B(i,j));
                    else
                        fprintf(fid,'\t%.6f,',B(i,j));
                    end
                end
```

This is simply a function with our B, A, and G coefficients calculated in the steps above.

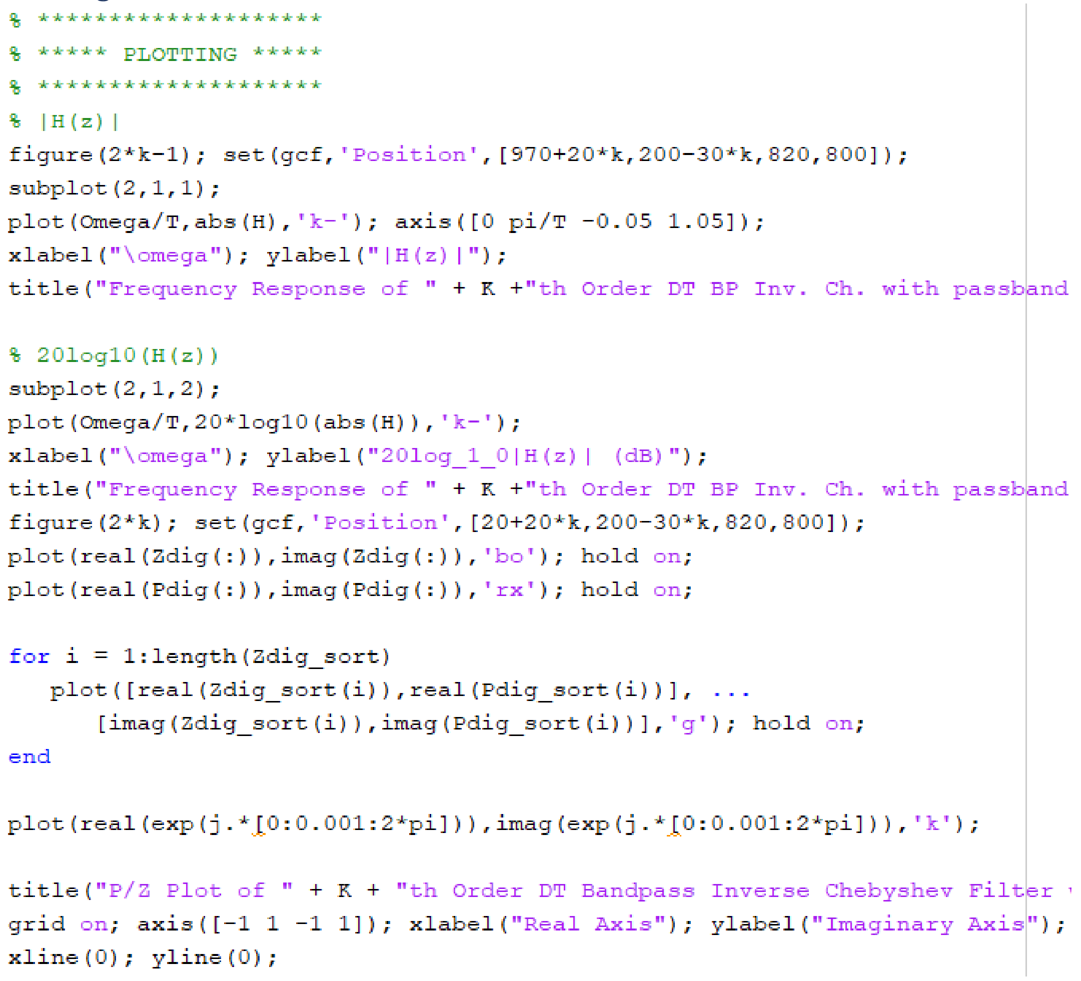
## Plots

Before we get to implementation, below is how I computed H(z) and plotted the Pole/Zero plots and Magnitude Response plots. I included the P/Z plots and Magnitude response for 250Hz, 500Hz, and 2000Hz that I did not include above, along with the Magnitude Response for the 1000Hz filter. These were all with K = 20

### Compute H(z)

```
Omega = linspace(0,pi,10001); H = 1;
% evaluate all 2nd order Zero polynomials over 0->Pi
for i = 1:length(Zdig2)
    H = H .* polyval(Zdig2(i,:),exp(1j*Omega));
end
% evaluate all 2nd order Pole polynomials over 0->Pi
for i = 1:length(Pdig2)
    H = H ./ polyval(Pdig2(i,:),exp(1j*Omega));
end
% multiply by gain factor
G = B(1)/A(1)*prod(1/c2-zk)/prod(1/c2-pk);
Gain = [Gain;G];
H = H.*G;
```

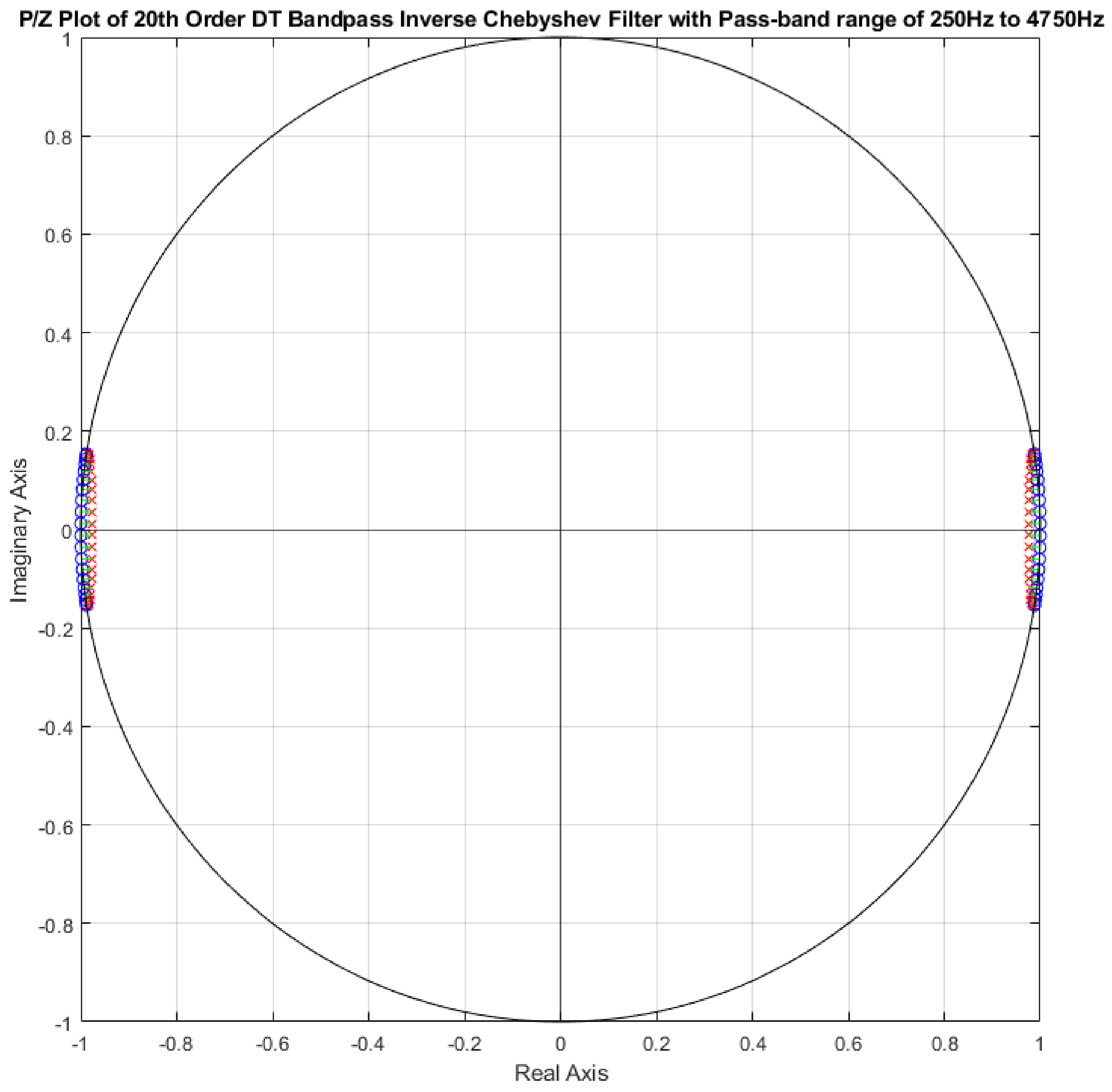
### Plotting

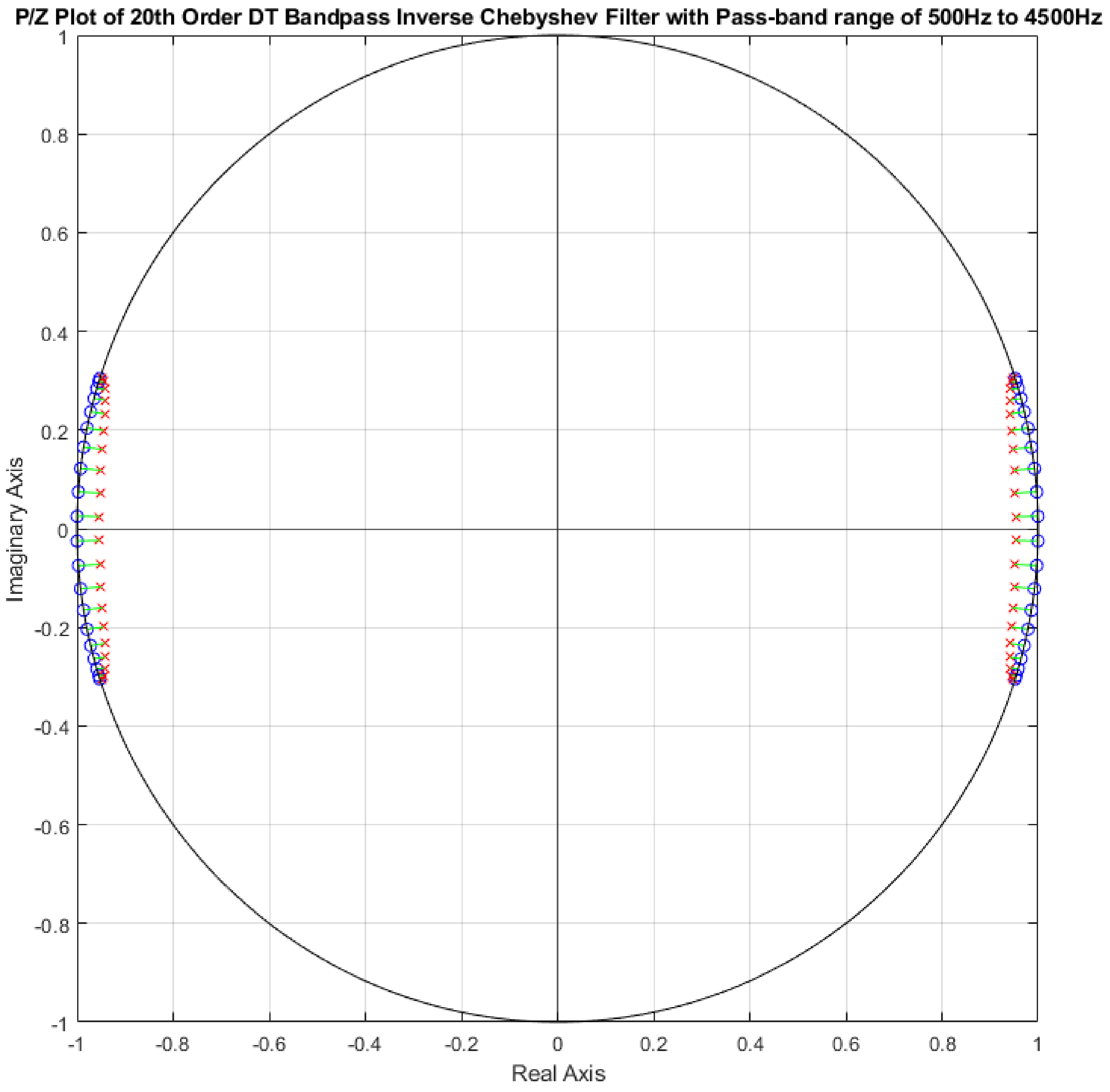


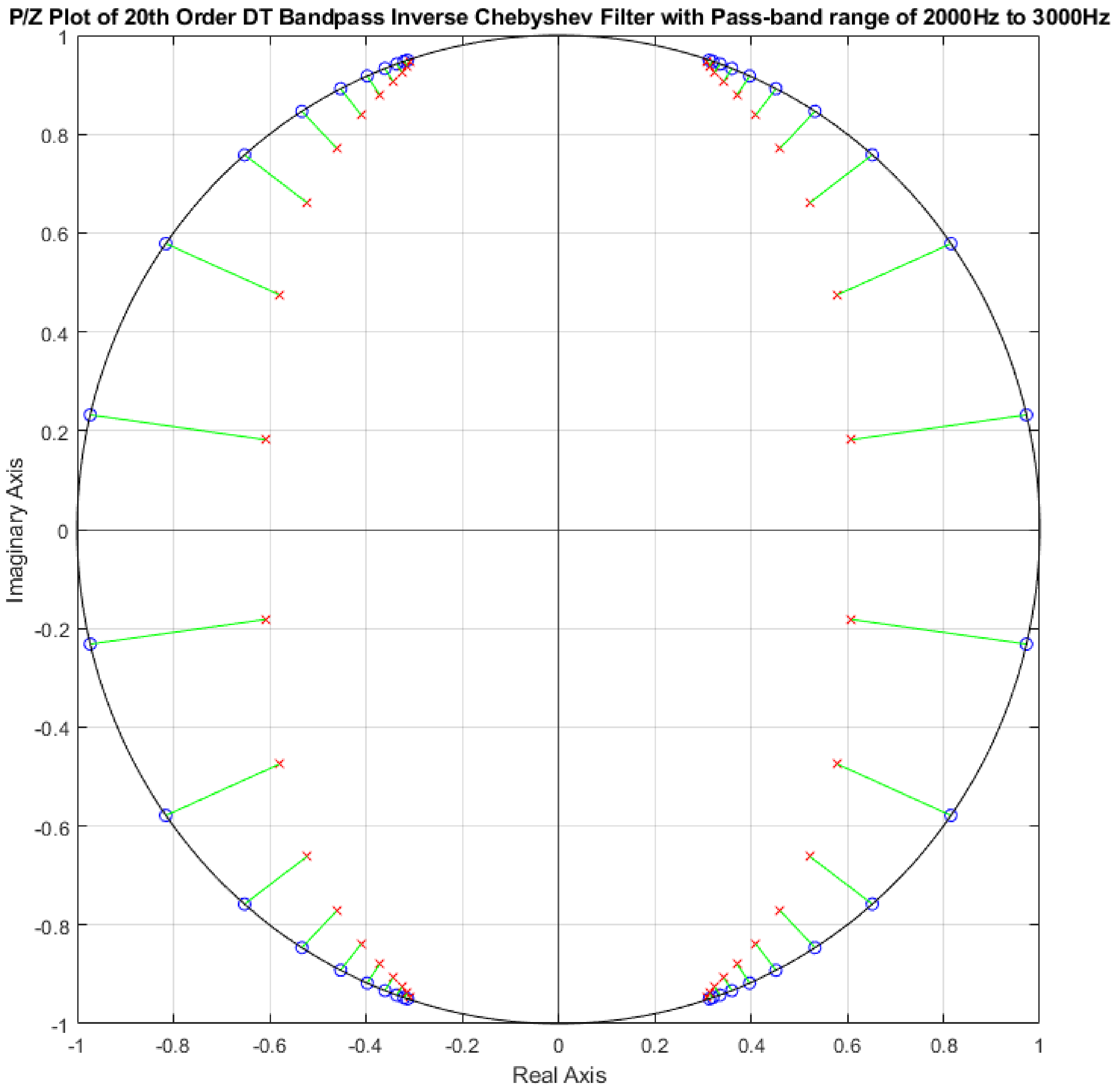
```
% ********************
% ***** PLOTTING *****
% ********************
% |H(z)|
figure(2*k-1); set(gcf,'Position',[970+20*k,200-30*k,820,800]);
subplot(2,1,1);
plot(Omega/T,abs(H),'k-'); axis([0 pi/T -0.05 1.05]);
xlabel("\omega"); ylabel("|H(z)|");
title("Frequency Response of " + K +"th Order DT BP Inv. Ch. with passband

% 20log10(H(z))
subplot(2,1,2);
plot(Omega/T,20*log10(abs(H)),'k-');
xlabel("\omega"); ylabel("20log_1_0|H(z)| (dB)");
title("Frequency Response of " + K +"th Order DT BP Inv. Ch. with passband
figure(2*k); set(gcf,'Position',[20+20*k,200-30*k,820,800]);
plot(real(Zdig(:)),imag(Zdig(:)),'bo'); hold on;
plot(real(Pdig(:)),imag(Pdig(:)),'rx'); hold on;

for i = 1:length(Zdig_sort)
   plot([real(Zdig_sort(i)),real(Pdig_sort(i))], ...
       [imag(Zdig_sort(i)),imag(Pdig_sort(i))],'g'); hold on;
end

plot(real(exp(j.*[0:0.001:2*pi])),imag(exp(j.*[0:0.001:2*pi])),'k');

title("P/Z Plot of " + K + "th Order DT Bandpass Inverse Chebyshev Filter '
grid on; axis([-1 1 -1 1]); xlabel("Real Axis"); ylabel("Imaginary Axis");
xline(0); yline(0);
```

With $f_{lower\ passband} = [250\ \ 500\ \ 1000\ \ 2000]$ and $K = 20$

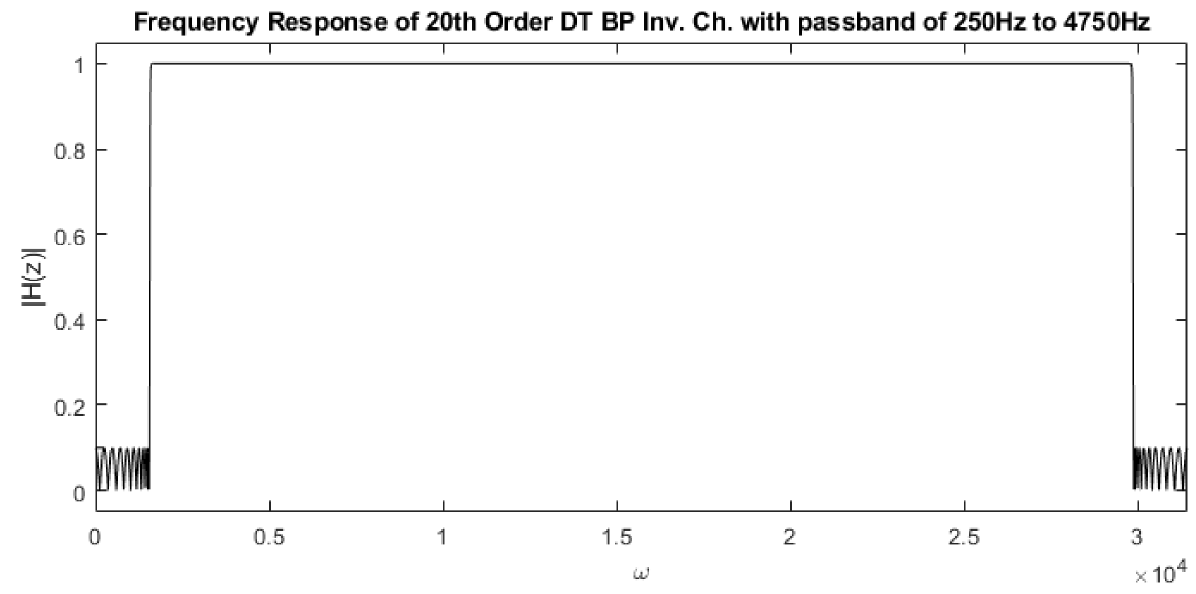
P/Z Plots:

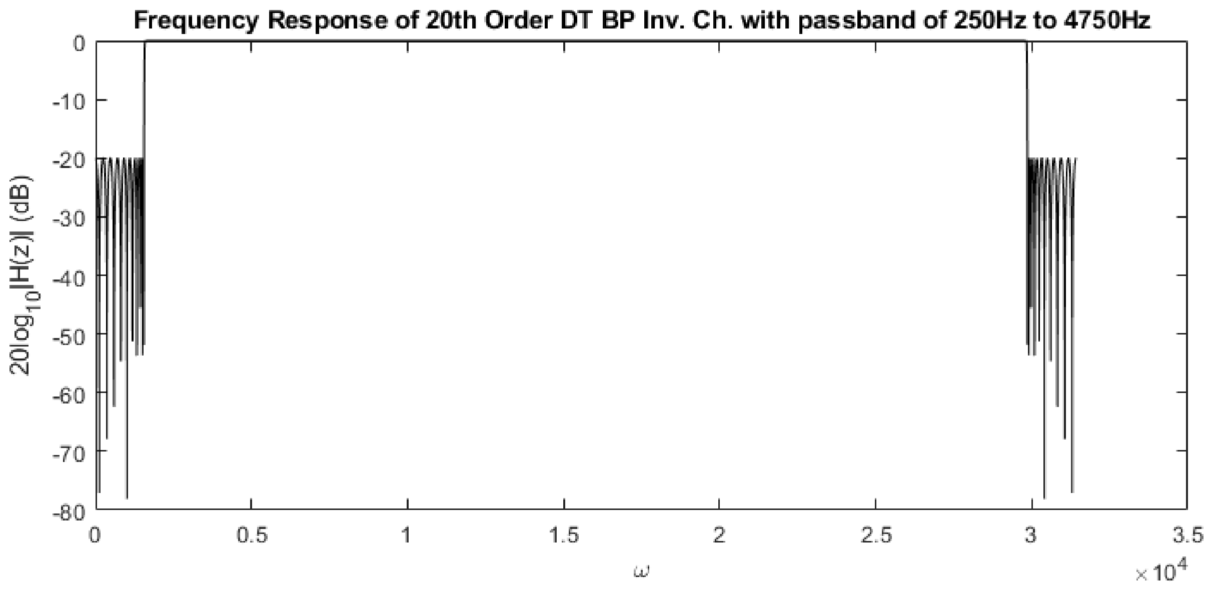


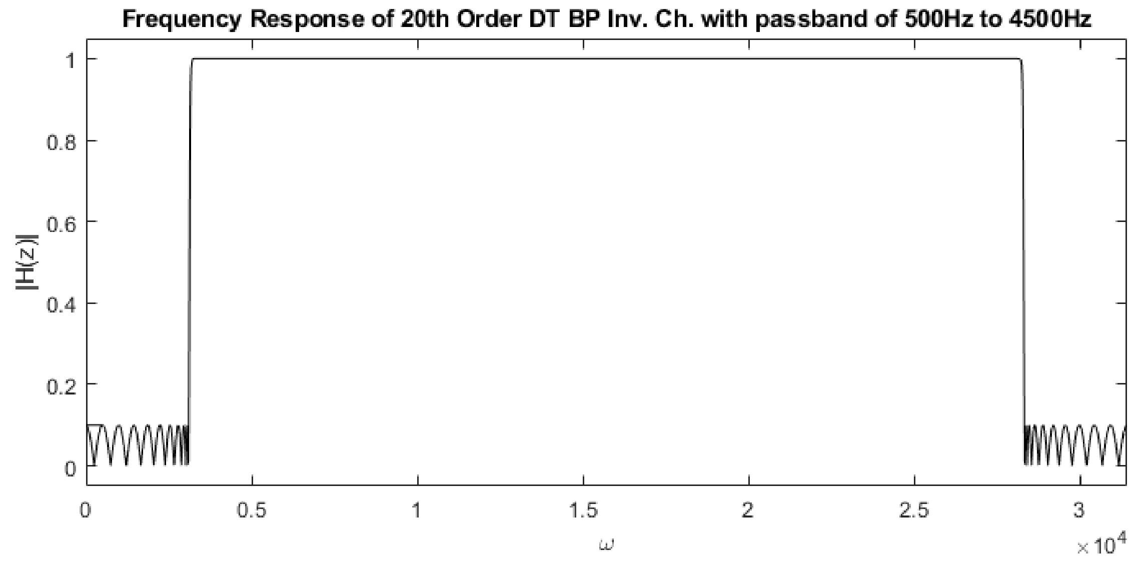

P/Z Plot of 20th Order DT Bandpass Inverse Chebyshev Filter with Pass-band range of 500Hz to 4500Hz
Imaginary Axis
Real Axis
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

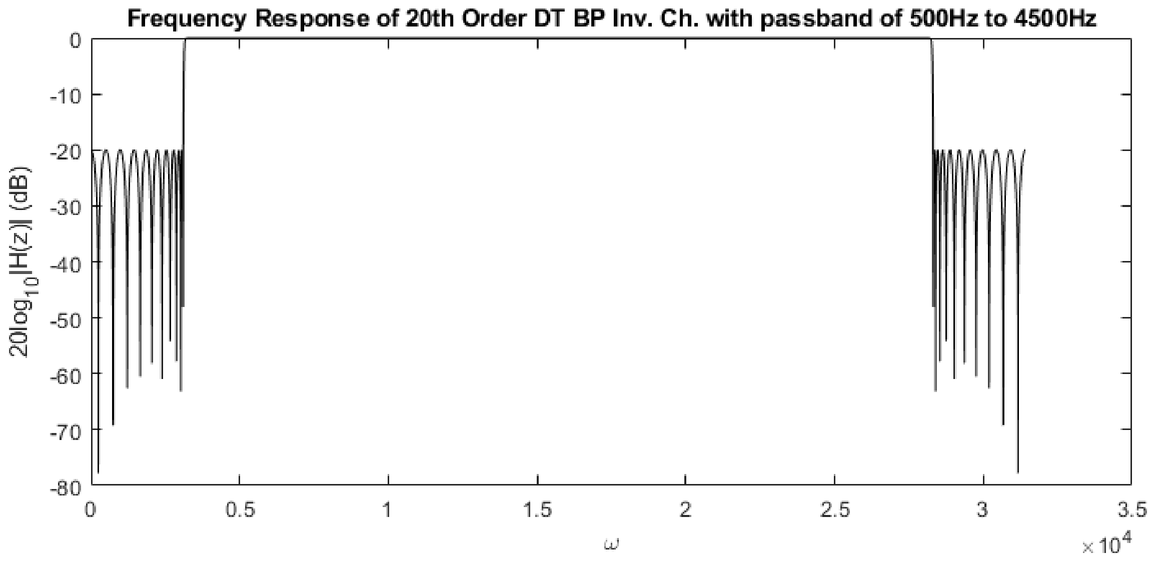

P/Z Plot of 20th Order DT Bandpass Inverse Chebyshev Filter with Pass-band range of 2000Hz to 3000Hz
Imaginary Axis
Real Axis
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

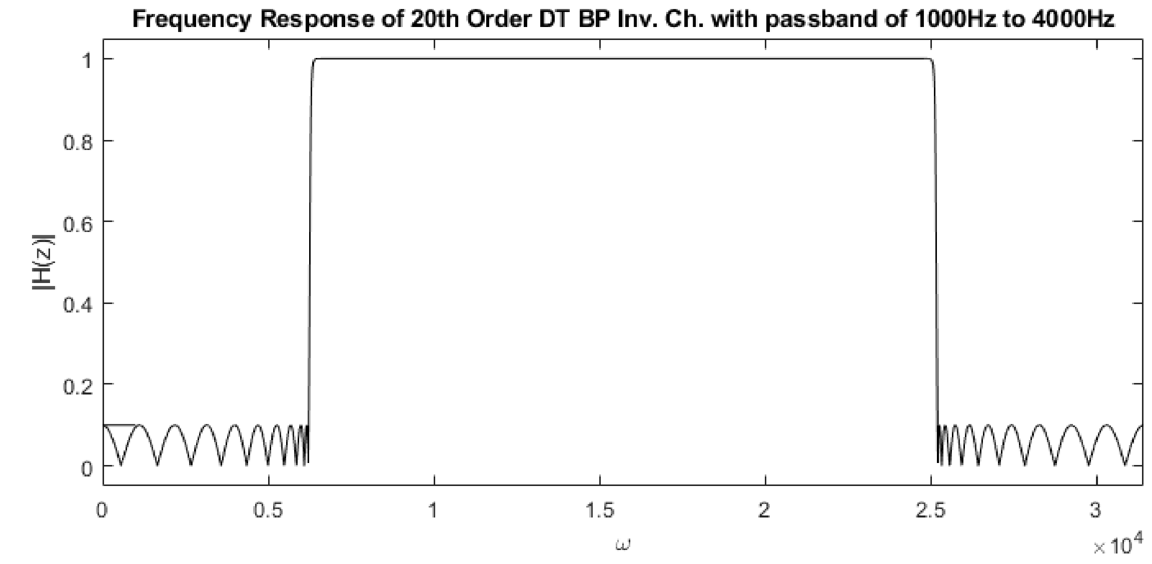
Magnitude Response Plots



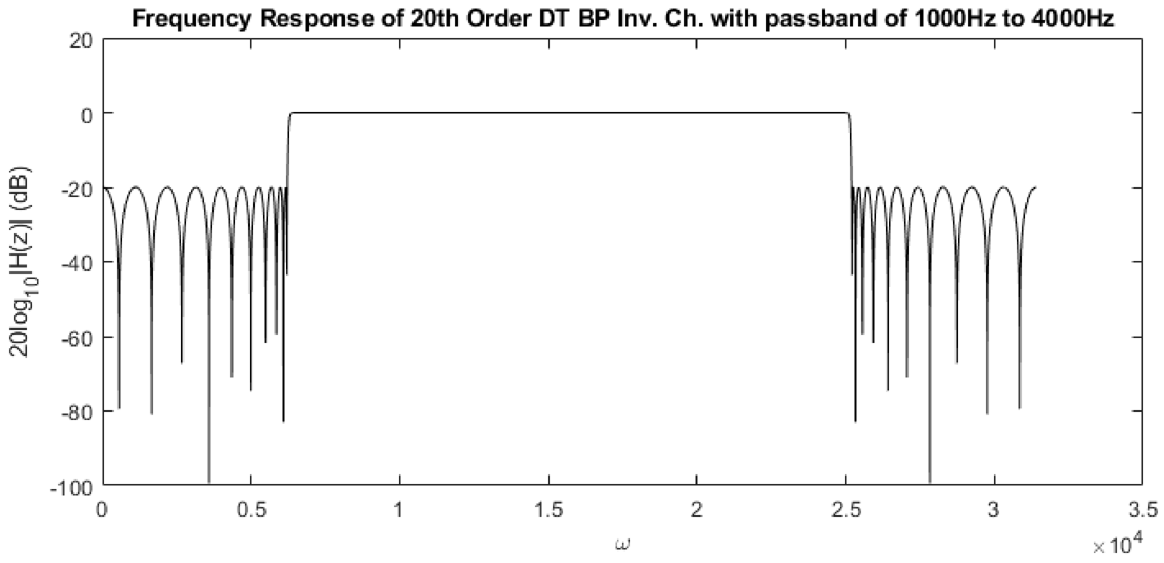



Frequency Response of 20th Order DT BP Inv. Ch. with passband of 500Hz to 4500Hz
|H(z)|
ω
×10^4

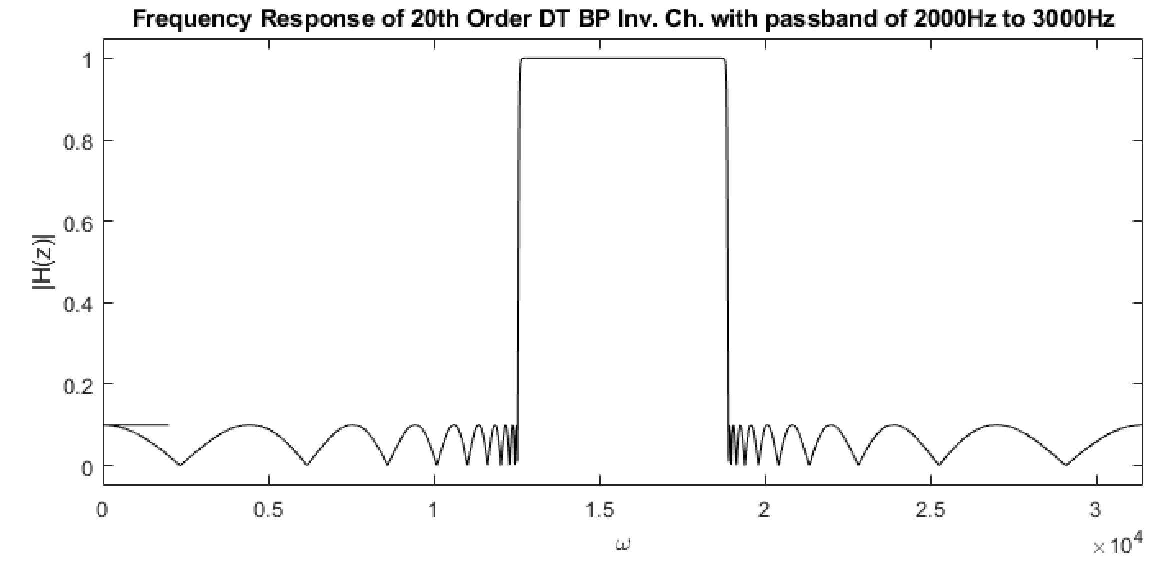

Frequency Response of 20th Order DT BP Inv. Ch. with passband of 500Hz to 4500Hz
20log10|H(z)| (dB)
ω
×10^4

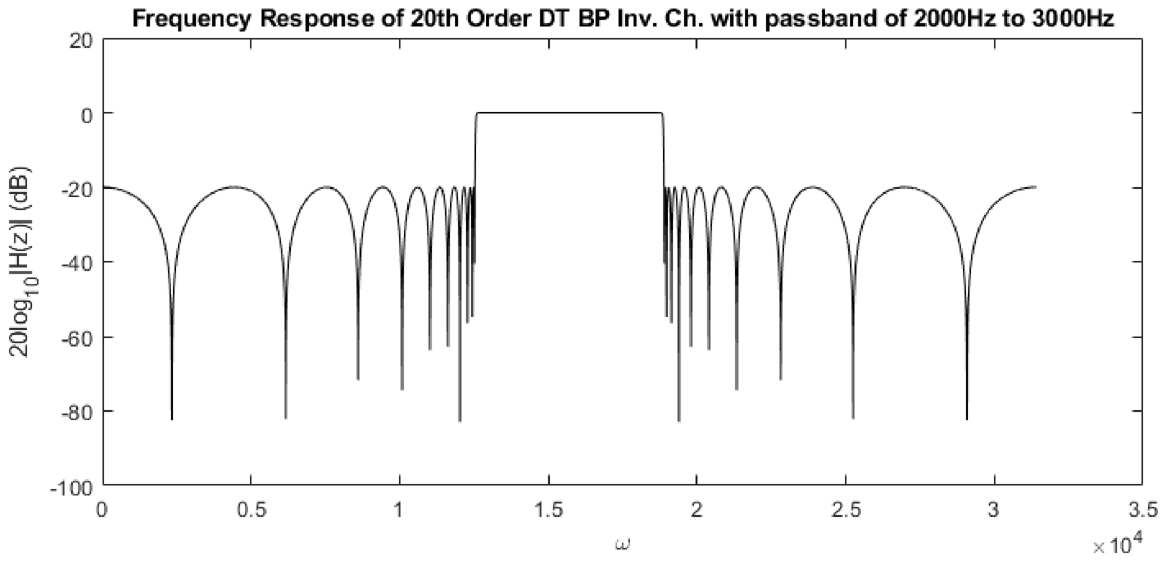


Frequency Response of 20th Order DT BP Inv. Ch. with passband of 1000Hz to 4000Hz
|H(z)|
0
0.2
0.4
0.6
0.8
1
0
0.5
1
1.5
2
2.5
3
ω
×10^4



Frequency Response of 20th Order DT BP Inv. Ch. with passband of 1000Hz to 4000Hz
20log10|H(z)| (dB)
20
0
-20
-40
-60
-80
-100
0
0.5
1
1.5
2
2.5
3
3.5
ω
×10^4


Frequency Response of 20th Order DT BP Inv. Ch. with passband of 2000Hz to 3000Hz
|H(z)|
0
0.2
0.4
0.6
0.8
1
0
0.5
1
1.5
2
2.5
3
ω
×10^4


Frequency Response of 20th Order DT BP Inv. Ch. with passband of 2000Hz to 3000Hz
20log10|H(z)| (dB)
20
0
-20
-40
-60
-80
-100
0
0.5
1
1.5
2
2.5
3
3.5
ω
×10^4

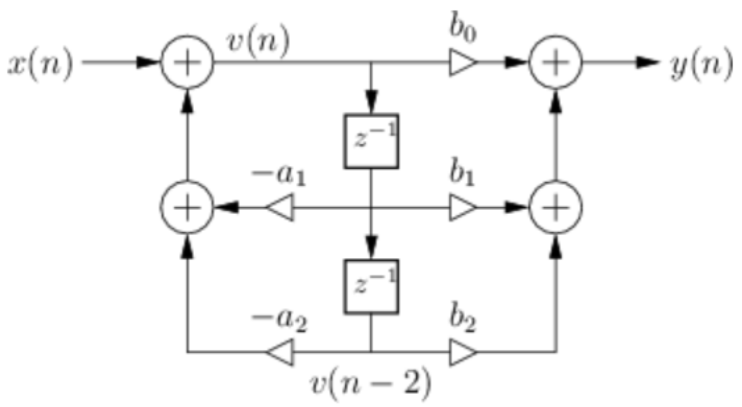
## Filter Coefficients

Below is the output of my GenerateHeader.m function in the coef.h header file:

```
#define K 20
float G[1] = {0.304332};
float B[1][K] = {
   {  1.627070,  -1.627070,  1.640619,  -1.640619,
      1.667352,  -1.667352,  1.706380,  -1.706380,  1.755964,
      -1.755964,  1.813064,  -1.813064,  1.872881,  -1.872881,
      1.928712,  -1.928712,  1.972558,  -1.972558,  1.996843,
      -1.996843   }
};

float A[1][K][2] = {
   {
      {1.611899,  0.988960},
      {-1.611899, 0.988960},
      {1.607148,  0.966968},
      {-1.607148, 0.966968},
      {1.614450,  0.944690},
      {-1.614450, 0.944690},
      {1.632260,  0.921800},
      {-1.632260, 0.921800},
      {-1.658497, 0.898257},
      {1.658497,  0.898257},
      {1.690352,  0.874464},
      {-1.690352, 0.874464},
      {-1.724173, 0.851446},
      {1.724173,  0.851446},
      {-1.755577, 0.830938},
      {1.755577,  0.830938},
      {-1.779945, 0.815234},
      {1.779945,  0.815234},
      {-1.793296, 0.806640},
      {1.793296,  0.806640}
   }
};

float buf[K][2] = {
{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}
};
```

## Implementation

Now is the fun part. Actually implementing our design onto a piece of hardware. Let's start with some pseudo-type-code in MATLAB. We know that we are implementing a DFII realization of our system in cascaded 2nd order stages. Below is a DFII realization structure.



From this we know we will need to store the former $v(n)$ values. Other things to consider are:

- Run-time filter selection via button
- Show interrupt timing from LED

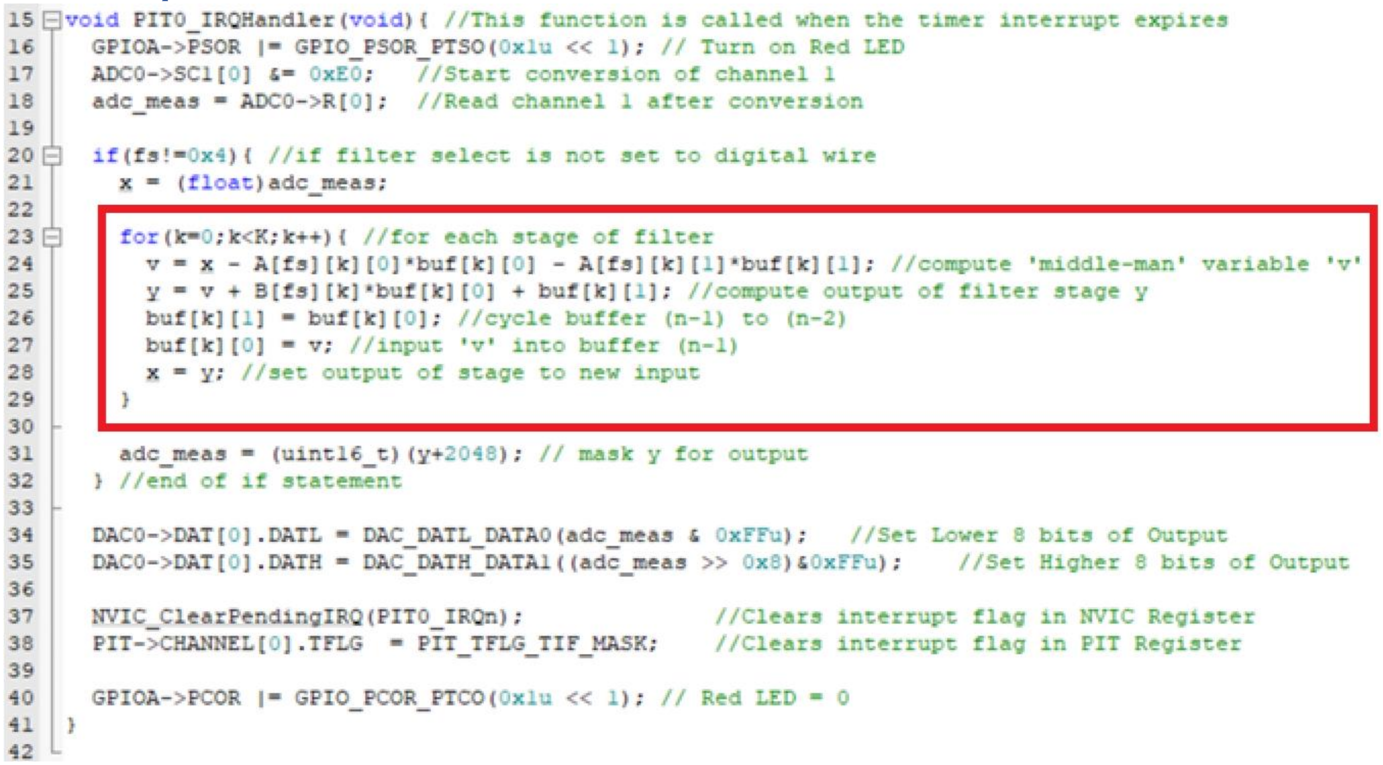
### Initial C-code

### Initially in MATLAB:

```
x = X(n*T); % adc input
x = real(Gain(filter_select))*x;
for k = 1:K % for each stage
    v = x - A_coef(filter_select,k,1).*buf(k,1) - A_coef(filter_select,k,2).*buf(k,2);
    y = v + B_coef(filter_select,k,1).*buf(k,1) + buf(k,2);
    buf(k,2) = buf(k,1);
    buf(k,1) = v;
    x = y;
end

Y = [Y;y];
```

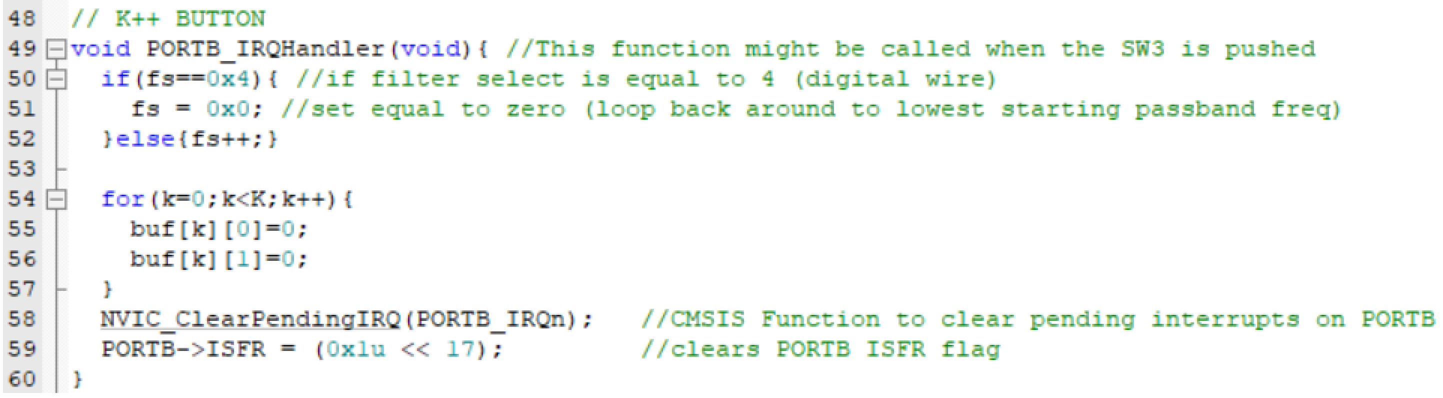
In Keil uVision IDE:

*Main Interrupt*



```c
void PIT0_IRQHandler(void){ //This function is called when the timer interrupt expires
  GPIOA->PSOR |= GPIO_PSOR_PTSO(0x1u << 1); // Turn on Red LED
  ADC0->SC1[0] &= 0xE0;   //Start conversion of channel 1
  adc_meas = ADC0->R[0];  //Read channel 1 after conversion

  if(fs!=0x4){ //if filter select is not set to digital wire
    x = (float)adc_meas;

    for(k=0;k<K;k++){ //for each stage of filter
      v = x - A[fs][k][0]*buf[k][0] - A[fs][k][1]*buf[k][1]; //compute 'middle-man' variable 'v'
      y = v + B[fs][k]*buf[k][0] + buf[k][1]; //compute output of filter stage y
      buf[k][1] = buf[k][0]; //cycle buffer (n-1) to (n-2)
      buf[k][0] = v; //input 'v' into buffer (n-1)
      x = y; //set output of stage to new input
    }

    adc_meas = (uint16_t)(y+2048); // mask y for output
  } //end of if statement

  DAC0->DAT[0].DATL = DAC_DATL_DATA0(adc_meas & 0xFFu);   //Set Lower 8 bits of Output
  DAC0->DAT[0].DATH = DAC_DATH_DATA1((adc_meas >> 0x8)&0xFFu);    //Set Higher 8 bits of Output

  NVIC_ClearPendingIRQ(PIT0_IRQn);            //Clears interrupt flag in NVIC Register
  PIT->CHANNEL[0].TFLG  = PIT_TFLG_TIF_MASK;  //Clears interrupt flag in PIT Register

  GPIOA->PCOR |= GPIO_PCOR_PTCO(0x1u << 1); // Red LED = 0
}
```

*Button Interrupt*

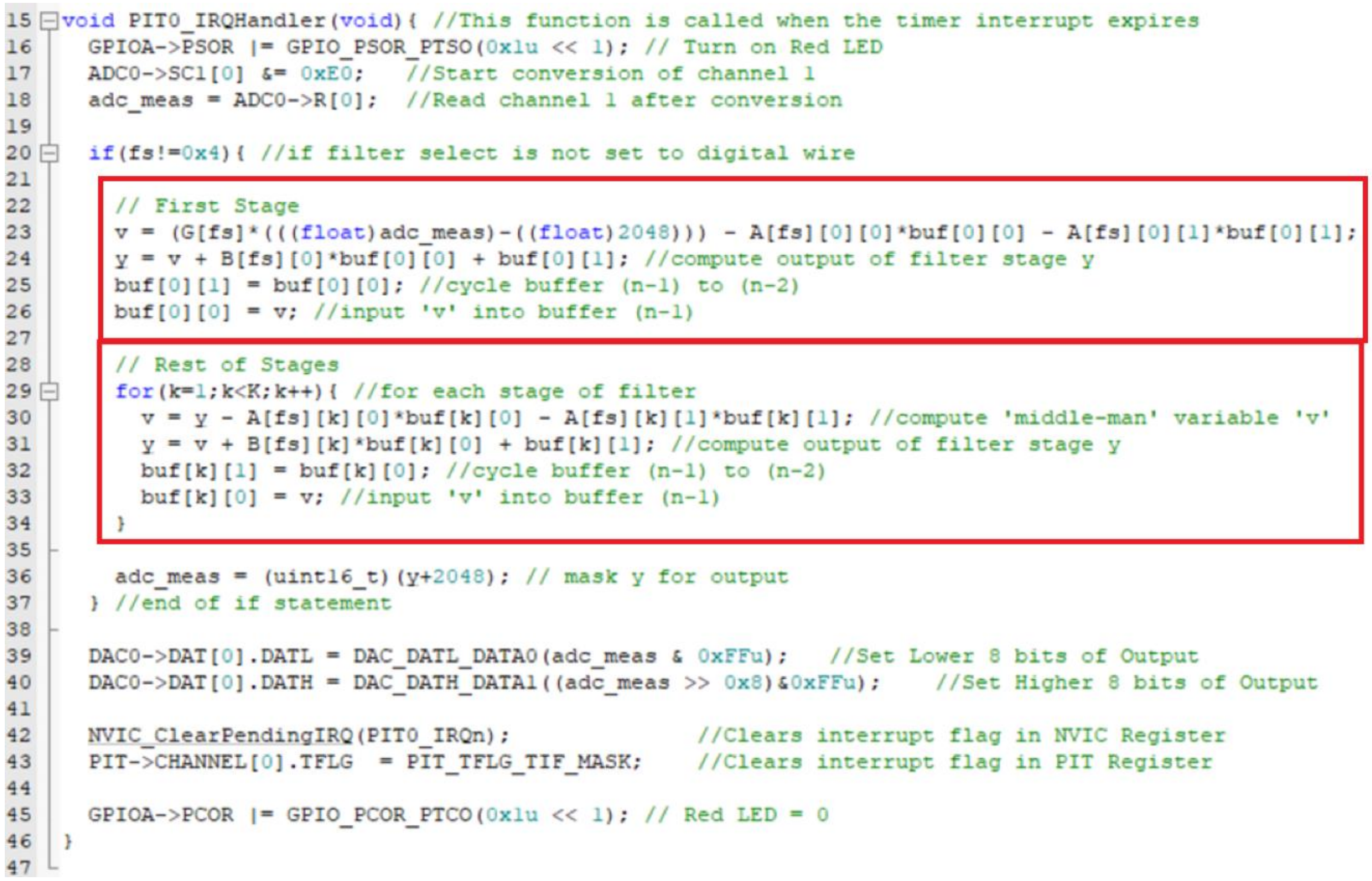


```c
// K++ BUTTON
void PORTB_IRQHandler(void){ //This function might be called when the SW3 is pushed
  if(fs==0x4){ //if filter select is equal to 4 (digital wire)
    fs = 0x0; //set equal to zero (loop back around to lowest starting passband freq)
  }else{fs++;}

  for(k=0;k<K;k++){
    buf[k][0]=0;
    buf[k][1]=0;
  }
  NVIC_ClearPendingIRQ(PORTB_IRQn);   //CMSIS Function to clear pending interrupts on PORTB
  PORTB->ISFR = (0x1u << 17);         //clears PORTB ISFR flag
}
```

## Optimizations/Revisions made

### Pulling out first stage

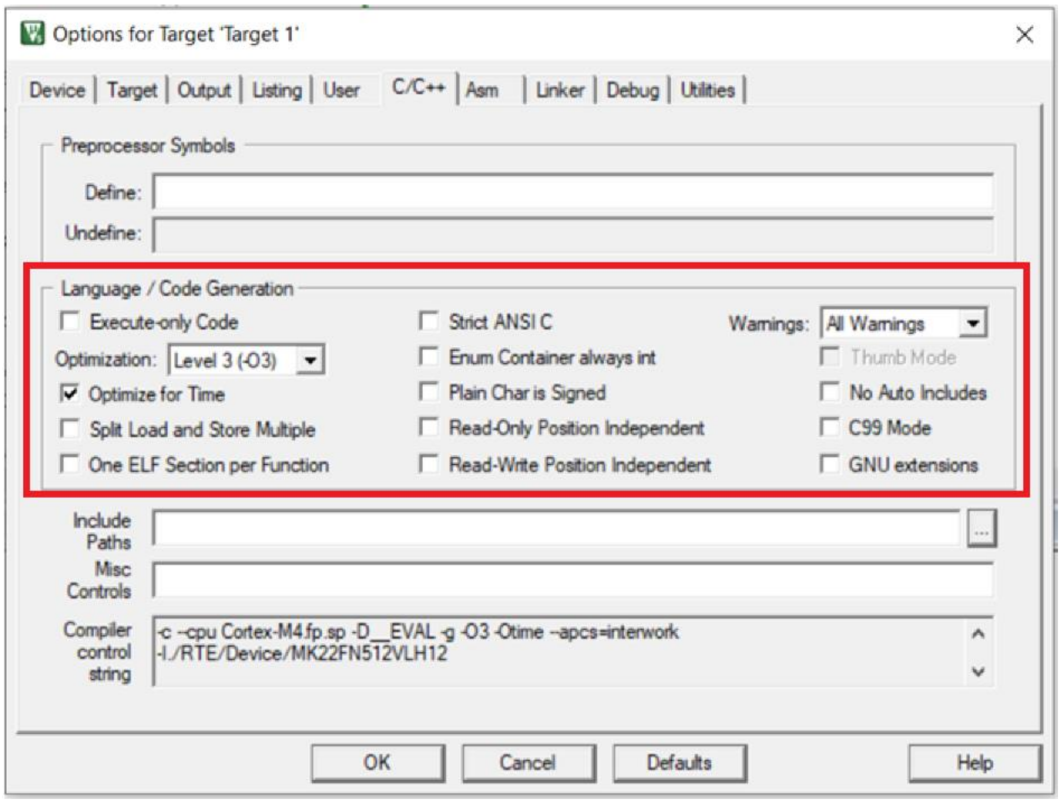
Most of my time in the main interrupt was from the large for-loop that calculates the next output value. I saw that if I took out the first stage of the for-loop, and nested together some of the calculations, I saved unnecessary assignment operations



```c
void PIT0_IRQHandler(void){ //This function is called when the timer interrupt expires
  GPIOA->PSOR |= GPIO_PSOR_PTSO(0x1u << 1); // Turn on Red LED
  ADC0->SC1[0] &= 0xE0;   //Start conversion of channel 1
  adc_meas = ADC0->R[0];  //Read channel 1 after conversion

  if(fs!=0x4){ //if filter select is not set to digital wire

    // First Stage
    v = (G[fs]*(((float)adc_meas)-((float)2048))) - A[fs][0][0]*buf[0][0] - A[fs][0][1]*buf[0][1];
    y = v + B[fs][0]*buf[0][0] + buf[0][1]; //compute output of filter stage y
    buf[0][1] = buf[0][0]; //cycle buffer (n-1) to (n-2)
    buf[0][0] = v; //input 'v' into buffer (n-1)

    // Rest of Stages
    for(k=1;k<K;k++){ //for each stage of filter
      v = y - A[fs][k][0]*buf[k][0] - A[fs][k][1]*buf[k][1]; //compute 'middle-man' variable 'v'
      y = v + B[fs][k]*buf[k][0] + buf[k][1]; //compute output of filter stage y
      buf[k][1] = buf[k][0]; //cycle buffer (n-1) to (n-2)
      buf[k][0] = v; //input 'v' into buffer (n-1)
    }

    adc_meas = (uint16_t)(y+2048); // mask y for output
  } //end of if statement

  DAC0->DAT[0].DATL = DAC_DATL_DATA0(adc_meas & 0xFFu);   //Set Lower 8 bits of Output
  DAC0->DAT[0].DATH = DAC_DATH_DATA1((adc_meas >> 0x8)&0xFFu);    //Set Higher 8 bits of Output

  NVIC_ClearPendingIRQ(PIT0_IRQn);              //Clears interrupt flag in NVIC Register
  PIT->CHANNEL[0].TFLG  = PIT_TFLG_TIF_MASK;    //Clears interrupt flag in PIT Register

  GPIOA->PCOR |= GPIO_PCOR_PTCO(0x1u << 1); // Red LED = 0
}
```

## Debug Optimizations

The second optimization I made was to increase the Keil uVision IDE's Code Generation Optimization level



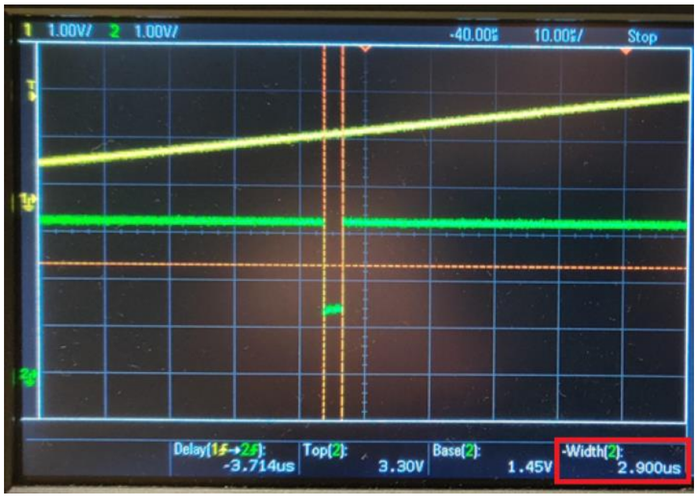
Options for Target 'Target 1'
Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities
Preprocessor Symbols
Define:
Undefine:
Language / Code Generation
Include Paths
Misc Controls
OK
Cancel
Defaults
Help

Execute-only Code [ ]
Optimization: Level 3 (-O3)
Optimize for Time [✓]
Split Load and Store Multiple [ ]
One ELF Section per Function [ ]
Strict ANSI C [ ]
Enum Container always int [ ]
Plain Char is Signed [ ]
Read-Only Position Independent [ ]
Read-Write Position Independent [ ]
Warnings: All Warnings
Thumb Mode [ ]
No Auto Includes [ ]
C99 Mode [ ]
GNU extensions [ ]

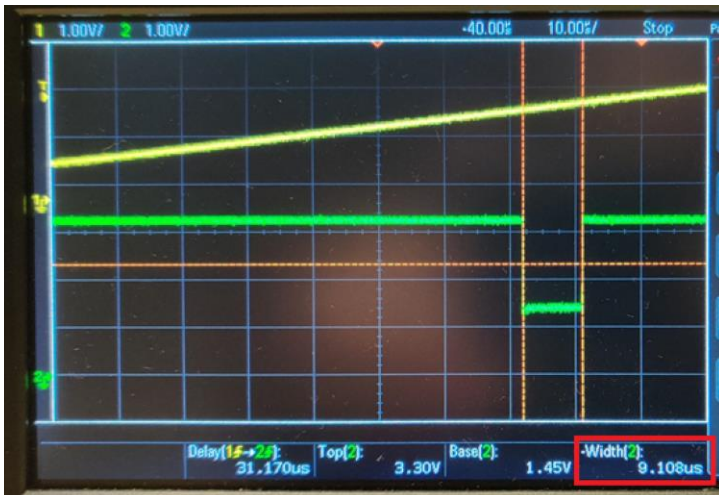
Compiler control string:
```
-c --cpu Cortex-M4.fp.sp -D__EVAL -g -O3 -Otime --apcs=interwork
-I ./RTE/Device/MK22FN512VLH12
```

## Testing Initial vs. Optimized C-code

By Probing the Red LED light that is programmed to turn on at the beginning of the ISR, and turn off at the end:



Before pulling out the first stage:

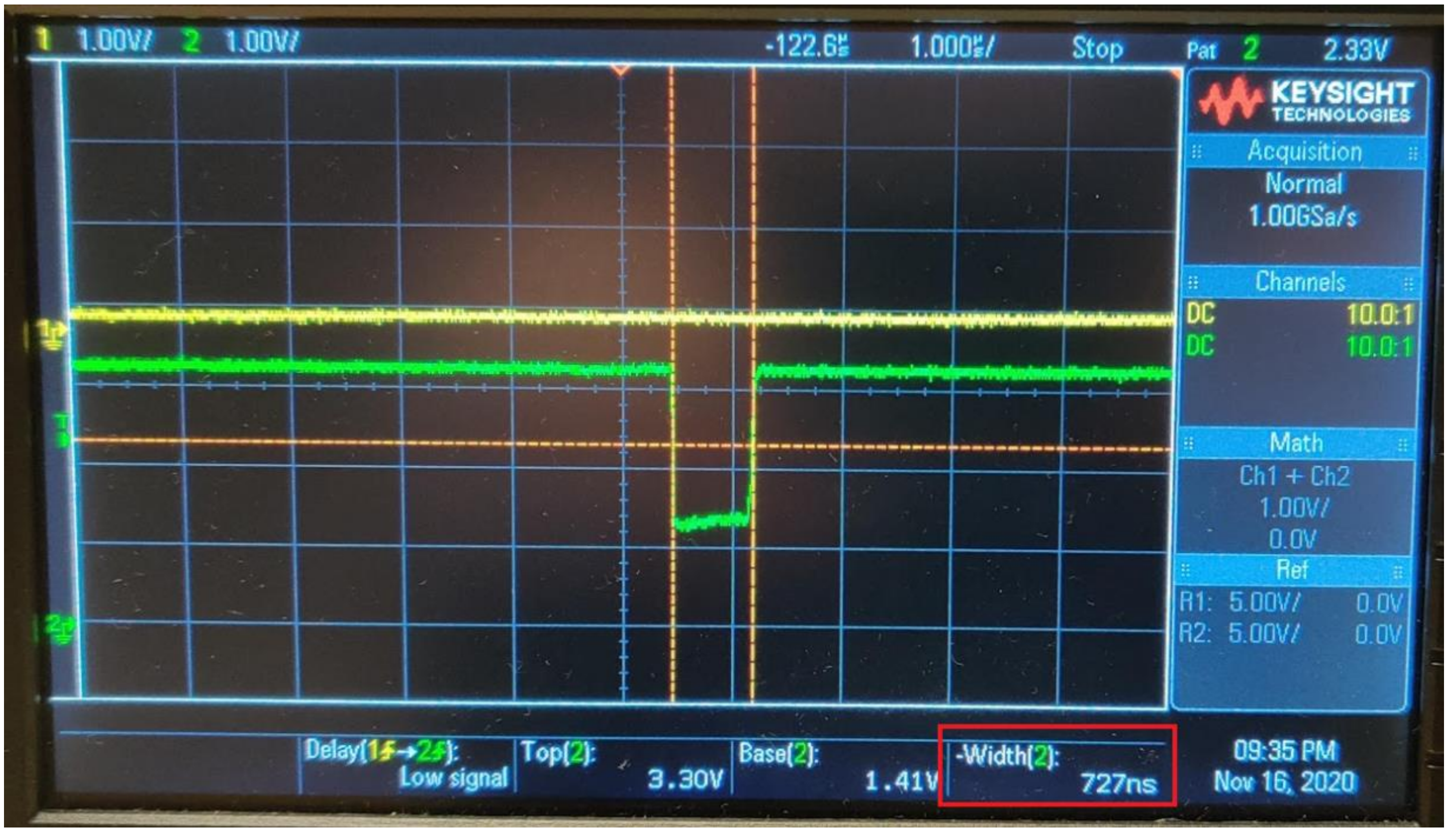


After pulling out the first stage:

This was with K = 256. This gave about 6us more time before I would miss timing

## Finding Highest Order Possible

At K = 256 I had 9.108us left. Next was K = 280.



There may be more debugger settings that could increase the amount of time in the interrupt before missing, but without that this is the highest order I can go.