SE 420: Lab 9 Report | Discrete Controller Design for a Magnetic Levitation Device

Ayush Sinha

Introduction

This experiment aims to control the position of a cantilever beam actuated through magnetic levitation using position feedback from Hall Effect sensors. The procedure begins with system identification followed by controller design in MATLAB, its simulation and subsequent implementation on actual system. As a final step, a slower trajectory is designed to replace the jerky step input and the controller's response is noted.

System Identification

System Description

The system consists of a cantilever beam with magnets on the free end. A current carrying coil causes force on the magnets and hence moves the beam. Hall Effect sensors output a voltage corresponding to the magnets' changing position (figure 1).

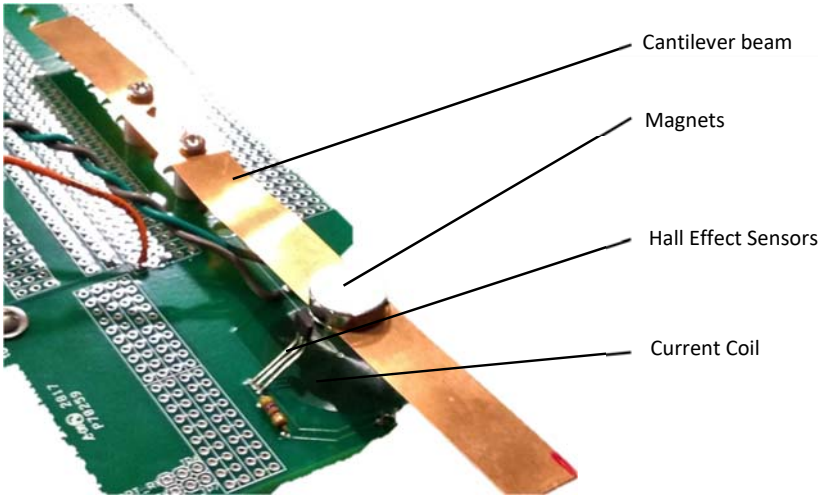

Figure 1: Magnetic Levitation Device

The system in figure 1 can be modelled as a spring-mass-damper system as shown in figure 2 with equation (1) as its mathematical model.

$$m\ddot{x} + c\dot{x} + kx = f(t) \tag{1}$$

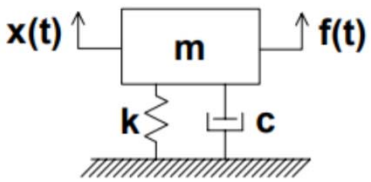

Figure 2: Simplified model

which gives the transfer function,

$$\frac{X(s)}{F(s)} = \frac{1}{k} * \frac{{}^{k}/_{m}}{s^2 + {}^{c}/_{m}\, s + {}^{k}/_{m}}$$

or,

$$\frac{X(s)}{F(s)} = Gain * \frac{\omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2}$$

Identification of Parameters

The voltage output of Hall Effect sensors is considered as output X(s) and control input (between -10 and 10), which is commanding the PWM output to motor drivers driving current in coil, is considered input F(s). C code is written to implement a step input switching between -1.5 and 0 every 15 seconds. The voltage output of Hall Effect sensors is observed using an oscilloscope (figure 3 and 4).

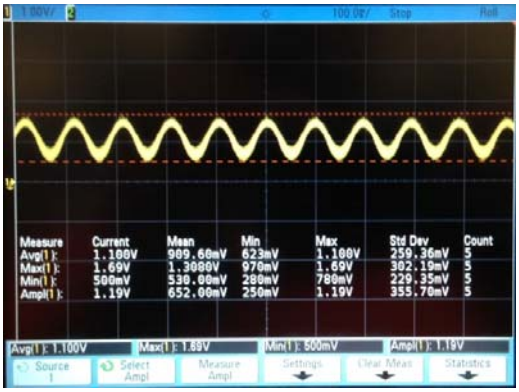

Figure 3: Output for input = 0

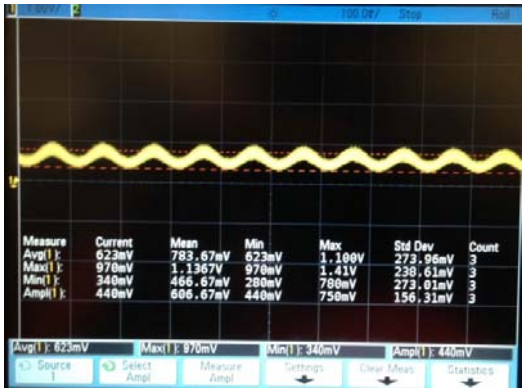

Figure 4: Output for input = -1.5

Since the vibrations persist for very long, the damping is assumed to be very small. For small damping ratio, ω_n is same as frequency of system's response. Hence, noting that output's frequency is nearly 10 Hz,

$$\omega_n = 2\pi * 10\, rad/s$$

Now, gain can be computed as ratio of response and input. However, the offset at zero input must be accounted. Hence, the gain is computed as,

$$Gain = \frac{0.623 - 1.100}{-1.5} = 0.325$$

Finally, the damping ratio is determined by trial and error. Simulated step responses for systems with above calculated ω_n and $Gain$ and different ξ (damping ratio) are compared with actual response on the oscilloscope. Figure 5 shows that ξ = 0.01 causes vibrations to die out much faster than the actual response. Figure 6 shows that system with ξ = 0.001 has vibrations of similar amplitude as the actual response. Thus, ξ = 0.001 is selected.

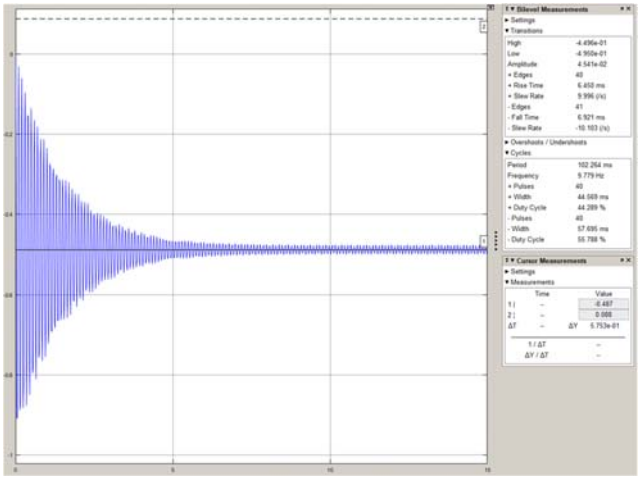

Figure 5: Simulated response for zeta = 0.01

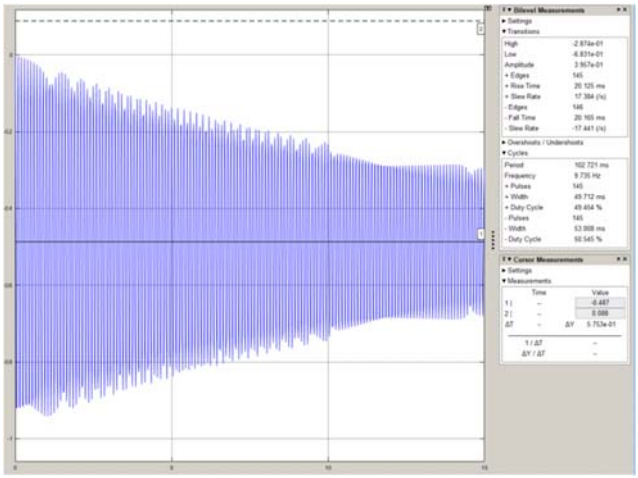

Figure 6: Simulated response for zeta = 0.001

Finally, the discrete model is determined using the following MATLAB commands,

```
%% Continuous Time Plant
K = 0.325; %Gain
w = 2*pi*10; %frequency (wn in rad/s)
ze = 0.001; %zeta - damping ratio
planttf = tf([K*w^2],[1 2*ze*w w^2]); %continuous plant
%% Discrete Time Plant
plantdtf = c2d(planttf,0.001,'zoh'); %discrete plant using ZOH
```

Controller Design

PID control scheme is selected for position control to ensure zero steady state error and minimal vibrations. The control law is as shown in equation (2). Note, e is error between desired sensor output and actual sensor output and v is derivative of actual sensor output.

$$u = K_p * e - K_d * v + K_i * \int e \tag{2}$$

The derivative term is approximated using $s \approx \frac{500s}{s+500}$ to avoid higher frequency noise and then converted to discrete form using Tustin approximation. Moreover, the emulation of PI control using Tustin approximation will lead to the following transfer function,

$K_{Loop} * \frac{z-zero_c}{z-1}$, with $K_{Loop} = \frac{2*\mathrm{Kp}+\mathrm{Ki}*\mathrm{T}}{2}$ and $zero_c = -(\frac{Ki*T-2*Kp}{2*Kp+Ki*T})$. K_{Loop} and $zero_c$ are the design values that are tuned in rltool to design the PI portion of the controller.

With the above setup and plant transfer function determined in earlier section, rltool in MATLAB along with above equations are used to find PID gains that achieve the desired characteristics as presented next.

1. Overshoot < 2%
2. Settling time < 150ms
3. 0% steady state error
4. Minimal oscillations at steady state

Figure 7 and 8 are screen captures from rltool session that led to determination of PID gains to achieve the above mentioned characteristics.

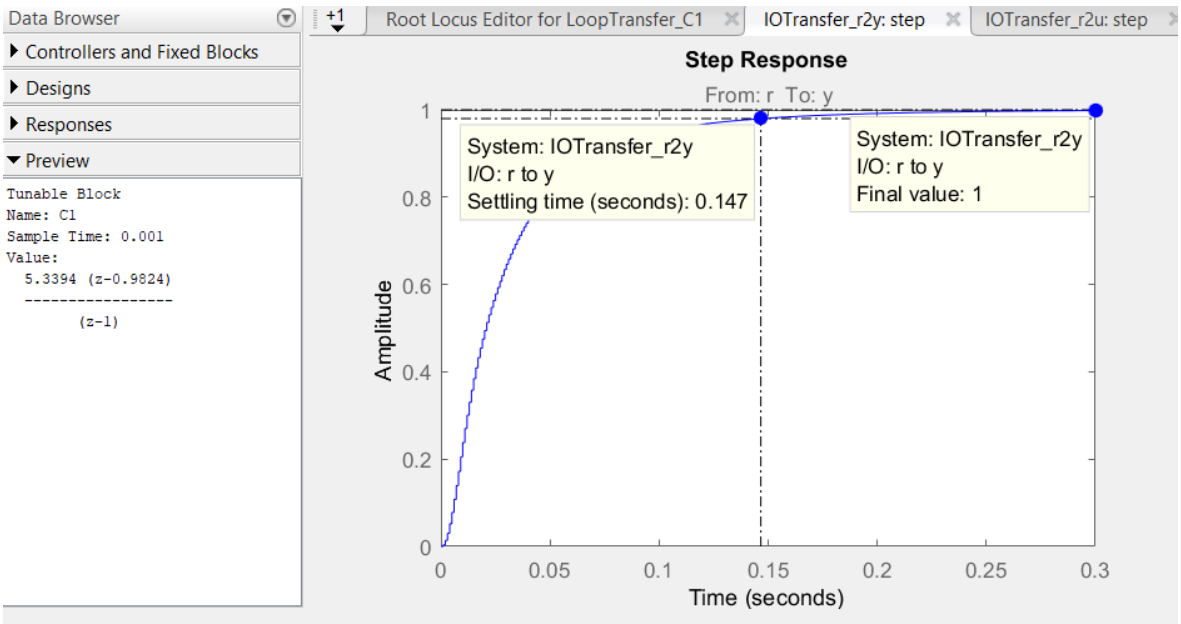

Figure 7: Step response achieving required characteristics and PI transfer function in left panel

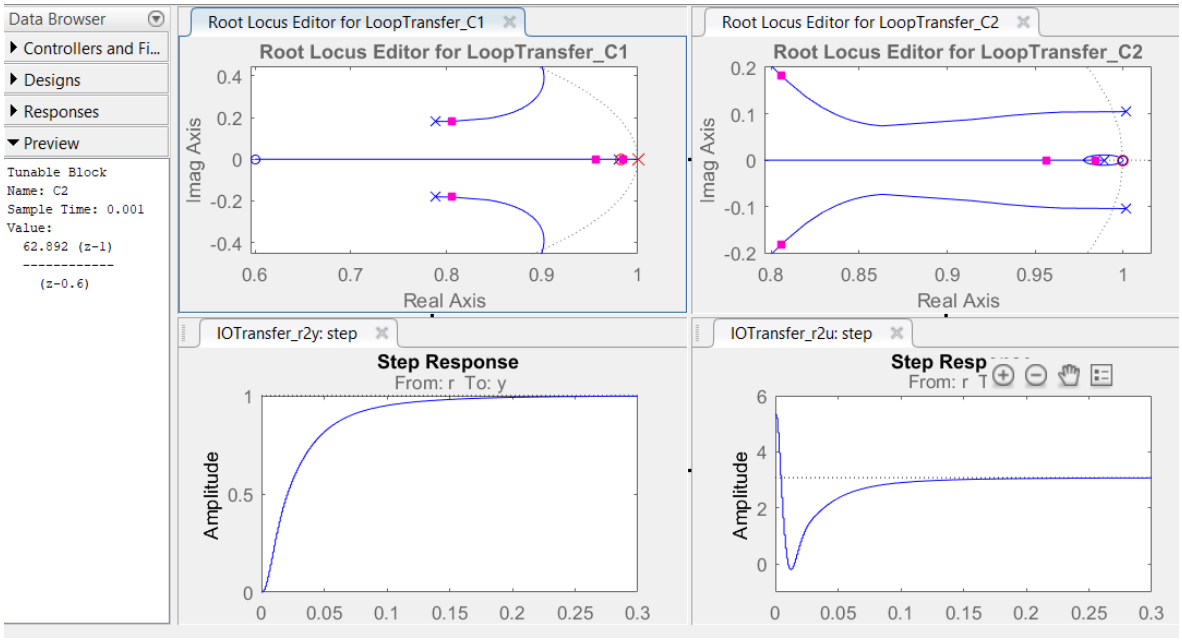

Figure 8: All plots in rltool and Derivative term transfer function in left panel

Controller Implementation and Results

The C implementation is presented in Appendix A (code snippet). Figure 9 is the Simulink model prepared to simulate the control law and even compare with the actual response. Figure 10 plots reference, simulated response and actual response to a step input.

A slower trajectory was designed to avoid sudden change in desired position and hence eliminate jerky transition. This slower trajectory was obtained by passing the square (step) input trajectory through a sixth order low-pass filter. The filter used is $\frac{20^6}{(s+20)^6}$ and its step response is as shown in figure 11. Finally, figure 12 plots the slower reference, simulated response and actual response. Observe that the actual response matches very closely with reference and simulated response for both trajectories.

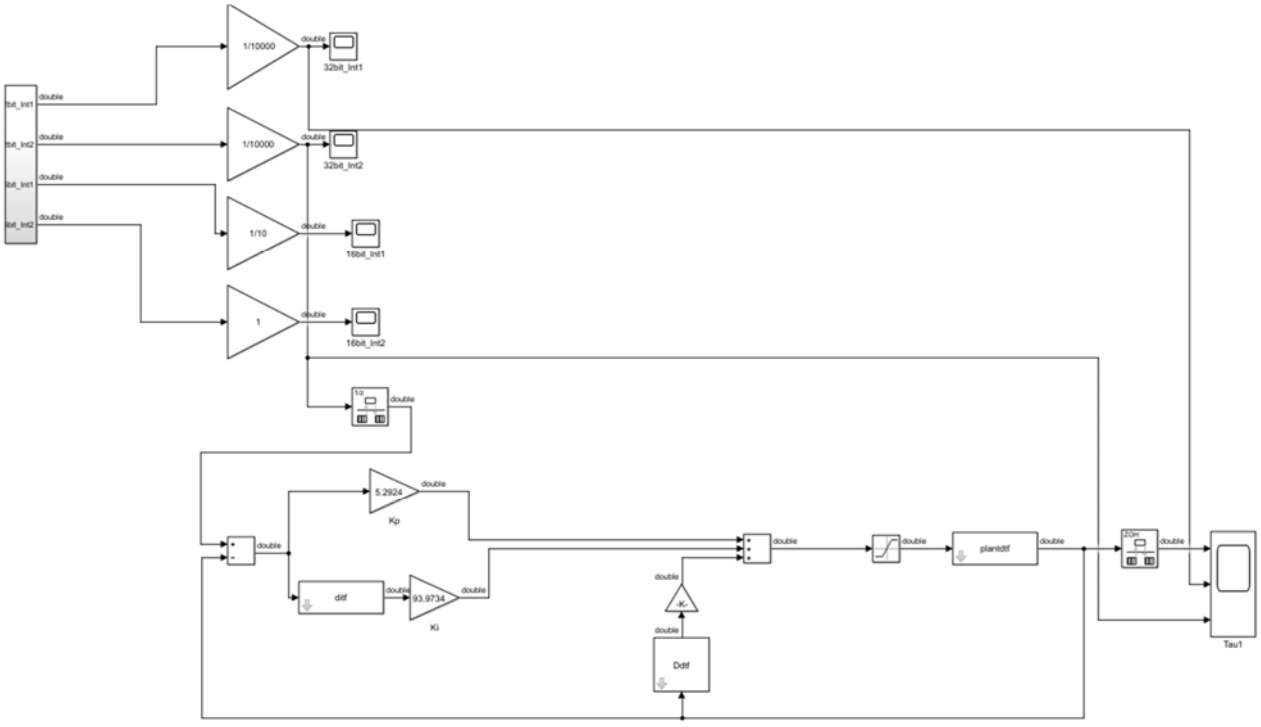

Figure 9: Simulink model

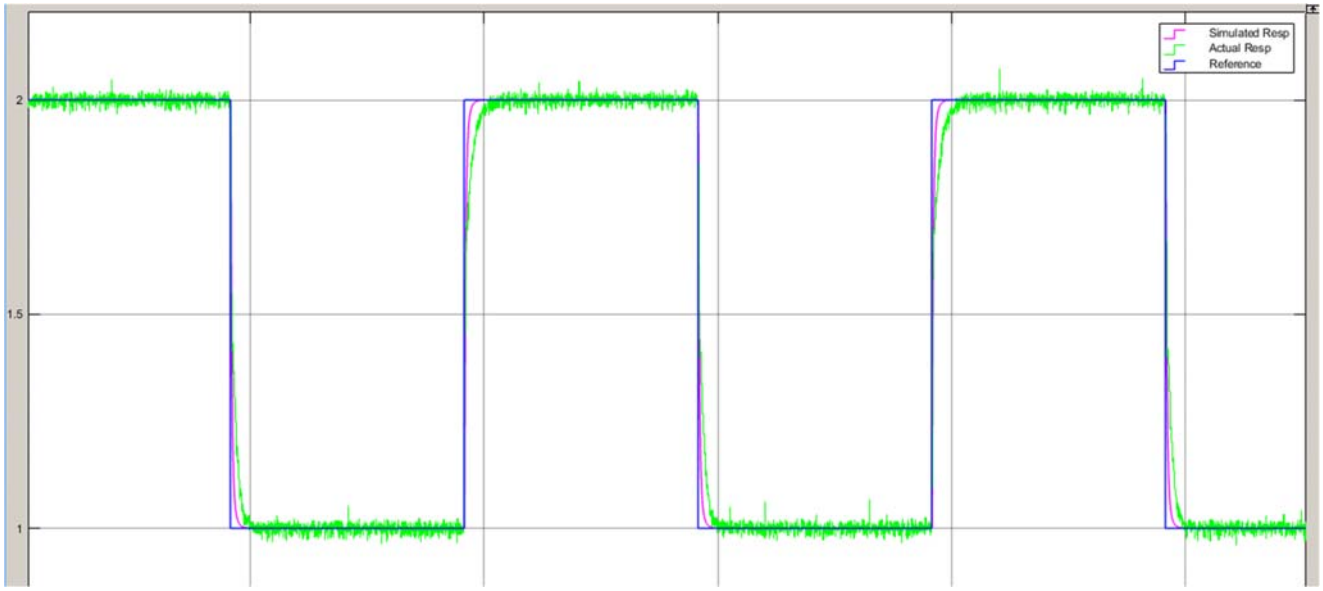

Figure 10: Reference, Simulated response, Actual response for step input

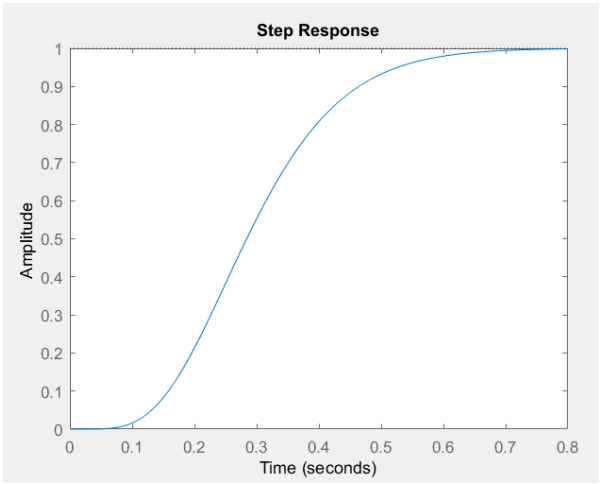

Figure 11: Step response of filter for slow trajectory

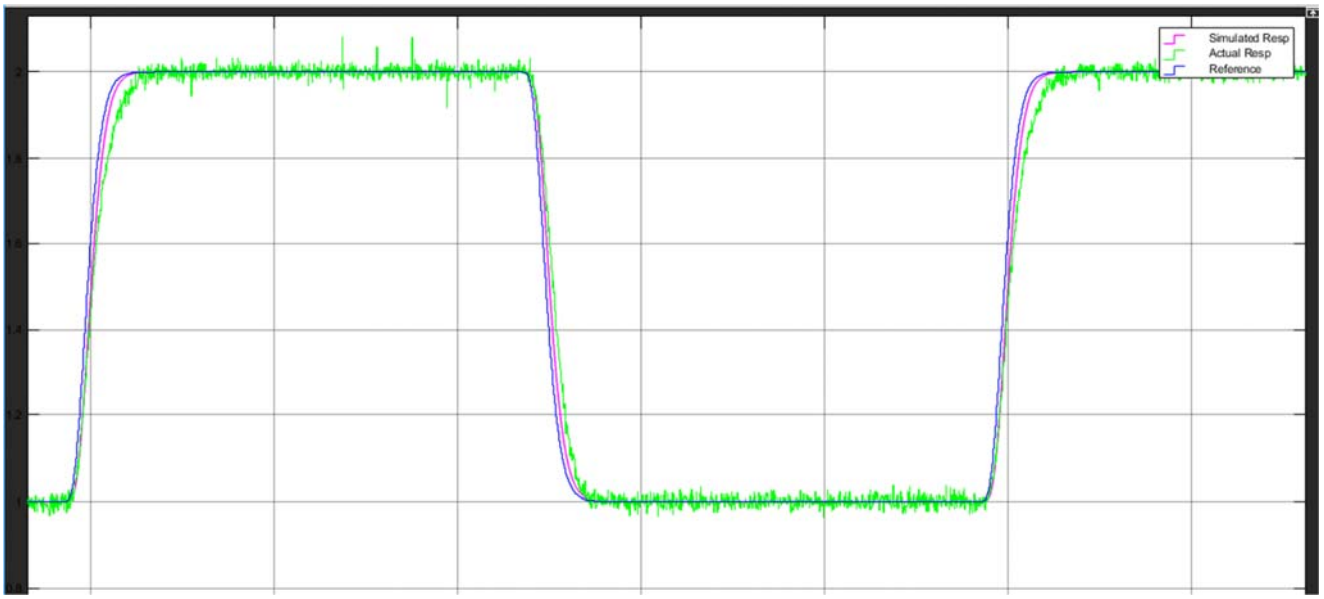

Figure 12: Reference, Simulated response, Actual response for slower trajectory

Appendix A: Snippet of C Code for microcontroller

```
//Variables
long timer = 0; // keeps track of time in ms
//reference signal generation
long double amp1=2; // reference high level in volts
long double amp2=1; // reference low level in volts
long double reference = 0; // stores reference value for each ms

// controls vars
long double u =0; // control effort
long double ADCvoltsb2 = 0; // Hall Effect sensors output in volts
long double vel_old = 0;
long double vel = 0; // derivative of sensor output
long double posn = 0; // sensor output
long double posn_old = 0;
long double error = 0; // error between desired posn and actual posn
long double error_old = 0;
long double integral = 0;
long double integral_old=0;

// PID Gains found thru MATLAB
long double Kp = 5.2924;
long double Kd = 0.1572;
long double Ki = 93.9734;

// Filter for slower trajectory
long double numfil[7]= {    9.4204523525420674e-13L,
                            5.6522714115252405e-12L,
                            1.4130678528813101e-11L,
                            1.8840904705084135e-11L,
                            1.4130678528813101e-11L,
                            5.6522714115252405e-12L,
                            9.4204523525420674e-13L};
long double denfil[7]= { 1.0000000000000000e+00L,
                        -5.8811881188118820e+00L,
                        1.4411822370355853e+01L,
                        -1.8835252998880922e+01L,
                        1.3846708268979292e+01L,
                        -5.4290064104116844e+00L,
                        8.8691688882963160e-01L};

long double r[7] = {0,0,0,0,0,0,0}; // stores reference values (square wave)
long double fr[7] = {0,0,0,0,0,0,0}; // stores filtered slower traj

// Generates Square wave
void ref(void){
    if(timer < 5000){
        reference = amp1;
    }
    else if((timer >= 5000)&&(timer < 10000)){
        reference = amp2;
    }
    else{
        reference = amp1;
        timer = 0;
    }
}

// Generates slower traj
void myfilter(void)
{
    r[6] = reference;
    fr[6]=-denfil[1]*fr[5]-denfil[2]*fr[4]-denfil[3]*fr[3]-denfil[4]*fr[2]-denfil[5]*fr[1]-
denfil[6]*fr[0];
```

```
fr[6]+=numfil[0]*r[6]+numfil[1]*r[5]+numfil[2]*r[4]+numfil[3]*r[3]+numfil[4]*r[2]+numfil[5]*r[1
]+numfil[6]*r[0];

    int i;
    for(i =0; i<6; i++){
        fr[i]=fr[i+1];
        r[i]=r[i+1];
    }
}

// Runs every 1 ms
void myclockfunc(void)
{
    AdcbRegs.ADCSOCPRICTL.bit.RRPOINTER = 0x10;
    AdcbRegs.ADCSOCFRC1.all |= 0x7;  // now starting 3 ADCB channels
}

void ADChwifunc(void)
{
    timer++;
    ref();
    myfilter();

    ADCB2result = AdcbResultRegs.ADCRESULT2;
    ADCvoltsb2 = 3.0*ADCB2result/4095.0; // sensor output in volts
    posn = ADCvoltsb2;
    vel = 0.6*vel_old + 400*(posn - posn_old); // derivative s = 500s/(s+500)
    error = fr[6] - posn;
    // error = reference - posn; // for square wave


    integral = integral_old + 0.001*(error + error_old)/2.0;
    u = Kp*error - Kd*vel + Ki*integral; // control law

    //saturation and anti-windup
    if((u < -10)){
        u = -10;
        integral = integral_old;
    }
    if((u > 10)){
        u = 10;
        integral = integral_old;
    }

    setEPWM8A(u); // sending control effort to PWM channel

    // storing old vals
    vel_old=vel;
    posn_old = posn;
    error_old = error;
    integral_old = integral;

    AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // clearing adc flag
}
```