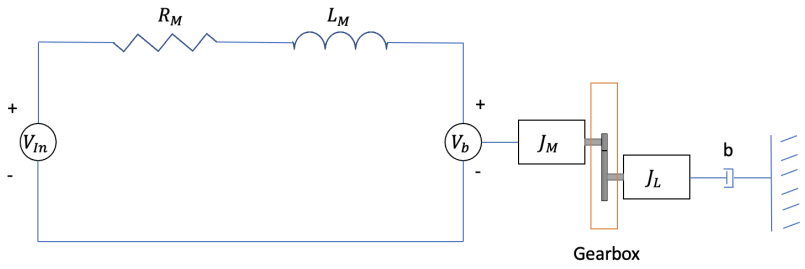
Industrial Automation

Mechanical Modeling, Analysis, and Implementation of a Position and Speed Controller for Brushed DC Motor Systems

By: Erid Pineda

May 9th, 2023



Contents

List of Figures

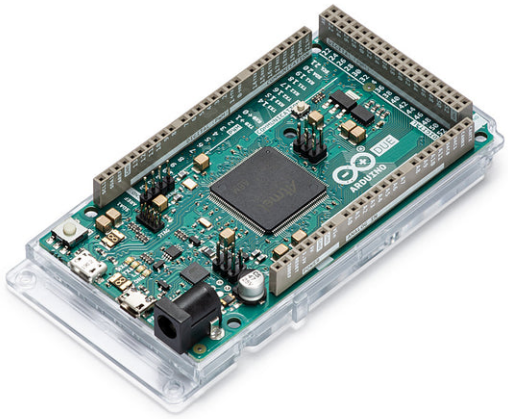
List of Tables

1 Introduction

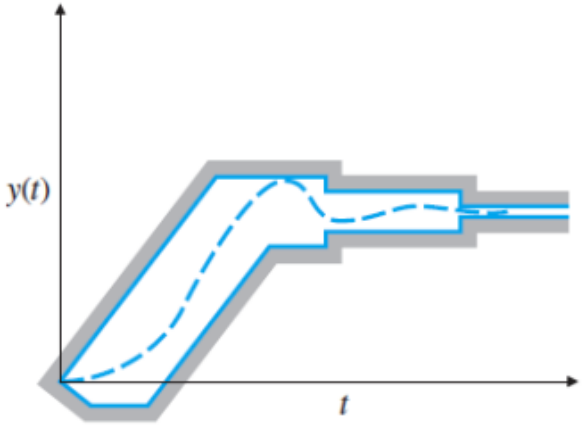
When a new project comes up and is in need of a controller there are general steps that must be taken to accomplish the project. Those steps are generally the major steps in control system design includes the following:

1. Understand the process and translate dynamic performance requirements into time, frequency, or pole-zero specifications.
2. Select sensors
3. Select actuators
4. Construct a linear model
5. Try simple controller (P, PID, Compensator, etc)
6. Evaluate/modify plant
7. Optimize controller (or more advanced controller)
8. Build computer mode and compute (simulate) the performance of the design
9. Build a prototype and evaluate

Throughout this course we have learned the necessary tools for industrial implementation of control systems and with existing knowledge of feedback control one can now go through the full design cycle. In this course steps 8 and 9 were learned which were composed of discrete-time signal processing, sampling theory, continuous time controller emulations, limitations of implementing continuous time controllers in discrete time and therefore on embedded systems.



(a) Micro-controller - Due



(b) Figures 10.1 (c) in FPE 8th edition

For this project most of the steps will be highlighted and implemented while building a position and speed controller for two systems: Brushed DC motor with low moment of inertia (SMOI) and with larger moment of inertia (LMOI).

1.1 Hardware selection

The hardware has been selected for this project and is a constant. This would be steps 2 and 3, following the general steps of control system design. The following hardware will be modeled and used to implement a controller.

Table 1: Hardware list

Hardware	Source	Purpose
Arduino Due	Arduino USA	Micro-controller will be used to implement controller
Motor Driver	Pololu	Driver will be used to control brushless DC motors
Motor	Pololu	Brushless DC motor with an encoder will be used to drive the fly wheel
Wheel Hub	Pololu	The flyhweel (smaller and larger diameters)

1.2 Arduino Due Specifications

The Arduino Due is based on a 32-bit ARM core micro-controller with a Atmel SAM3X8E ARM-Cortex-M3 CPU. The most telling specification on the data sheet of this micro-controller is this CPU's clock speed. This is an important specification because as it governs the speed at which the process executes instructions. This processor has an 84MHz clock (max).

TECH SPECS

Microcontroller	AT91SAM3X8E
Operating Voltage	3.3V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-16V
Digital I/O Pins	54 (of which 12 provide PWM output)
Analog Input Pins	12
Analog Output Pins	2 (DAC)
Total DC Output Current on all I/O lines	130 mA
DC Current for 3.3V Pin	800 mA
DC Current for 5V Pin	800 mA
Flash Memory	512 KB all available for the user applications
SRAM	96 KB (two banks: 64KB and 32KB)
Clock Speed	84 MHz
Length	101.52 mm
Width	53.3 mm
Weight	36 g

Figure 2: Arduino Due Micro-Controller Tech Specifications

2 Model Derivation

The eleco-mechanical model below characterizes the motor from pololu found in table 1.

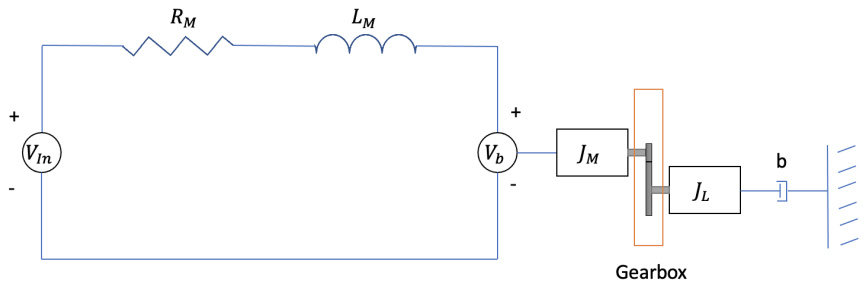


Figure 3: Motor Model

As shown in the figure this motor has a gear box integrated within this particular package. To derive the

transfer function from V_{in} to θ, this model can be broken up into its electrical and mechanical components. The assumptions made in this lumped-mass derivation are found below.

- Motor is linear
- Shaft is rigid
- No torsional spring effects present
- Shaft has not mass

2.0.1 Electrical Model

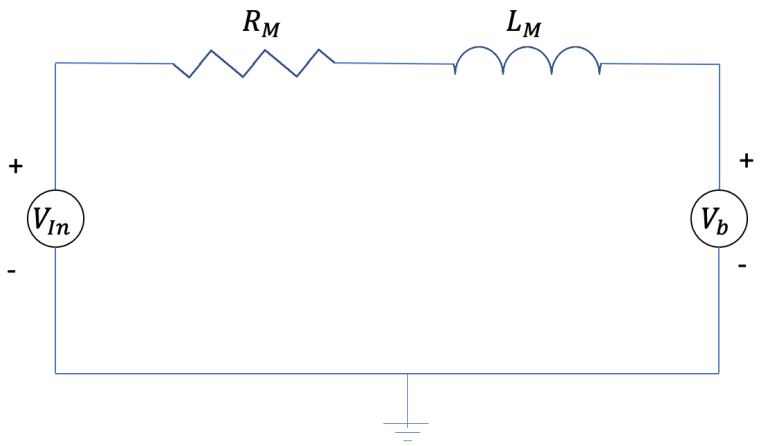


Figure 4: Motor Model - Electrical

This electrical model is composed of components below.

- V_{in} - voltage source
- R_M - Resistor
- L_M - Inductor
- V_B - Back EMF

When using kirchoffs it can be show that the voltage in the inner loop in figure 4 is found to be:

$$V_M - V_R - V_L - V_B = 0 \tag{1}$$

Then substituting each components equivalent voltage expressions the final is found below.

$$V_M - IR_M - L_M \frac{dI_M}{dt} - V_B = 0 \tag{2}$$

Where V_B is found below and describes the physical limitation of a particular motor.

$$V_B = K_B \omega_M \tag{3}$$

2.0.2 Mechanical Model

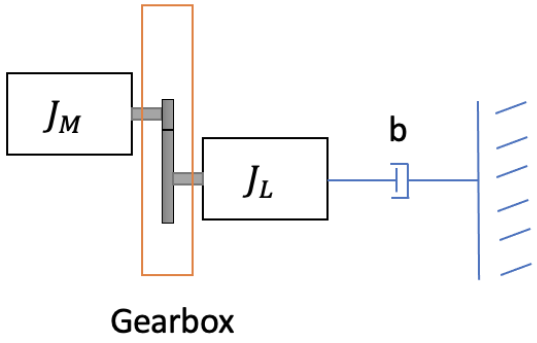


Figure 5: Motor Model - Mechanical

The Mechanical portion of the motor model is composed of mainly the 4.4:1 gearbox with an energy loss dampener attached at the end. As seen in the figure above there will be two FBDs that will be needed. Those two diagrams are found below.

(a) FBD of motor-shaft

(b) FBD of load-shaft

Figure 6: Mechanical model FBDs

For the FBD found in figure 6a the governing equation is found below, where the torque required to accelerate the motor shaft.

$$J_M \dot{\omega}_M = T_M - T_1 \tag{4}$$

Where T_1 is the reaction torque due to a moving load.

As for the FBD for the load-shaft, its governing equation includes the exit torque (T_{ext}) subtracted by its reaction torque T_2 and the energy loss term.

$$J_L \dot{\omega}_L = T_{ext} - T_2 - T_B \tag{5}$$

These two shaft are connected by gearing seen more clearly in the figure below.

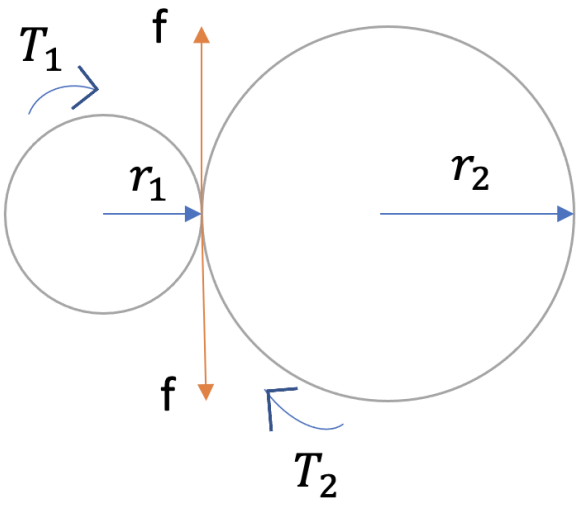


Figure 7: Motor Model - gearing

It was found that the two torques are related by the tangential force f.

$$r_1 f = -T_1$$
$$r_2 f = -T_2$$

$$T_1 = T_2(\frac{1}{N}) \tag{6}$$

Where N is the ratio of r_2 and r_1 and with this relationship also yield the following two expressions relating speed and acceleration.

$$\omega_M = -N\omega_L \tag{7}$$

$$\dot{\omega}_M = -N\dot{\omega}_L \tag{8}$$

Utilizing equations 5, 7, and 8 to yield the physical model below.

$$(J_L + N^2 J_M)\dot{\omega}_L = T_{ext} - NT_M - T_B \tag{9}$$

$$(J_{eq})\dot{\omega}_L = T_{ext} - NT_M - T_B \tag{10}$$

where T_M is dependent on the torque constant K_T.

$$T_M = K_T I_M \tag{11}$$

2.0.3 Electromechanical Model

To characterize the electromechanical model seen on figure 3 to acquire the transfer function from V_{in} to θ each equation can translated to the Laplace domain. Below the equation contains the desired voltage for a given speed.

$$V_M(s) = R_M I_M(s) + L_M s I_M(s) + K_B \omega_M(s) \tag{12}$$

Substituting eq 11 the equation below is obtained.

$$V_M(s) = R_M(\frac{T_M}{K_T}) + L_M s(\frac{T_M}{K_T}) + K_B \omega_M(s) \tag{13}$$

$$V_M(s) = (R_M + L_M s)(\frac{T_M}{K_T}) + K_B \omega_M(s) \tag{14}$$

The mechanical model can now be integrated below by substituting equation 10.

$$V_M(s) = (R_M + L_M s)(\frac{-\frac{1}{N} J_{eq} s^2 \theta_L(s) - \frac{1}{N} bs\theta_L + \frac{1}{N} T_{ext}(s)}{K_T}) + K_B \omega_M(s) \tag{15}$$

Then finally arriving at the transfer function below.

$$\frac{\theta_L(s)}{V_M(s)} = \frac{1}{\frac{L_M J_{eq}}{NK_T} s^3 + \frac{R_M J_{eq}}{NK_T} s^2 + (\frac{R_M b}{NKT} + K_B N)s} \tag{16}$$

The equation above is of 3rd order.

Then to find the transfer function from ω_L to V_M the derivative can be taken by reducing the order of the equation above to a second order transfer function.

$$\frac{\omega_L(s)}{V_M(s)} = \frac{1}{\frac{L_M J_{eq}}{NK_T} s^2 + \frac{R_M J_{eq}}{NK_T} s + \frac{R_M b}{NKT} + K_B N} \tag{17}$$

To figure out what happens as the gearbox ratio changes the terms associated with the s^2 term must divided out to achieve the standard form.

$$\frac{\omega_L(s)}{V_M(s)} = \frac{\frac{NK_T}{L_M J_{eq}}}{\frac{NK_T}{L_M J_{eq}}} \frac{1}{\frac{L_M J_{eq}}{NK_T} s^2 + \frac{R_M J_{eq}}{NK_T} s + \frac{R_M b}{NKT} + K_B N} \tag{18}$$

$$\frac{\omega_L(s)}{V_M(s)} = \frac{\frac{NK_T}{L_M J_{eq}}}{s^2 + \frac{R_M J_{eq}}{L_M J_{eq}} s + \frac{R_M b}{L_M J_{eq}} + \frac{N^2 K_B K_T}{L_M J_{eq}}} \tag{19}$$

In this form it can be compared to the general second form transfer function found below.

$$G(s) = \frac{\omega_n^2}{s^2 + (2\zeta\omega_n)s + \omega_n^2} \tag{20}$$

It appears that as N increases the natural frequency increases, and so the follwoing characteristic of the system are influenced.

Rise Time:

$$t_r = \frac{1.8}{\omega_n} \tag{21}$$

Peak Time:

$$t_p = \frac{\pi}{\omega_d} = \frac{\pi}{\sqrt{1-\zeta^2}\omega_n} \tag{22}$$

Both the rise time and peak time increase and as a result the poles move father from the real axis. As they move father with a greater N then the more oscillatory the system becomes and therfore less and less stable.

3 Hardware Interface

In order to get the motor running on this setup the first task that needs to be done is assembling the Pololu A4990 Dual Motor Driver Shield. This motor driver is shipped to its user with the components in the image below.

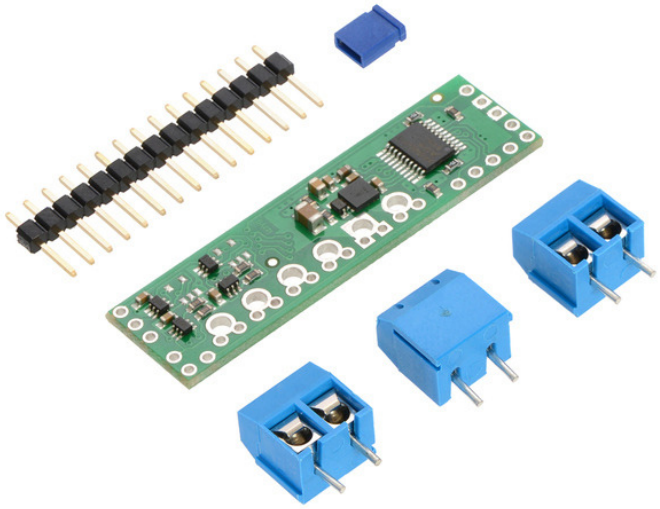


Figure 8: Pololu A4990 Dual Motor Driver Shield

The header pins and the 2-pin, 5mm terminal block are mounted the motor driver can now be mounted on the Arduino DUE. The shield plugs into the Arduino digital pins 6, 7, 8, 9, and 10 on one side and Arduino VIN, GND, GND, and 5V/VCC on the other as indicated on the PCB of the shield. In this project the motor encoder will be using the 3.3v header on the DUE. At its current state the shield will need an additional modification. The upper-left corner of the shield partially blocks the Arduino's 3.3V pin and so will need to be removed.

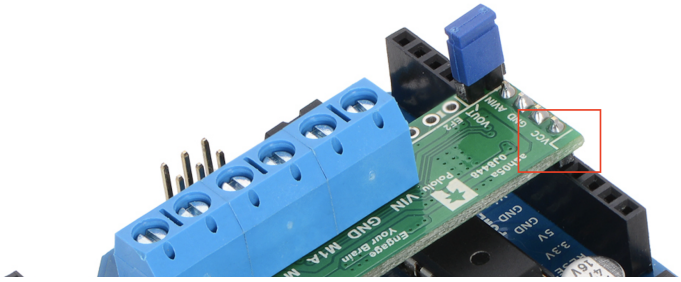


Figure 9: Pololu A4990 Dual Motor Driver Shield - blocked 3.3v

Once the corner is removed and plugged into the Due the motor can be connected to the motor controller. This particular brushed DC motor comes with 48 CPR Encoder and so will have additional signal that will need to communicate with the Arduino DUE.

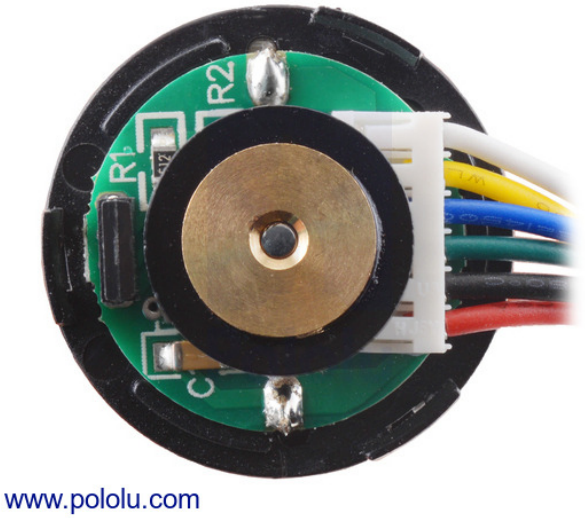


Figure 10: Pololu Metal Gearmotor with 48 CPR Encoder

The table below describes the wires functions found in the figure above.

Table 2: Pololu Brushed DC Motor Wire functions

Color	**Function**
Red	Motor Power
Black	Motor Power
Green	Encoder GND
Blue	Encoder Vcc
Yellow	Encoder Output A
White	Encoder Output B

Now with the function of each wire known the two can be connected together. To get this done jumper wires with corresponding color were used. This was done below.

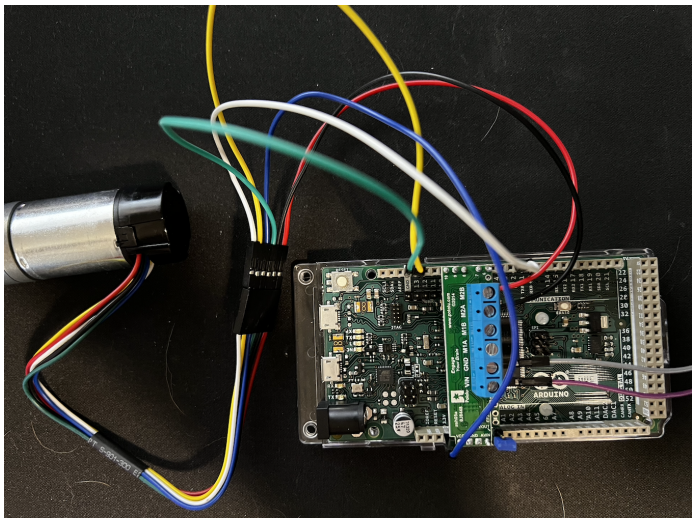


Figure 11: Pololu Motor shield with Metal Gearmotor

One thing to note is the pins located on top of the motor controller were not connected given that the Due will not be powered from the power supply that will be connected to the shield by the purple and grey jumpers. So with the configuration above the motor can be ran once connected with a 12V power supply. An Agilent single output DC power supply was used for this project.

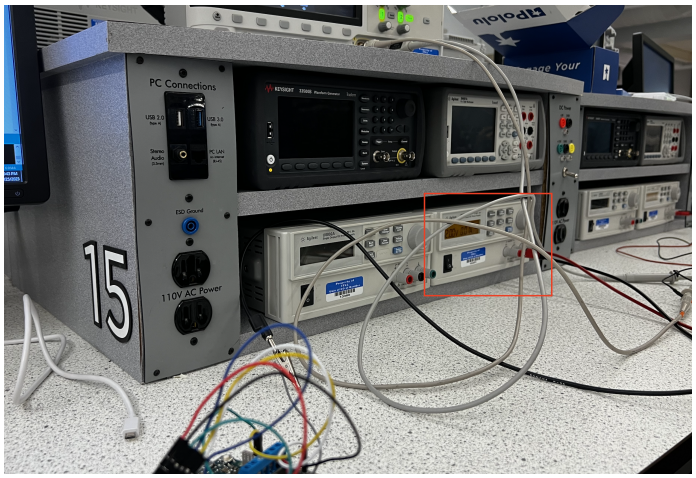


Figure 12: Agilent single output DC power supply

By running the following script using the Arduino IDE the motor can be powered.

```
digitalWrite(dir, LOW);
analogWrite(spd, 4000);
```

The variables 'dir' for direction and 'spd' for speed needed to be set to their corresponding pin-outs. In this case the motor power and GND pins were connected to the M2A and M2B terminals on the motor shield. This meant that pins 10 and 8 are used for speed and direction respectively. Once done the 'digitialWrite' function can be used to send a high or low signal, which drives the motor in a clock-wise direction or counter-clock-wise direction. In this case 'Low' corresponds to the clock-wise direction. With the direction set a speed can be selected by commanding a PWM signal to drive the motor. The script provided was set to a 12bit resolution providing 4095 levels to select, with 0 providing 0% duty cycle and 4095 100% duty cycle. The top of the that range was selected in this instance.

3.1 Encoder Setup

The first thing to look at was the encoder output reading, given that the required input voltage needed to be between 3.5 and 20v and only supplying it with 3.3v. The output of both encoders was found below.

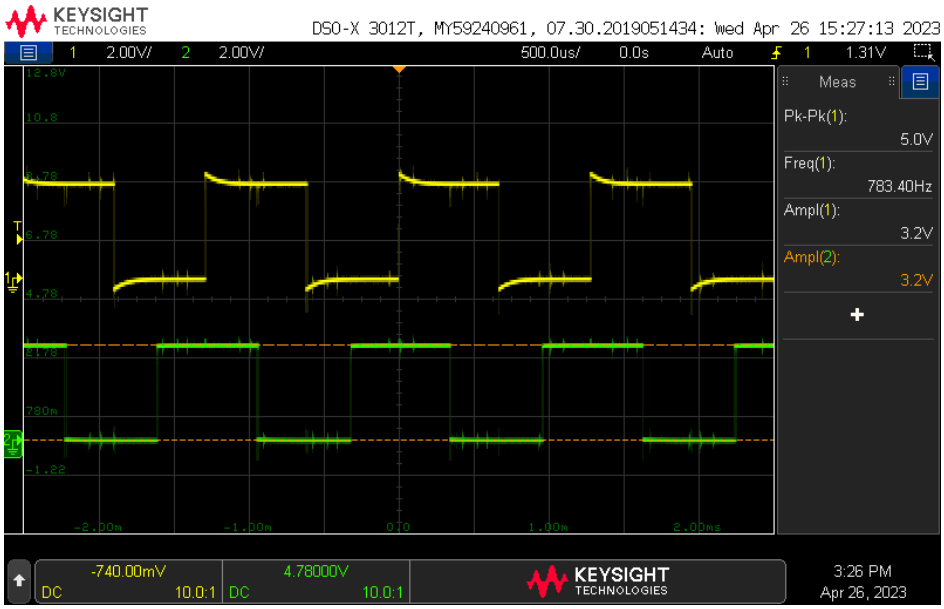


Figure 13: Encoder A and B output with 3.3v V_{cc}

The outputs are found to be approximately 90°out of phase, as intended. But one of the channels is found to exhibit a semi-square wave behavior as seen in the figure above.

The Arduino Due features a Quadrature Decoder that will need to be enabled to perform system identification then design a feedback controller. The Due has a timer counter that embeds a quadrature decoder (QDEC) connected in front of the timers and driven by TIOA0, TIOB0 and TIOB1 inputs. When enabled, the QDEC performs the input lines filtering, decoding of quadrature signals and connects to the timers/-counters in order to read the position and speed of the motor through the user interface[3]. The code below was used to receive encoder position measurements to Digital pins 2 and 13.

```
// 1 of 3 external clock inputs
  REG_PMC_PCER0 = PMC_PCER0_PID27; // PMC (Power Mangement Controller) must be configured to enable timer counter clock,
                                   // Enabled by writing PCER0 (Peripheral clock enable 0) and Peripheral ID 27 clock selected
  REG_TC0_CMR0 = TC_CMR_TCCLKS_XC0;  // Interrupt TC0 slected when PID27 is choosen Selects external clock trigger - channel 0 for speed and position
                                     // TCCLKS (TC Channel Mode Resgister) - External clock signal XC0 choosen
// Block mode regester used -  defines external clock inputs, allows them to be chained
  REG_TC0_BMR = TC_BMR_QDEN // Quadrature Decoder Enabled
              | TC_BMR_POSEN // Position enabled
              | TC_BMR_EDGPHA; // Edge on PHA Count mode -  edges detected on both PHA and PHB

  REG_TC0_CCR0 = TC_CCR_CLKEN // TC_CCR - Channel Control Register enabled
               | TC_CCR_SWTRG; // Software trigger enabled
```

Figure 14: Arduino Encoder counts - Code Commented

4 System Identification

The transfer function was derived earlier in this report but there are a few issues with utilizing it beyond its symbolic form, mainly that the coefficients (b, L_M, R_M, K_T, K_B) will need to be found to build the a usable plant for this geared motor. The simpler method is to use system identification. This blackbox approach uses statistical methods to build mathematical models of dynamical systems from its measured output data. That data is then fitted to extract parameters which would make your model closely match the data set.

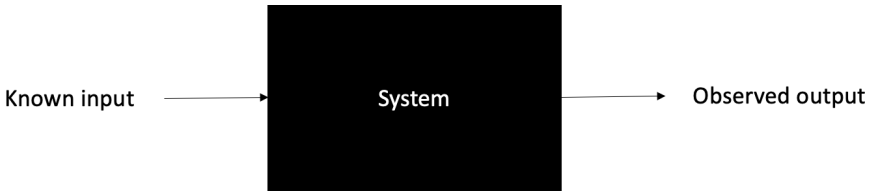


Figure 15: System Identification

This can be done for this geared motor. Although the second order plant found earlier can be used, it can be approximated to a first order system because it is known that the second order system is composed dominate pole and a really fast pole. These two poles corresponds to the mechanical and electrical parts of the system. By applying some voltage V_{in} using PWM, an ouput from the motor can be observed on the serial to find the two constants K and σ in the first order approximation found below.

$$G_\omega(s) = K\frac{\sigma}{s+\sigma} \tag{23}$$

To perform this method on this geared motor, the encoder onboard must be used. This two-channel hall effect encoder is used to sense the rotation of a magnetic disk thats tied to the motor shaft, providing a resolution 48 counts per revolution. The motorshaft is connected to a 4.4:1 gearbox at the order end allowing for a final resolution at the output-shaft of about 211 counts per revolution. These counts were outputted to the serial monitor in the Arduino IDE and copied to a text file for post processing using MATLAB.

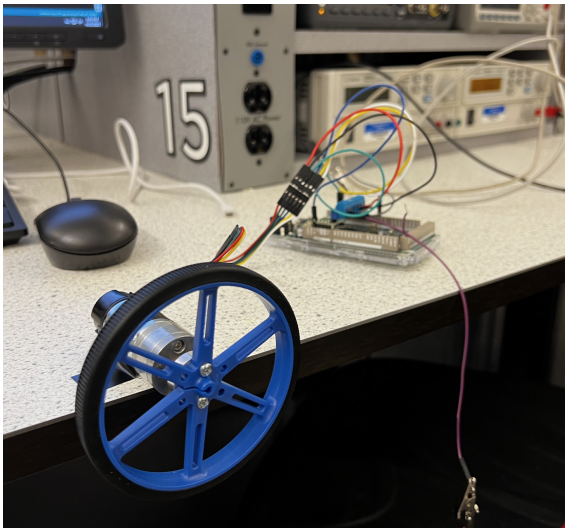


Figure 16: Data collection set up

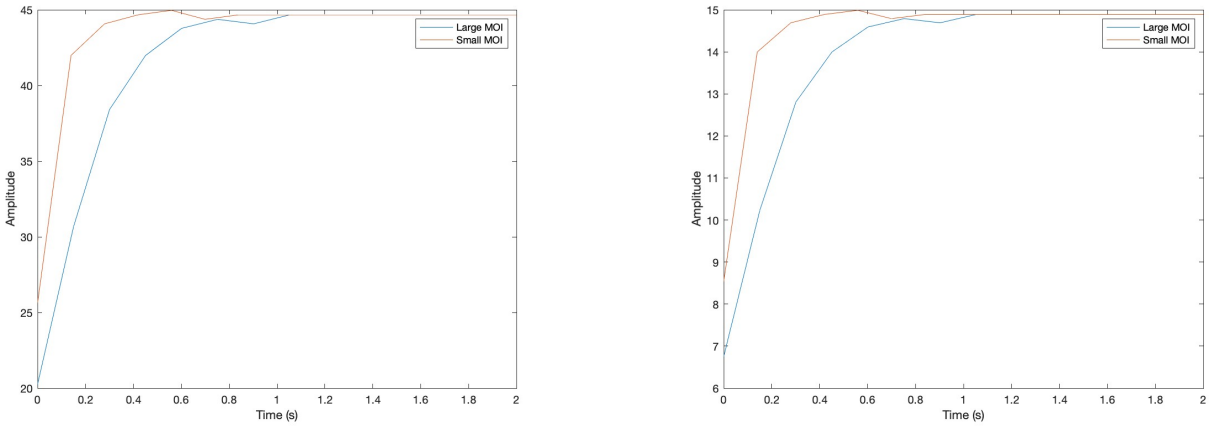
```
N=0;
Speed_BW=length(t);
for N = 1:length(Counts_BW)-1
    Speed_BW(N) = (Counts_BW(N+1) - Counts_BW(N))/((0.1));
end
N=0;
Speed_SW=length(t2);
for N = 1:length(Counts_SW)-1
    Speed_SW(N) = (Counts_SW(N+1) - Counts_SW(N))/(0.1);
end
```

Figure 17: MATLAB Script used to perform derivative

In the MATLAB script above, the derivative was performed on this data set by a future and current value divided by the difference in time for each count sample. The timer in the script was set to update the count every 100,000μs, which equates to 0.1s. The resulting responses from this gear motor and fly wheel systems were then obtained and are found in the figure below. The raw output was plotted in figure 18a while the true response is found next to it given the form of the transfer function below.

$$\frac{\omega(s)}{V_{in}(s)} = K\frac{\sigma}{s+\sigma} \tag{24}$$

When acquiring that data that lead to the step response below, a pwm of 1000 was used. This pwm equates to roughly 3V and therefore the data-set must divided by 3 to acquire to identify K and σ



(a) Response from raw data

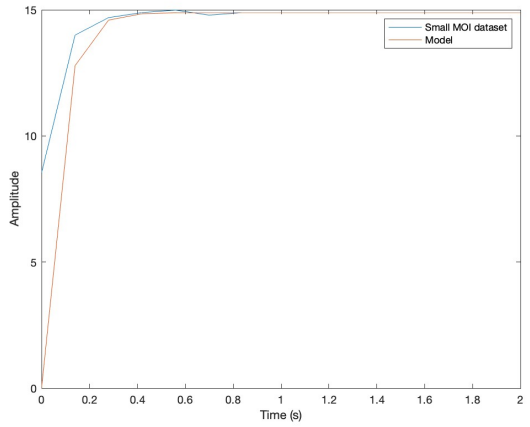
(b) Response dividing data-set by V_{in}

Figure 18: System Identification Results

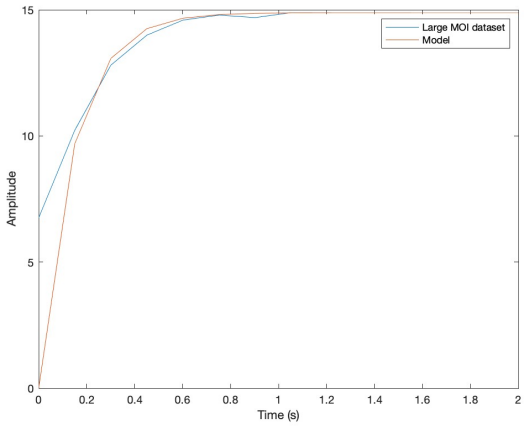
The step response for both inertia were collected and were found to make sense. The system with the smaller flywheel reached a constant speed a bit quicker than the larger diameter flywheel. To then find the coefficients of the first order approximation (eq. 18) the inverse Laplace transform must be taken. This was found be the equation below.

$$y(t) = K(1 - exp(-\sigma t)) \tag{25}$$

The fitting was done iteravly by reading off the gain K (steady state value) and changing σ until it closely matched the step response obtained from the data set.



(a) Fitted Small MOI



(b) Fitted Large MOI

Figure 19: Fitted step responses for small/large MOI

For the small and large MOI system the final values for K and σ were found to be:

$$y(t) = 14.89(1 - exp(-14t)) \tag{26}$$

$$y(t) = 14.89(1 - exp(-7t)) \tag{27}$$

Then translating this over to the first order approximation of the plant:

$$G_{\omega,S-MOI}(s) = 14.89\frac{14}{s+14} \tag{28}$$

$$G_{\omega,L-MOI}(s) = 14.89\frac{7}{s+7} \tag{29}$$

5 Feedback Control

Now that the plant was obtained, continuous time controllers can be designed. Two controllers will be designed. A PD controller will be deisgned of position and a PI controller will be designed for speed. To convert the plants over to position the derivative can simply be taken of the system by multiplying by $1/s$. This is done for both small and large MOI plants below.

$$G_{\theta,S-MOI}(s) = \frac{G_{\omega,S-MOI}(s)}{s} = 14.89\frac{14}{s(s+14)} \tag{30}$$

$$G_{\theta,L-MOI}(s) = \frac{G_{\omega,L-MOI}(s)}{s} = 14.89\frac{7}{s(s+7)} \tag{31}$$

The Closed-loop step responses were then looked over below.

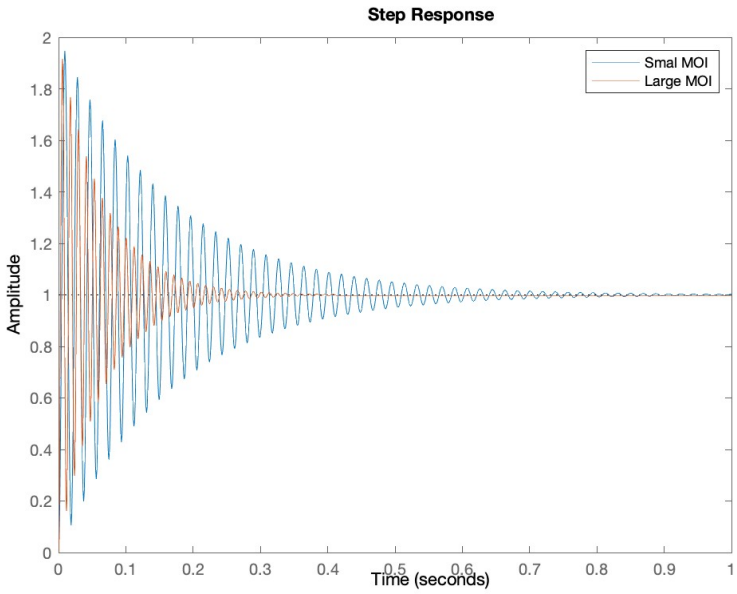


Figure 20: Closed-Loop Step Responses for small/large MOI

With no controller both plants under feedback appeared to have nearly 100% overshoot but with different settling times. The system with large MOI settled nearly 2x quicker. From the figure alone its clear that a controller would be needed to improve these two systems performance.

5.1 PD Controller - Position

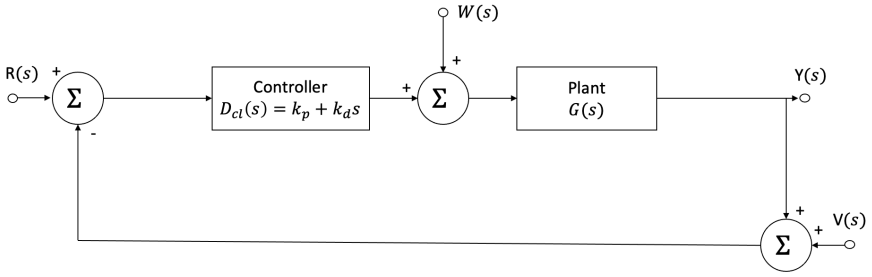


Figure 21: Closed-Loop Step Responses for small/large MOI

A PD controllers are designed to improve steady state error and increase damping which this system is need of. The PD controller in this section will be used for both plants. A unity feedback control system is found above where $D_{cl}(s)$ is equal to:

$$D_{cl}(s) = K_p + K_d s \tag{32}$$

Where the closed loop transfer from R(s) to Y(s) is equal to:

$$G_{cl}(s) = \frac{D_{cl}(s)G(s)}{1 + D_{cl}(s)G(s)} \tag{33}$$

The closed loop transfer function when substituting $D_{cl}(s)$ and $G(s)$ we get

$$G_{cl}(s) = \frac{208.4(K_p + K_d)}{s^2 + s(208.4K_d + 14) + 208.4K_p} \tag{34}$$

The equation above can then be compared to that of a general second order system to find desired system performance characteristics. Before doing so performance specification must be set and those have been set in the table below.

Table 3: Performance Specifications

Parameters	**Specification**
Overshoot (%)	≤ 10
Settling Time (s)	$\leq 1s$
Rise Time (s)	N/A
Steady State Error (step)	0

For the desired OS, a dampening coeff (ζ) can be found.

$$\zeta \geq \frac{ln(M_p)}{\sqrt{\pi^2 + (ln(M_p))^2}} \tag{35}$$

Where M_p is the minimum OS.

With this dampening value, θ_{min} can also be found to further clarifying our performance region.

$$\theta_{min} = sin^{-1}(\zeta) \tag{36}$$

$$\begin{aligned} \theta_{min} &= sin^{-1}(0.50) \\ &= 30^\circ \end{aligned}$$

Given that the system needs to have a settling time of less than 1s. A desired ω can be found.

$$t_s = \frac{4.6}{\zeta\omega_n} \tag{37}$$

$$\omega_n \geq \frac{4.6}{\zeta t_s} \tag{38}$$

Table 4: Performance Region Values

Parameters	**Specification**	**Desired Values**
Overshoot (%)	10	$\zeta \geq 0.49$
Settling Time (s)	1s	$\omega_n \geq 15$ rad/s
Rise Time (s)	N/A	N/A
Steady State Error (step)	0	N/A

With the guidance from the table above the a dampening value of 0.9 and natural frequency of 100 rad/s were selected. The appropriate K_p and K_d values can now be found by comparing to the general form of a second order system found below.

$$G(s) = \frac{\omega_n^2}{s^2 + (2\zeta\omega_n)s + \omega_n^2} \tag{39}$$

To find K_d:

$$\begin{aligned} 208.4K_d + 14 &= 2\zeta\omega_n \\ K_d &= \frac{2\zeta\omega_n^2 - 14}{208.4} \\ K_d &= \frac{2(0.9)(100) - 14}{208.4} \\ K_d &= 0.79 \end{aligned}$$

Then to find K_p

$$\begin{aligned} 208.4K_p &= \omega_n^2 \\ K_p &= \frac{\omega_n^2}{208.4} \\ K_p &= \frac{100^2}{208.4} \\ K_p &= 47.93 \end{aligned}$$

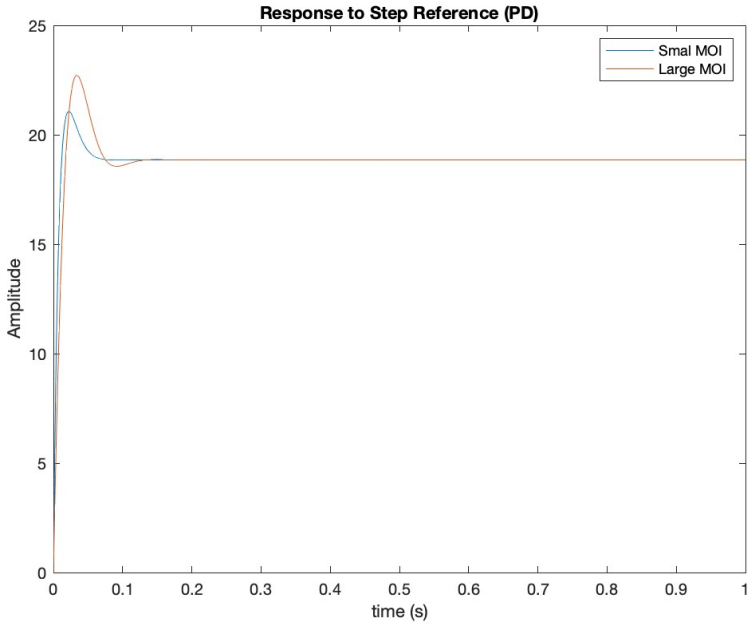


Figure 22: CL response - PD control

The response found up above was found to produce more overshoot than desired at a little above 10% and more so with the system with the larger fly wheel. So below the dampening ratio was increased to 1, then the resulting K_d constant doubled to achieve the final PD controller. The final simulated step resonse is found below.

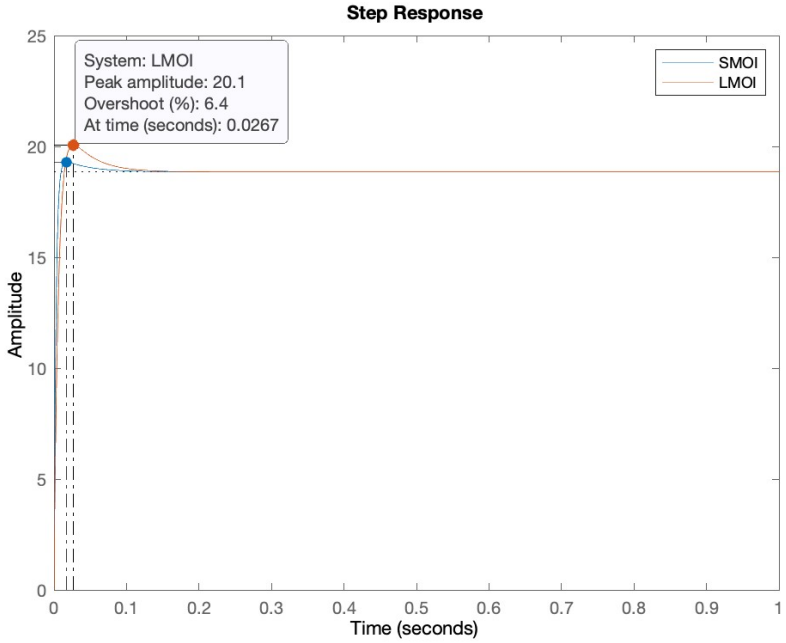


Figure 23: CL response - PD control

5.2 PI Controller - Speed

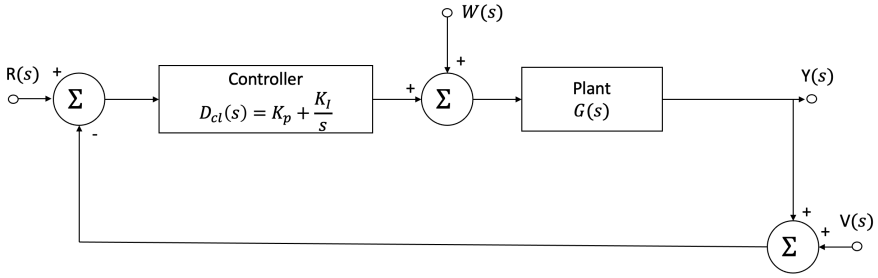


Figure 24: Closed-Loop Step Responses for small/large MOI

A PI controllers are designed to improve steady state error by increasing system type but at the cost of increased overshoot. The PI controller in this section will be used for both plants. A unity feedback control system is found above where $D_{cl}(s)$ is equal to:

$$D_{cl}(s) = K_p + \frac{K_i}{s} \tag{40}$$

Where the closed loop transfer from R(s) to Y(s) is equal to:

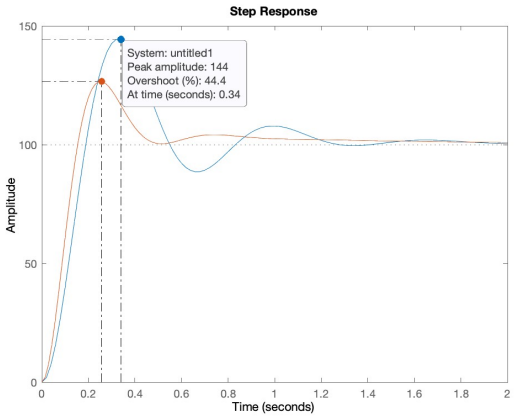
$$G_{cl}(s) = \frac{D_{cl}(s)G(s)}{1 + D_{cl}(s)G(s)} \tag{41}$$

The closed loop transfer function when substituting $D_{cl}(s)$ and $G(s)$ we get

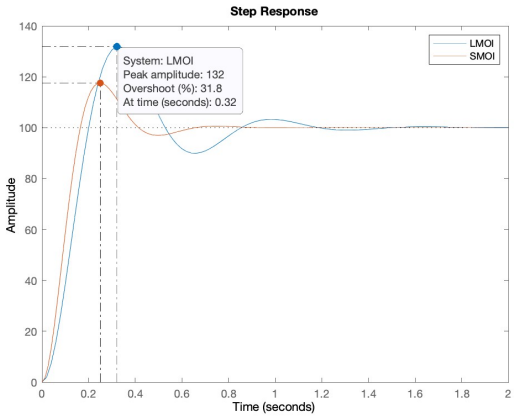
$$G_{cl}(s) = \frac{208.4(K_p s + K_i)}{s^3 + 14s^2 + 208.4K_p s + 208.4K_i} \tag{42}$$

The equation above can then be compared to that of a general second order system to find desired system performance characteristics. The same performance specifications were set for this PI controller found in table 3 in the previous section.

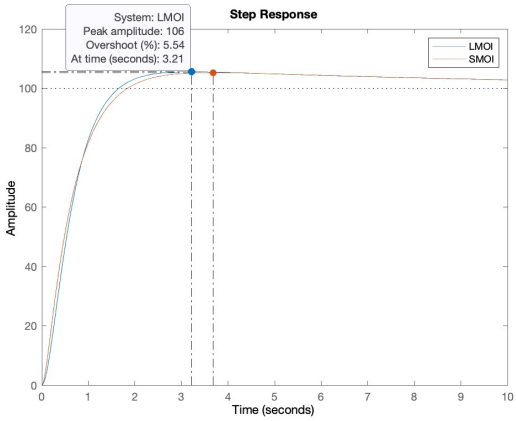
This PI Controller was done iteratively by first selecting a K_p and K_i of 1 then seeing the resulting step response with the controller on the plants.



(a) $K_p = 1, K_i = 1$



(b) $K_p = 1, K_i = 0.1$



(c) $K_p = 0.1, K_i = 0.1$

Figure 25: PI Controller

The final PI Controller was found with final values of K_p and K_i of 0.1 and 0.01 respectively.

6 Emulation

At this point the plant was identified and a PD/PI controller was developed and both under feedback lead to a stable response. To transfer this into a usable form emulation must be used. Emulation in a general

sense is used to imitate the behavior and in this case its continuous time controls system imitating digital control systems. Its desired to convert a continuous system to discrete with out any change to the system ideally, so its behavior is the exact same or similar in both domains.

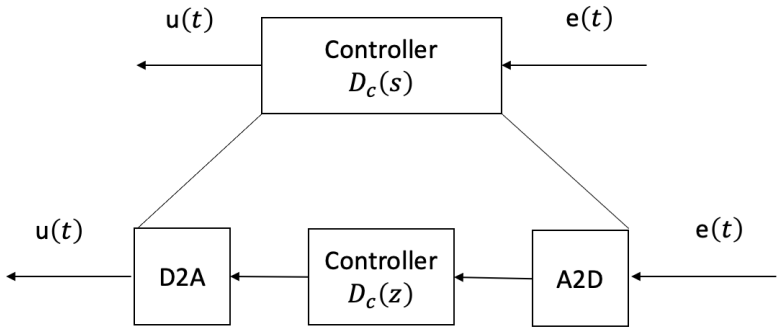


Figure 26: Continuous-time approximation

There are 2 methods that will be discussed: Forward Euler's and Tustin's. These 2 methods allow for discrete time approximation of any continuous time transfer function. By allowing substituting the following for s.

Forward Difference:

$$s = \frac{z-1}{T} \tag{43}$$

Tustin:

$$s = \frac{2(z-1)}{T(z+1)} \tag{44}$$

During the course of this project the controller was found to be unstable and so a pole was added to the PD controller. This new PD controller was found below where τ places the pole times farther from the zero location.

$$C_{PD}(s) = \frac{K_p + K_d s}{\tau s + 1} = K_d \frac{s + \frac{Kp}{K_d}}{\tau s + 1} \tag{45}$$

This pole was added after the design of the controller.

6.1 Forward Euler's

6.1.1 PD Controller

To convert the continuous controller C(s) to a discrete controller c(z), simply plug in the approximation (eq. 26) for s. The sampling time T will be 1ms.

$$C_{PD}(s = \frac{z-1}{T}) = 1.785s + 47.94$$
$$= 1.785(\frac{z-1}{T}) + 47.94$$
$$= 1.785(\frac{z-1}{0.001}) + 47.94$$

So the emulated discrete time controller is found to be:

$$C_{PD}(z) = 1785(z - 0.9731) \tag{46}$$

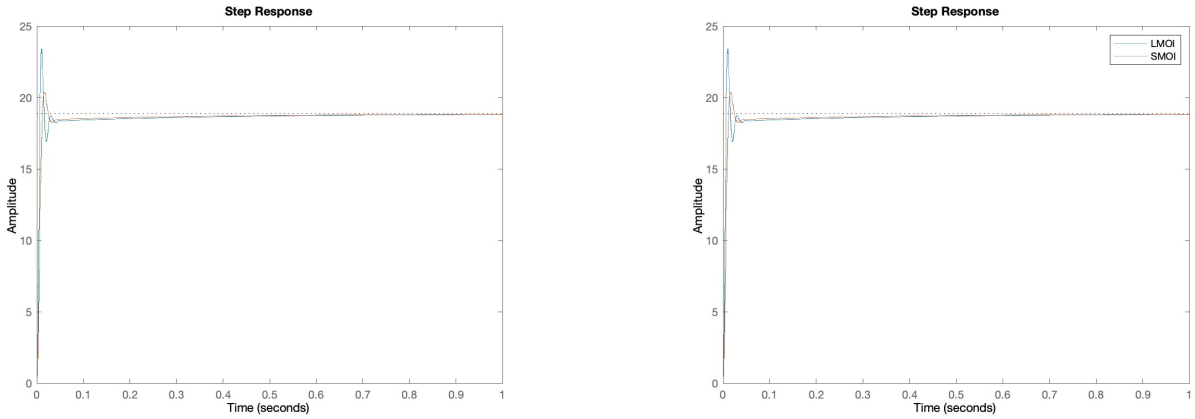
After the pole was added:

$$C_{PD}(z) = \frac{4793(z - 0.9731)}{z + 1.68} \tag{47}$$

This controller was found to be unstable with the gain of 47, so it was lowered by a factor of 10.

$$C_{PD}(z) = \frac{479.39(z - 0.9973)}{z - 07314} \tag{48}$$

This controller was then tested by translating the plant using the 'C2D' function and matlab. A final stable controller using forward Euler is found below.



(a) Forward Eulers, $T_s = 0.1$

(b) Forward Eulers, $T_s = 0.001$

Figure 27: Feedback discrete time step response - Forward Eurler

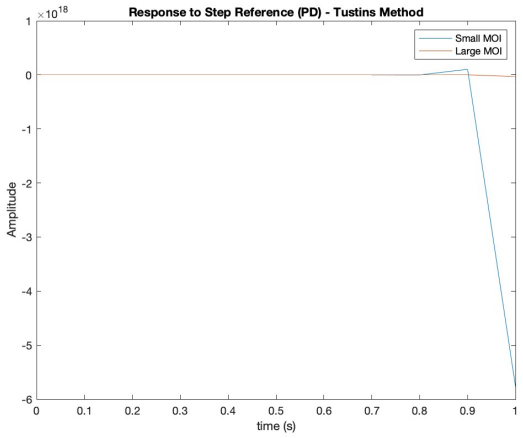
6.2 Tustin's

6.2.1 PD Controller

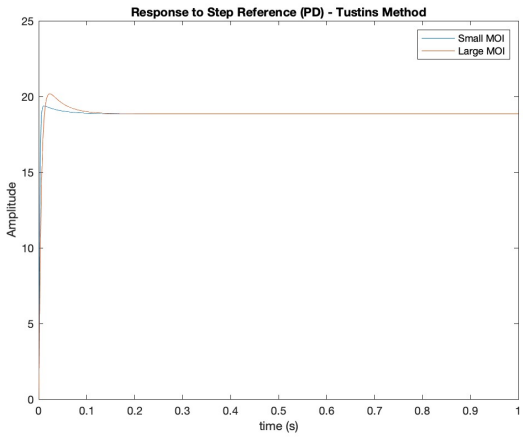
The same procedure can be done to approximate the continuous time controller using eq. 27. But in this case MATLAB's built-in 'c2d' function will be used to obtain the discrete time controller. The resulting z-transform is found below.

$$C_{PD}(z) = \frac{2074z - 2019}{z + 0.1463} \tag{49}$$

The discrete response of with this controller is found below.



(a) Tustin, $T_s = 0.1$



(b) Tustin, $T_s = 0.001$

Figure 28: Feedback discrete time step response - Tustins

7 Micro-Controller Implementation

7.1 PD Controller

The analysis was done in the previous section and has proven to be stable. The controller can now be implemented on the Arduino Due. This is simply done by converting the z-transform for the controller to a difference equation. Then solving for the output. This was done below.

$$C_{PD}(z) = \frac{U(z)}{E(z)} = \frac{2074z - 2019}{z + 0.1463} \tag{50}$$

The controller above that will be implemented will be the one for which Tustin's approximation was used to emulate, given that the Forward Euler method is a non-causal method.

$$\begin{aligned} \frac{U(z)}{E(z)} &= \frac{2074z - 2019}{z + 0.1463} \\ E(z)(2074z - 2019) &= U(z)(z + 0.1463) \end{aligned}$$

Each z in the equation is a time delay so the difference equation is found to be:

$$2074zE[k+1] - 2019E[k] = 0.1463U[k] + U[k+1]$$

Then shifting each K by 1:

$$2074E[k] - 2019E[k-1] = 0.1463U[k-1] + U[k]$$

The final implementable form for the .

$$U[k] = 2074E[k] - 2019E[k-1] - 0.1463U[k-1] \tag{51}$$

The difference equations was then translated to the Arduino IDE for testing.

```
// Controller/ Difference Equation
//V = 2074*rad_Err - 2019*rad_err_prev - 0.1463*V_prev; // Difference equation kp =47, Kd =0.7-  a lot of jitter
//V = 388*rad_Err - 368*rad_err_prev + 0.57*V_prev; // Difference equation kp =47, Kd =0.7-  a lot of jitter
V = 46.85*rad_Err - 46*rad_err_prev + 0.94*V_prev; // Difference equation kp =4.7
//V = 0.4797*rad_Err - 0.4797*rad_err_prev + 0.99995*V_prev; // Difference equation kp =0.0048
//V = 19.93*rad_Err - 19.93*rad_err_prev + 0.9777*V_prev; // Difference equation kp =0.2 - Steady state of 2*pi?

rad_err_prev = rad_Err;
V_prev = V;
```

Figure 29: Controllers translated to Arduino IDE

This particular controller was found to produce a jittery response, likely due to motor saturation so the gain was lowered to 4.7 and produced much better response.

7.2 PI Controller

The same process was used to find the difference equation. The discrete controller for the PI controller is found below, in this case only using Tustin's approximation. Three total controller were used during testing. The one below was found using a sampling time of 0.001s.

$$C_{PD}(z) = \frac{U(z)}{E(z)} = \frac{0.1z - 0.1}{z - 1} \tag{52}$$

The second controller tested was found by reducing the sampling time to 0.01s.

$$C_{PD}(z) = \frac{U(z)}{E(z)} = \frac{0.1001z - 0.09995}{z - 1} \tag{53}$$

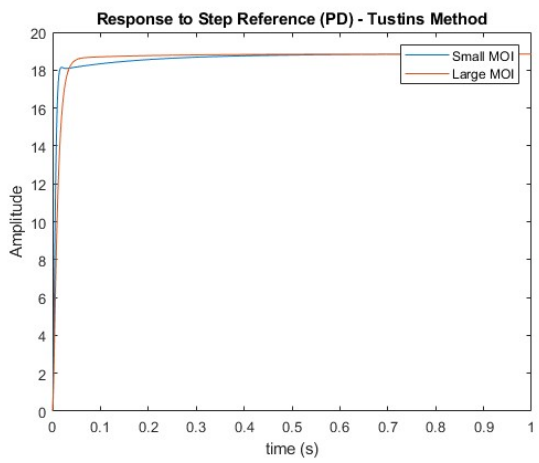
The third controller was tested by using the same sampling time of 0.01s and an addition of a poles that is 10x greater than the zero.

$$C_{PD}(z) = \frac{U(z)}{E(z)} = \frac{0.8337z^2 + 0.0008333z - 0.8329}{z^2 - 0.3333z - 0.6667} \tag{54}$$

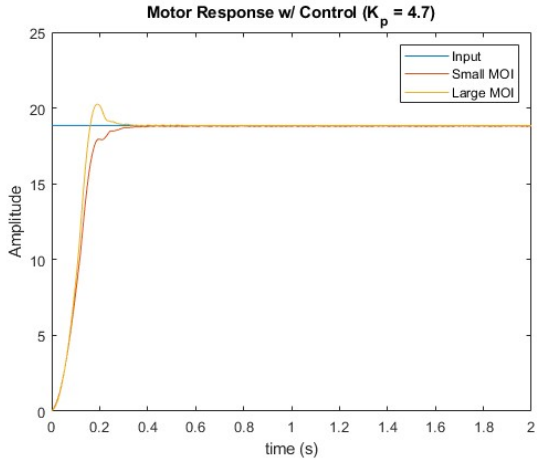
8 Results

8.1 PD Controller - Position

There was two controllers that provided results. The one found below was with the gain K_p of 4.7. The input to the system chosen to be 6π, so the final value shall approach 18.



(a) Discrete response, $K_p = 4.7$, $K_d = 0.81$

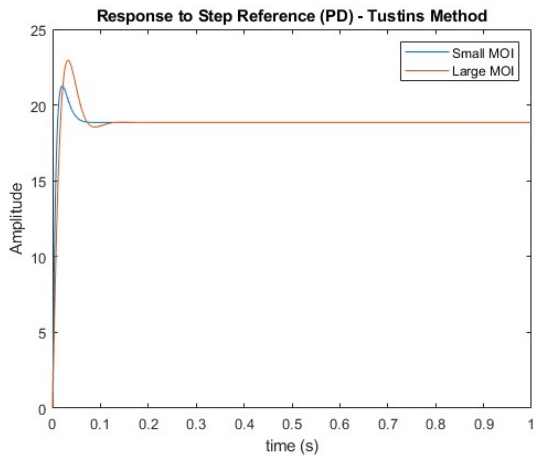


(b) Findings when tested

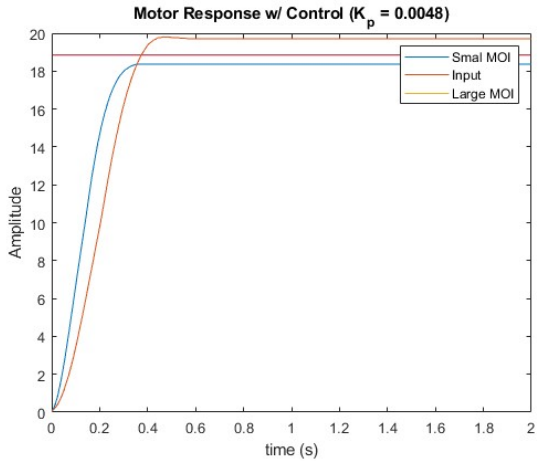
Figure 30: Simulation vs test $K_p = 4.7$, $K_d = 0.81$

In testing the results did not match the simulation very well. The larger fly wheel system tended to overshoot roughly 10% each time this test was ran while the smaller fly wheel system did not over shoot like its simulation but did not exactly match.

In this next the K_p value was set really low to reduce overshoot and decrease rise time. The findings are found below.



(a) Discrete response, $K_p = 0.0047$, $K_d = 0.81$



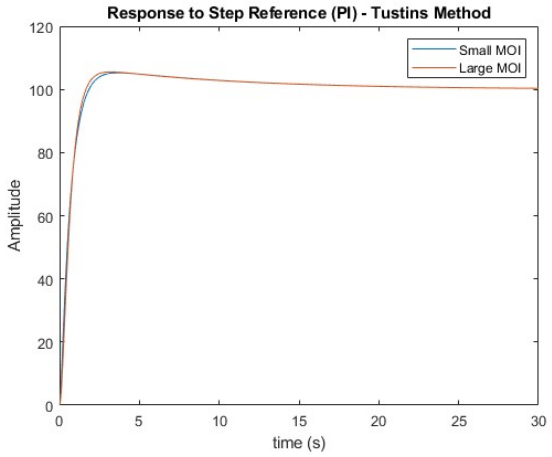
(b) Findings when tested

Figure 31: Simulation vs test $K_p = 0.0047$, $K_d = 0.81$

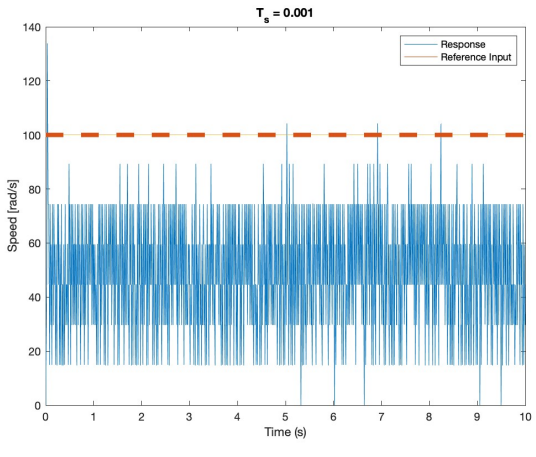
These results also did not match the simulation. In this case the smaller fly wheel system performed better than the simulation resulting in no overshoot and settling to the input with less than 1% error. While the larger fly wheel system overshoot by nearly 12% and settled not to far from its overshoot value.

8.2 PI Controller - Speed

The speed controller results were found to be nosy. There were a few attempts made to reduce the noise. The first set of figures found below was tested with a sampling time of 1000hz.



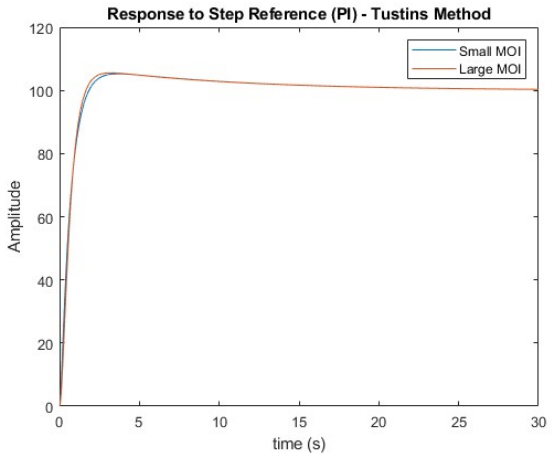
(a) Discrete response, $K_p = 0.1$, $K_I = 0.01$



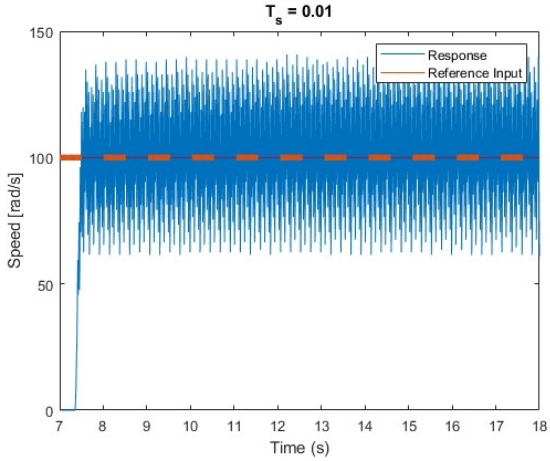
(b) Findings when tested at 1000hz

Figure 32: Simulation vs test ($K_p = 0.1$, $K_i = 0.01$, $T_s = 0.001s$)

The results in figure 32b show a rather large overshoot and settling to around 50 rad/s. It is possible that during testing I may have tried multiple values including an input of 50rad/s, given that it settles nearly there.



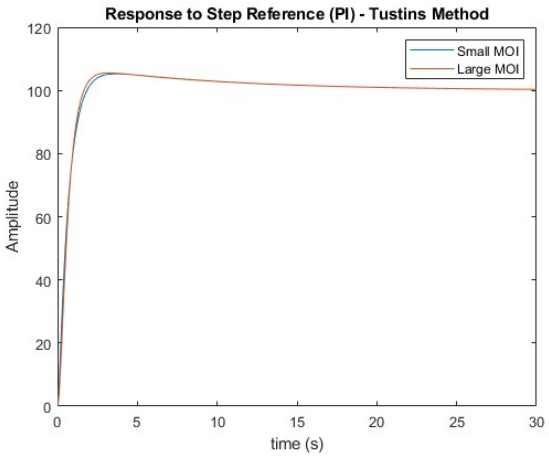
(a) Discrete response, $K_p = 0.1$, $K_I = 0.01$



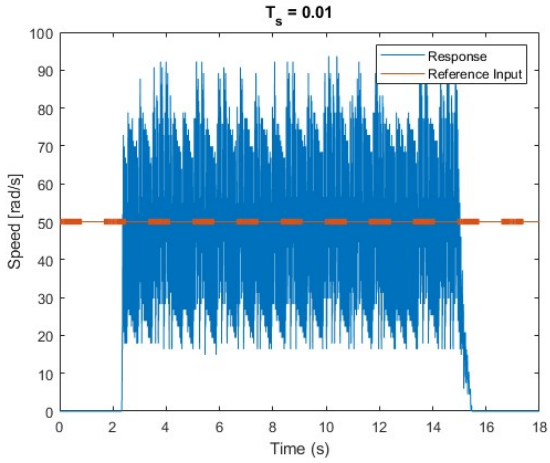
(b) Findings when tested at 100hz

Figure 33: Simulation vs test ($K_p = 0.1$, $K_i = 0.01$, $T_s = 0.01s$)

In the second round of testing, the sampling time was reduced to see if it would help the noise issue but that did not happen. The noisy response does appear to settle near the 100 rad/s command given. The overshoot and settling time is not easily seen in the figure.



(a) Discrete response, $K_p = 0.1$, $K_I = 0.01$



(b) Findings when tested at 100hz, with a pole 10x greater than the zero

Figure 34: Simulation vs test ($K_p = 0.1$, $K_i = 0.01$, $T_s = 0.01s$)

In the final test the results were just as noisy if not more with teh addition of a the pole that its 10x farther from the zero.

It is plausible that to achieve a less noisy response the speed of the motor will need to be used in the quadrature setup, rather than using the position reading.

9 References

1. https://www.pololu.com/product/4861
2. https://www.pololu.com/product/2512
3. SAM3X Datasheet

9.1 Matlab Code

To run code use mlx files, the m file is only for general overview.

```
clc; clear all; close all;
% Motor Counts
Counts_BW = importdata('Copy_of_Motor_bw_counts.txt'); % [counts/ms]
Counts_BW = (2*pi*Counts_BW/(211)); % [rad/ms]

Counts_SW = importdata('Motor_sw_counts.txt'); % [counts/ms]
Counts_SW = (2*pi*Counts_SW/(211)); % [rad/ms]

t = linspace(0,40,length(Counts_BW));
t2 = linspace(0,40,length(Counts_SW));

N=0;

Speed_BW=length(t);

for N = 1:length(Counts_BW)-1

    Speed_BW(N) = (Counts_BW(N+1) - Counts_BW(N))/((0.1)); % 0.1s intervals

end

N=0;

Speed_SW=length(t2);

for N = 1:length(Counts_SW)-1

    Speed_SW(N) = (Counts_SW(N+1) - Counts_SW(N))/(0.1); % 0.1s intervals


end

% Responses
figure;
plot(t(1:end-1), Speed_BW) % dividing by V_in
hold on
plot(t2(1:end-1), Speed_SW)
legend('Large MOI', 'Small MOI')
xlabel('Time (s)')
ylabel('Amplitude')
xlim([0 2]);

% Inverse L of second order TF
K_hat = 44.66/3; % divide by 3v
```

```
sigma_SMOI = 14;
sigma_LMOI = 7;
y = K_hat*(1 - exp(-sigma_LMOI*t)) % Large MOI
y2 = K_hat*(1 - exp(-sigma_SMOI*t2)) % Small MOI

figure;
plot(t(1:end-1), Speed_BW/3)
hold on
plot(t , y)
legend('Large MOI dataset', 'Model')
xlabel('Time (s)')
ylabel('Amplitude')
xlim([0 2]);

figure;
plot(t2(1:end-1), Speed_SW/3)
hold on
plot(t2, y2)
legend('Small MOI dataset', 'Model')
xlabel('Time (s)')
ylabel('Amplitude')
xlim([0 2]);

%modelfun = @(b,t) b(1)*(1-exp(-b(2)*t));
%beta0 = [10000, 0.0001];

%beta = nlinfit(t,Counts_BW',modelfun,beta0)

%y_fit = beta(1)*(1-exp(-beta(2)*t));

%figure;
%plot(t, y_fit)
Plant

s = tf('s');

k = K_hat;

G_omega = k*(sigma_SMOI/((s+sigma_SMOI)))
G_theta_SMOI = k*(sigma_SMOI/(s*(s+sigma_SMOI)))
G_theta_LMOI = k*(sigma_LMOI/(s*(s+sigma_LMOI)))


figure;
step(G_omega, 5)
figure;
step(G_theta_SMOI, 5)
figure;
rlocus(G_theta_SMOI)

G_theta_CL_SMOI = feedback(G_theta_SMOI,1)
G_theta_CL_LMOI = feedback(G_theta_LMOI,1)

figure;
```

```
step(G_theta_CL_SMOI)
hold on
step(G_theta_CL_LMOI)
legend('Smal MOI', 'Large MOI')
Controller
kP = 10000/208.6;
kD = 168/208.4;%2*(186/208.4);

tau  = 1/((kP/kD)*100)

C_PD = (kP + kD*s)

figure;
pzmap(C_PD)

% Plot Response to Step Reference
R = 6*pi;
[y_pd_SMOI, t4] = step(R * feedback(C_PD * G_theta_SMOI, 1), 1);
[y_pd_LMOI, t5] = step(R * feedback(C_PD * G_theta_LMOI, 1), 1);

figure;
rlocus(C_PD*G_theta_CL_SMOI)

figure;
plot(t4, y_pd_SMOI)
hold on
plot(t5, y_pd_LMOI)
title('Response to Step Reference (PD)');
xlabel('time (s)')
ylabel('Amplitude')
legend('Smal MOI', 'Large MOI')
%xlim([0 2])
C_PI_z = c2d(C_PI, Ts, 'Tustin')

G_theta_z_SMOI = c2d(G_theta_SMOI, Ts)
G_theta_z_LMOI = c2d(G_theta_LMOI, Ts)

figure;
rlocus(C_PI_z*G_theta_z_SMOI)
xlim([-1 2])


[y_pi_z_SMOI, t_z] = step(R * feedback(C_PI_z * G_theta_z_SMOI, 1), 30);
[y_pi_z_LMOI, t_z2] = step(R * feedback(C_PI_z * G_theta_z_LMOI, 1), 30);

figure;
plot(t_z, y_pi_z_SMOI)
title('Response to Step Reference (PI) - Tustins Method');
xlabel('time (s)')
ylabel('Amplitude')
hold on
plot(t_z2, y_pi_z_LMOI)
legend('Small MOI', 'Large MOI')C_PI_z = c2d(C_PI, Ts, 'Tustin')

```

```
153 G_theta_z_SMOI = c2d(G_theta_SMOI, Ts)
154 G_theta_z_LMOI = c2d(G_theta_LMOI, Ts)
155
156 figure;
157 rlocus(C_PI_z*G_theta_z_SMOI)
158 xlim([-1 2])
159
160
161 [y_pi_z_SMOI, t_z] = step(R * feedback(C_PI_z * G_theta_z_SMOI, 1), 30);
162 [y_pi_z_LMOI, t_z2] = step(R * feedback(C_PI_z * G_theta_z_LMOI, 1), 30);
163
164 figure;
165 plot(t_z, y_pi_z_SMOI)
166 title('Response to Step Reference (PI) - Tustins Method');
167 xlabel('time (s)')
168 ylabel('Amplitude')
169 hold on
170 plot(t_z2, y_pi_z_LMOI)
171 legend('Small MOI', 'Large MOI')
172
173 Ts = 0.001; % TC set to 1000us -> 1ms in Arduino IDE
174
175 Hz_forward = c2d_euler(C_PD/(tau*s+ 1),Ts,'forward')
176 Hz_forward_f = c2d_euler(C_PD/(tau*s+ 1),Ts,'forward','zpk')
177
178 C_PD_z = c2d(C_PD/(tau*s+ 1), Ts, 'Tustin')
179
180 G_theta_z_SMOI = c2d(G_theta_SMOI, Ts)
181 G_theta_z_LMOI = c2d(G_theta_LMOI, Ts)
182
183 figure;
184 rlocus(C_PD_z*G_theta_z_SMOI)
185 xlim([-1 2])
186
187 [y_pd_z_Eulers, t_z3] = step(R * feedback(Hz_forward * G_theta_z_SMOI, 1), 1);
188
189 [y_pd_z_SMOI, t_z] = step(R * feedback(C_PD_z * G_theta_z_SMOI, 1), 1);
190 [y_pd_z_LMOI, t_z2] = step(R * feedback(C_PD_z * G_theta_z_LMOI, 1), 1);
191
192 figure;
193 plot(t_z, y_pd_z_SMOI)
194 title('Response to Step Reference (PD) - Tustins Method');
195 xlabel('time (s)')
196 ylabel('Amplitude')
197 hold on
198 plot(t_z2, y_pd_z_LMOI)
199 %xlim([0 2])
200 legend('Small MOI', 'Large MOI')
201 Motor Response with Controller
202
203 % Big wheel
204 MRes_BW_kp5 = importdata('BWMOI_Kp5.txt');
205 t_kp5 = linspace(0,3,length(MRes_BW_kp5));
206
```

```
MRes_BW_kp0048 = importdata('BMOI_kp0048.txt');
t_kp5_4 = linspace(0,10,length(MRes_BW_kp0048));

% Small wheel
MRes_SW_kp5 = importdata('SWMOI_Kp5.txt');
t_kp5_2 = linspace(0,10,length(MRes_SW_kp5));

MRes_SW_kp0048 = importdata('SMOI_kp0048.txt');
t_kp5_3 = linspace(0,10,length(MRes_SW_kp0048));

Input_r = 6*pi*ones(length(t_kp5_3));


figure;
plot(t_kp5, 6*pi*ones(length(t_kp5))) % ref
hold on
%figure;
plot(t_kp5_2, MRes_SW_kp5)
title('Motor Response w/ Control (K_{p} = 4.7)');
xlabel('time (s)')
ylabel('Amplitude')
hold on
plot(t_kp5, MRes_BW_kp5)
xlim([0 2])
legend('Input', 'Small MOI', 'Large MOI')

figure;
plot(t_kp5_3, Input_r)
hold on
plot(t_kp5_3, MRes_SW_kp0048)
title('Motor Response w/ Control (K_{p} = 0.0048)');
xlabel('time (s)')
ylabel('Amplitude')
hold on
plot(t_kp5_4, MRes_BW_kp0048)
xlim([0 2])
legend('Input', 'Small MOI', 'Large MOI')
```

9.2 Arduino Code - Position Control

```
#include "DueTimer.h"

#define COUNTS_PER_ROTATION 211
#define pi 3.1416

//**********************************
int count_q = 0;      // quadrature decoder hardware counts
int spd = 10; // speed
int dir = 8;  // direction 'High' - negative counts (CCW), "LOW" - positive Counts (CW)
int pos = LOW;

//****************Intializing*****************
float count_q_prev = 0;
float V_prev = 0;
float rad_err_prev = 0;

float rad_Input;
```

```
float rad_Output;
float rad_Err;
float V_req_pwm;

//*****************Voltage*****************
//Maximum motor voltage
float V_max = 12;
//Minimum motor voltage
float V_min = 0;
float V = 0;          // set initial voltage to zero

void setup() {

  Serial.begin(9600); //115200 or 9600
  analogWriteResolution(12);    // set DAC output to maximum 12-bits
  Timer3.attachInterrupt(update).start(1000); // start ISR timer3 (not used by quad decoder and PWM) at 1 ms, which is 1kHz
                                              // 1000us = 1ms
  pinMode(spd, OUTPUT);
  pinMode(dir, OUTPUT);

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

  // 1 of 3 external clock inputs
    REG_PMC_PCER0 = PMC_PCER0_PID27; // PMC (Power Mangement Controller) must be configured to enable timer counter clock, Enabled by
                                     // writing PCER0 (Peripheral clock enable 0) and Peripheral ID 27 clock selected
    REG_TC0_CMR0 = TC_CMR_TCCLKS_XC0;  // Interrupt TC0 slected when PID27 is choosen
    Selects external clock trigger - channel 0 for speed and position
                                      // TCCLKS (TC Channel Mode Resgister) - External clock signal XC0 choosen
  // Block mode regester used -  defines external clock inputs, allows them to be chained
    REG_TC0_BMR = TC_BMR_QDEN // Quadrature Decoder Enabled
                | TC_BMR_POSEN // Position enabled
                | TC_BMR_EDGPHA; // Edge on PHA Count mode -  edges detected on both PHA and PHB

    REG_TC0_CCR0 = TC_CCR_CLKEN // TC_CCR - Channel Control Register
                 | TC_CCR_SWTRG; // Software trigger
}

void loop() {
  count_q = REG_TC0_CV0; // Position measurement is read from thetimer counter value register, cv contains counter value in real time
  //Serial.println(6*pi);
  //Serial.print(" ");
  Serial.println((2*pi*(count_q))/COUNTS_PER_ROTATION);
}

void update() {
  count_q = REG_TC0_CV0;

  rad_Input = 6*pi; // desired output [rad]
  rad_Output = (2*pi*(count_q))/COUNTS_PER_ROTATION; // [rad]

  rad_Err = rad_Input - rad_Output; //error

  // Controller/ Difference Equation
  //V = 388*rad_Err - 368*rad_err_prev + 0.57*V_prev; // Difference equation kp =47 -  a lot of jitter
  V = 46.85*rad_Err - 46*rad_err_prev + 0.94*V_prev; // Difference equation kp =4.7
  //V = 0.4797*rad_Err - 0.4797*rad_err_prev + 0.99995*V_prev; // Difference equation kp =0.0048
  //V = 19.93*rad_Err - 19.93*rad_err_prev + 0.9777*V_prev; // Difference equation kp =0.2 - Steady state of 2*pi?
```

```
76    rad_err_prev = rad_Err;
      V_prev = V;
78
      pos  = LOW; //  Positive counter values
80
      if (V < V_min) {
82        pos  = HIGH; // if input voltage is negative flip direction
        }
84
      digitalWrite(dir, pos); // direction can change dependent on V
86
      V_req_pwm = fabs(V)*(4095/12); // Voltage must remain postive for pwm
88
      if (V_req_pwm> 4095)
90    {
        V_req_pwm = 4095;
92    }

94    analogWrite(spd, V_req_pwm);

96  }
```

Code/ref.ino

9.3 Arduino Code - Speed Control

```
    #include "DueTimer.h"
2
    #define COUNTS_PER_ROTATION 211.2
4   #define pi 3.1416

6   //**********************************
    int count_q = 0;      // quadrature decoder hardware counts
8   int spd = 10; // speed
    int dir = 8;  // direction 'High' - negative counts (CCW), "LOW" - positive Counts (CW)
10  int pos = LOW;
    //****************Intializing*****************
12  float count_q_prev = 0;
    float V_prev = 0;
14  float V_prevprev = 0;
    float rad_err_prev = 0;
16  float Prev_rad_rev;
    float Spd_Err_prev = 0;
18  float Spd_Err_prevprev = 0;

20  float rad_Input;
    float rad_Output;
22  float rad_Err;
    float V_req_pwm;
24  float rad_rev;
    float Spd;
26  float Spd_Input;
    float Spd_Output;
28  float Spd_Err;

30  //****************Voltage*****************
    //Maximum motor voltage
32  float V_max = 12;
    //Minimum motor voltage
34  float V_min = 0;
    float V = 0;          // set initial voltage to zero
36
    void setup() {
38
      Serial.begin(9600); //115200 or 9600
40    analogWriteResolution(12);    // set DAC output to maximum 12-bits
```

```
  Timer3.attachInterrupt(update).start(10000); // start ISR timer3 (not used by quad decoder
     and PWM) at 1 ms, which is 1kHz
  pinMode(spd, OUTPUT);
  pinMode(dir, OUTPUT);

  // 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;
}

void loop() {
 count_q = REG_TC0_CV0;
 rad_rev = (2*pi*(count_q))/COUNTS_PER_ROTATION; // [rad]
 Spd_Output = (rad_rev - Prev_rad_rev)/(0.01); //[rad/s]
 Prev_rad_rev = rad_rev;

 Serial.println(Spd_Output);
 // analogWrite(DAC0, Spd_Output)

}

void update() {

  count_q = REG_TC0_CV0;
  rad_rev = (2*pi*(count_q))/COUNTS_PER_ROTATION; // [rad]

  Spd_Input = 50; //[rad/s]
  Spd_Output = (rad_rev - Prev_rad_rev)/(0.01); //[rad/s]

  Spd_Err = Spd_Input - Spd_Output;

  // Controller/Difference Equation
 // V = Spd_Err - 0.9999*Spd_Err_prev + 1*V_prev; // Difference equation kp =1, ki=0.1
 // V = Spd_Err - 0.9995*Spd_Err_prev + 1*V_prev; // Difference equation kp =1, ki=0.1

 // Used tau - pole after design
  V = 0.8337*Spd_Err + 0.0008333*Spd_Err_prev - 0.8329*Spd_Err_prevprev + 0.3333*V_prev +
    0.6667*V_prevprev; // Difference equation kp =1, ki=0.1

  Spd_Err_prev = Spd_Err;
  Spd_Err_prevprev = Spd_Err_prev;
  Prev_rad_rev = rad_rev;
  V_prev = V;
  V_prevprev = V_prev;

  pos  = LOW; //  Positive counter values

if (V < V_min) {
      pos  = HIGH; // if input voltage is negative flip direction
    }

  digitalWrite(dir, pos); // direction can change dependent on V

  V_req_pwm = fabs(V)*(4095/12); // Voltage must remain postive for pwm

  if (V_req_pwm> 4095)
  {
    V_req_pwm = 4095;
  }
```

```
108   analogWrite(spd, V_req_pwm);

110
   //Serial.println(Spd_Output);
112
}
```

Code/Speed_control.ino