

Lab 6 – Discrete Control Lab

Submitters:

Or Shaul

Saray Sokolovsky

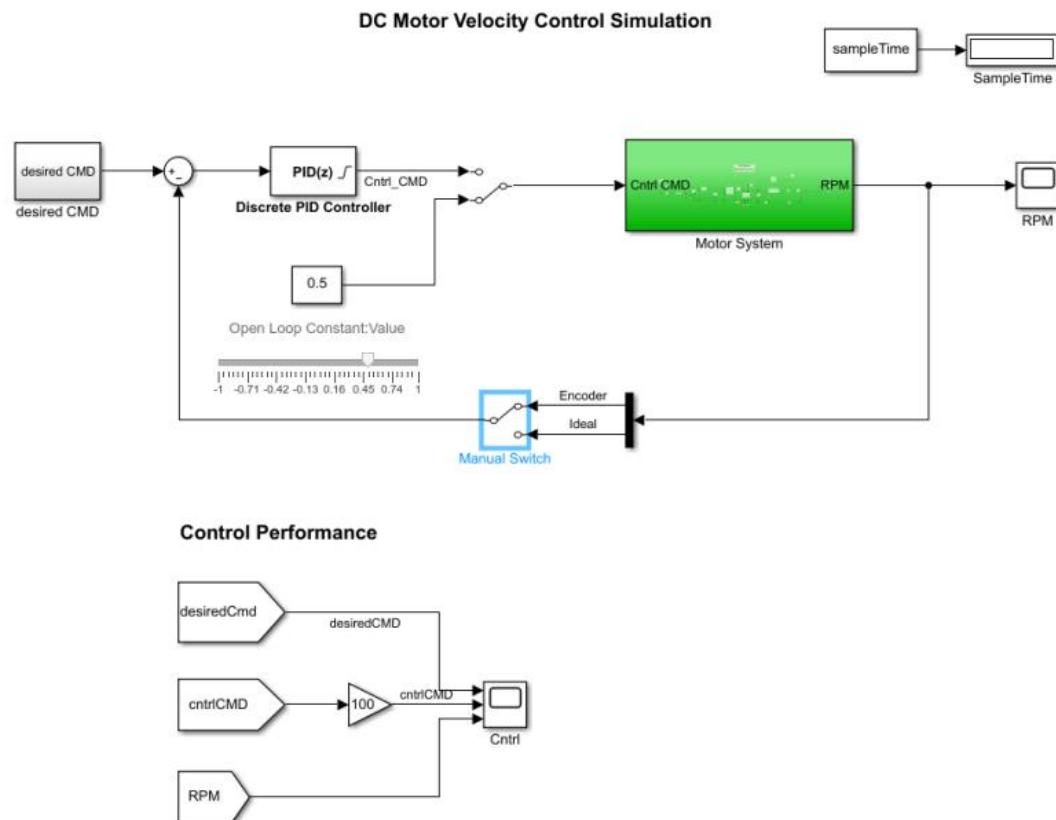
Introduction:

Modern control systems are realized using a micro controller unit (MCU) and implemented in code. Since an MCU is a discrete system, the control implementation is also a discrete control system. In this experiment, we implemented a simple motor control system using an Arduino development board with the Arduino development environment (IDE).

Discrete Control Part 1 – Simulation

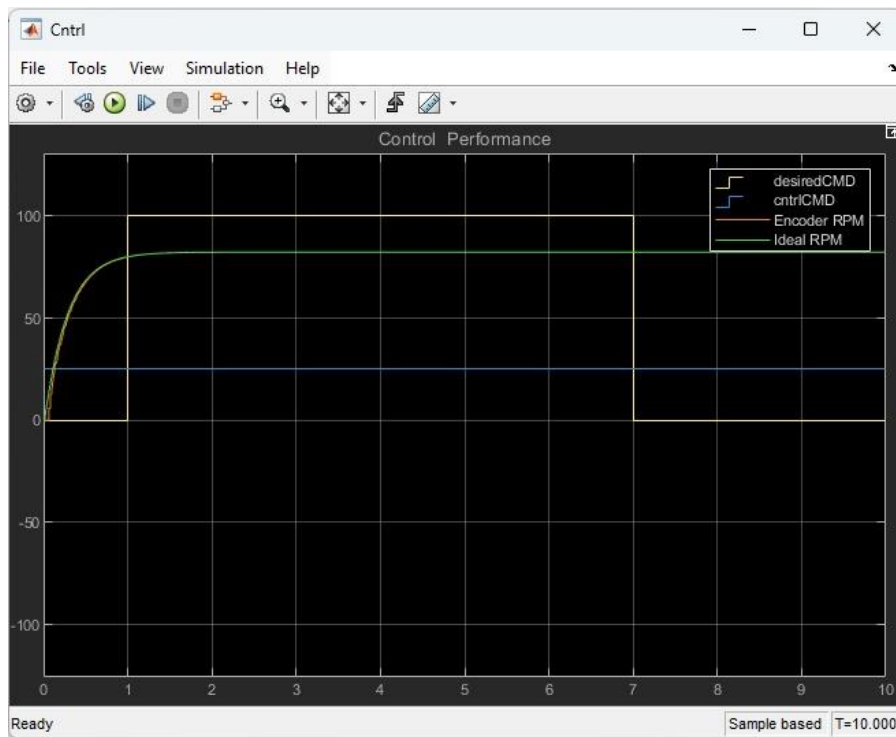
Throughout this section we are going to explore the concepts of a DC motor system and learn the implementation of a discrete velocity motor control by employing a simulation in Simulink. We will get familiar with the concepts of a sampled system and its influence on the control performance.

3.1 Matlab Simulation review, Open loop

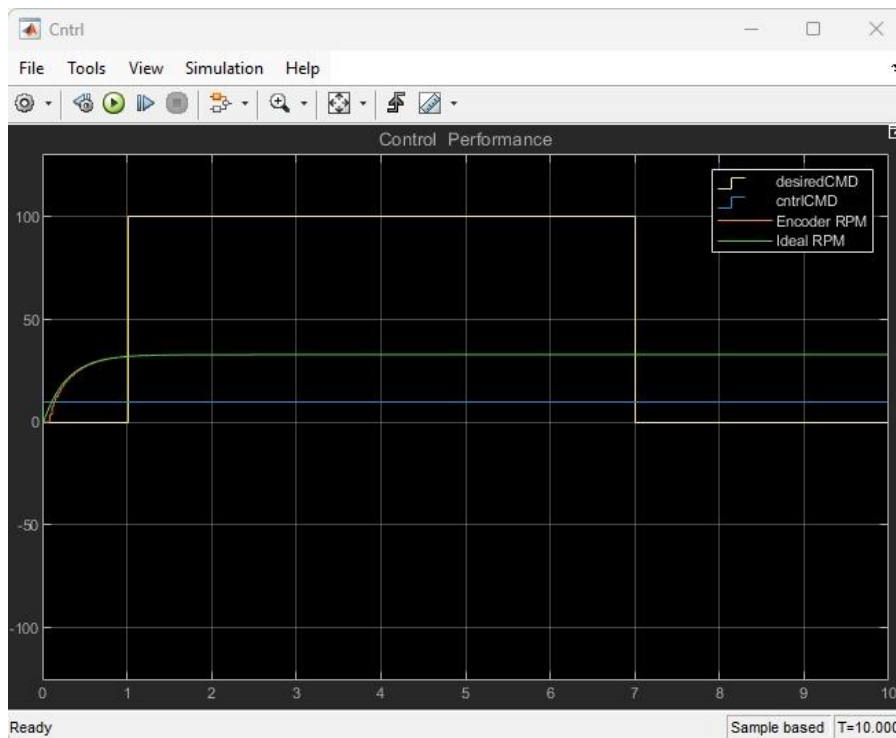


Running the model in open loop with various motor command values:

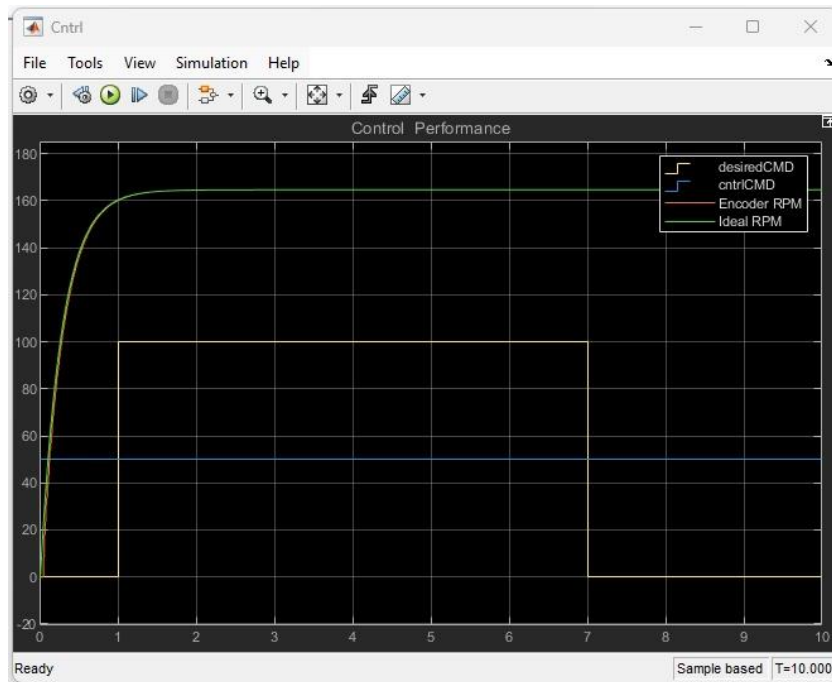
- Open loop with motor command at 0.25 and sample time of 0.01 [sec]:



- Open loop with motor command at 0.1 and sample time of 0.01 [sec]:



- Open loop with motor command at 0.5 and sample time of 0.01 [sec]:

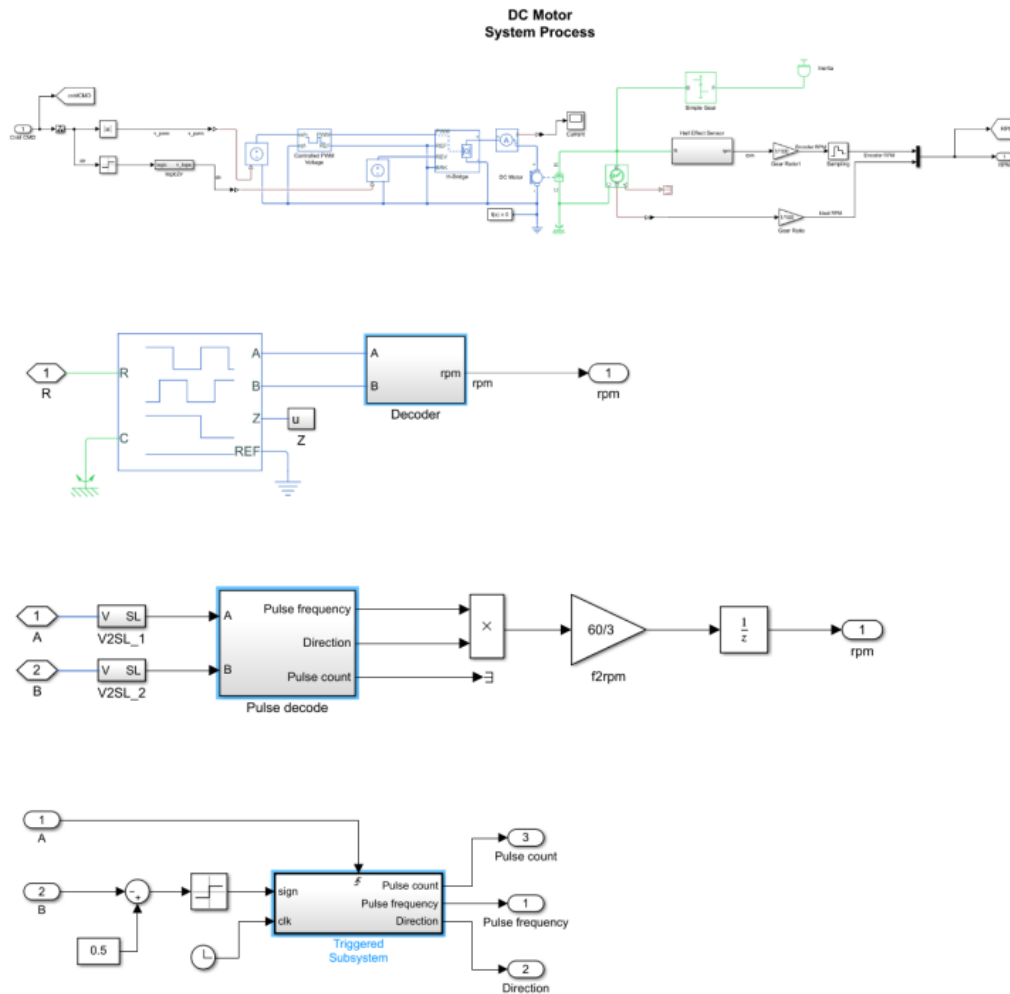


- Open loop with motor command at 0.25 and sample time of 0.1 [sec]:



We notice the jumps in the red line- as expected, we can see that lowering the sampling time causes lower resolution in the Encoder RPM calculations.

The simulation subsystem named Motor System:



The encoder generates pulse signals corresponding to the motor shaft rotation. The decoder interprets these pulses to determine the pulse frequency and direction. The pulse frequency is then converted into RPM using a scaling factor. So overall, the final motor velocity in RPM is obtained by integrating the scaled pulse frequency. The Motor System get a control command as input from the PID controller and outputs the RPM which is the motor velocity. The control command goes through an H-bridge and into the Hall Effect sensor which is used to measure the magnitude of a magnetic field and to provide feedback on the rotor's position. To measure the RPM, we use a Pulse Encoder that generates pulses in response to the rotation on the motor shaft from the Hall Effect Sensor's output. We need to add the gear ratio to calculate the shaft RPM.

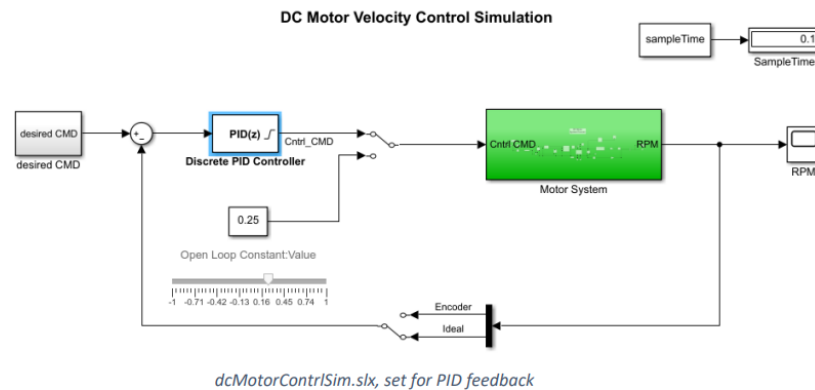
The formula for calculating the RPM is:

$$Motor - RPM = \frac{Encoder\ pulses}{num\ pulses\ per\ revolution} \cdot \frac{60}{Time\ intervals}$$

$$Shaft - RPM = Motor - RPM \cdot Gear - Ratio$$

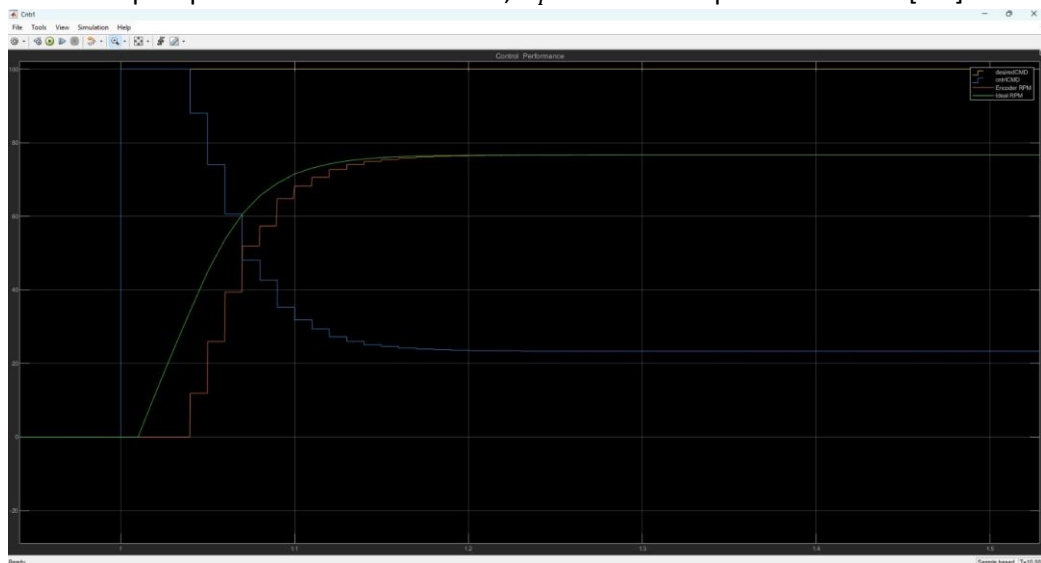
3.2 Close loop proportional control

For this part of the experiment, we will consider the system as a black box - without a known continuous model and learn how to manually tune the controller to achieve the desired system response. First route the model switches to use the signals from the PID controller output, and for the feedback the Ideal RPM value, as depicted in the figure below. Make sure to change back the simulation sample time to 0.01 sec.

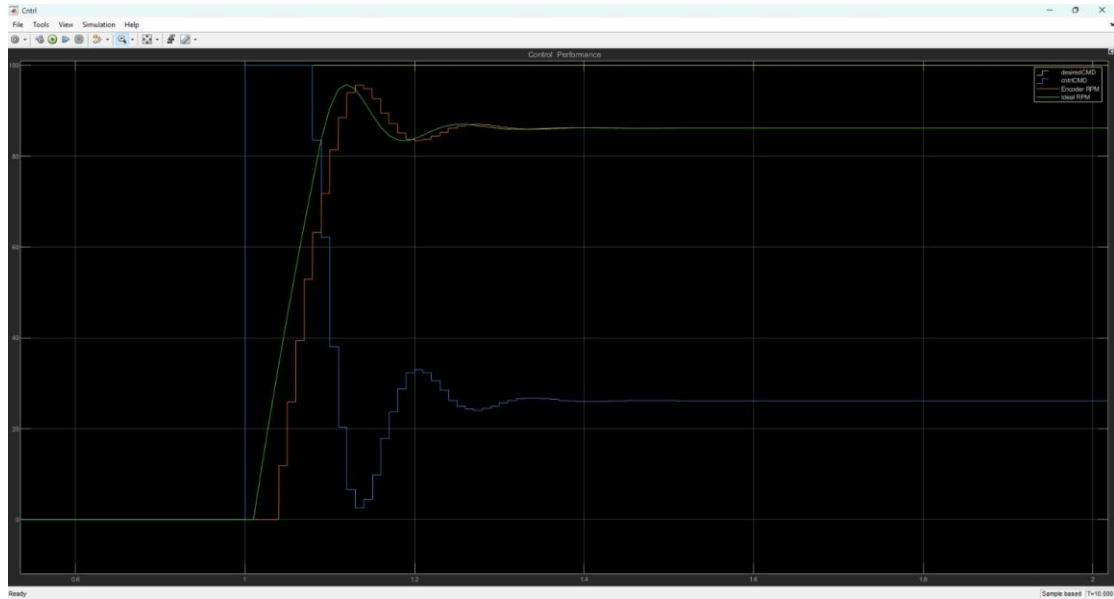


The PID controller was set to proportional gain only $K_P = 0.01$, Running the simulation with various gains:

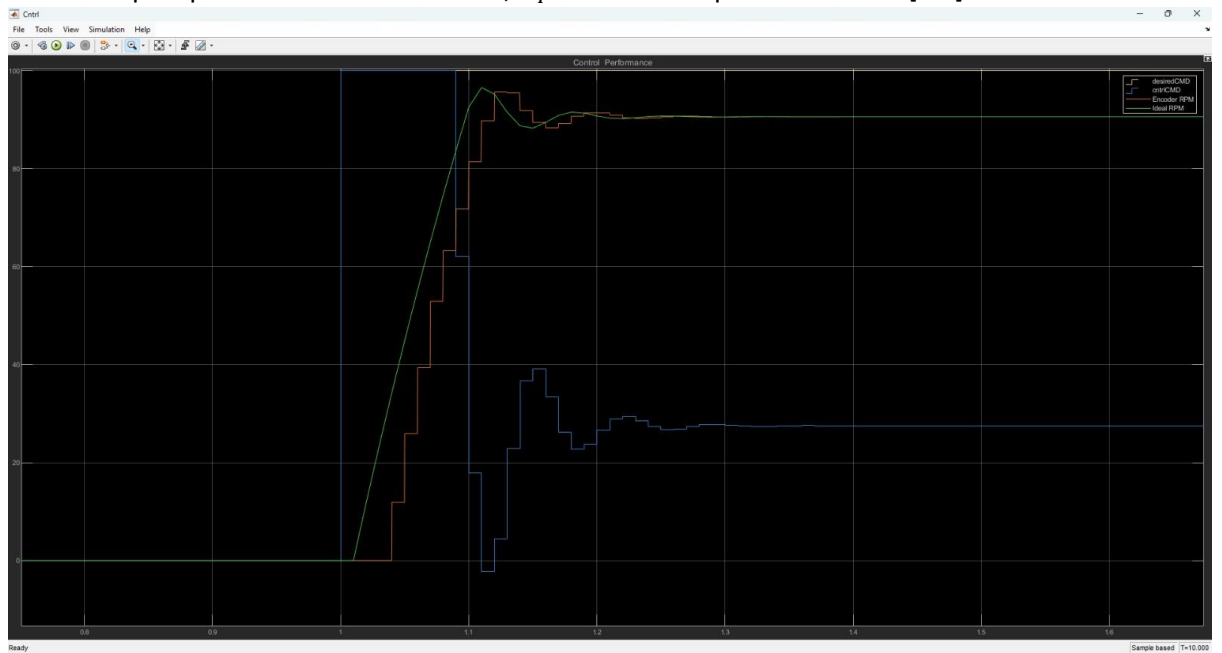
- Closed loop response Ideal RPM feedback, $K_P = 0.01$ sample time of 0.01 [sec]:



- Closed loop response Ideal RPM feedback, $K_p = 0.025$ sample time of 0.01 [sec]:

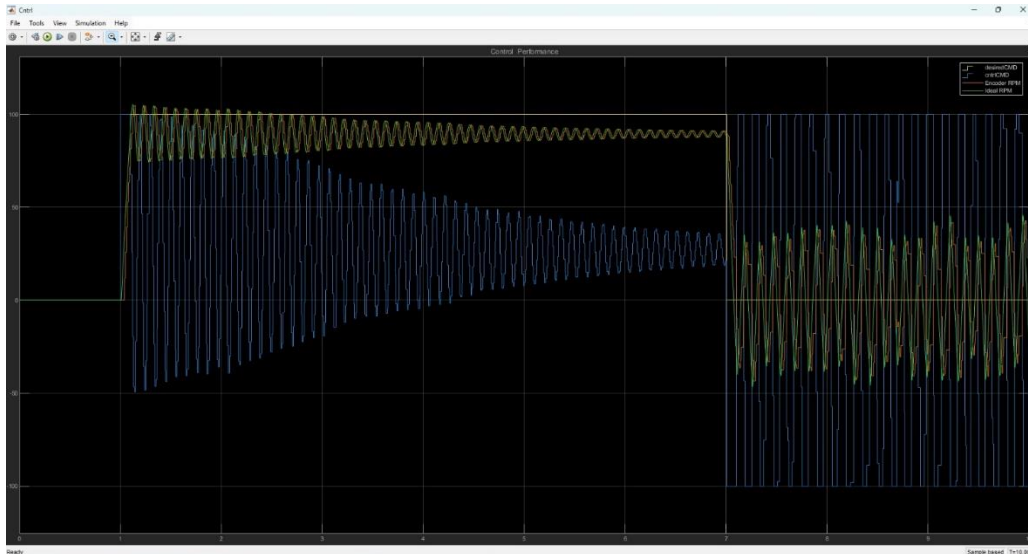


- Closed loop response Ideal RPM feedback, $K_p = 0.05$ sample time of 0.01 [sec]:



We can notice that we are getting a slight overshoot in the closed loop response. Switching back to the Encoder feedback, we can see that the system has lost stability.

- Closed loop response Encoder RPM feedback, $K_P = 0.05$ sample time of 0.01 [sec]:

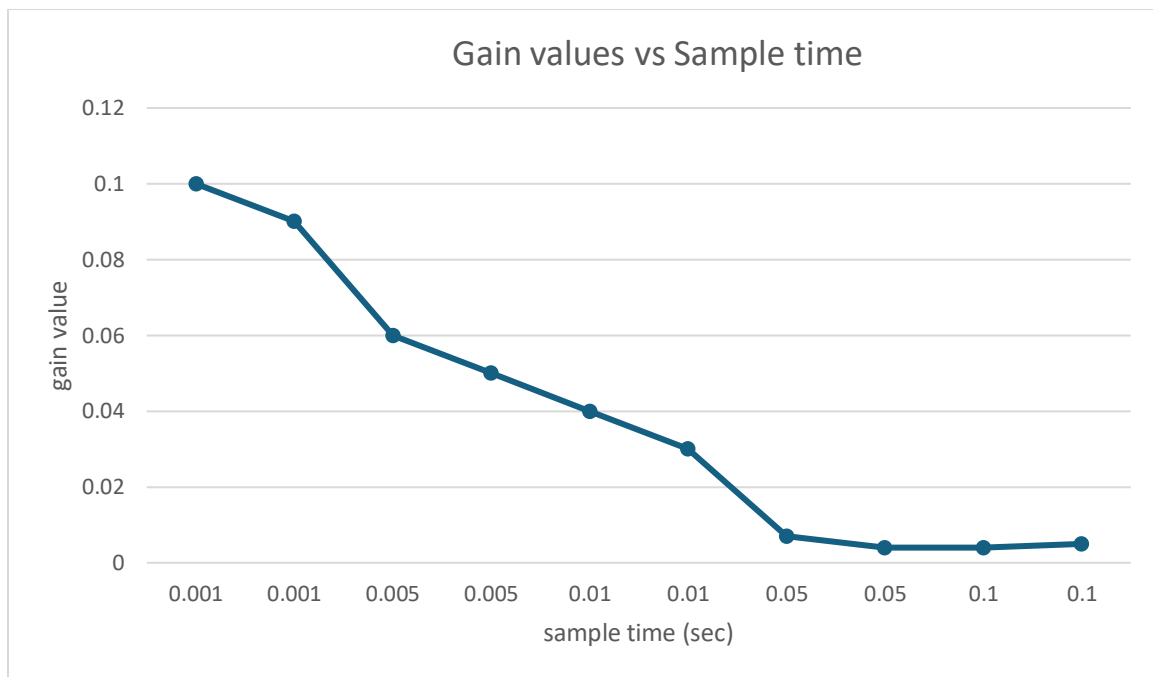


We ran the simulation for various time samples and gain values and the results are:

Sample time (sec)	Gain value	Best gain
0.1	0.005	0.004
0.1	0.004	
0.05	0.004	0.007
0.05	0.007	
0.01	0.03	0.03
0.01	0.04	
0.005	0.05	0.06
0.005	0.06	
0.001	0.09	0.1
0.001	0.1	

For best gain, the parameters were in which gain are we getting a slight overshoot and fast rise time. From the table we can clearly notice that as the sample time decrease, we need to increase the Gain value.

To show that visually we can plot of the gain value vs. sample time:



From this plot, a good rule of thumb is- as the sample time decrease, we should increase the value of the proportional gain in the PID controller.

3.3 Close loop PID tuning

Our objective is to improve the response by adding the integral and differential component of the PID controller. We set the model to encoder feedback and sample Time variable of 0.01sec. We would like to eliminate the steady state error by adding an integral gain value. Tune the system until you get a rise time of 0.1 sec, overshoot of up to 10% and settling time of 0.2 sec.

$$K_p = 0.02, K_i = 0.05$$



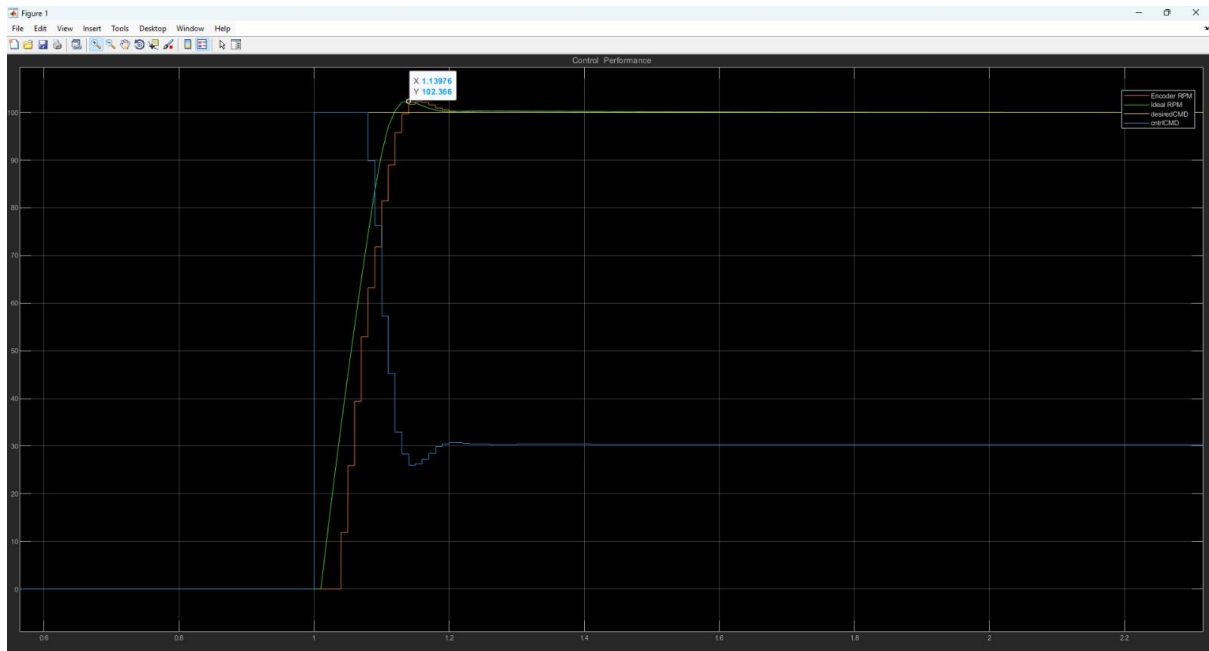
We can see from the graph that we indeed satisfy all the requirements. We can see that the steady state error is indeed eliminated:

$$OS = 6.599\%, t_s = 0.2(sec), t_r = 0.1(sec)$$

We needed to slightly reduce the proportional gain because both k_i and k_p cause an increase in the overshoot and in the settling time. Now we want to decrease the overshoot even more, so we add a differential gain. a good starting point will be:

$$k_d = k_p \cdot Time - sample = 0.02 \cdot 0.01 = 0.0002$$

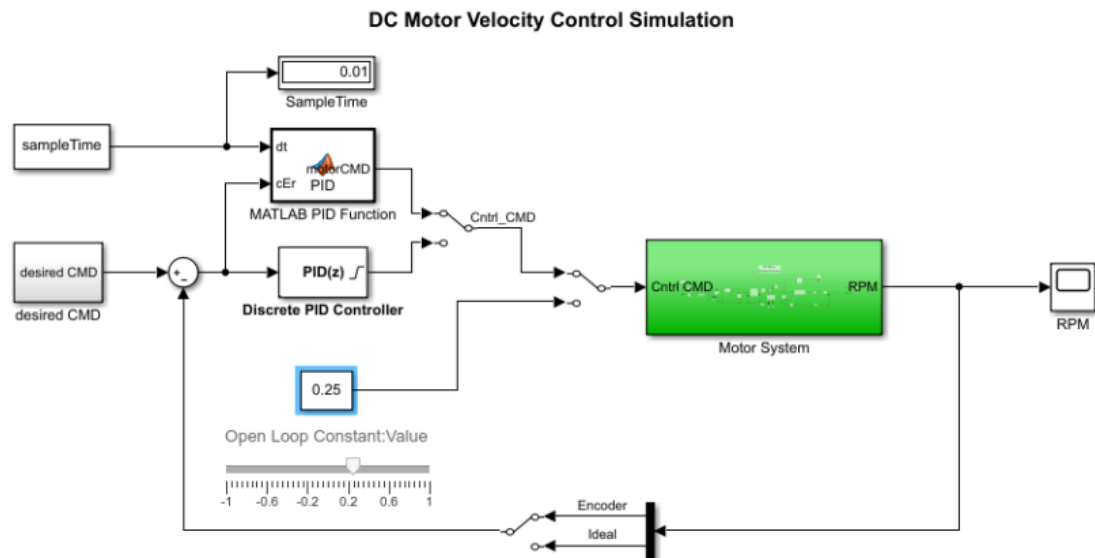
$$k_p = 0.02, k_i = 0.05, k_d = 0.0001:$$



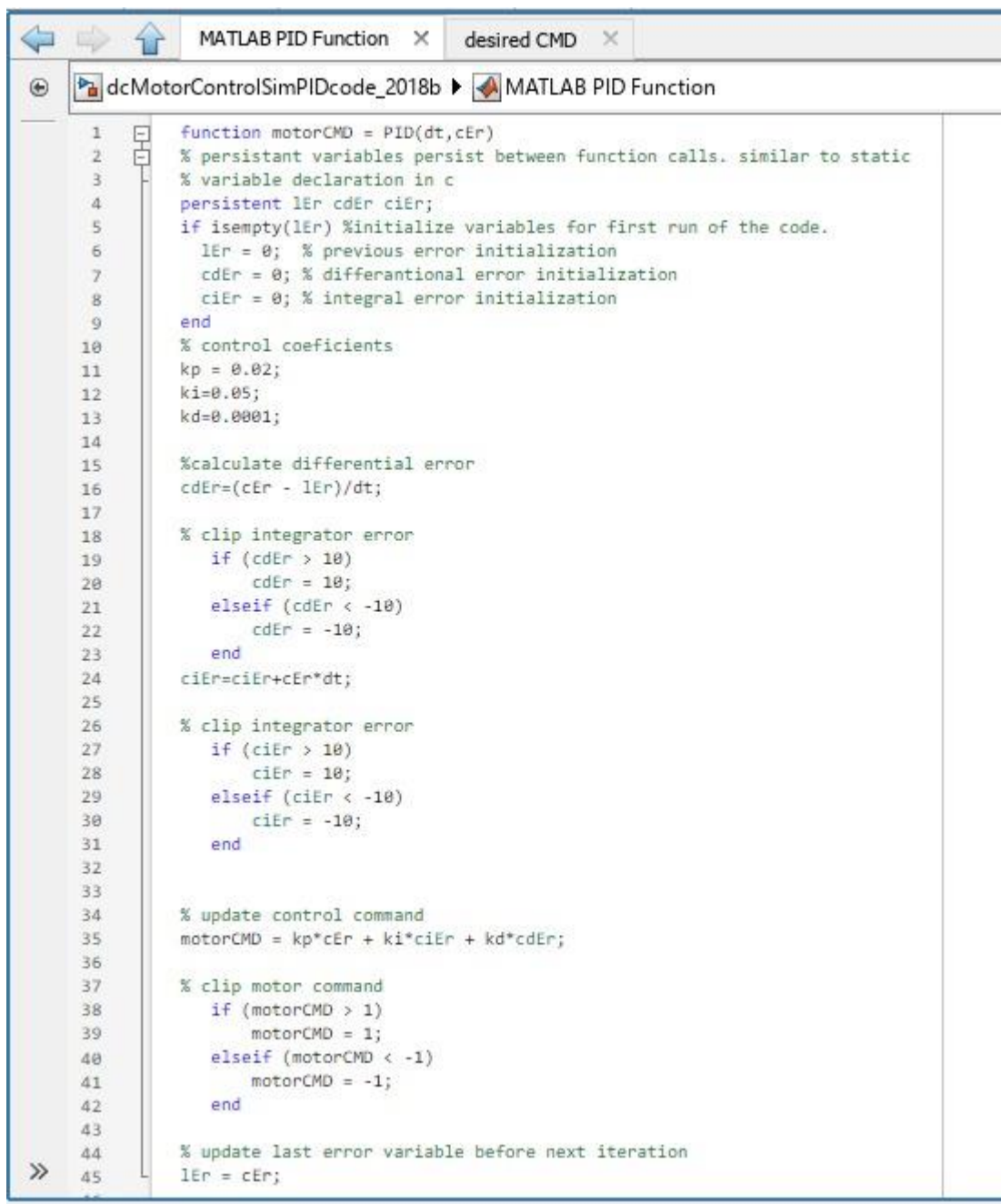
As expected, we now see that the overshoot is lower, O.S = 2.2%

3.4. PID code implementation

For this part we are using Simulink model dcMotorControlSimPIDcode.slx. We route the system Control command to the MATLAB PID function, which has the implementation of proportional gain only.



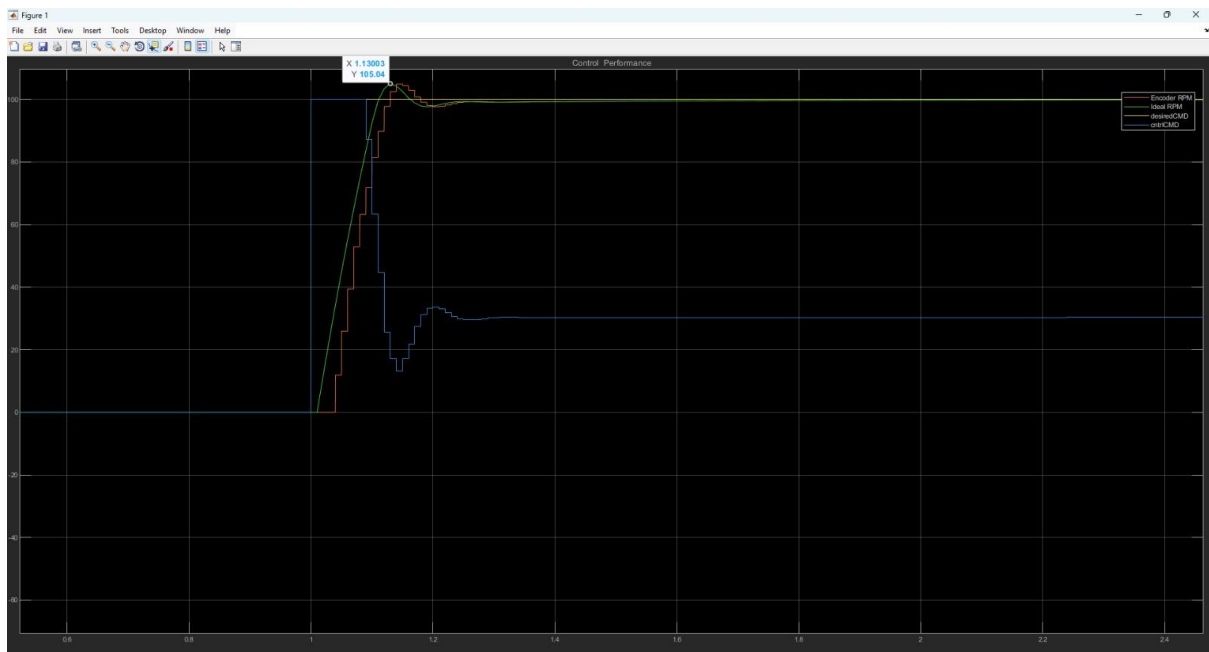
The Updated MATLAB function with k_p , k_i , k_d values:



The image shows a MATLAB editor window with two tabs: 'MATLAB PID Function' and 'desired CMD'. The active tab is 'MATLAB PID Function', which contains a script named 'dcMotorControlSimPIDcode_2018b'. The script defines a function 'motorCMD = PID(dt, cEr)' that implements a PID control algorithm. The code includes initialization for persistent variables (lEr, cdEr, ciEr), control coefficients (kp, ki, kd), and logic for calculating differential error, clipping integrator error, updating the control command, and clipping the motor command. The function is designed to be called repeatedly, with the error variable 'lEr' being updated at the end of each iteration.

```
1 function motorCMD = PID(dt, cEr)
2 % persistent variables persist between function calls. similar to static
3 % variable declaration in c
4 persistent lEr cdEr ciEr;
5 if isempty(lEr) %initialize variables for first run of the code.
6     lEr = 0; % previous error initialization
7     cdEr = 0; % differantional error initialization
8     ciEr = 0; % integral error initialization
9 end
10 % control coefficients
11 kp = 0.02;
12 ki=0.05;
13 kd=0.0001;
14
15 %calculate differential error
16 cdEr=(cEr - lEr)/dt;
17
18 % clip integrator error
19 if (cdEr > 10)
20     cdEr = 10;
21 elseif (cdEr < -10)
22     cdEr = -10;
23 end
24 ciEr=ciEr+cEr*dt;
25
26 % clip integrator error
27 if (ciEr > 10)
28     ciEr = 10;
29 elseif (ciEr < -10)
30     ciEr = -10;
31 end
32
33 % update control command
34 motorCMD = kp*cEr + ki*ciEr + kd*cdEr;
35
36 % clip motor command
37 if (motorCMD > 1)
38     motorCMD = 1;
39 elseif (motorCMD < -1)
40     motorCMD = -1;
41 end
42
43 % update last error variable before next iteration
44 lEr = cEr;
45
```

The response from the code implementation:



We can see that we got a bigger overshoot of 5% instead of 2.2% using the regular PID controller. Also, we can notice that the settling time is larger than before. but the steady state error is still eliminated as before.

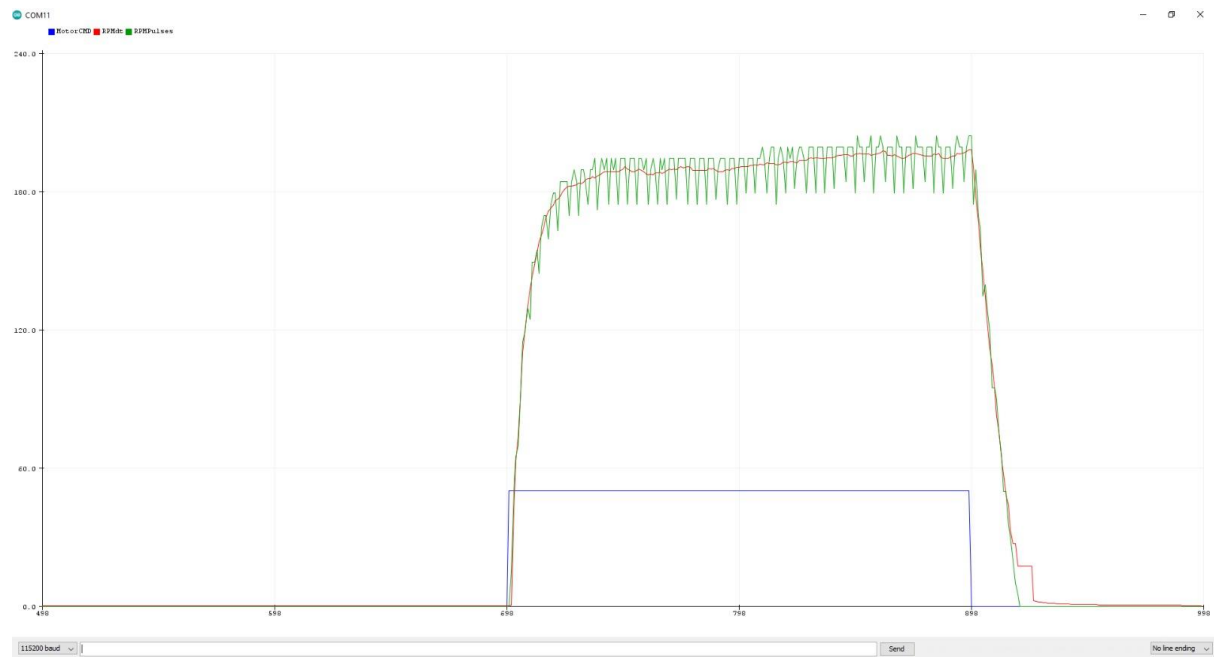
In summary, the first method of the PID controller gives a better response than the MATLAB implementation.

Discrete Control Part 2 – Hardware

4.3 Open loop velocity response & Encoder readings

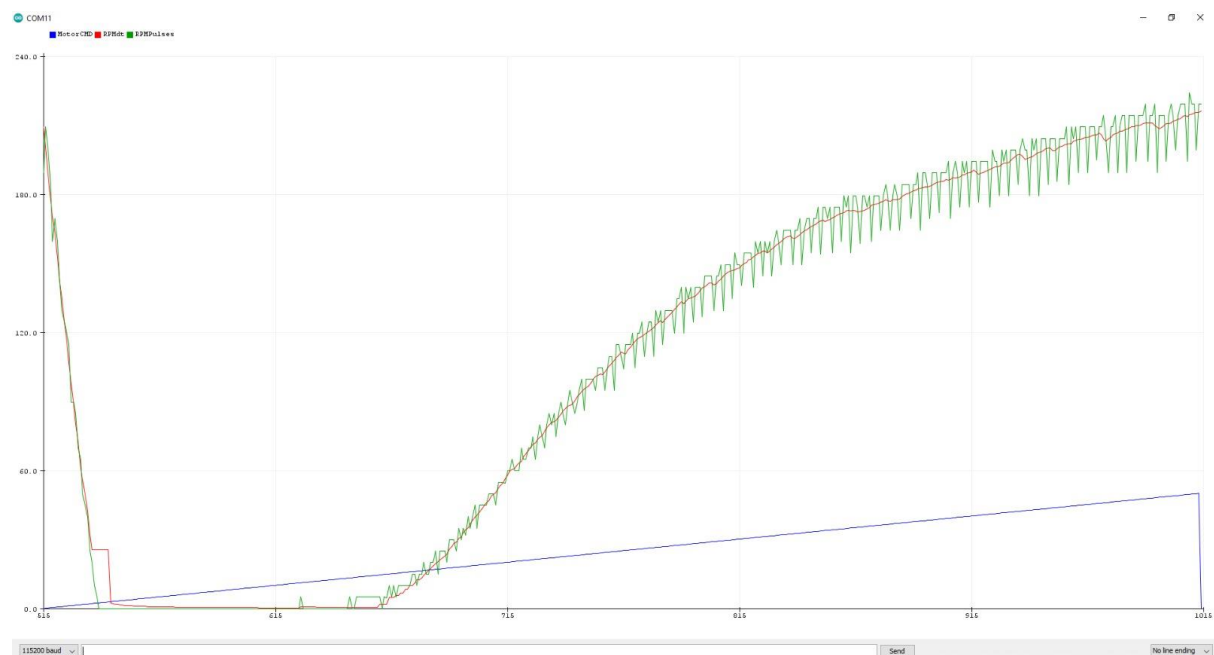
For this part, we worked with ArduMotorControlEx2.ino sketch in Arduino environment. The parameter GEAR_RATIO was updated to be 100.37 in accordance to the gear ration written on the box (1:100.37).

We got that the step response with a gain of 0.5 is:



The RPM in this gain was 200.

Changing the command to the rampCMD by uncommenting this function and with open loop gain of 0.5 we got the following response:



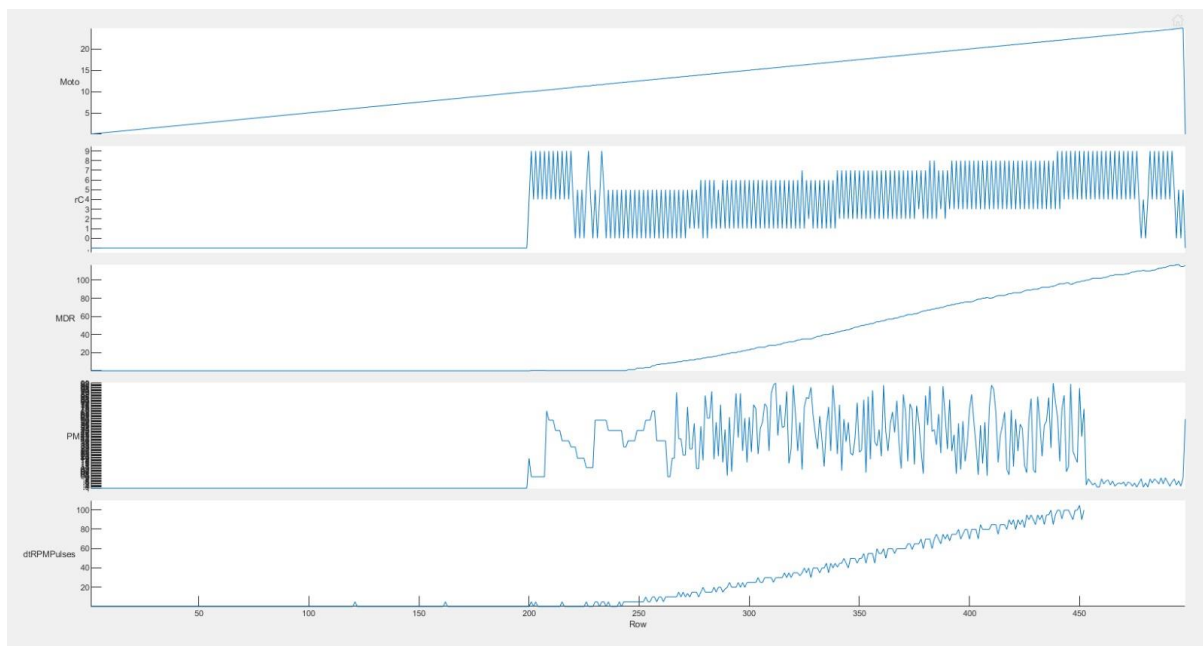
again, as with the step response. the RPM is close to 200 for this gain value.

What is the minimal command for the motor to start moving? we got the for a minimal value of 0.15 gain, the motor started moving.

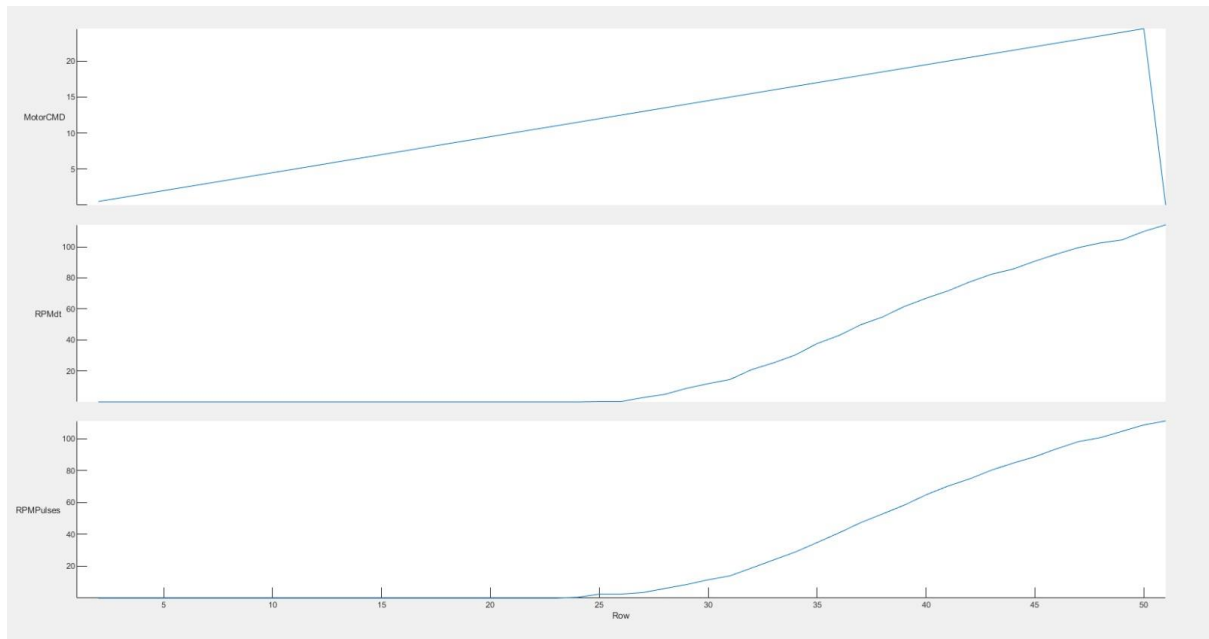
What is the maximum velocity? The maximal motor velocity measured was 250 RPM

What can you say about the two methods for velocity measurements? Which is better at high speeds and which for lower speeds, how the control frequency influences the results? The two methods were, RPMdt and RPM pulses. From the results of the simulation, we observed that for lower speeds, RPM pulses method was better and for high speeds, RPMdt method was more precise and smooth. When the control frequency was set to a lower value, the effect was that the rise time was higher, and when the control frequency was set to a higher value, the effect was that the rise time was lower.

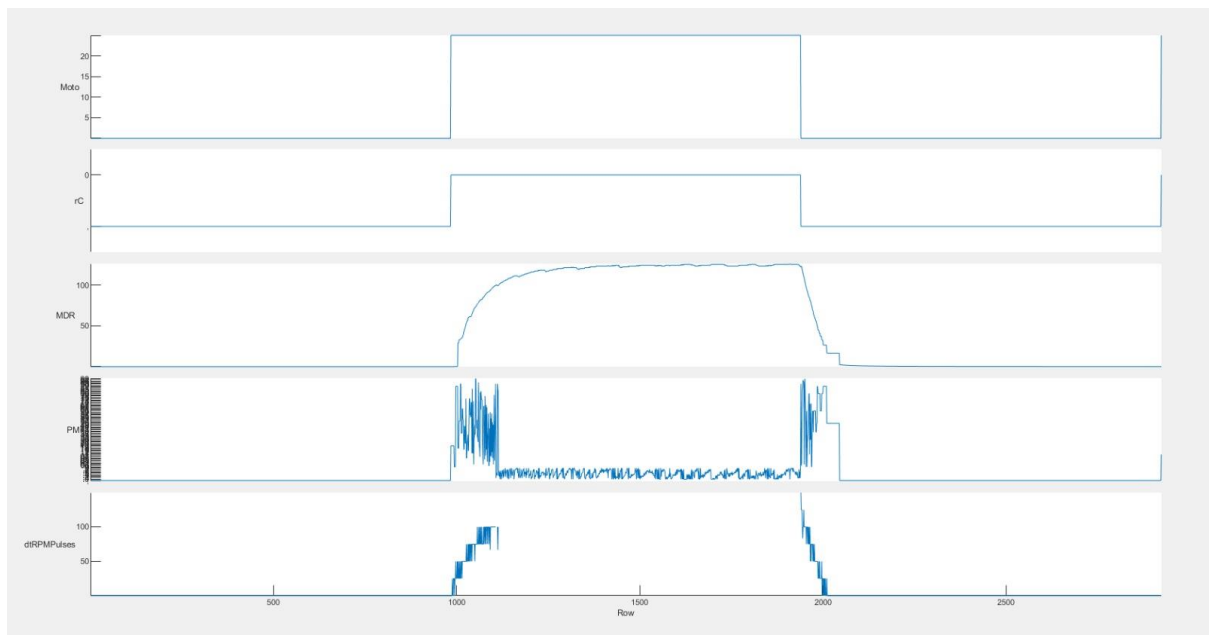
We prepared recordings of a ramp response with period of 5 seconds, gain of 0.25 at two control periods, 10ms and 100ms. By importing the simulation data to MATLAB script we generated plots comparing the measured RPM of both methods. Plot of RPMdt and RPM pulses methods vs MotorCMD for control period of 10 [ms]:



Plot of RPMdt and RPM pulses methods vs MotorCMD for control period of 100 [ms]:



We can see that for control period of 100 ms, we got more accurate results for both methods. Now we will plot the same plot for simulation of a step response at a gain of 0.25 and period of 2 seconds.



The rise time is measured from a value of 10% to 90% . From the plot we got that the rise time is:

$$T_{10\%} = 1000[ms]$$

$$T_{90\%} = 1400[ms]$$

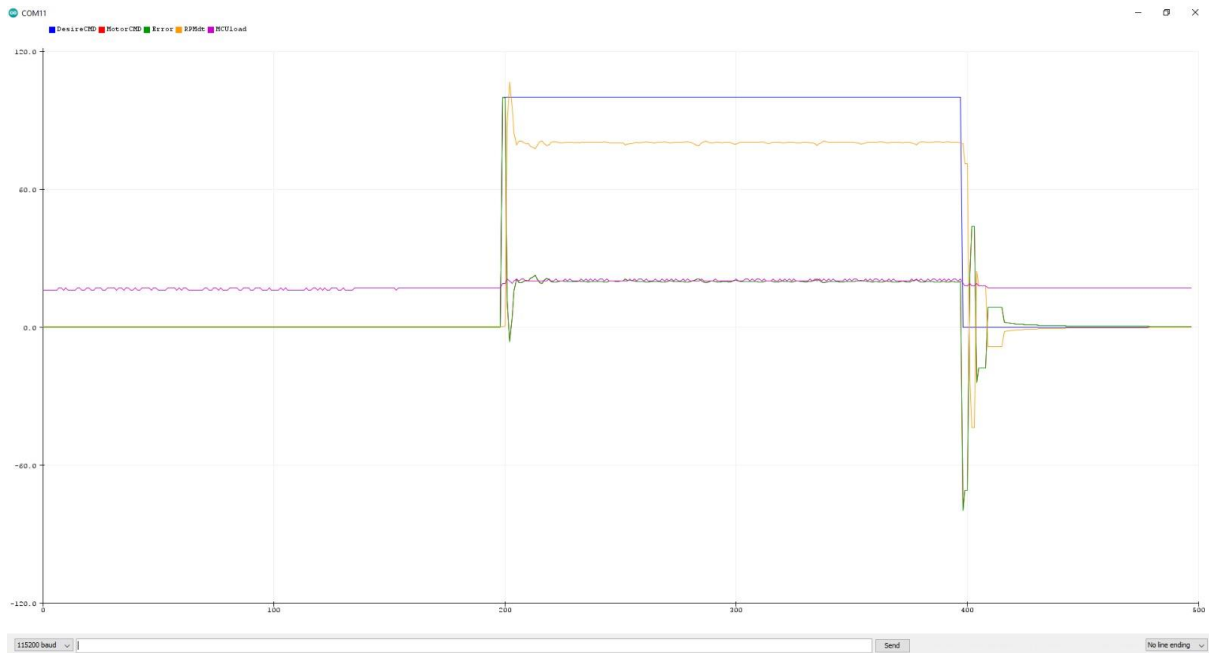
Therefore, the rise time is:

$$T_{rise} = T_{90\%} - T_{10\%} = 1400 - 1000 = 0.4[ms]$$

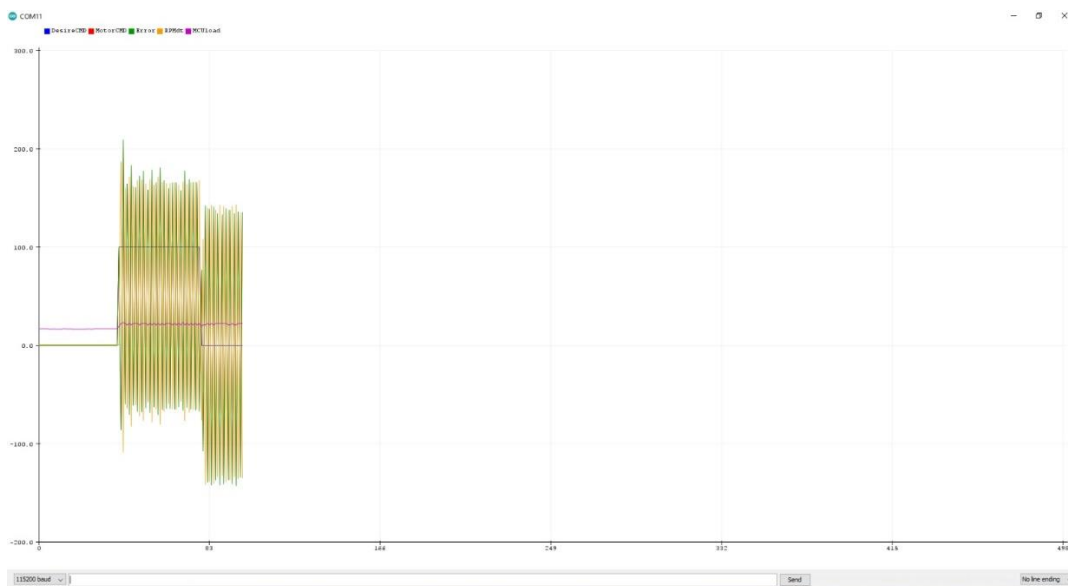
4.4 Close loop proportional velocity control

For this section we will implement a simple proportional control command and tune it's gain. For this part, we worked with ArduMotorControlEx3.ino sketch in Arduino environment. We ran the simulation for step response of approximately 1/3 of the maximum velocity at open loop. Using the hint, the gain was set to $K_p = 0.01$

The response we got in the Serial Plotter tool for DT_CONTROL_MILLIS 10:



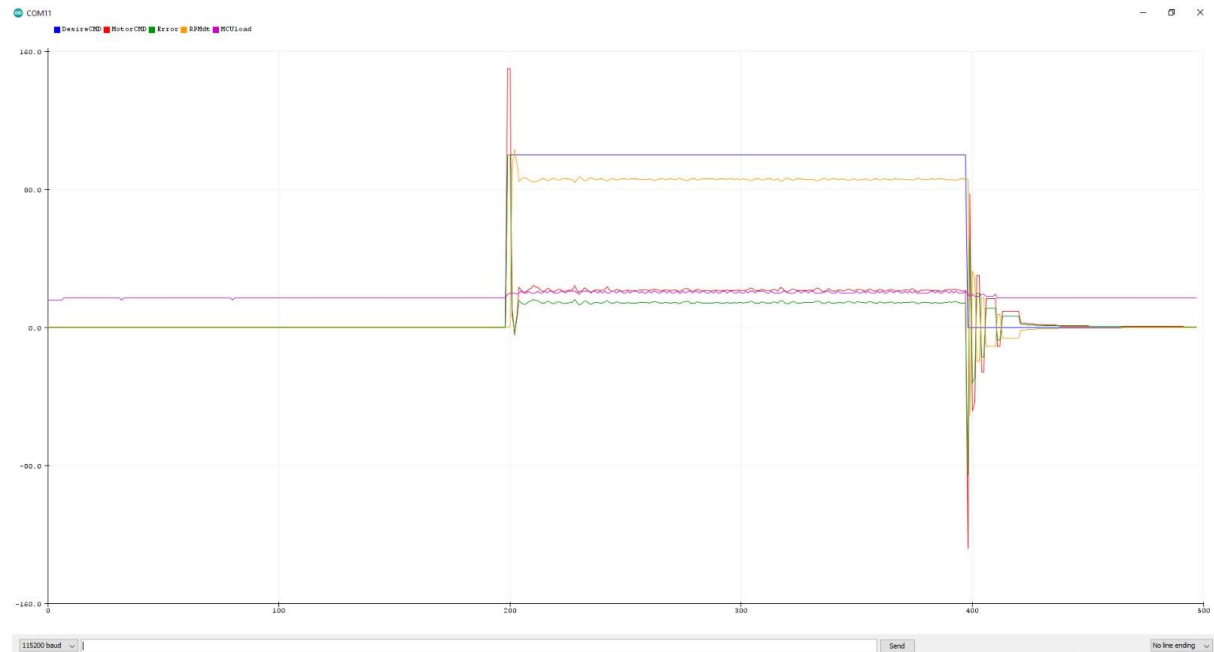
The response we got in the Serial Plotter tool for DT_CONTROL_MILLIS 50:



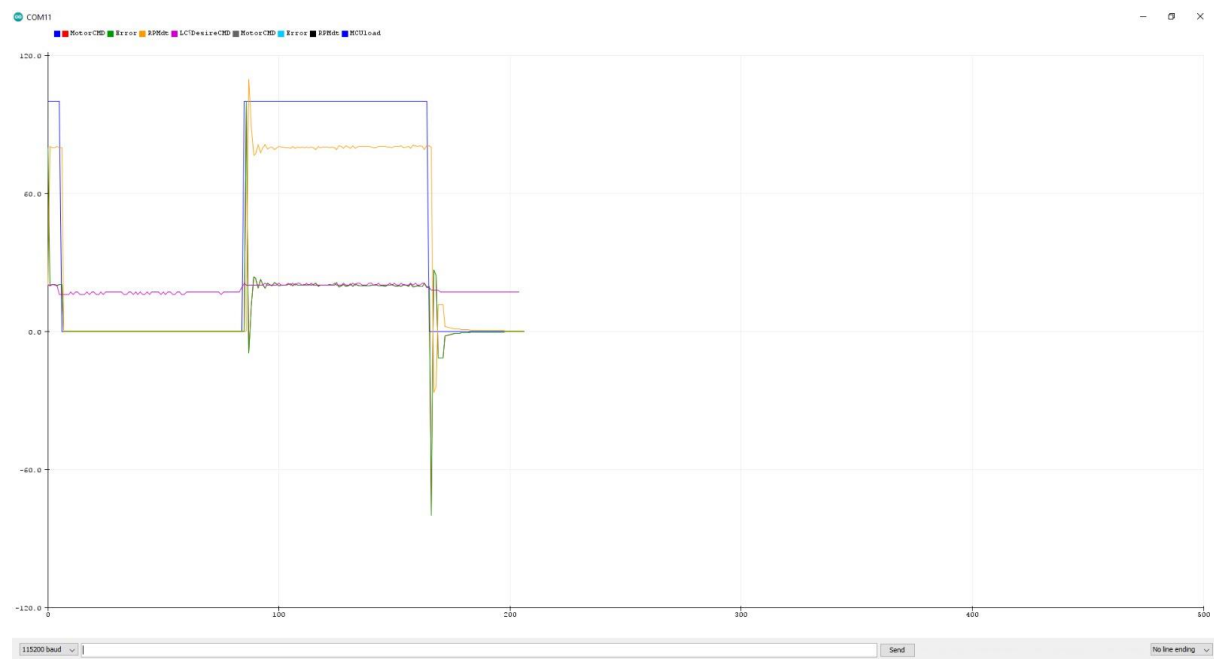
Meaning we are not stable for the same gain value.

Running the simulation for 10/25/50 control frequencies, we adjusted the gain value until the response was stable:

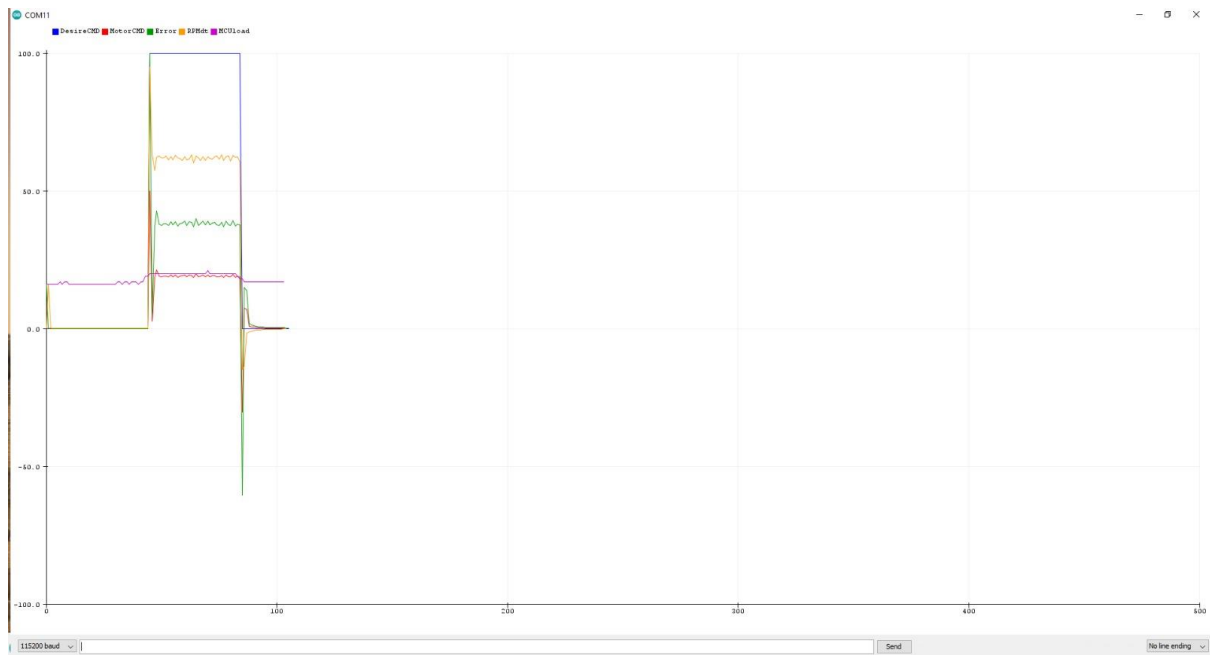
For 10 - $K_p = 0.01$



For 25 - $K_p = 0.015$

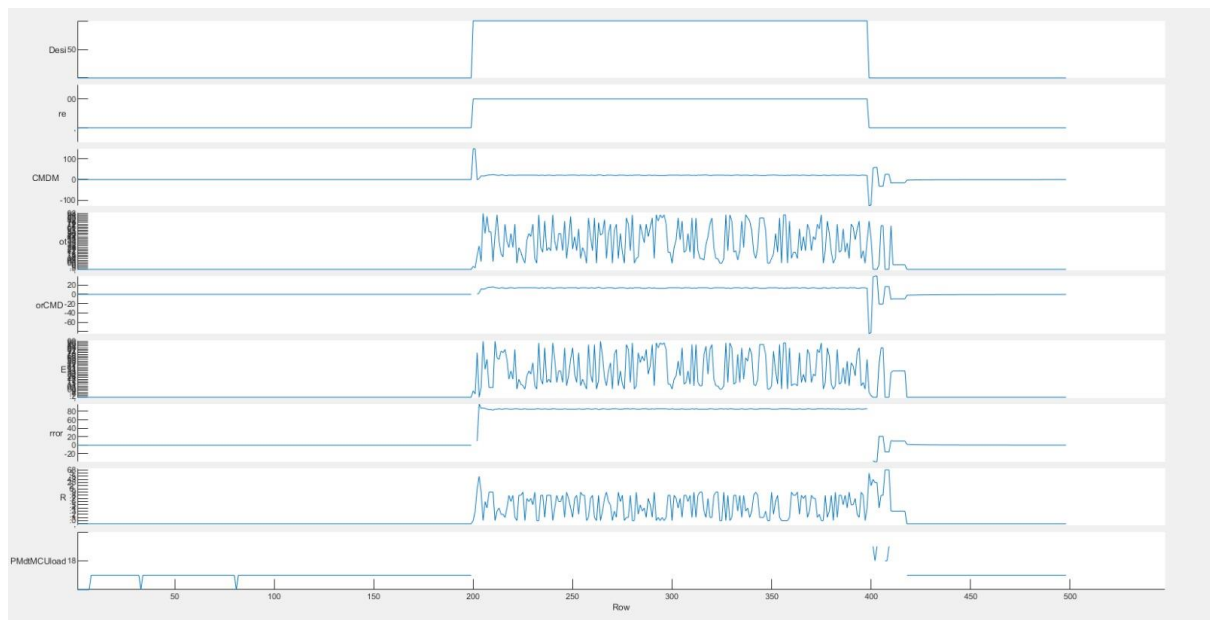


For 50 - $K_p = 0.02$

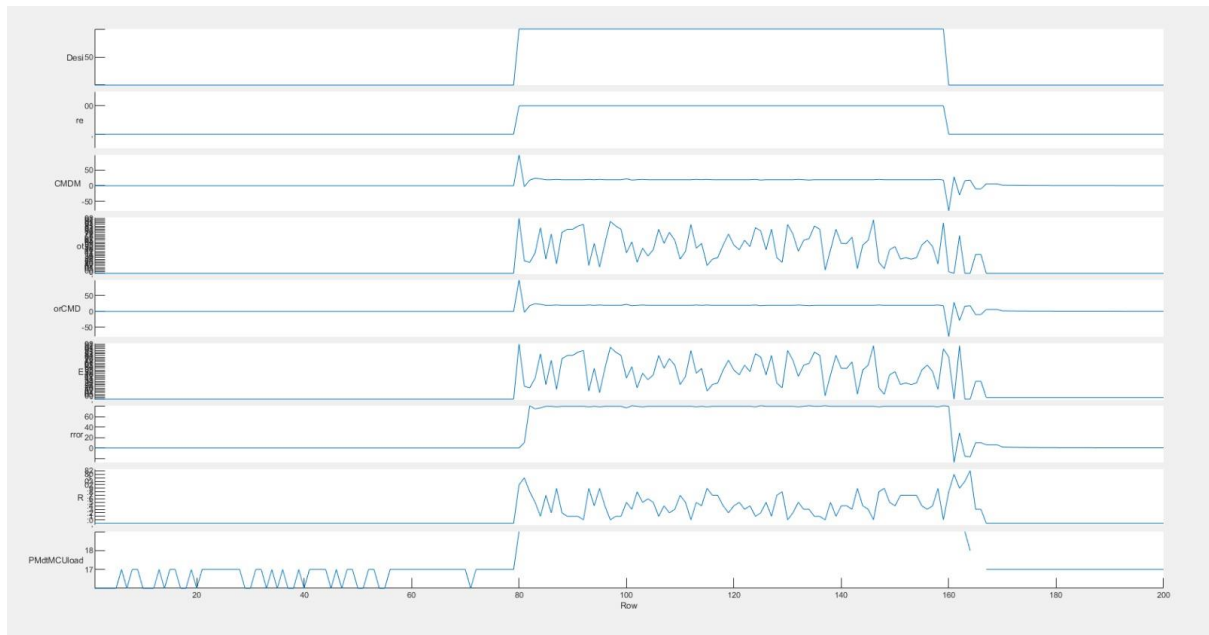


generate a plot containing the 3 step responses generated previously and determine the rise time for each.

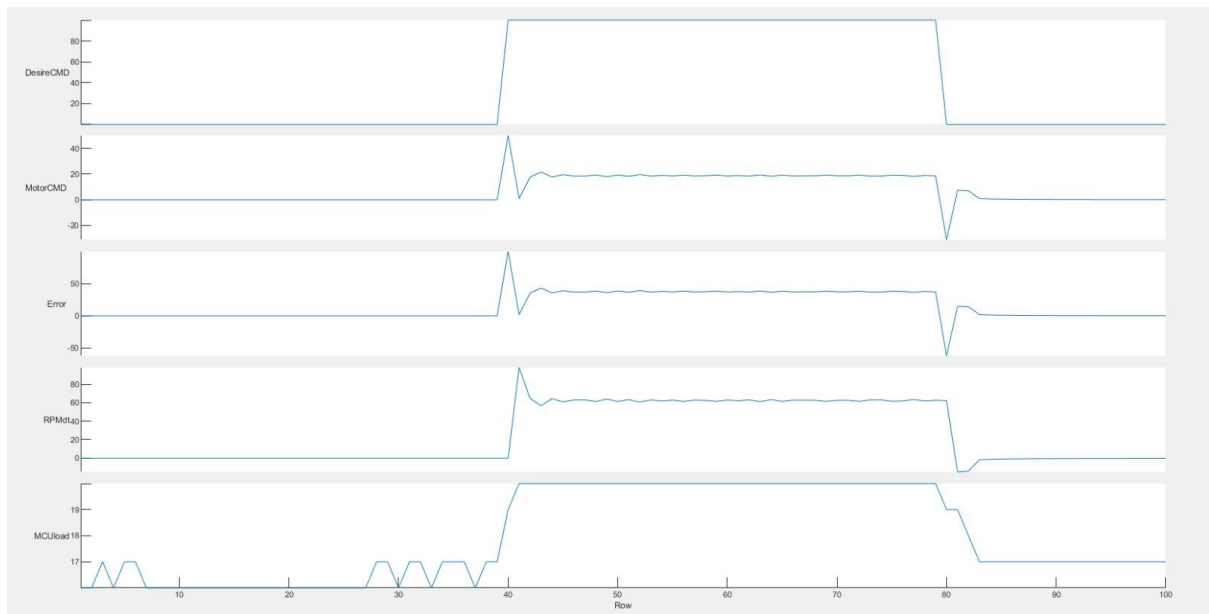
step response for DT_CONTROL_MILLIS 10:



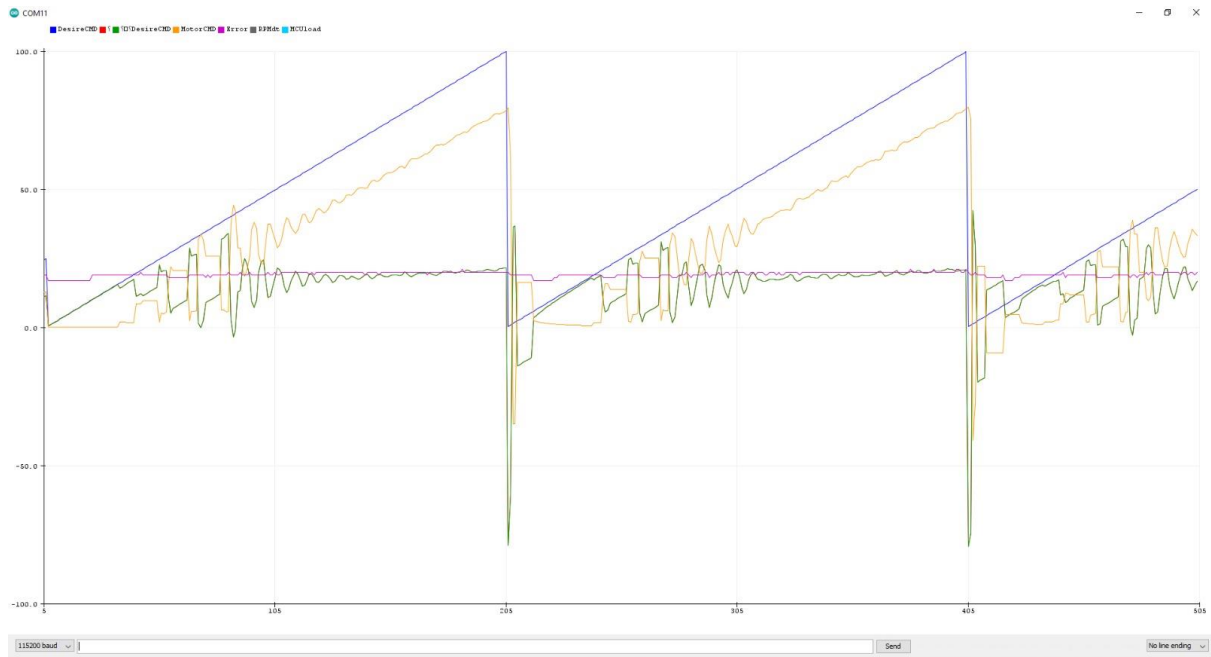
step response for DT_CONTROL_MILLIS 25:



step response for DT_CONTROL_MILLIS 50:



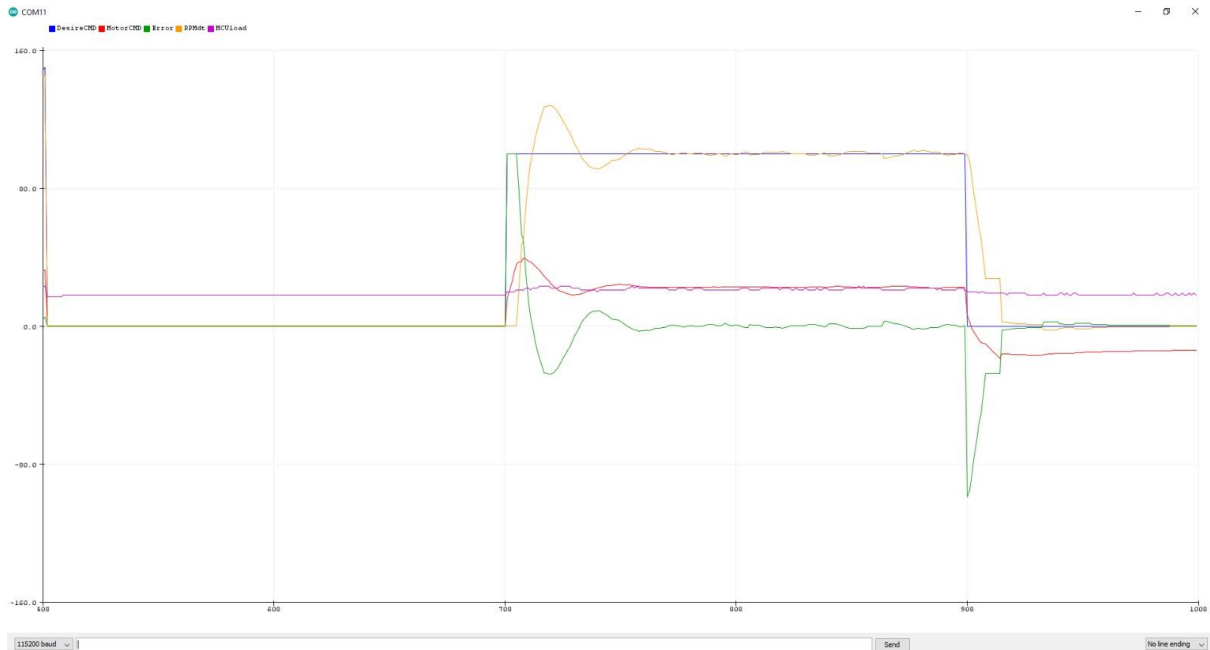
Adjust the proportional gain value until a steady response is achieved for the ramp command like the plot below. We will be using this value 0.01:



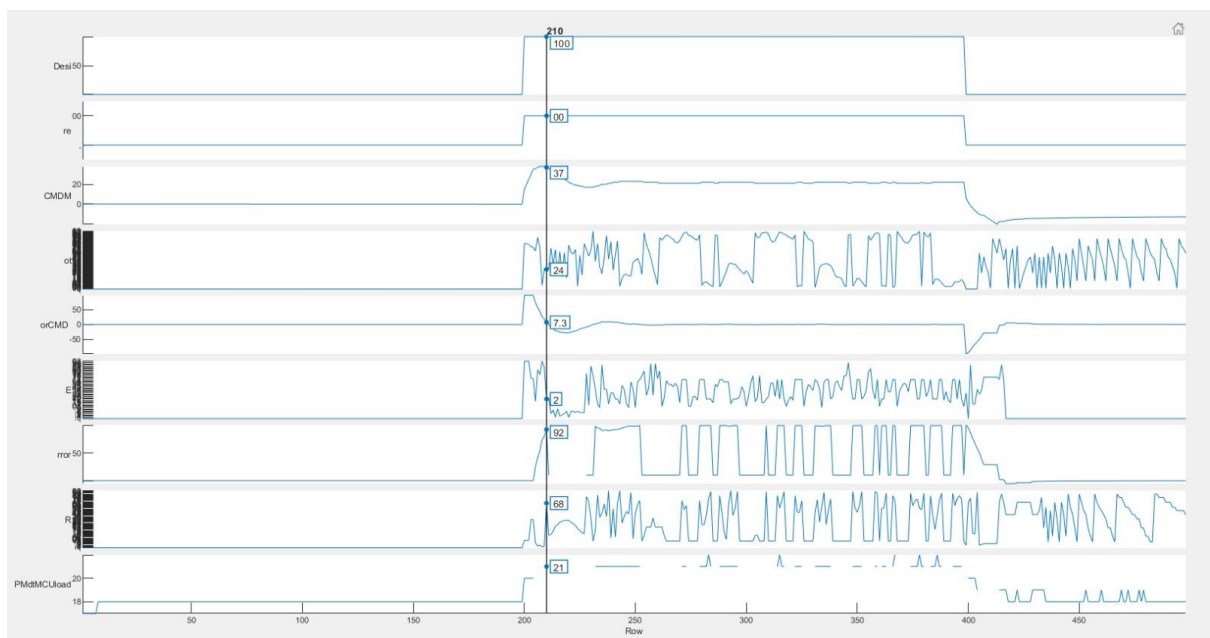
4.5 Close loop PID velocity control

For this section we opened the ArduMotorControlEx4.ino sketch in the Arduino environment. The values were tuned until a slight overshoot was achieved for a step response of $\sim 1/3$ open loop maximum velocity and control interval of 10 milliseconds.

$$k_p = 0.01, k_i = 0.05, k_d = 0$$



MATLAB step response for DT_CONTROL_MILLIS 10:

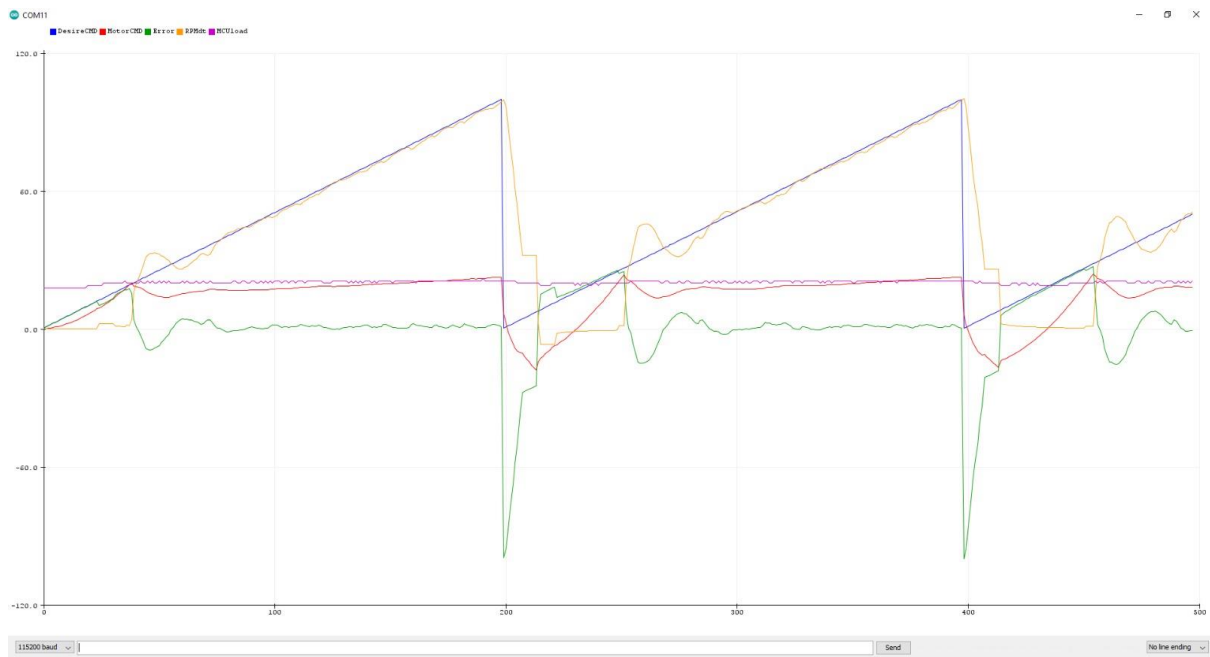


overshoot: 15%

settling time: 0.25 [sec]

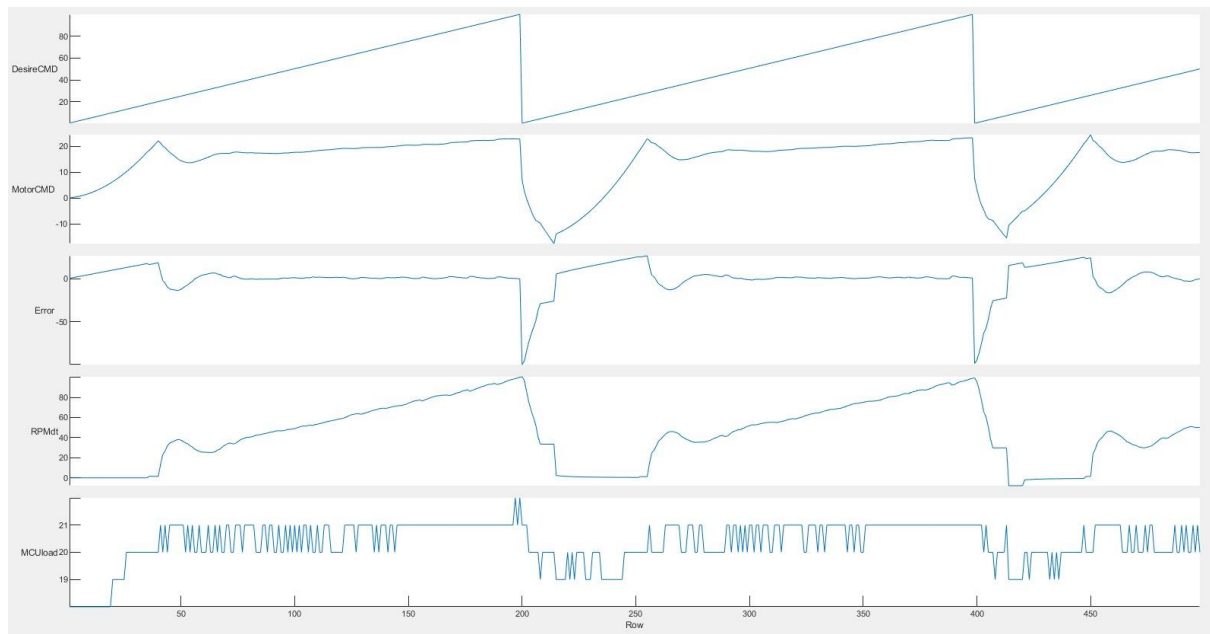
rise time: 0.1 [sec]

Now, running the ramp response for the same values we get:



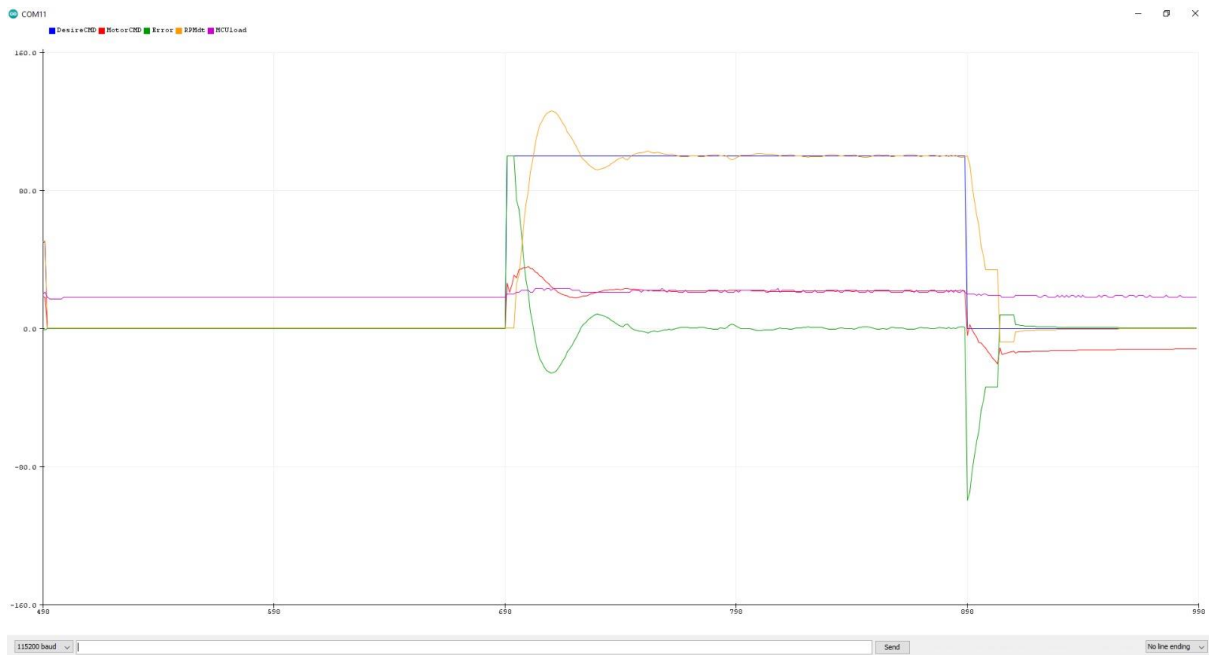
From this plot we can see that at the start of the ramp, we are below the minimal voltage needed to rotate the motor and overcome the static friction, therefore we can see that the data for RPMdt remains constant and not affected from the ramp. Until we get to the minimal voltage required to rotate the motor and then there is a good overlap between the RPMdt and DesireCMD.

Plotting the imported data in MATLAB we get:

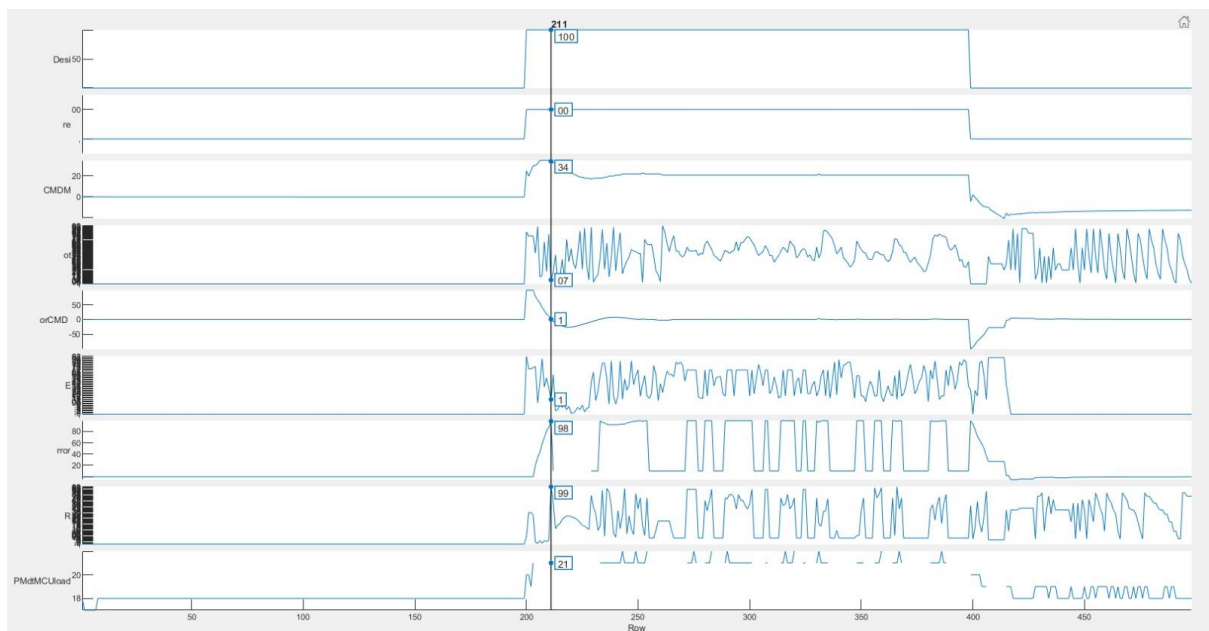


Step response with the values:

$$k_p = 0.01, k_i = 0.05, k_d = 0.00001$$



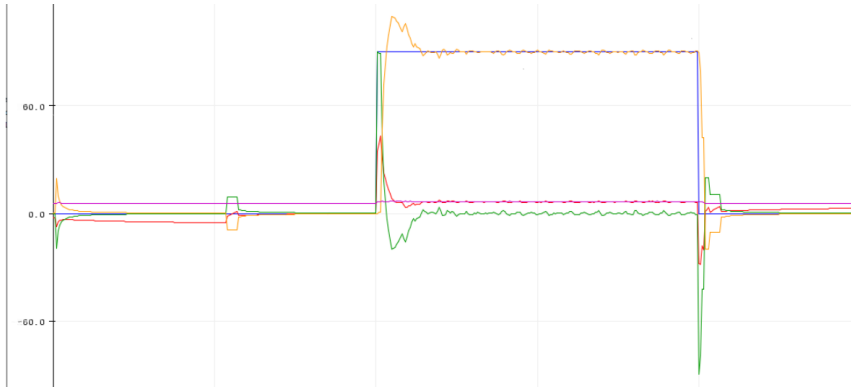
Plotting the imported data in MATLAB we get:



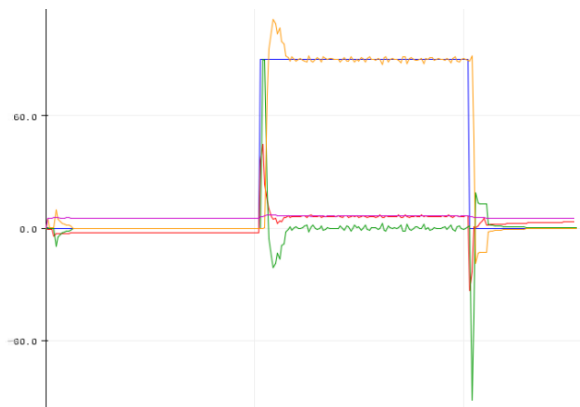
We can clearly notice that now the error is almost zero as expected.

4.6 Control loop frequency influence on control performance

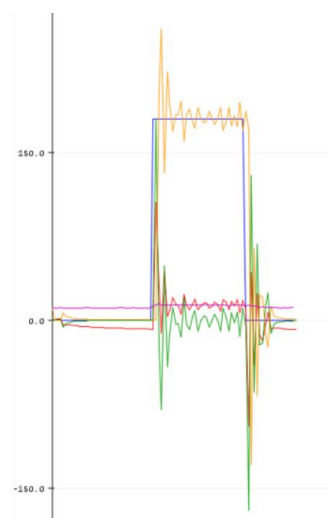
For this section we will use the PI coefficients found at the previous section with the differential gain set to zero. (PI Controller). We ran a series of step responses at $\sim 1/3$ open loop maximum velocity at various Control intervals `DT_CONTROL_MILLIS`. Starting with 10ms and increasing the value until we lose stability. Step response for `DT_CONTROL_MILLIS` 10:



Step response for `DT_CONTROL_MILLIS` 30:



Step response for `DT_CONTROL_MILLIS` 60:



For The MATLAB step response for DT_CONTROL_MILLIS 10, DT_CONTROL_MILLIS 30 and DT_CONTROL_MILLIS 60 we didn't had enough time to create them in the lab. So unfortunately, we can't know for sure the overshoot, settling time and rise time but we will approximate based on the Arduino graphs:

For DT_CONTROL_MILLIS 10 - overshoot: 15% settling time: 0.25 [sec] rise time: 0.1 [sec]

For DT_CONTROL_MILLIS 30 - overshoot: 20% settling time: 0.25 [sec] rise time: 0.1 [sec]

For DT_CONTROL_MILLIS 60 - overshoot: 45% settling time: there in none rise time: 0.1 [sec]

We can assume that for control period of 60 [ms] we lose stability. the overshoot is significant, and we don't have a settling time. For a control period of 10 [ms], we can assume and see that it has the smallest overshoot and a faster settling time. Therefore, we will suggest control loop frequency is 10.