Controlling Position of Different Inertias with a DC Motor

Cole Sheyka

May 2023

1 Abstract

The purpose of this project was to use the quadrature decoder on a bushed, DC motor to read its position, then control its position based on a user input. Two different intertias attached to the shaft of the DC motor to determine systems response and corresponding transfer function. Determination of the open-loop transfer function allows for a difference equation to control an inertias final position on a spinning motor with the Arduino IDE.

2 Introduction

By virtue of two Hall effect sensors, the motor's quadrature encoder is capable of reading the shaft position in units of [counts/cycle]. The encoder outputs two square waves offset in phase by π with a 50% duty cycle to pins 2 and 13 on the Due. An example of encoder outputs is shown in Figure 1 below.

The B channel in Figure 1 has been offset by a value of -3.5 such that both channels can be seen easily. Otherwise, the range of voltage values for the channels is [0 3.3]. It can be derived that the amount of counts per one cycle is 211, meaning that 211 counts/cycle is equivalent to 2π rads.

An H-bridge motor driver was used to transfer power to the DC motor via the Vin pin on an Arduino Due. With this motor driver, it is possible to control two motors at the same time, although only one was used for this experiment. Moreover, the motor driver allows the user to control speed with a PWM signal, and direction by writing a digital pin to be HIGH or LOW. Depending on how the DC motor was wired, sending a 1 to the directional pin would make the motor spin counter-clockwise or clockwise, resulting in the encoder counting in positive or negative numbers.

Once the hardware was wired correctly, data had to be collected to determine the transfer function of the motor such that a proper proportional-derivative (PD) controller could be developed to control position of the motor as specified by user input.

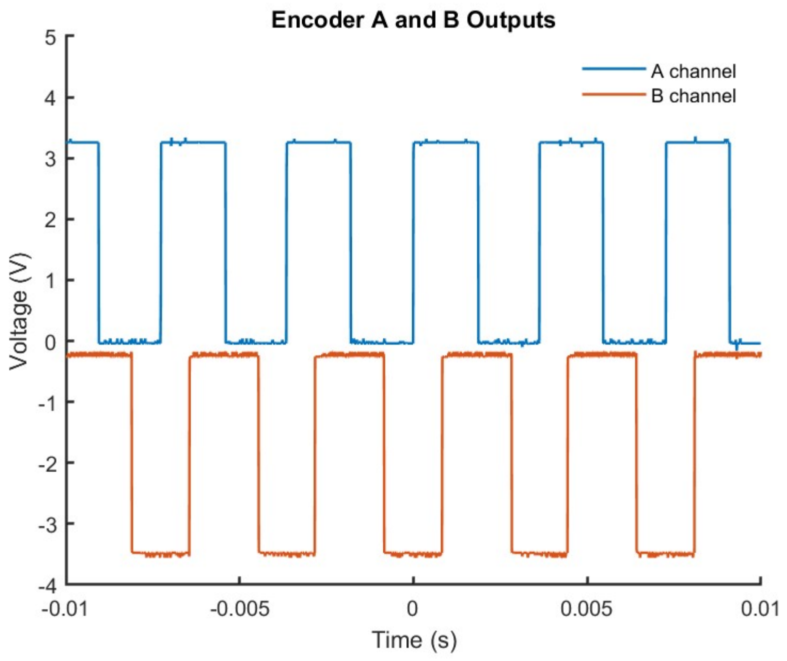


Figure 1: Block diagram with both plants in feedback with a controller

3 Experimental Set Up

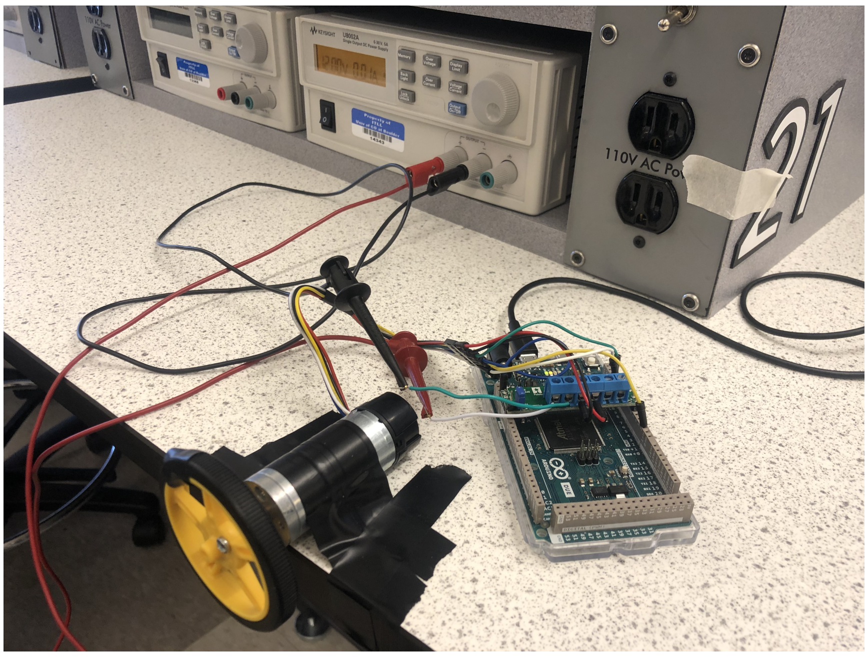


Figure 2: Experimental set up with small inertia shown

Figure 2 shows the small inertia attatched to the DC motor being studied in these experiments. The board shown in the Arduino Due. The power pins for the motor are wired to a modular motor driver also shown above. 12 Vcc was pulled from a DC power supply through the Vin pin of the motor driver. The emmulation power pin was plugged into the Due's 3.3 V power supply to ensure that the ESC is communicating at the same logic levels as the Due. This results in the A and B channels of the emmulator to count in a voltage range of [0 3.3] Vdc.

4 Determining the Model

To get the transfer function of the motor, 12V was applied to the Vin pin of the motor driver and an Arduino sketch was uploaded to give the system speed and directional instructions. In the Arduino sketch, an interrupt was used to tell the CPU to update what it's doing. In order to collect data, an interrupt function calls a void update function every 0.01 seconds to write speed and direction to the motor. Either a small or large inertia was allowed to rotate until it was determined that the shaft was rotating at some constant rate. Intuitively, for the larger inertia, the system takes more time to reach steady-state and vise versa for the smaller inertia.

The motor's emulator outputs counts/rev to the serial monitor of the Due every time the void update function is called. Then, this data was saved to a .txt file for further analysis. This positional data was saved and imported to MATLAB for further analysis. In MATLAB, a derivative difference equation was used to convert rads to rads/s. After dividing the resulting speed array by the voltage applied to the motor then plotting the results, a step response of the plant can be attained.

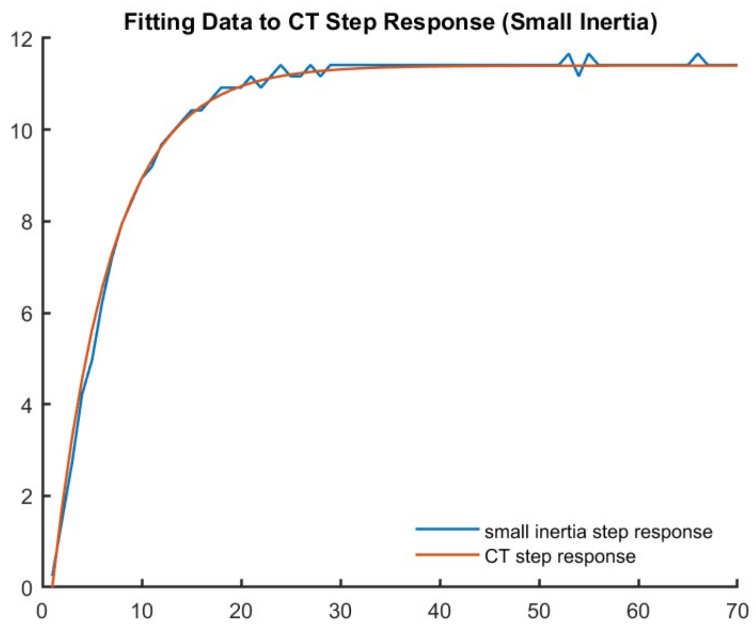


Figure 3: Fitting a curve to small inertia step response

Note that in Figures 3 and 4, the x axis represents indexing in the data array, not time. For example

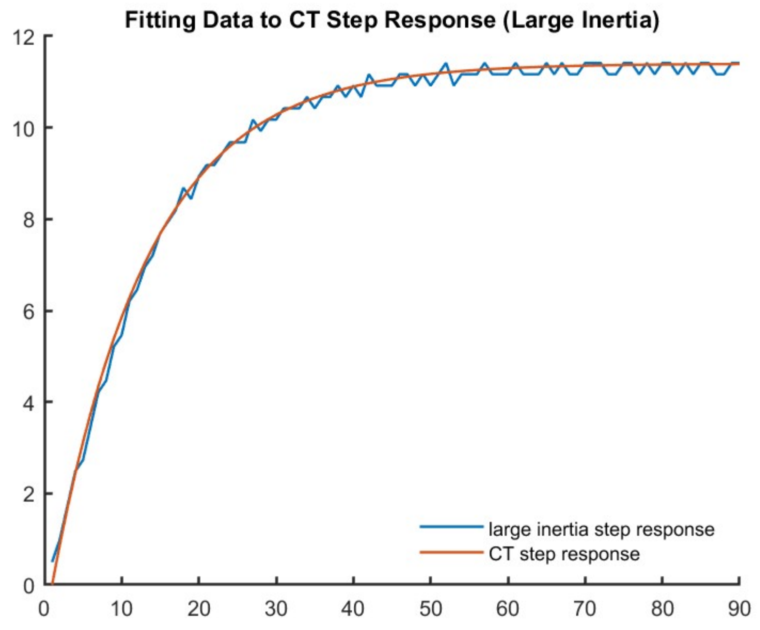


Figure 4: Fitting a curve to large inertia step response

with the large inertia, 90 samples of data were recorded in 0.9 seconds. Repeating this process for both inertias allows us to fit a discrete curve to the model and back out the DC gain and pole location for a first-order approximation of the second-order plant. The equation used to fit these curves is

$$G = k(1 - e^{-\sigma t}) \tag{1}$$

where G is the plant, k is the DC gain of the system, and σ is the pole location. The value for the DC in both cases was easily found by recording what the plant converges to in the figures above. The pole locations were found by iteratively adjusting the value and referring back to the model until the curves match. For each inertia the DC gain for the plant was the same, although the pole locations changed roughly by a factor of 2.

These parameters for the respective inertias were used to build two transfer functions for the large and small inertias in continuous time. These two plants were also emmulated using ZOH in MATLAB to have two discrete time transfer functions.

5 Determining the Controller

After the necessary variables of the plant had been recorded, a PD controller was designed to control desired position of the motor. In continuous time, the PD controller is non-realizable since there is no pole in the denominator. It can be written as

$$C(s) = \frac{U(s)}{E(s)} = sk_d + k_p \tag{2}$$

where $C(s)$ is the continuous-time, open loop PD controller, k_d is the derivative gain, and k_p is the proportional gain. The proportional gain for both inertias was found by tuning a PID block in Simulink. This can be seen in Figure 6.

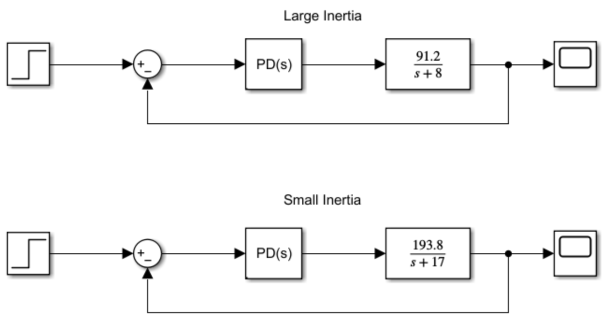
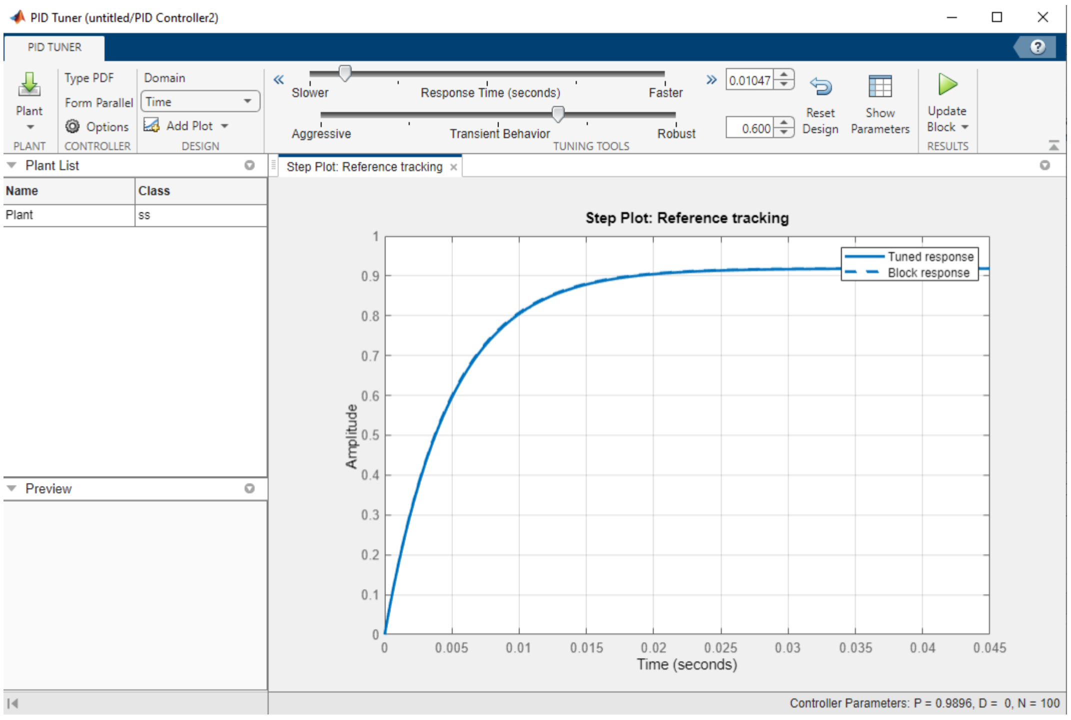


Figure 5: Block diagram with both plants in feedback with a controller



Controller Parameters: P = 0.9896, D = 0, N = 100

Figure 6: Block diagram with both plants in feedback with a controller

The response time slider was adjusted until the two curves seen in Figure 6 matched each other. By observing the P value seen in the bottom left corner, the proportional gain value can be recorded. In the case of these experiments, although two different plants were derived, the proportional gain value is the same to control both inertias. Therefore, it was determined that only one controller had to be designed to control both inertias.

The open-loop transfer function, L, is then

$$L = C * G \tag{3}$$

which is true for both the continuous time and discrete time cases. The closed-loop transfer function is then

$$H = \frac{L}{1 + L}. \tag{4}$$

To get the emmulation curve seen in the figures below, the feedback() function in MATLAB was used to simulate closed loop feedback with L. Then, by using the step() function, the corresponding step response can be seen. Note that stepping the closed loop transfer function, H, accomplishes the same task.

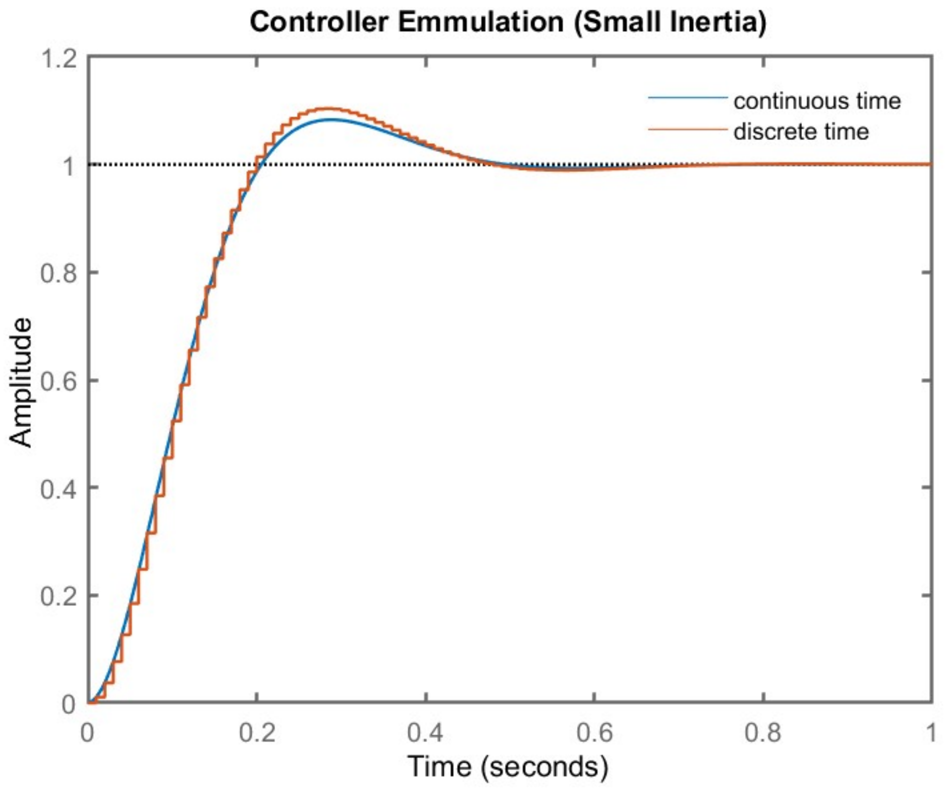


Figure 7: Closed-loop emmulation of L(s) (small inertia)

After emmulating the controller in discrete time using Tustin's method in MATLAB, the appropriate derivative gain can iteratively determined by comparison to to the continuous time controller. Instead of simply passing an unrealizable C(s) into the c2d() function in MATLAB, a low pass filter was added

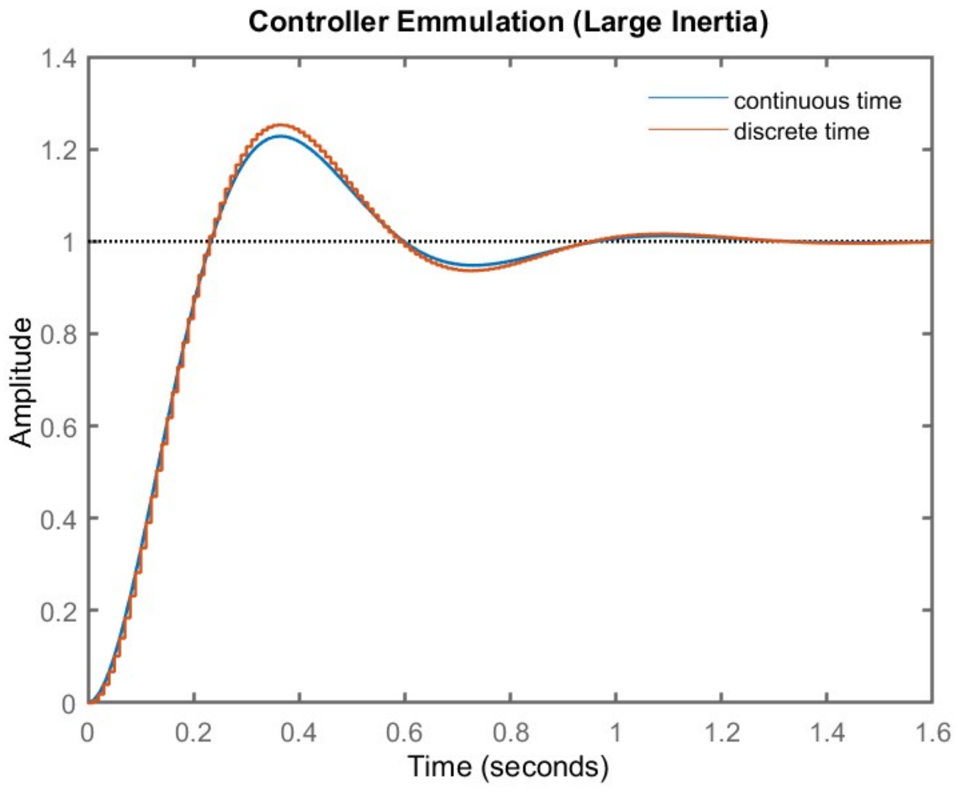


Figure 8: Closed-loop emmulation of L(s) (large inertia)

to make the controller realizable in continuous time, and therefore in discrete time. To reiterate, only the discretized contoller has this low pass filter while the continuous time controller does not. For this experiment, two controllers were used for analysis, one being in discrete time with a low pass filter, and the other being in continuous time.

Besides using Tustin's or ZOH method with the c2d() function in MATLAB, there are other solvers which accomplish nearly the same task, like Euler's forward or backwards method for example. However, Euler's methods are not built into the c2d() function so the user has to define the C(z) by hand if this solver is desired. A Bode plot of the original controller, and two emmulated controllers using Tustin's and backwards method are shown below.

For this experiment, Tustin's method was chosen when determining the closed-loop transfer function, and therefore the difference equation which was implemented into the Arduino IDE to control position.

After it was determined that the closed-loop transfer function was emmulated properly, the $C(z)$ had to be turned into code via a difference method such that the Due could update to a specified final position. The emmulated controller in terms of z given the chosen proportional and derivative gains is

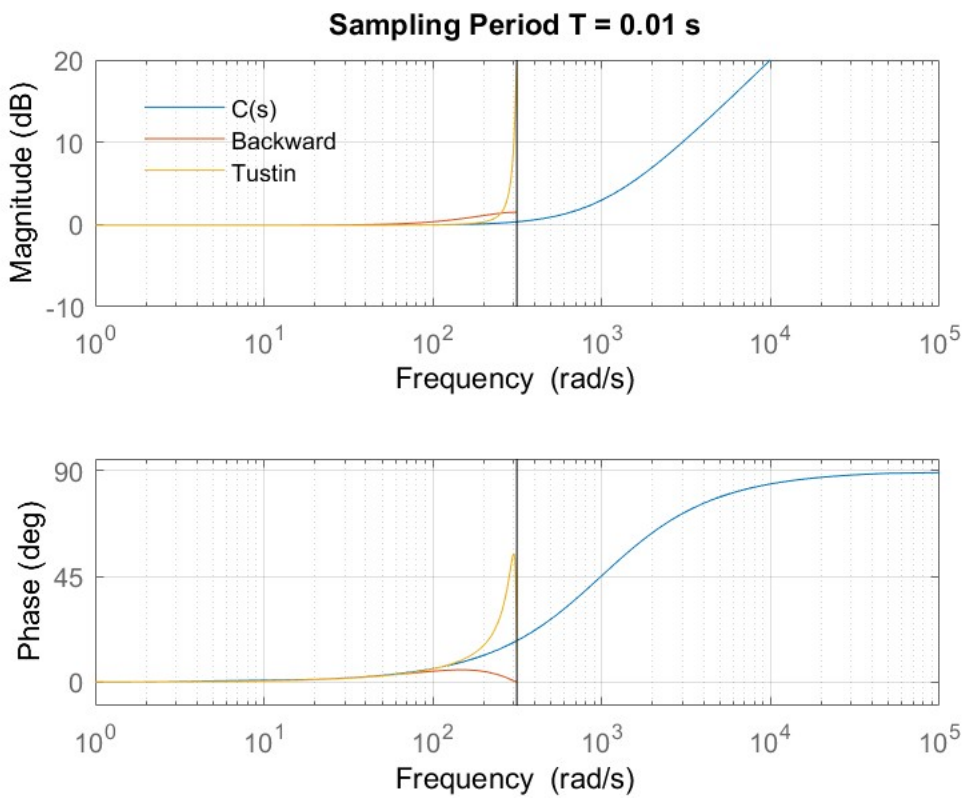


Figure 9: Bode plot with different solvers

$$C(z) = \frac{1.166z + 0.774}{z + 0.9604} \tag{5}$$

and the corresponding difference equation is

$$u[k] = 1.166e[k] + .774e[k-1] - .9604u[k-1]. \tag{6}$$

Equation 6 is what is converted into code, then implemented into the Arduino IDE under the void update function seen in the appendix.

6 Arduino IDE

This section further explains the implementation of the PD controller in the Arduino IDE. To accomplish this, 4 floating point variables were initialized globally to hold each value of u and e seen in equation 6. The desired final position was defined, and a speed variable was also initialized.

In the void setup function, pins 9 and 7 were initialized as outputs for the Due (since motor channels 1A and 1B were used with the motor driver), and specific clocks were identified to interact with registers directly on the Due. This allows the CPU to easily read positional data from the encoder via an interrupt function without the need for extra functions and loss of computational efficiency. Experimentally, it was found that a baud rate of 115,200 was effective for capturing data accurately.

The void update function is called by the interrupt function in the set up every 0.01 seconds. This function calculates the output of the PD controller, which is in units of volts. A quick conversion to a decimal value in the range of 12 bits and a variable change allows mapping between the output of the controller and desired position. The difference between current position and desired position, or error, is proportional to the output of the controller. Therefore, as the shaft on the motor spins and approaches a desired position, the error decreases and so does power to the motor. Once the inertia has rotated to its final position, the shaft will stop rotating.

7 Conclusion

We were successfully able to emmulate a continuous time PD controller and implement it into the Arduino IDE to control the final position of a spinning shaft from a DC motor. This was done by first identifying the transfer function of the plane being studied, finding the closed-loop step response, and using a difference equation to represent the emmulation in the Arduino IDE. Once the Arduino code is compiled, the CPU will call an update function to interpret distance from the final value.

8 MATLAB Code

```
clc, clear, close all;

% count data
counts = readmatrix("encoder.csv");
tc = counts(:,1);
A_ch = counts(:,2);
B_ch = counts(:,3);

% plot A and B channel
figure;
hold on
plot(tc,A_ch,'linewidth',1.2)
plot(tc, B_ch-3.5,'linewidth',1.2)
legend('A channel', 'B channel')
legend boxoff
ylim([-4,5])
title('Encoder A and B Outputs')
xlabel('Time (s)')
ylabel('Voltage (V)')
set(gca,'linewidth',1.2)
hold off

% large inertia
LI = readmatrix("large_inertia.txt");
LI = (2*pi/211)*LI; % convert to rad

LIwC = readmatrix('LI_with_C.txt'); % already in rads
pos = 2*pi; % desired position
twc = (1:length(LIwC))'*0.01;
LIwC = [twc LIwC/pos];
```

```
% small inertia
SI = readmatrix('small_intertia.txt');
SI = (2*pi/211)*SI; % convert to rad

% derivative (rad/s)
Ts1 = 0.01;
for i = 1:length(LI)-1
    LI(i) = (LI(i+1) - LI(i))/ Ts1;
end
for i = 1:length(SI)-1
    SI(i) = (SI(i+1) - SI(i))/ Ts1;
end
% for i = 1:length(LIwC)-1
%     LIwC(i) = (LIwC(i+1) - LIwC(i))/ Ts1;
% end

V_in = 12;  % V
LI = LI(1:90)./V_in;
SI = SI(1:70)./V_in;

% time vectors
tLI = (0:length(LI)-1)' * Ts1;
tSI = (0:length(SI)-1)' * Ts1;
xlabel = linspace(0,40,Ts1);



% parameter fitting
% large inertia
k_LI = 11.4;
s_LI = 8;
step_LI = k_LI*(1 - exp(-s_LI*tLI));

% small inertia
```

```
k_SI = 11.4;
s_SI = 17;
step_SI = k_SI*(1 - exp(-s_SI*tSI));

% transfer fcns
G_LI = k_LI*tf(s_LI,[1,s_LI,0]);   % large inertia position
G_SI = k_SI*tf(s_SI,[1,s_SI,0]);   % small inertia position

% rlocus(G_LI)



% % feedback loop
% L = C_LI*G_LI;
%
% T1 = feedback(L,1);
% step(T1, 10)
% rlocus(L)



% PD controller: small inertia
kd = 0.001;
kp = 0.9896;
tau = (10*(kp/kd))^-1;

% backward euler method
Ts = 0.01; % s
z = tf('z', Ts); % Define z as a discrete time variable

C_zLI = ((z-1)/(Ts*z))*kd + kp;

% controller info
C_DT = tf([kd kp],[tau,1]);
C_CT = tf([kd kp],1);
```

```
% tustins method and controller
TM_C = c2d(C_DT,Ts,'tustin');

% zoh for plant emmulation
TM_GLI = c2d(G_LI,Ts,'zoh');
TM_GSI = c2d(G_SI,Ts,'zoh');

% getting openn loop L values for cont and discrete time
L_LI = G_LI*C_CT;  % continuous time
Lz_LI = TM_GLI*TM_C;   % discrete time

L_SI = G_SI*C_CT;    % continuous time
Lz_SI = TM_GSI*TM_C;   % discrete time


% closed-loop tf
H_LI = L_LI/(1+L_LI);


figure;
plot(LIwC(:,1),LIwC(:,2),'linewidth',1.2)

figure;
hold on
step(feedback(L_LI,1))
step(feedback(Lz_LI,1))
% plot(LIwC,'linewidth',1.2)
% step(H_LI) note that this is the same as the above line
legend('continuous time','discrete time')
legend boxoff
title('Controller Emulation (Large Inertia)')
set(gca,'linewidth',1.2)
set(findall(gcf,'type','line'),'linewidth',1.2)
hold off

figure;
hold on
step(feedback(L_SI,1))
step(feedback(Lz_SI,1))
% step(LD_LI)
legend('continuous time','discrete time')
legend boxoff
title('Controller Emmulation (Small Inertia)')
set(gca,'linewidth',1.2)
set(findall(gcf,'type','line'),'linewidth',1.2)
hold off

% figure;
% step(G_CL)

figure;
hold on
plot(LI,'linewidth',1.2)
plot(step_LI,'linewidth',1.2)
title('Fitting Data to CT Step Response (Large Inertia)')
legend('large inertia step response','CT step response', ...
    'location','southeast')
legend boxoff
set(gca,'linewidth',1.2)
hold off



figure;
hold on
plot(SI,'linewidth',1.2)
plot(step_SI,'linewidth',1.2)
title('Fitting Data to CT Step Response (Small Inertia)')
legend('small inertia step response','CT step response', ...
    'location','southeast')
legend boxoff
set(gca,'linewidth',1.2)
hold off

% figure;
% plot(TM_C)



opts1=bodeoptions('cstprefs');
opts1.PhaseVisible = 'off';
opts1.YLim={[-10 20]};
opts1.XLim={[0 10^3]};

opts2=bodeoptions('cstprefs');
opts2.MagVisible = 'off';
opts2.YLim={[-10 95]};
opts2.XLim={[0 10^3]};

subplot(2,1,1)
hold on
bodeplot(C_CT,opts1);
bodeplot(C_zLI,opts1);
bodeplot(TM_C,opts1);
grid on
title('Sampling Period T = 0.01 s')
legend('C(s)','Backward','Tustin','location','northwest')
legend boxoff
set(xlabel(''),'visible','off');
hold off
```

```
subplot(2,1,2)
hold on
bodeplot(C_CT,opts2);
bodeplot(C_zLI,opts2);
bodeplot(TM_C,opts2);
grid on
title('')
hold off

% figure;
% look at open loop to assess closed loop
% rlocus(LD_LI)
% %
% figure;
% margin(LD_LI)

% figure;
% step(LD_LI)
```

9 Arduino Code

```
#include "DueTimer.h"
int count_q = 0;        // quadrature decoder hardware counts
float pos =2*PI;
//float setpoint = pos - 0.25;  // large inertia OL input shaping and setpoint
float setpoint = pos - 0.35;  // small inertia OL input shaping and setpoint
float u1 = 0;
float u2 = 0;
float e1 = 0;
float e2 = 0;
float sped = 0;



void setup() {

  pinMode(9,OUTPUT);
  pinMode(7,OUTPUT);

  analogWriteResolution(12);     // set DAC output to maximum 12-bits
  Timer3.attachInterrupt(update).start(1000); // start ISR timer3 (not used by quad deco

  // setup for encoder position measurement (Digital pins 2 and 13)
  // This is described in Chapter 36 of the SAM3X8E datasheet

    REG_PMC_PCER0 = PMC_PCER0_PID27;
    REG_TC0_CMR0 = TC_CMR_TCCLKS_XC0;

    REG_TC0_BMR = TC_BMR_QDEN
                | TC_BMR_POSEN
                | TC_BMR_EDGPHA;

    REG_TC0_CCR0 = TC_CCR_CLKEN
                 | TC_CCR_SWTRG;
```

```
Serial.begin(115200); //115200 or 9600
}

void loop() {
//  count_q = REG_TC0_CV0;
//  float rad = (2*PI/211.2)*count_q;
//  Serial.println(rad);

}

// update the control output using the difference equation
void update() {

  count_q = REG_TC0_CV0;
  float rad = (2*PI/211.2)*count_q;
  int pwm = 4095;

  // calculate the error signal
  e1 = setpoint - rad;
  u1 = 1.166*e1 + .774*e2 - .9604*u2;

  // convert output to PWM
  sped = u1*pwm/12;

  if (sped > pwm){
    sped = pwm;
  }

  if (sped < 0){
    sped = -sped;
  }
  if (u1 < 0){
    digitalWrite(7,LOW);
    analogWrite(9,sped);
  }
  else{
    digitalWrite(7,HIGH);
    analogWrite(9,sped);
  }

  // update the previous error signal and control output for the next time step
  e2 = e1;
  u2 = u1;
  Serial.println(rad);

}
```