Speed and Position control for a Brushed DC Motor

Venkata Mahesh Reddy Guddeti

May 2023

Abstract

This report describes the process of constructing a PD controller for controlling the position and a PI controller for controlling the velocity of a brushed DC motor with two different-sized wheels which thus changes the moment of inertia.

1 Introduction

Our goal in this project is to develop a PD controller for both the small and big wheels of a brushed DC motor to control the position and a PI controller to control the velocity for only the small wheel. To achieve this various steps must be taken which involve:

1. System identification, to determine the plant transfer function for both the small and big wheels
2. Controller design, this involves designing a controller in the continuous domain and checking for our desired parameters, and then converting it from a continuous to a discrete domain, $C(s)toC(z)$, using emulation methods(Forward and Tustin) and checking for stability in the z domain.
3. We now have to take our $C(z)$ and convert it into code so that we can implement it on the Arduino DUE

The above three steps are major steps toward designing a controller for our required system, other steps involve setting up our hardware properly, and simulation of our system to check whether the controller works as intended. Especially for the third step, where we have to implement our transfer function on the Arduino, we have to make sure that we have a consistent sampling time and for this reason, we use the DUE timer which allows us to set up interrupts at our desired sampling rate. The code snippet for the DUE timer is given below with comments indicating what each register does:

```
    void setup() {
  // the quadrature encoder set up
  REG_PMC_PCER0 = PMC_PCER0_PID27;   // activate clock
  REG_TC0_CMR0 = TC_CMR_TCCLKS_XC0;  // select XC0 as clock source

  //Turn on the position measurement mode with no filters and the quadrature encoc
  REG_TC0_BMR = TC_BMR_QDEN
              | TC_BMR_POSEN
              | TC_BMR_EDGPHA;

  // enable the clock (CLKEN=1) and reset the counter (SWTRG=1)
  REG_TC0_CCR0 = TC_CCR_CLKEN | TC_CCR_SWTRG;
}

// The encoder position is then accessed from
void loop() {
  int newPosition = REG_TC0_CV0; // Read the encoder position from register
}
```

As already discussed, we are using a brushed DC motor for this experiment and before we get into all the details about deriving its plant transfer function let us discuss how a brushed DC motor works. A magnetic field and an electric current in a wire coil interact to drive a brushed DC motor. A stator and a rotor are the two primary parts of the motor.

A permanent magnet or an electromagnet is housed in the stator, a component of the motor that is stationary. On the other hand, the rotor is a revolving component made up of a wire coil and a commutator.

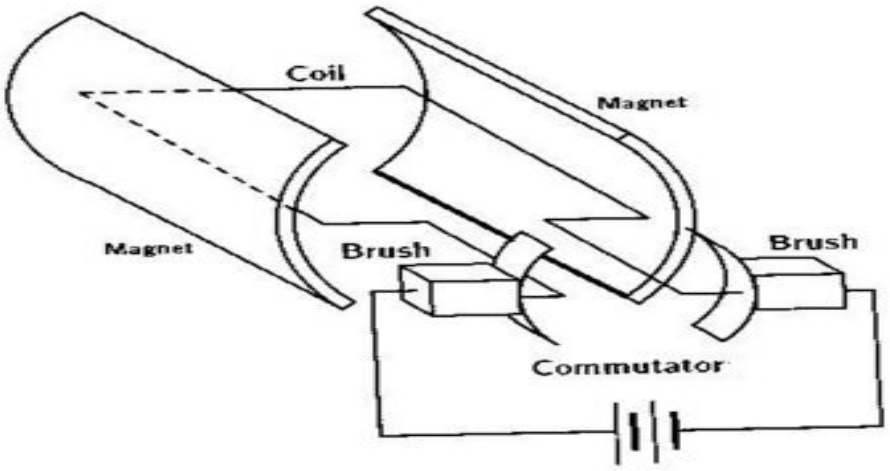
Magnetic field is produced when an electric current passes through the wire coil. The rotor rotates as a result of magnetic field interactions between the stator and rotor.

The commutator is a device that permits a particular direction of current flow through the wire coil, which in turn causes the rotor to revolve in that direction. Two metal plates mounted to the rotor and separated by an insulator make up the commutator.

The two stationary brushes that are connected to the power supply come into contact with the metal plates of the commutator when the rotor revolves. The rotor's wire coil receives energy from the brushes, creating a magnetic field that interacts with the magnetic field of the stator to keep the rotor rotating.

Every time the commutator brushes come into contact with the opposing plates, the current flowing through the wire coil is reversed. The magnetic field's direction changes as a result of this reversal of current in the coil's wire, which also alters the direction of the rotor's spin.

In general, the commutator and brushes of a brushed DC motor manage the interaction of a magnetic field and an electric current in a wire coil, which causes the rotor to rotate.

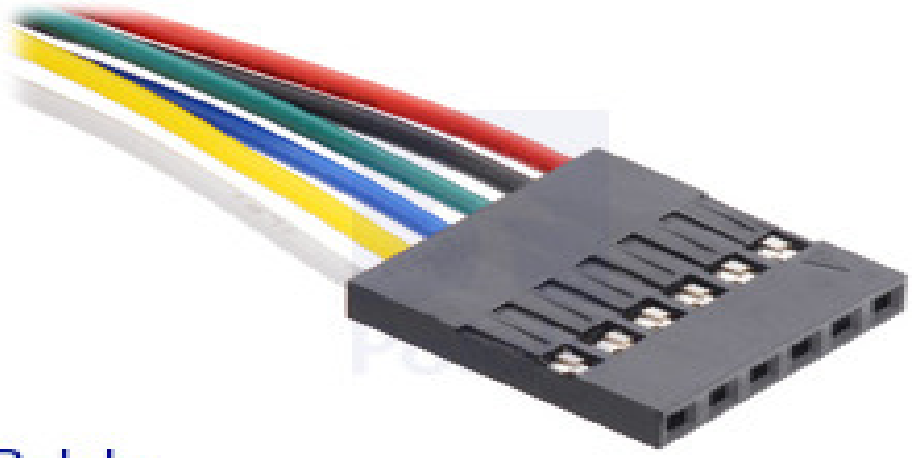


2 Lumped Parameter modeling for Brushed DC Motor

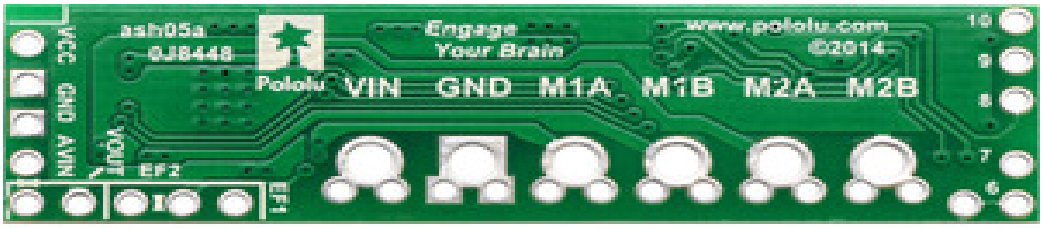
As is well known, the Transfer Function of the Motor System is represented by the equation Gw(s) = omega(s)/Vin(s) for the volts-in angular rate. Variable N is also included in the transfer function because the motor also includes a gear train. Thus, from the diagram, we can deduce that R is a resistor, L is an inductor, Vin and Vout are input and output voltage of the circuit having current I, and T is the torque applied to the external motor by the system, while damper b also has an impact on the torque of the motor. The following are a few equations derived from the motor laws

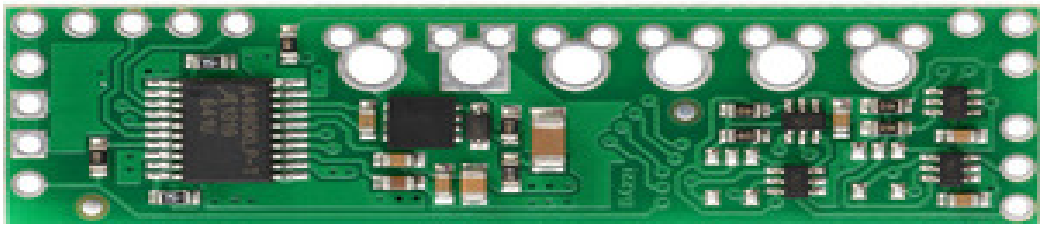
3 Setting up the hardware

To be able to conduct the experiment we must first set up our hardware. First, we have to solder the motor driver at the appropriate location as indicated in the diagram. This motor driver can drive two motors, from the data sheet of the Motor driver, we can find out the pins that should be used on the Arduino DUE for a specific motor. In our case, we are using pins 10, 8 which control the speed and direction respectively for a motor plugged into the Motor 2 channel on the motor driver. Our DC motor brushed motor consists of an Encoder which allows us to determine the position and the angular velocity of the motor, the encoder can read upto 48 counts per revolution but as the gear ratio is $4.4 : 1$, the total counts become 211.2 ($4.4 * 48$). This motor can take upto 12V and the various pins are indicated as follows:



Pololu


ash05a
0J8448
Pololu
Engage
Your Brain
www.pololu.com
©2014
VIN
GND
M1A
M1B
M2A
M2B
VCC
GND
VIN
VOUT
EP2
EP1
10
9
8
7
6



Color	Function
Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 V to 20 V)
Yellow	encoder A output
White	encoder B output

This motor driver can be very conveniently placed on our Adruino due at the appropriate pins. I am using a DC voltage supplier to provide the necessary voltage to drive the motor (12V). The encoder takes 3.3 volts from the Arduino, and the logic power for the motor driver is supplied by the 12V dc output. The Arduino can be back powered by the motor driver.

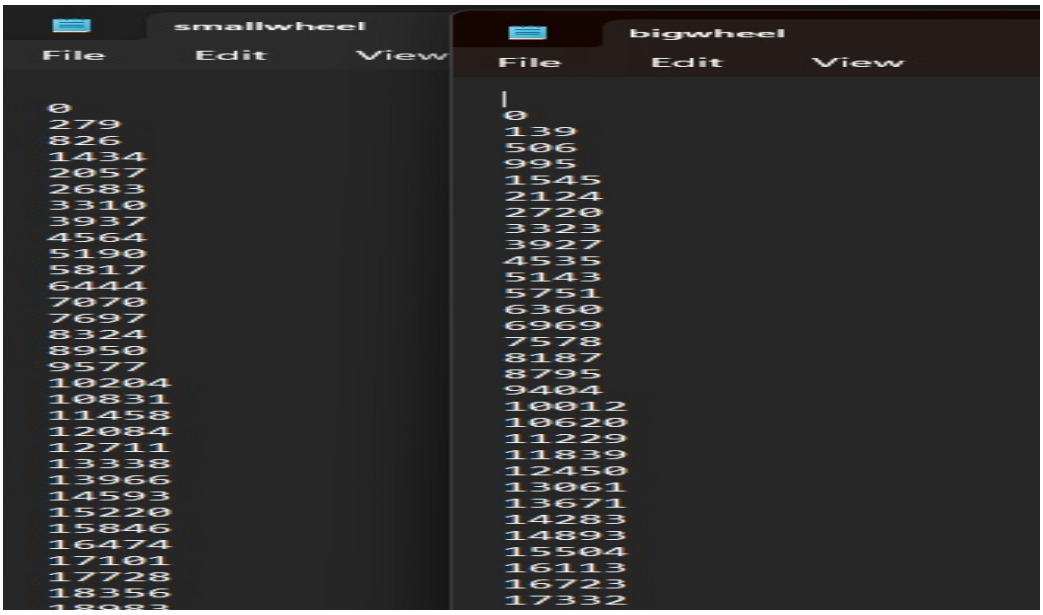
/electrical circuit diagram

4 System Identification

We have seen in section two the derivation of the plant transfer function using a lumped parameter model for our DC brushed motor, however, this model does not accurately accommodate all the nuances involved in a real mechanical system, and also the inertia of the different-sized wheels. To be able to accurately determine our plant transfer function, we need to experimentally derive it. This involves first deriving the plant transfer function for the angular velocity of the plant and from this transfer function we have to determine the transfer function for the position. We are approximating the plant transfer function to be first order because the pole for the mechanical system would be much closer to the imaginary axis than the pole of the electrical system.

To first determine the transfer function of the plant for angular velocity, we have to let the motor spin at a particular angular velocity by giving a step input while it is connected to our Arduino DUE and collect data on the

number of counts passed at every sampling period from the serial monitor and from this data we can determine the angular velocity. The sampling time I used for the collection of the encoder data was 1 ms.



After the collection of this data, we need to find the angular velocity and have to find an appropriate transfer function that could fit the curve from our collected data. Our transfer function is given as follows:

$$G(s) = K * \frac{\sigma}{s + \sigma}$$

This is the transfer function for our plant for angular velocity, where K is the value of the steady state divided by the applied voltage and sigma is the rise time for our plant.

$$K = S.S/V_a$$

The inverse laplace transform is given by:

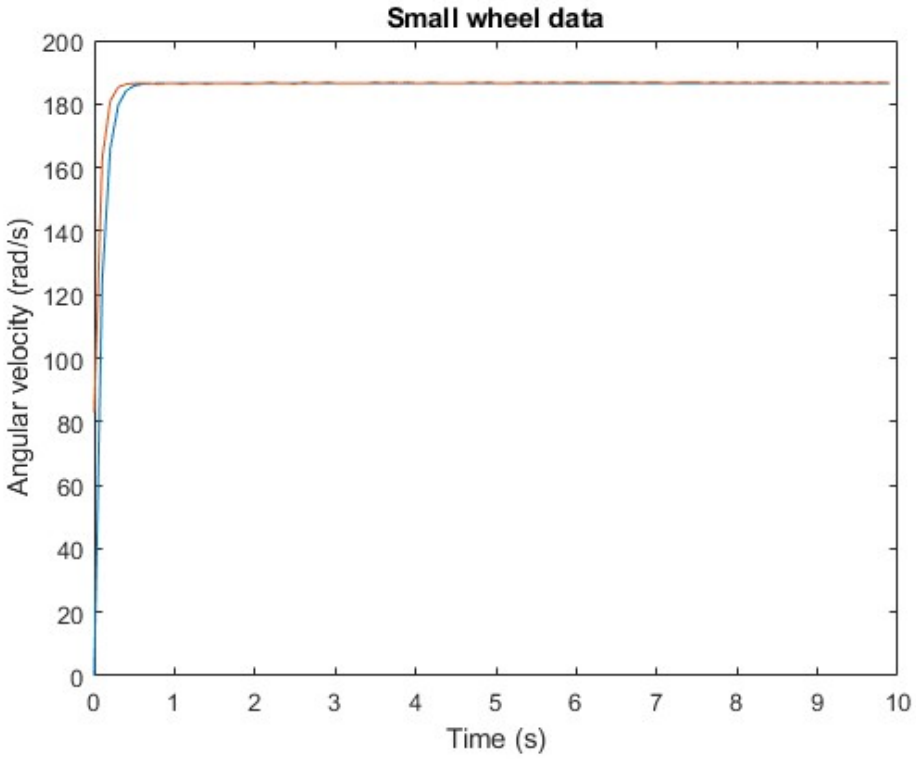
$$g(t) = V_a * K * (1 - \exp(-\sigma * t))$$

To be able to check whether the predicted transfer function works, I took the inverse Laplace transform and compared it with the data we collected by plotting it. For this, I used the following code in MATLAB as shown below: Matlab code for small wheel:

```
clear;
close all;
clc;

% Read position data from file
j = readmatrix('smallwheel.txt');

% Define system parameters
N = 4.4;                    % Gear ratio
counts = 48;        % Encoder resolution
T_s = 0.1;                  % Sampling time

% Convert position data to angular velocity
rad = 2*pi*j/(N*counts);   % Convert position to radians
omega = diff(rad)/T_s;              % Calculate angular velocity

% Create time vector
t = (0:length(omega)-1)*T_s;

% Plot angular velocity data
plot(t, omega);
xlabel('Time (s)');
ylabel('Angular velocity (rad/s)');
title('Small wheel data');
```

```
v_app = 12;
k_hat = ss/v_app;

r =0.9*(omega(100));
p = 0.1*(omega(100));

for i = 1:numel(omega)
    if omega(i) > p
        ti = t(i);
        break
    end
end

for i = 1:numel(omega)
    if omega(i) > r
        tq = t(i);
        break
    end
end

tr = tq - ti;
sig = 2.2/tr;

y = v_app*k_hat*(1-exp(-sig*t));

figure(1)
plot(t, y)
hold on
plot(t, omega)
hold off

G_w_small = k_hat*(sig/(s + sig));

figure(2)
step(G_w_small)
```

The plots comparing the step response for the expermental result and the simulated result is shown below:

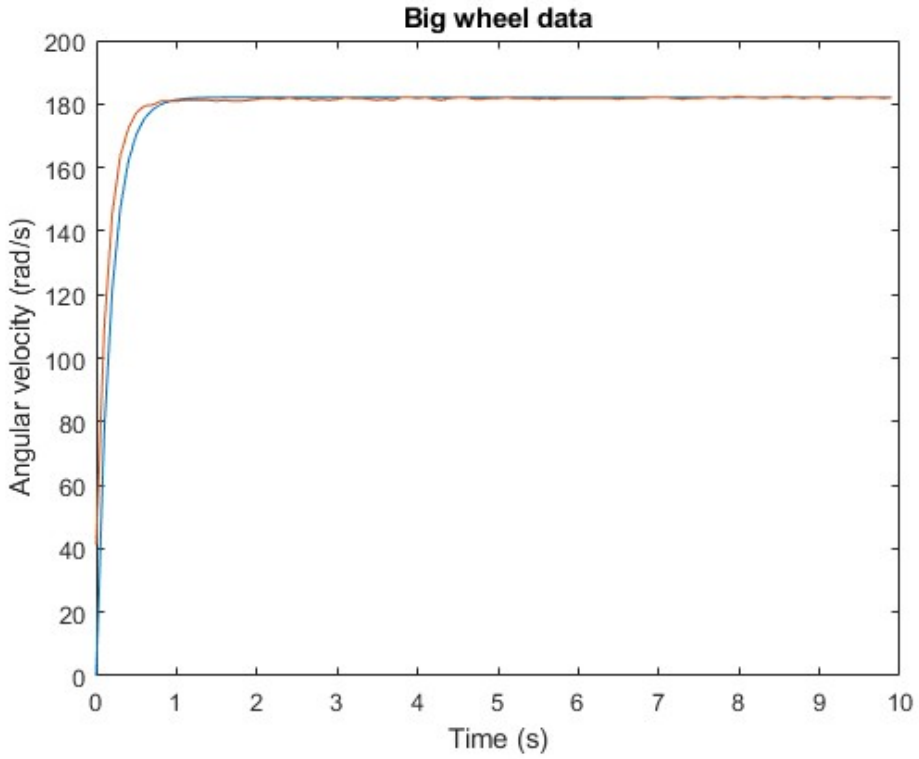


Matlab code for Big wheel:

```
1
2   clear;
3   close all;
4   clc;
5
6   % Read position data from file
7   j = readmatrix('bigwheel.txt');
8
9   % Define system parameters
10  N = 4.4;                    % Gear ratio
11  counts = 48;        % Encoder resolution
12  T_s = 0.1;                  % Sampling time
13
14  % Convert position data to angular velocity
15  rad = 2*pi*j/(N*counts);   % Convert position to radians
16  omega = diff(rad)/T_s;              % Calculate angular velocity
17
18  % Create time vector
19  t = (0:length(omega)-1)*T_s;
20
21  % Plot angular velocity data
22  plot(t, omega);
23  xlabel('Time (s)');
24  ylabel('Angular velocity (rad/s)');
25  title('Big wheel data');
```

```
omega = omega(1:100);
ss = omega(100);
plot(t, omega);


v_app = 12;
k_hat = ss/v_app;

r =0.9*(omega(100));
p = 0.1*(omega(100));

for i = 1:numel(omega)
    if omega(i) > p
        ti = t(i);
        break
    end
end

for i = 1:numel(omega)
    if omega(i) > r
        tq = t(i);
        break
    end
end

tr = tq - ti;
sig = 2.2/tr;

y = v_app*k_hat*(1-exp(-sig*t));

figure(1)
plot(t, y)
hold on
plot(t, omega)
xlabel('Time (s)');
ylabel('Angular velocity (rad/s)');
title('Small wheel data');

G_w_big = k_hat*(sig/(s + sig));

figure(2)
step(G_w_big)
```

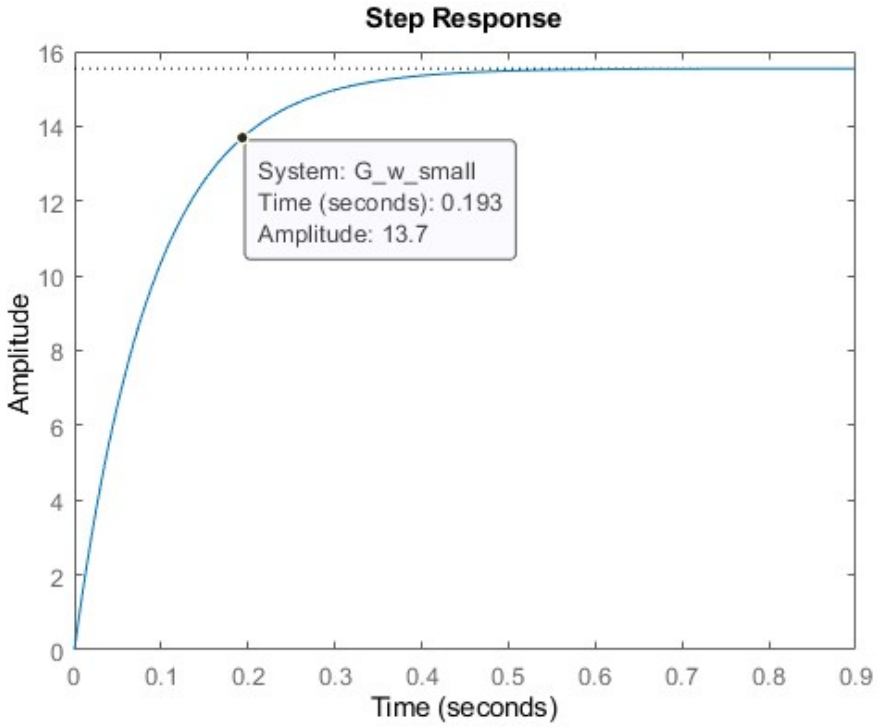
The plots comparing the step response for the expermental result and the simulated result is shown below:



The transfer function that I got for both the small and big wheels is given as:

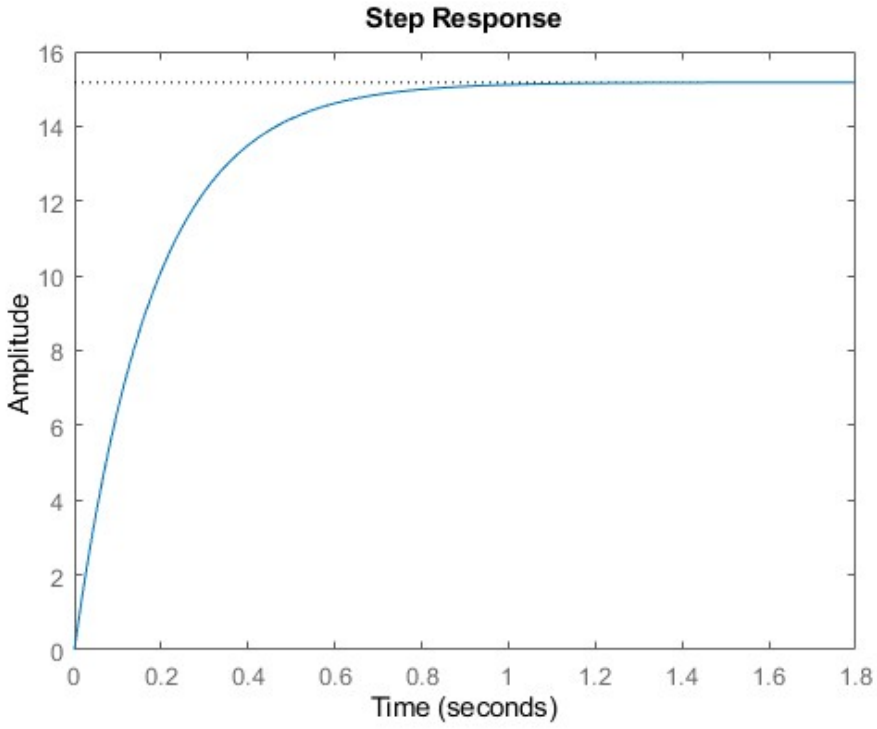
$$G(s)_s = \frac{171}{s+11}$$

The step response for this transfer function is:



$$G(s)_b = \frac{83.45}{s + 5.5}$$

The step response for this transfer function is:



5 Designing controller in the continuous domain for position

Now that we have the plant transfer function for the angular velocity, we need to now find a way to convert this into the transfer function that describes the transfer function for position. We can derive it as follows:

$$\omega = \dot{x}$$

The angular velocity is the derivative of the position. Now taking the laplace transform on both sides we get:

$$\omega(s) = s * X(s)$$

$$X(s) = \frac{\omega(s)}{s}$$

So, the transfer function for the plant for position for both the wheels is given by:

$$G(s)_s = \frac{171}{s^2 + 11 * s}$$

$$G(s)_b = \frac{83.45}{s^2 + 5.5 * s}$$

Now that we have the required transfer function, let us design an appropriate controller to control the position. Our requirement is to design a PD controller or a Proportional-Derivative controller which is given by:

$$C(s) = K_p + K_d * s$$

But this controller is not realizable, as a transfer function with only a zero can not be implemented, so we have to approximate this transfer function by placing the pole at the far left position so that it has minimal influence on our PD controller. The modified transfer function is given by:

$$C(s) = \frac{K_p + K_d * s}{\tau * s + 1}$$

where τ is given by:

$$\tau = 10 * \frac{K_d}{K_p}$$

As there are no particular system requirements while designing this controller, we can choose the values of K_p and K_d to make a suitable controller that can respond quickly enough and has a low steady-state error. Although, the steady state error would be hard to remove given that this is a PD controller and not a PI controller where the integrator term would eliminate the steady-state error.

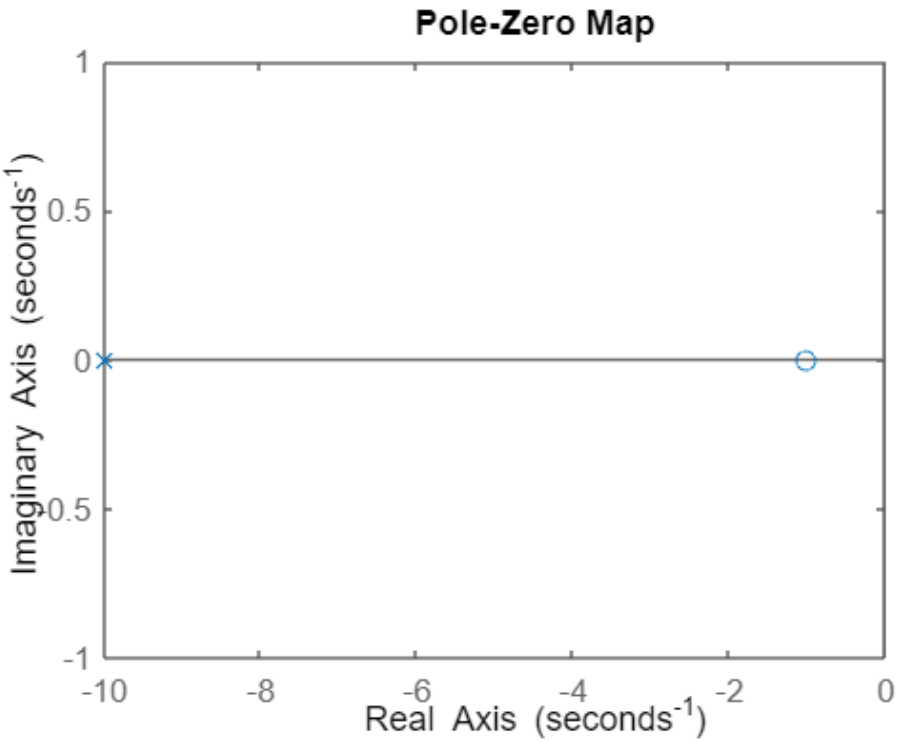
5.1 small wheel

Kp = 1 and Kd =10

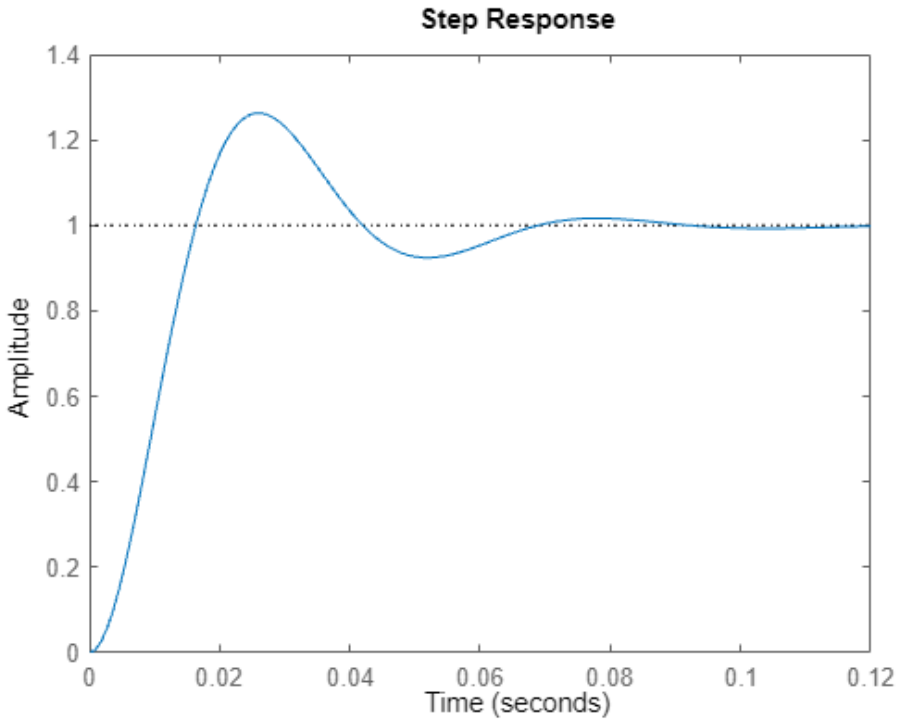
$$C(s) = \frac{s + 10}{0.01 * s + 1}$$

The values of K_p and K_d that I have chosen and also the code and response time of this controller with the plant for the small wheel are given by: /insert matlab code and plots

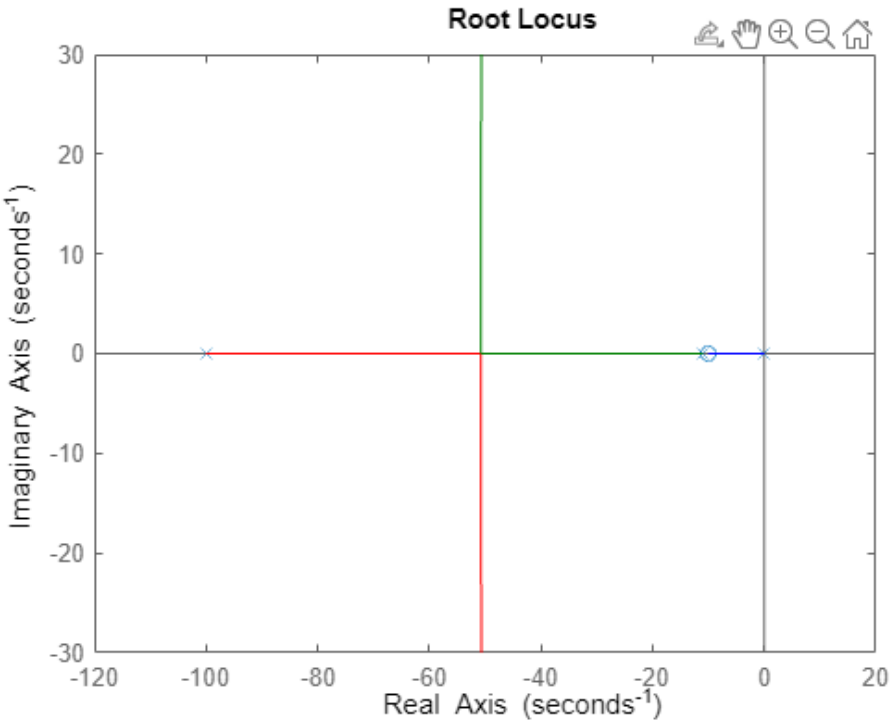
```
% PD controller
t1 = 0.01
s = tf('s')
Kd = 1;
Kp = 10;
Tau = 1/((Kp/Kd)*10)

C_s = (Kp + Kd*s)/(s*Tau +1);
figure;
pzmap(C_s)

G_w_small
G_s = G_w_small/s
fl = C_s*G_s
step(feedback(fl,1))
rlocus(fl)
```



Pole-Zero map for controller



Feedback step response for C(s)



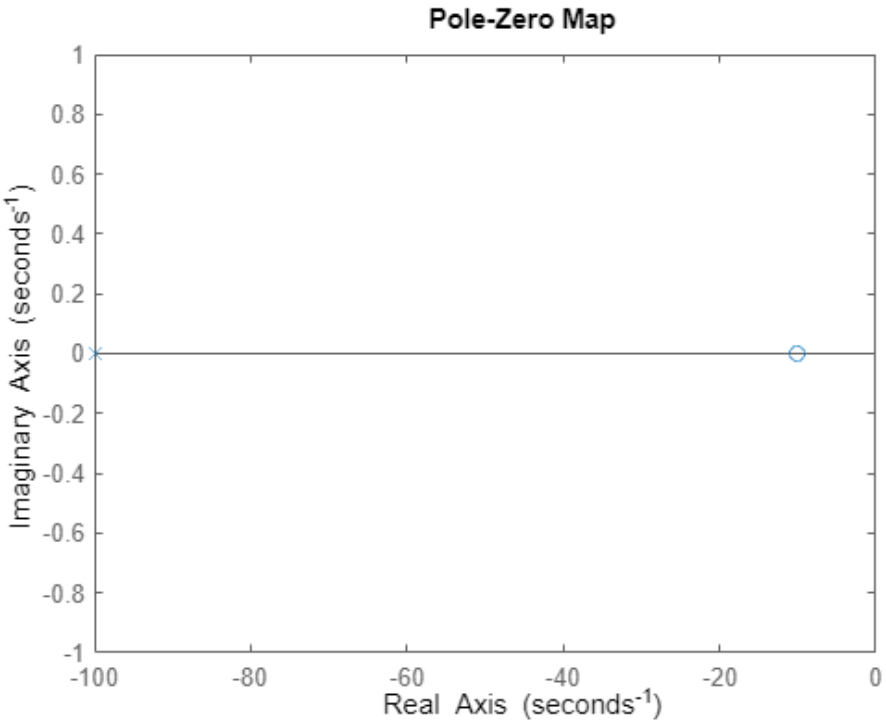
Root locus for T(s)

5.2 Big Wheel

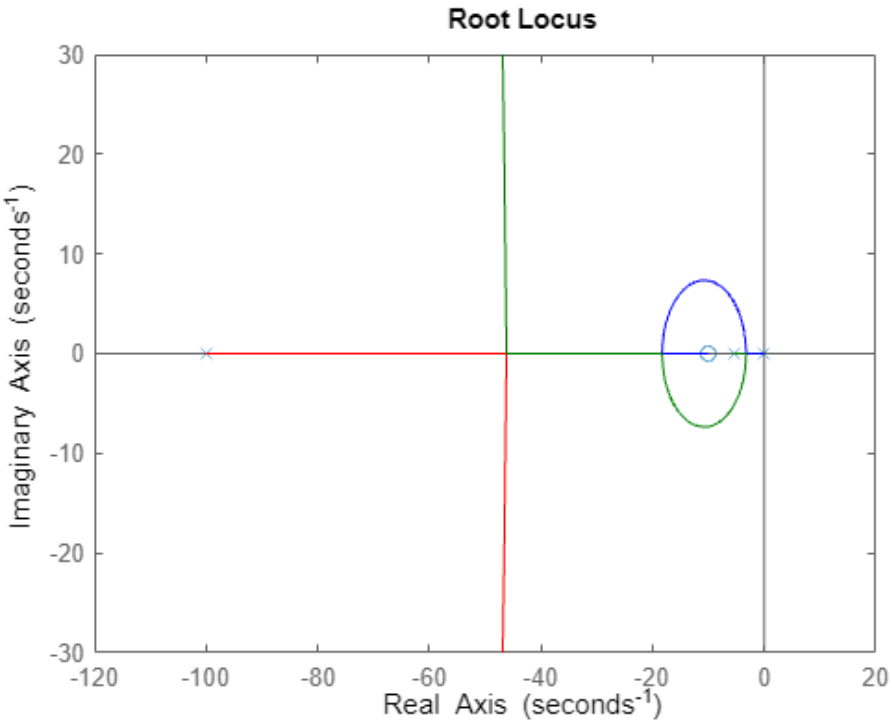
Now for the big wheel, the Matlab code contains the Kp and Kd values, the values along with the code and plots given below: The Kp = 0.1 and Kd = 1

$$C(s) = \frac{0.1 * s + 1}{0.01 * s + 1}$$

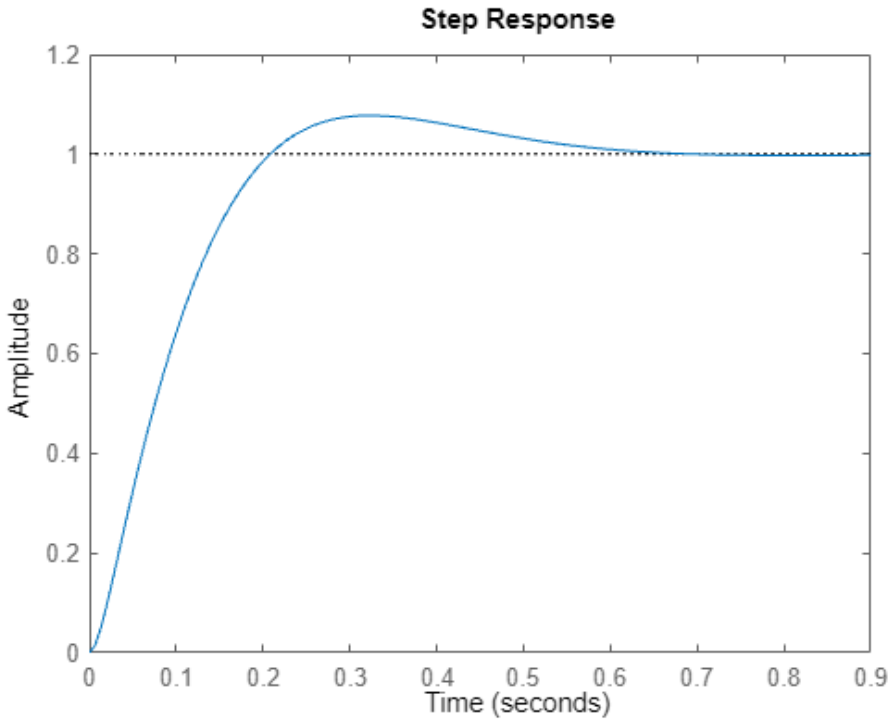
```
% PD controller
t1 = 0.01
s = tf('s')
Kd = .1;
Kp = 1;
Tau = 1/((Kp/Kd)*10)

C_s = (Kp + Kd*s)/(s*Tau +1);
C_s
figure;
pzmap(C_s)

G_w_big
G_s = G_w_big/s
fl = C_s*G_s
step(feedback(fl,1))
rlocus(fl)
```



The pole zero map for C(s)



The root locus for $C(s) * G(s)$



6 Designing controller in the continuous domain for Angular velocity

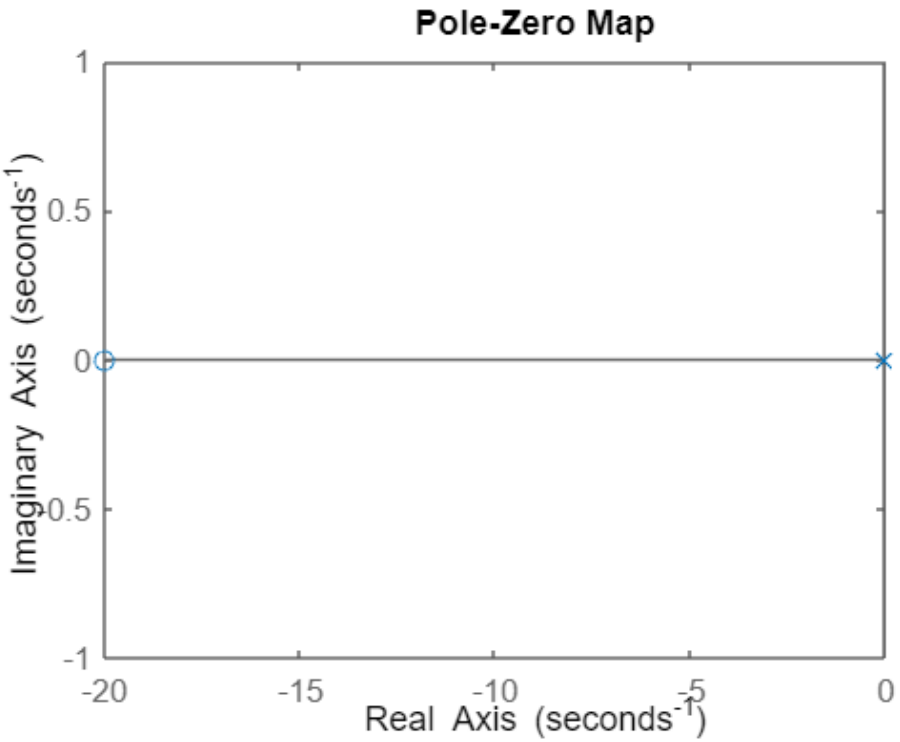
Now, for the velocity control, we have to design a PI controller. The transfer function for a PI controller is given by:

$$C(s) = K_p + \frac{K_i}{s}$$

Through various experimental runs the values for K_p and K_i are 10 and 0.5 respectively.

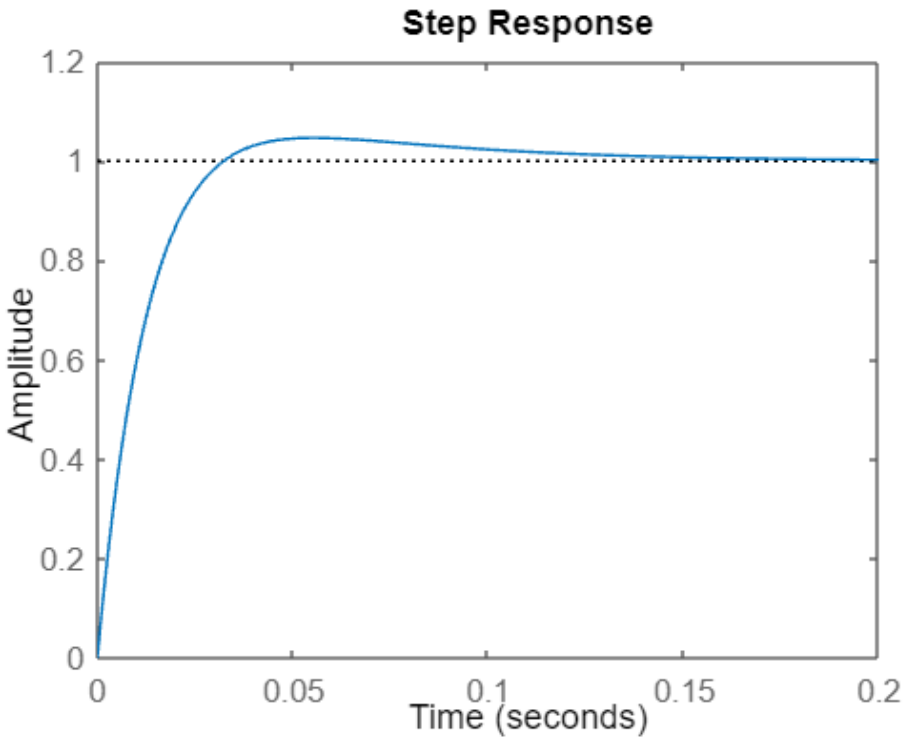
$$C(s) = \frac{0.5 * s + 10}{s}$$

```
% PI controller
t1 = 0.01
s = tf('s')
Ki = 10;
Kp = 0.5;


C_s = Kp * ((s + (Ki/Kp))/s)
%C_s = (s+1)/(s+10)
figure;
pzmap(C_s)
%rlocus(C_s)
G_0_s = G_w_small
step(G_w_small)
fl = C_s*G_0_s
step(feedback(fl,1))
step(feedback(G_0_s, 1))

```



Pole zero map for C(s)

The step response for this controller with the plant is given as:



Thus, I will go forward using this controller.

7 Emulation and various emulation methods

Emulation is the process of converting the our transfer function from the continuous domain to the discrete domain using various approximation methods. Three methods include the Euler forward method, Euler backward method and Tustin method, of all three Tustin's approximation is the most popular method as we will see why.

Let us derive the Euler's forward method:

The Euler's forwar method is dgiven by:

$$U(k+1) = U(k) + e(k) * T$$

Now, applying the z-transform on both sides we get:

$$z * U(z) = U(z) + E(z) * T$$

$$C(z) = \frac{U(z)}{E(z)} = \frac{T}{z-1}$$

The derivative in the laplace domain is given by s. Now, equating the laplace domain equivilant to the z domain equivilant gives us:

$$C(s) = \frac{U(s)}{E(s)} = \frac{1}{s}$$

$$s = \frac{z-1}{T}$$

The Tustin's approximation is given by:

$$S = \frac{2}{T} * \frac{z-1}{z+1}$$

8 Controller in the discrete domain for position

We have to add a zero-order hold in front of G(s) to make the discrete signal continuous as G(s) is continuous. We can use the c2d function in Matlab to acheive this.

8.1 Small Wheel

Using both these methods let us convert our continuous time controller transfer function to a dicrete time transfer function using sampling time as 0.01 seconds. Using the approxiamtion for s by euler's forward method you get:

$$s = \frac{z-1}{T}$$

$$C(s) = \frac{s+10}{0.01 * s + 1}$$

Substitute s using Euler's forward method and we get:

$$C(z) = \frac{100 * z - 90}{z}$$

For Tustin's method we can use the MATLAB c2d function. The Matlab code to convert the continuous time to discrete time is given as:

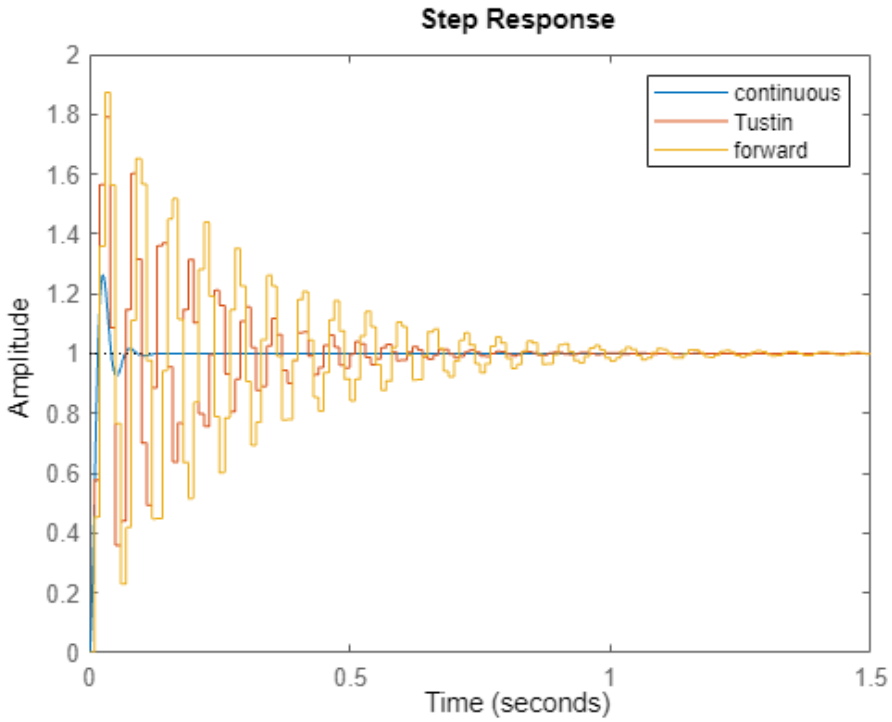
$$C(z) = \frac{70 * z - 63.33}{z - 0.333}$$

The Matlab code to convert into a discrete domain is:

```
% Conversion to z domain
C_Z_forward = c2d_euler(C_s,t1,'forward')
C_Z_tustin = c2d(C_s, t1, 'tustin')
G_Z = c2d(G_s,t1,'zoh')

T_Z_tustin = C_Z_tustin * G_Z
T_Z_forward = C_Z_forward * G_Z

step(feedback(fl,1))
hold on
step(feedback(T_Z_tustin, 1))
step(feedback(T_Z_forward, 1))


legend('continuous', 'Tustin', 'forward')
```

The response time of this controller is quick as it is a high-gain controller and as a result, the steady-state error is nearly zero and has a lot of oscillations. I will continue with this controller. You can see the difference between Euler and Tustin, in Tustin's method the oscillations dampen out much more quickly than Euler's method.

Let us now compare the continuous time step response with the discrete time step response for both methods,



As you can see, using the forward method approximation and Tustin's method seems stable. Euler's forward method does not map the continuous time to discrete time exactly and the controller may go unstable in the discrete-time unlike in our case. But Tustin's method does approximate from the continuous time to the discrete-time well as seen by our plot.

8.2 Big Wheel

Using both these methods let us convert our continuous time controller transfer function to a dicrete time transfer function using sampling time as 0.01 seconds. Using the approxiamtion for s by euler's forward method you get:

$$s = \frac{z-1}{T}$$

$$C(s) = \frac{0.1 * s + 1}{0.01 * s + 1}$$

Substitute s using Euler's forward method and we get:

$$C(z) = \frac{10 * z - 9}{z}$$

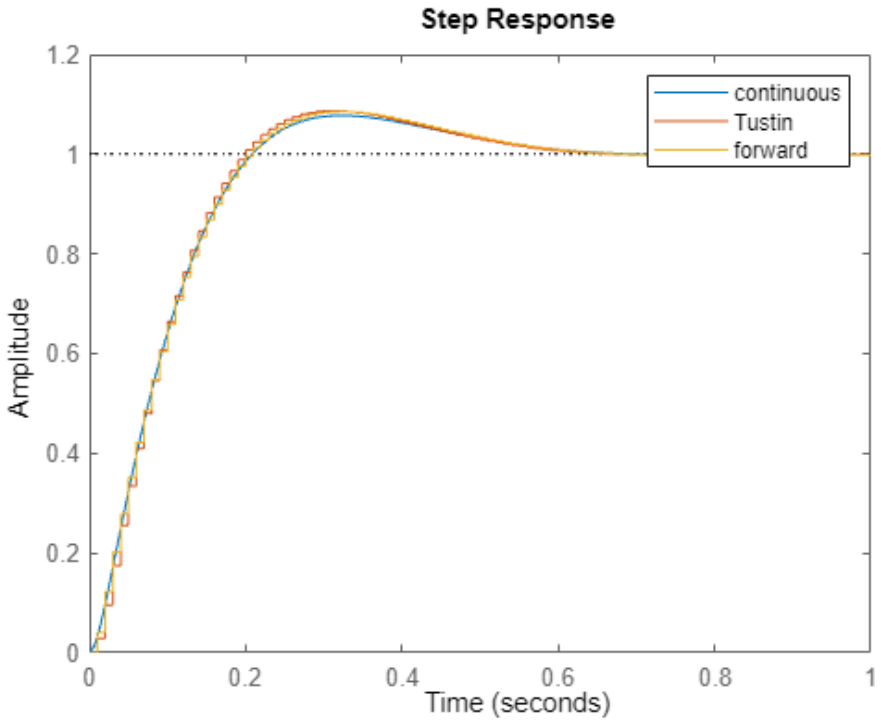
For Tustin's method we can use the MATLAB c2d function. The Matlab code to convert the continuous time to discrete time is given as:

$$C(z) = \frac{7 * z - 6.33}{z - 0.333}$$

The Matlab code to convert into a discrete domain is:

```
% Conversion to z domain
C_Z_forward = c2d_euler(C_s,t1,'forward')
C_Z_tustin = c2d(C_s, t1, 'tustin')
G_Z = c2d(G_s,t1,'zoh')

T_Z_tustin = C_Z_tustin * G_Z
T_Z_forward = C_Z_forward * G_Z

step(feedback(fl,1))
hold on
step(feedback(T_Z_tustin, 1))
step(feedback(T_Z_forward, 1))


legend('continuous', 'Tustin', 'forward')
```

The response time of this controller is quick as it is a high-gain controller and as a result, the steady-state error is nearly zero and the oscillations are are also minimal. The forward and tustin's methods both seem to be working similarly.



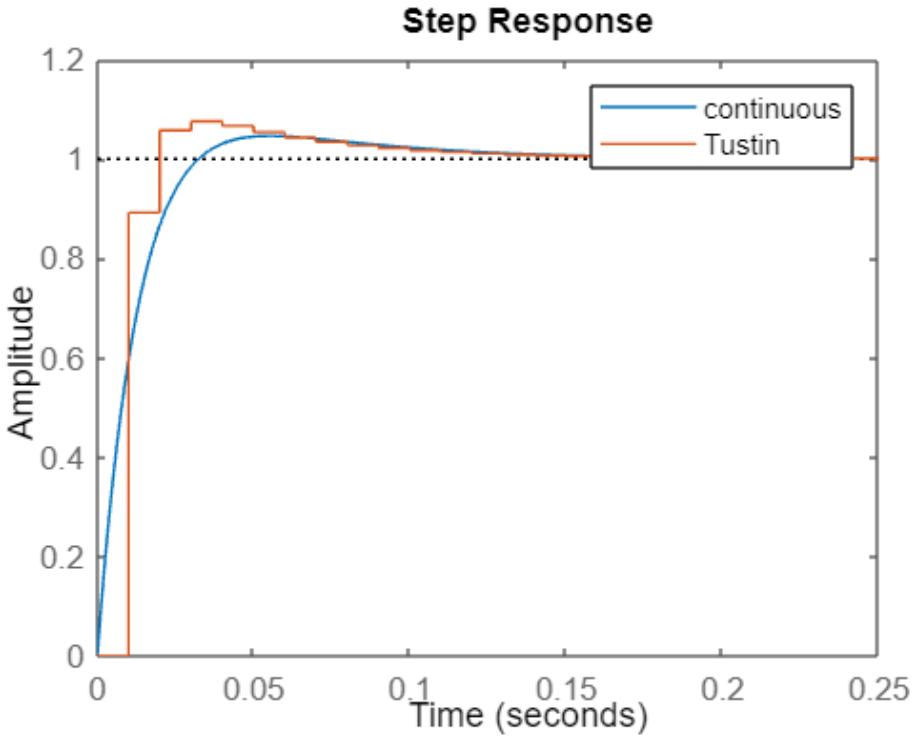
9 Controller in the discrete domain for Angular velocity

Using both these methods let us convert our continuous time controller transfer function to a dicrete time transfer function. For Tustin's method we can use the MATLAB c2d function. The Matlab code to convert the continuous time to discrete time is given as:

```
% Conversion to z domain
C_Z_tustin = c2d(C_s, t1, 'tustin')
G_Z = c2d(G_0_s,t1,'zoh')

T_Z_tustin = C_Z_tustin * G_Z

step(feedback(fl,1))
hold on
step(feedback(T_Z_tustin, 1))


legend('continuous', 'Tustin')
```

$$C(z) = \frac{0.55 * z - 0.45}{z - 1}$$

The step response using this discrete transfer function is given as:



10 Arduino Implementation

To be able to implement this on our arduino, we need to convert our C(z) transfer function into code which can be done as follows: Let our transfer function be:

$$C(z) = \frac{a * Z - b}{z - c} = \frac{U(z}{E(z)}$$

Now, transposing U(z) and E(z) we get:

$$a * z * E(z) - b * E(z) = U(z) * z - c * U(z)$$

Applying the inverse Z transform:

$$a * e(k+1) - b * e(k) = u(k+1) - c * u(k)$$

Putting $k = k + 1$, we get:

$$u(k) = c * u(k-1) + a * e(k) - b * e(k-1)$$

The above equation is the code that will control our plant to give the desired output. The required values of a, b, and c are determined from our C(z) function. The Arduino code is given below: /insert arduino code

In this code we are mapping our U(k) from 0 to 12 volts and also the full scale of the PWM signal which is 4095.

10.1 Position control

10.1.1 small wheel

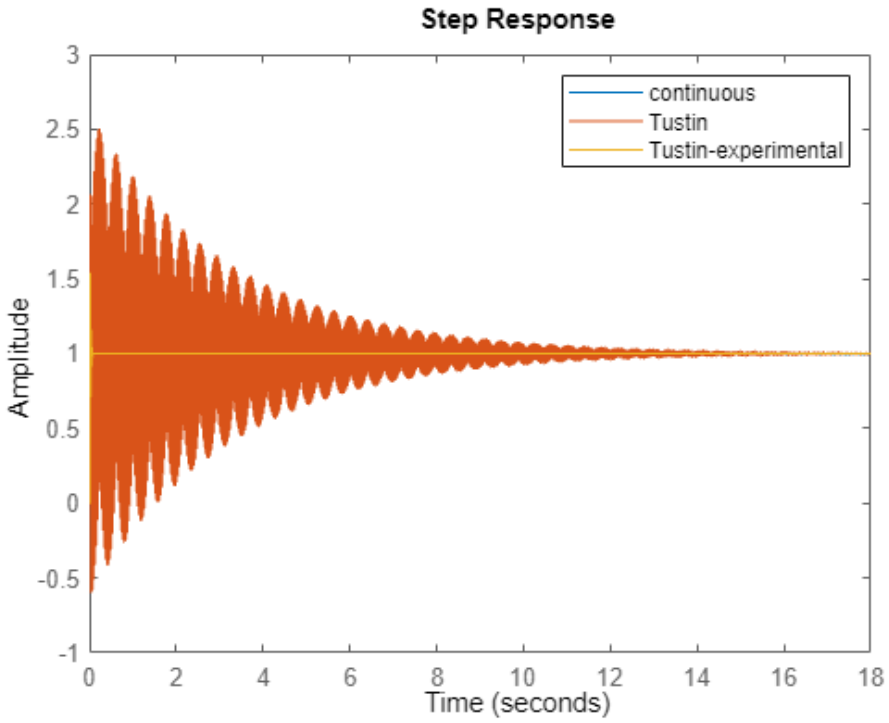
The Arduino code for implementing the position control is given by:

```
  #include "DueTimer.h"

#define PI 3.142

int count_q_p = 0;
float U = 0;
float E_p = 0;
float E = 0;
bool dir = LOW;
int pwm = 0;
float a = 70;
float b = 63.33;
float c = 0.333;
float reference = 2* PI;
int count_q_n = 0;
float f1 = 0;


void setup() {

  Serial.begin(115200); //115200 or 9600
  analogWriteResolution(12);    // set DAC output to maximum 12-bits
  Timer3.attachInterrupt(update).start(1000); // start ISR timer3 (not used by qua

  // 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;

  pinMode(10, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(DAC1, OUTPUT);

}


void loop() {

}


void update() {
  count_q_p = REG_TC0_CV0;

  f1 = (2*PI*count_q_p)/(211.2);
  E_p = E;
  E = reference - f1; //radians
  U = c*U + a*E + b*E_p;

  if(U < 0)
  {
    dir = LOW;
    if (U <-12){
      U= -12;
    }
    pwm = 4095*(-1*U/12);
  }
```

```
    if(U > 0)
  {
    dir = HIGH;
    if (U>12){
      U= 12;
    }
    pwm = 4095*(1*U/12);
  }

// control action is implemented here

  digitalWrite(8,dir);

  analogWrite(10,pwm);

  analogWrite(DAC1, feedback_vel);

  Serial.println(feedback_vel);


}
```

Plot for small wheel



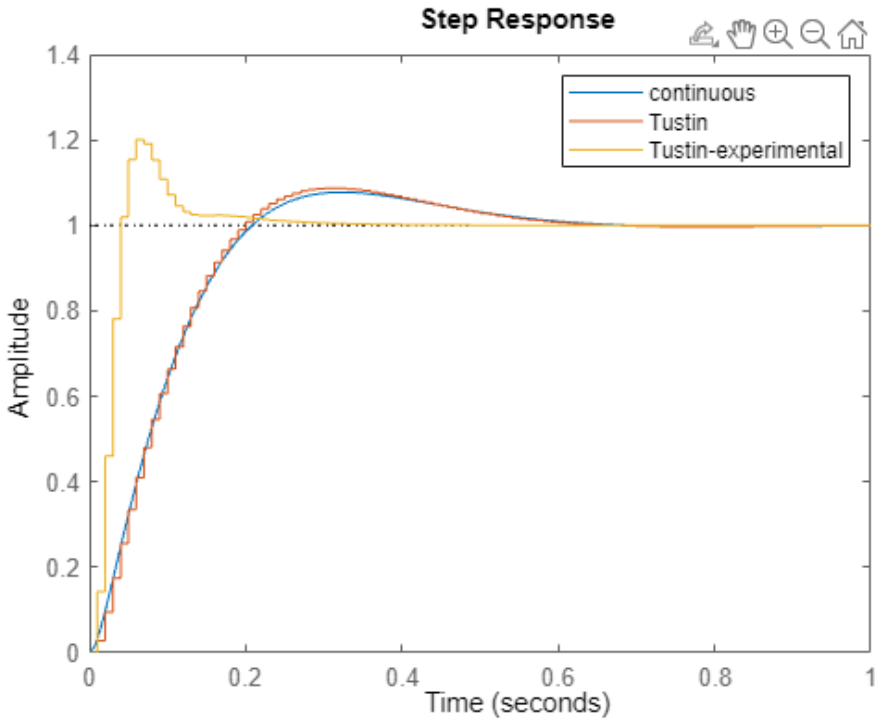
10.1.2 bigwheel

```
    #include "DueTimer.h"

#define PI 3.142

int count_q_p = 0;
float U = 0;
float E_p = 0;
float E = 0;
bool dir = LOW;
int pwm = 0;
float a = 7;
float b = 6.33;
float c = 0.333;
float reference = 2* PI;
int count_q_n = 0;
```

```
float f1 = 0;


void setup() {

  Serial.begin(115200); //115200 or 9600
  analogWriteResolution(12);    // set DAC output to maximum 12-bits
  Timer3.attachInterrupt(update).start(1000); // start ISR timer3 (not used by qua

  // 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;

  pinMode(10, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(DAC1, OUTPUT);

}


void loop() {

}


void update() {
  count_q_p = REG_TC0_CV0;

  f1 = (2*PI*count_q_p)/(211.2);
```

```
  E_p = E;
  E = reference - f1; //radians
  U = c*U + a*E + b*E_p;

  if(U < 0)
  {
    dir = LOW;
    if (U <-12){
      U= -12;
    }
    pwm = 4095*(-1*U/12);
  }
    if(U > 0)
  {
    dir = HIGH;
    if (U>12){
      U= 12;
    }
    pwm = 4095*(1*U/12);
  }

// control action is implemented here

  digitalWrite(8,dir);

  analogWrite(10,pwm);

  analogWrite(DAC1, feedback_vel);

  Serial.println(feedback_vel);


}
```

Plot for big wheel



10.2 Speed control

The Arduino code for implementing the velocity control is given by: TThe response to a 100 rad/sec input can be seen in the serial monitor which seems to provide a pretty accurate response given that it is a PI controller.

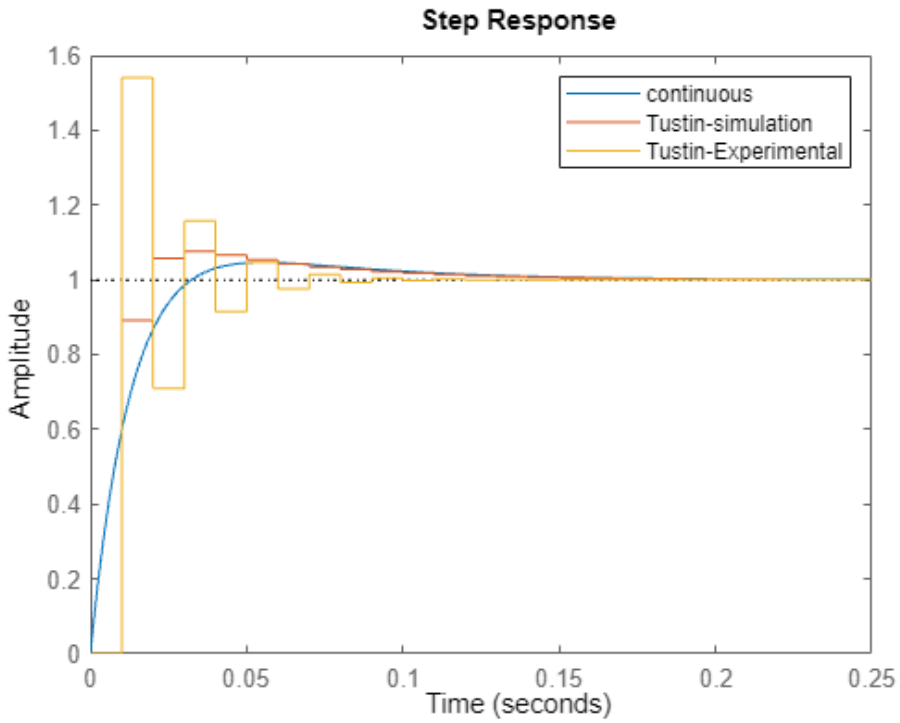
```
    #include "DueTimer.h"

#define PI 3.142

int count_q_p = 0;
float U = 0;
float E_p = 0;
float E = 0;
bool dir = LOW;
int pwm = 0;
float a = 0.55;
```

```
float b = 0.45;
float c = 1;
float reference = 100;
float feedback_vel  = 0;
int count_q_n = 0;
float f1 = 0;
float f2 = 0;

void setup() {

  Serial.begin(115200); //115200 or 9600
  analogWriteResolution(12);    // set DAC output to maximum 12-bits
  Timer3.attachInterrupt(update).start(1000); // start ISR timer3 (not used by qua

  // 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;

  pinMode(10, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(DAC1, OUTPUT);

}


void loop() {

}
```

```
void update() {
  count_q_n = count_q_p;
  count_q_p = REG_TC0_CV0;

  f1 = (2*PI*count_q_n)/(211.2);
  f2 = (2*PI*count_q_p)/(211.2);
  feedback_vel = (f2 - f1)/0.01;

  E_p = E;
  E = reference - feedback_vel; //radians
  U = c*U + a*E + b*E_p;

  if(U < 0)
  {
    dir = LOW;
    if (U <-12){
      U= -12;
    }
    pwm = 4095*(-1*U/12);
  }
    if(U > 0)
  {
    dir = HIGH;
    if (U>12){
      U= 12;
    }
    pwm = 4095*(1*U/12);
  }

// control action is implemented here

  digitalWrite(8,dir);

  analogWrite(10,pwm);

  analogWrite(DAC1, feedback_vel);
```

```
  Serial.println(feedback_vel);

}
```

The plots from the oscilloscope for angular velocity control versus the simulated response are given:



From the plots it can be understood can be seen that the simulation and the experimental plots are similar which tells us that the transfer function approximation for our plant is pretty accurate.