Ala'alddin Al-Migdad & Matt McDade

Dr. Jay Adams

Embedded Scientific Computing Project (Part 1 & 2)

30 November 2018

Part 1 - Discretization of a Continuous-Time Butterworth Filter

In part one of this project, we were tasked with discretizing and choosing a computational structure for a fourth-order Butterworth filter. We were given poles to discretize, Tustin's method was used replacing s with $\frac{2}{\Delta t}\frac{z-1}{z+1}$. This gave us a discrete time approximation to compare to the continuous time function. From there, we found an appropriate number of fraction bits to accurately define our discretized function with an error on the poles of less than 5% using the best suited computational structure.

Finding the S-Domain Transfer Function

To find the transfer function from the given poles and frequency, we first solved for ω_c using the frequency to evaluate the true pole locations. The given poles were as follows:

$$s \ \varepsilon \ \{\omega_c e^{(i\frac{5\pi}{8})}, \omega_c e^{(i\frac{7\pi}{8})}, \omega_c e^{(i\frac{9\pi}{8})}, \omega_c e^{(i\frac{11\pi}{8})} \}$$

We then converted these poles into their real and imaginary parts using euler's identity

$$s_1 = \omega(cos(\tfrac{5\pi}{8}) + i\ sin(\tfrac{5\pi}{8})) = \omega(-0.3827 + i\ 0.9239)$$

$$s_2 = \omega(cos(\tfrac{7\pi}{8}) + i\ sin(\tfrac{7\pi}{8})) = \omega(-0.9239 + i\ 0.3827)$$

$$s_3 = \omega(cos(\tfrac{9\pi}{8}) + i\ sin(\tfrac{9\pi}{8})) = \omega(-0.9239 + i\ 0.3827)$$

$$s_4 = \omega(cos(\tfrac{11\pi}{8}) + i\ sin(\tfrac{11\pi}{8})) = \omega(-0.3827 + i\ 0.9239)$$

Pairing up the poles that have the same real and imaginary values to create two second-order factors, and placing those in our resulting transfer function:

$$H_C(s) = \frac{M}{(s^2+0.7654s*\omega+1.0000505\omega^2)(s^2+1.8478s*\omega+1.0000505\omega^2)}$$

Calculating an appropriate multiple M to achieve a DC-Gain of 16 by setting s to 0,

$$16 = H(0) = \frac{M}{(1.0000505\omega^2)(1.0000505\omega^2)}$$

$$M = 16.001616040804\omega^4 = 1.1683 * 10^6$$

Determining an Appropriate Sample Time / Discretizing the continuous time filter.

Tustin's Method was used to convert the transfer function from continuous time to discrete time where $s = \frac{2}{\Delta t}\frac{z-1}{z+1}$, with $\Delta t = 0.0005$ seconds. This Δt was chosen for better calculations and simplicity. After substituting for s on the continuous time function, the now discrete time function after plugging in the values of M, ω and Δt is as follows:

$$H_d(z) = \frac{0.15955904(z+1)^4}{(z^2-1.12051z+0.575922)(z^2-0.862021z+0.21238)}$$

Generating and Comparing Frequency Responses of Both Filters

In Matlab, a graph of the frequency response was generated for each transfer function that was found and they were both compared. After analyzing both graphs it was found that the graphing seemed correct since both Figure 1 looked identical to Figure 2. The DC gain multiplier M was calculated in order to shift the graph to start at 16 in which translates in the graph to be approximately 24, since 20log(16) is about 24dB. Using M it is noticeable that Figures 1 and 2 both have a DC gain of approximately 24dB.

The graph after approximating our discrete-time coefficients with a [6, -5] fixed-point representation, which is described in the next step, can be seen in Figure 3. Our full Matlab implementation can be found in Appendix A.

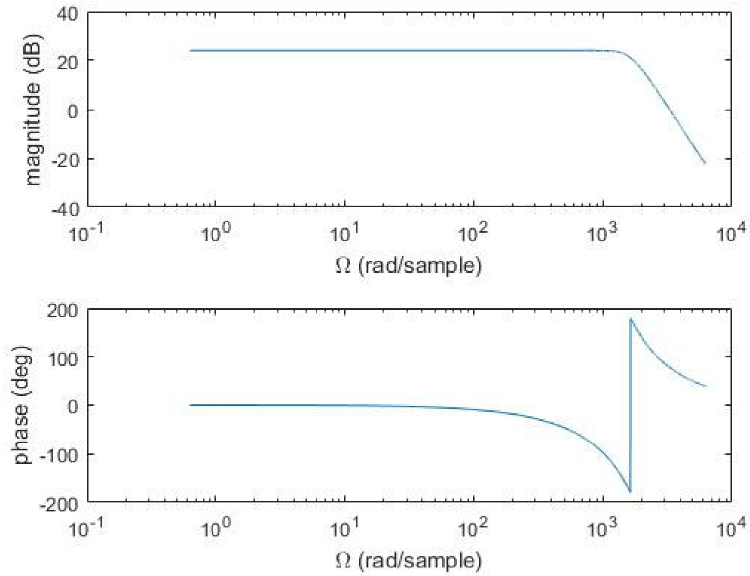

Figure 1: Continuous Time Frequency Response

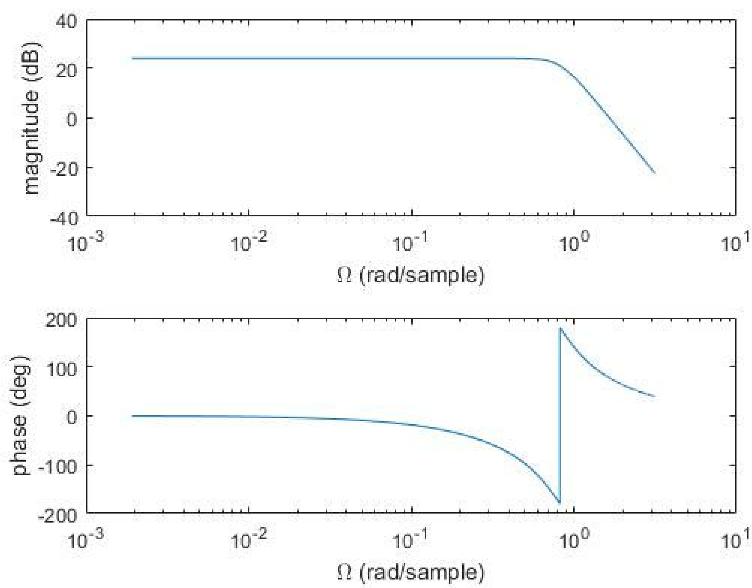

Figure 2: Discrete Time Frequency Response (Euler's Method)

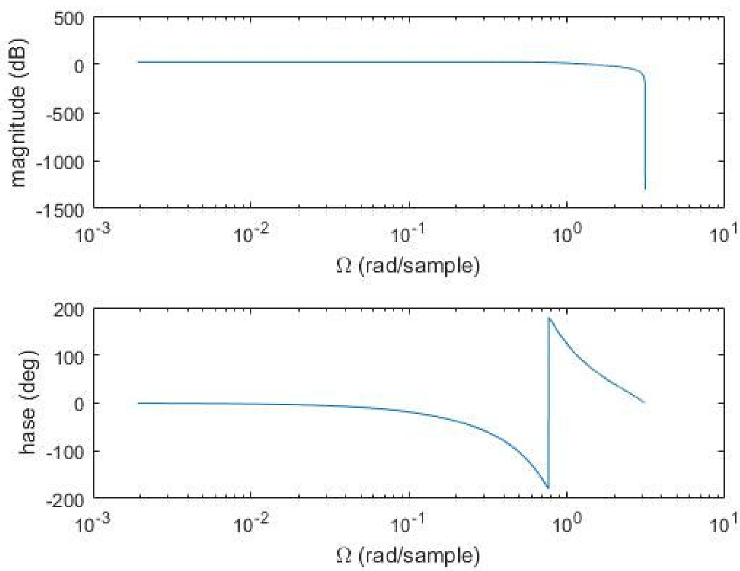

Figure 3: Discrete Time Frequency Response Using a Fixed-Point [6, -5] Approximation

Choosing a Computational Structure for the Filter

Direct Form 2 was chosen to represent this filter since we discretized using Tustin's approximation. Using one of the direct form implementations, we are approximating and accumulating error of only 4 coefficients total, instead of the 8 that we would have to approximate had we used a Coupled Form structure. To calculate how we could best approximate the coefficients with less than 5% error, we chose to use 5 fraction bits. The approximated coefficients are as follows:

$$a_1 = 1.125 \qquad a_0 = 0.5625$$

for the first polynomial, and

$$a_1 = 0.875 \qquad a_0 = 0.21875$$

for the second.

The largest of these, a_1 , can be represented using 1 integer bit, and 5 fraction bits. This makes our $[n, -f]$ representation turn out to be [6, -5] .

Below are the Direct Form 2 diagrams for this approximation:

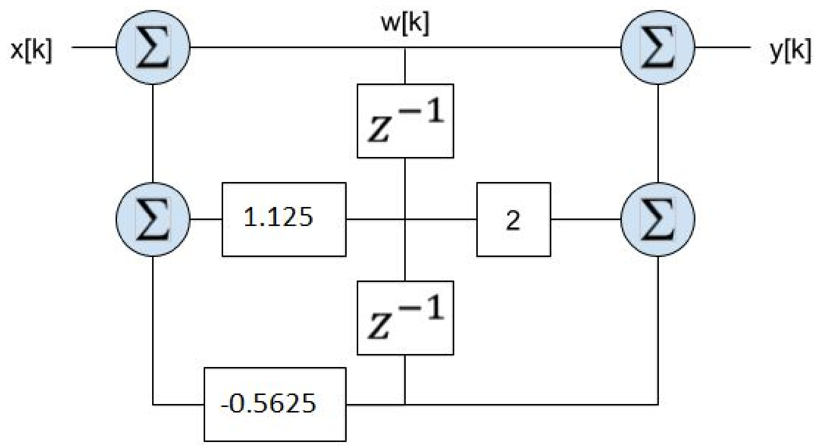

Figure 4: Direct Form 2 implementation of the first polynomial in the denominator

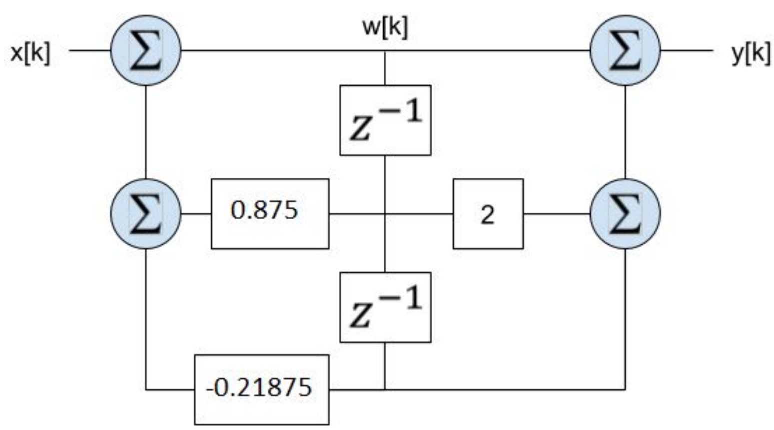

Figure 5: Direct Form 2 implementation of the second polynomial in the denominator

Part 2: Implementing the Discretized Digital Filters in C-Code

Writing C-Code to Implement Our Filters

To implement both discrete filters, non-approximated and approximated, using the C programming language, all we had to do was create floating point variables for the step response iterations, multiply them with our coefficients appropriately, and combine them how they appear in our part 1 implementation. If we were to calculate the step responses for these filters, the ykm1, and ykm2 values would change every step to refine the function. The code for our implementation is shown below:

```c
void main() {
    // These would change per iteration when calculating step response
    float xk = 1,
          ykm1 = 0,
          ykm2 = 0;

    // Discrete - Actual coefficients
    float yk1_actual = xk - 1.12051*ykm1 + 0.575922*ykm2;
    float yk2_actual = xk - 0.862021*ykm1 + 0.21238*ykm2;

    // Discrete - Approximated coefficients using 5 fraction bits
    float yk1_approx = xk - 1.125*ykm1 + 0.5625*ykm2;
    float yk2_approx = xk - 0.875*ykm1 + 0.21875*ykm2;
}
```

Appendix

A. Matlab Code for Part 1 (Produces Figures 1-3):

```
clc;
clear all;
close all;

w = 1643.838;
t = .0005;
m = 16.001616040804 * w .^4 ;


OmegaCont=linspace(0,2000*pi,10001); % Omega for cont
OmegaDisc=linspace(0,pi,w); % Omega for discrete

% Continuous
s = 1i * OmegaCont;
num = m;
den1 = (s.^2 + 0.7654*s*w + 1.0000505* w.^2);
den2 = (s.^2 + 1.8478*s*w + 1.0000505* w.^2);
den = den1 .* den2;
Hc = num ./ den;

% Plot continuous
figure(1)
subplot(211)
semilogx(OmegaCont,20*log10(abs(Hc)))
xlabel('\Omega (rad/sample)')
ylabel('magnitude (dB)')
subplot(212)
semilogx(OmegaCont,angle(Hc)*180/pi)
xlabel('\Omega (rad/sample)')
ylabel('phase (deg)')


% Discrete
z = exp(1i*OmegaDisc);
s = log(z)/t;
den1 = (s.^2 + 0.7654*s*w + 1.0000505* w.^2);
den2 = (s.^2 + 1.8478*s*w + 1.0000505* w.^2);
den = den1 .* den2;
Hd = num ./ den;

% Plot Discrete
figure(2)
subplot(211)
```

```
semilogx(OmegaDisc,20*log10(abs(Hd)))
xlabel('\Omega (rad/sample)')
ylabel('magnitude (dB)')
subplot(212)
semilogx(OmegaDisc,angle(Hd)*180/pi)
xlabel('\Omega (rad/sample)')
ylabel('phase (deg)')


% Discrete approximation
z = exp(1i*OmegaDisc);
% t and w already plugged in
num = ((z + 1).^4) * 0.15955904;
den1 = (z.^2 - 1.125.*z + 0.5625);
den2 = (z.^2 - 0.875.*z + 0.21875);
den = den1 .* den2;
Hda = num ./ den;

% Plot discrete approximation
figure(3)
subplot(211)
semilogx(OmegaDisc,20*log10(abs(Hda)))
xlabel('\Omega (rad/sample)')
ylabel('magnitude (dB)')
subplot(212)
semilogx(OmegaDisc,angle(Hda)*180/pi)
xlabel('\Omega (rad/sample)')
ylabel('hase (deg)')
```