Lab 03 - Time-Varying Resonant Filters

Ron Guglielmone

PROBLEM 1(a)

A resonant low-pass filter has the following transfer function:

$$H(s) = \frac{1}{(s/\omega_c)^2 + \frac{1}{Q}(s/\omega_c) + 1}.$$

The magnitude frequency response and phase response for various values of frequency and Q were plotted using the following MATLAB script (Figure 1).

```
% Ron Guglielmone
% MUSIC 424, CCRMA, Stanford University
% April 26, 2017
%
% HW 3 - Problem 1(a)

clear all;
close all;

% Constants:
wc = 2*pi*1000;          % Radians/second
Q = 2.^(-4:2);           % Unit-less
N = length(Q);           % Loop end
B = [0 0 1];             % Numerator Coeffs.

for i = 1 : N

    % Calculate denom. coeffs.
    A = [1 1/(wc*Q(i)) 1/(wc^2)];
    % Calculate freq. response
    [H,W] = freqs(B,A);
    % Add magnitude to plot:
    subplot(2,1,1);
    loglog(W, abs(H));
    hold all
    % Add phase to plot:
    subplot(2,1,2);
    semilogx(W, angle(H));
    hold all
end
```

Figure 1, MATLAB script for problem 1(a).

The following plots for phase and magnitude response were seen for values of Q ranging from Q = 2^{-2} up to Q = 2^{4} (Figure 2).

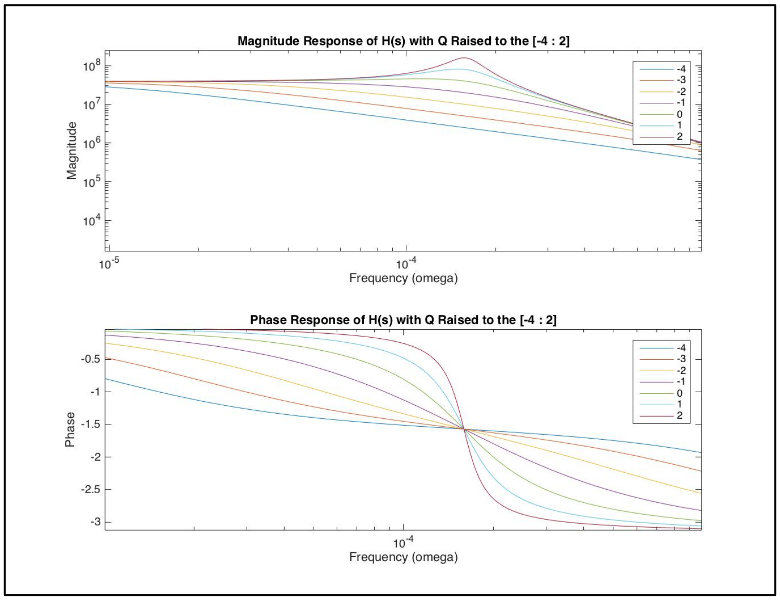

Figure 2, magnitude and phase for varying values of Q.

Next, frequency was varied from $2\pi 1000(2^{[-2]})$ up to $2\pi 1000(2^{[2]})$ (Figure 3).

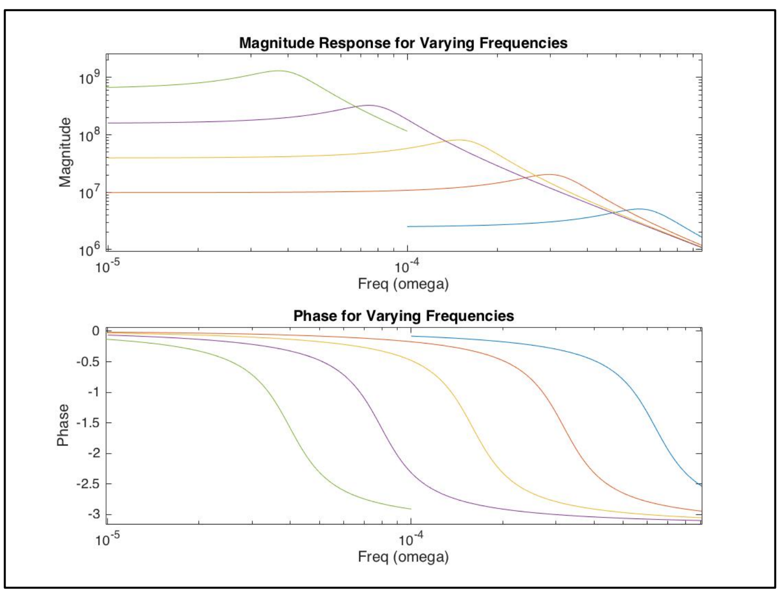

Figure 3, magnitude and phase for varying frequencies.

Finally, the poles and zeros for each case were plotted in Figures 4 and 5.

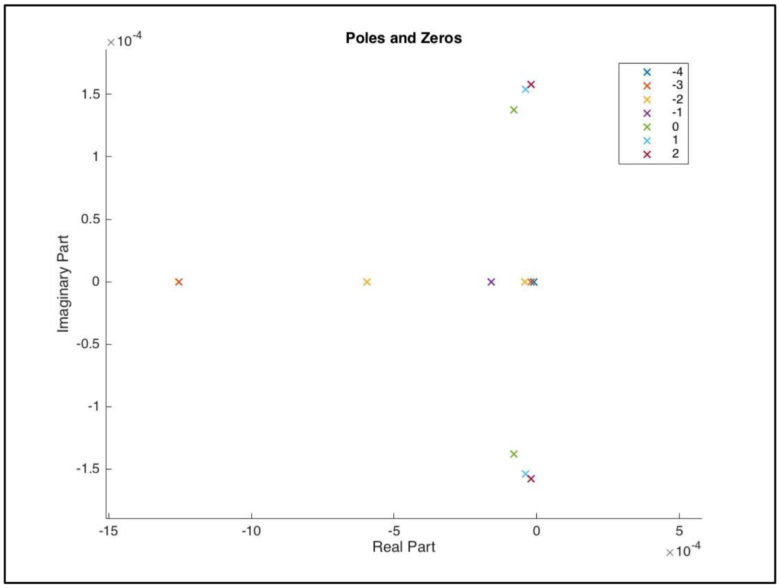

Figure 4, poles and zeros for varying Q.

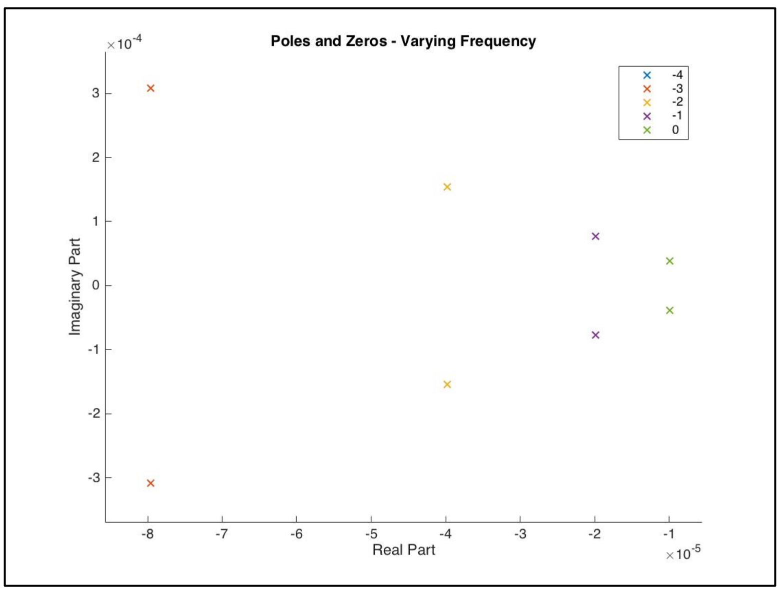

Figure 5, poles and zeros for varying frequency.

PROBLEM 1(b)

The bilinear transform has been used to map our analog filter into a digital representation (Figure 6).

$$H(s) = \frac{1}{(\frac{s}{w})^2 + (\frac{1}{Q})(\frac{s}{w}) + 1}$$

We let $s = (\frac{2}{T})(\frac{1-z^{-1}}{1+z^{-1}})$

H(s) becomes H(z)...

$$H(z) = \frac{1}{\left(\frac{(\frac{2}{T})(\frac{1-z^{-1}}{1+z^{-1}})}{w}\right)^2 + (\frac{1}{Q})\left(\frac{(\frac{2}{T})(\frac{1-z^{-1}}{1+z^{-1}})}{w}\right) + 1}$$

Figure 6, bilinear transform mapping from s to z.

Then, the bi-quad coefficients were implemented in C++ (Figure 7).

```
///////////////BEGIN////////////////
//TODO: design analog filter based on
//input gain, center frequency and Q
//////////////START/////////////////

b0 = 1.0;
b1 = 0.0;
b2 = 0.0;
a0 = 1.0;
a1 = 1.0 / (center * qval * 2 * pi);
a2 = 1.0 / (center * center * 2 * pi);

///////////////END////////////////////
// TODO: apply bilinear transform
//////////////START///////////////////

double T = 1/fs;

az0 = ( a0*T*T + 2*a1*T + 4*a2 );
az1 = ( 2*a0*T*T - 8*a2 ) / az0;
az2 = ( a0*T*T - 2*a1*T + 4*a2 ) / az0;

bz0 = ( b0*t*T + 2*b1*T + 4*b2 ) / az0;
bz1 = ( 2*b0*T*T - 8*b2 ) / az0;
bz2 = ( b0*T*T - 2*b1*T + 4*b2 ) / az0;

az0 = 1;

///////////////END////////////////////
```

Figure 7, bi-quad coefficients in C++.

PROBLEM 2(a)

Filter parameter controls have been fed into leaky integrators to help smooth changes between different states (Figure 8).

```
//////////////////////////////////////////
// Leaky integrator (same as Lab 01) /////
//////////////////////////////////////////

void setTau(float tau, float fs) {
    //////////////START/////////////////
    a1 = exp( -1.0 / ( tau * fs ) );
    b0 = 1 - a1;
    //////////////END/////////////////
}

void reset() {
    // reset filter state
    z1=0;
}
void process (float input, float& output) {
    //////////////START/////////////////
    z1 += b0 * (input - z1);
    output = z1;
    //////////////END/////////////////
}
```

Figure 8, updated "SlewedParameter" class.

PROBLEM 2(b)

An LFO was implemented using the sin( ) of a phase counter (Figure 9).

```
float WahWah::LFO(float f0)
{
    // TODO: Implement
    //////////////START/////////////////

    // Initialize:
    float phase = 0;
    // Increment by delta:
    float change = 2*pi*f0/fs;
    // Mod 2pi for phase:
    phase = fmodf(phase+change,2*pi);
    // Lookup next sample:
    float sample = sin(phase);
    return sample;

    ///////////////END////////////////////

}
```

Figure 9, LFO implementation in C++.

PROBLEM 2(c)

It seems like the frequency computer is already coded, and I'm not sure what to change. Still, some modest changes in the following sections were made (Figure 10).

```
void WahWah::processReplacing [… etc etc… ]

        // slew control signals
        float freqSlew, gainSlew, qSlew;
        fcSlewer.process(FcValue, freqSlew);
        gSlewer.process(GainValue, gainSlew);
        qSlewer.process(QValue, qSlew);

        // modulation signal
        float modulationSignal  = frequencyComputer(freqSlew, RateValue, DepthValue);
}

// Not sure what to change in here:
float WahWah::frequencyComputer(float fc, float rate, float depth)
{
    float floSignal = pow(depth, .5*LFO(rate));

    // Slew the modulation signal as well
    modSlewer.process(floSignal, floSignal);

    return fc*floSignal;

}
```

PROBLEM 2(d)

I did not have time to finish this, but the peak detector portion of the code is presented below (Figure 11).

```
PeakDetector() {

[… etc …]

    // Process one block (one sample):
    void process (float input, float threshold, float& output) {

        // Test (?) above or below threshold:
        if ( fabs( input ) > levelEstimate ) {

            // "Attack-state" update equation:
            levelEstimate += b0_a * ( fabs( input ) - levelEstimate );
        }

        else {

            // Release to threshold:
            levelEstimate += b0_r * ( threshold - levelEstimate );
        }

        // Update output:
        output = levelEstimate;
    }
};
```

Figure 11, peak detector scheme.