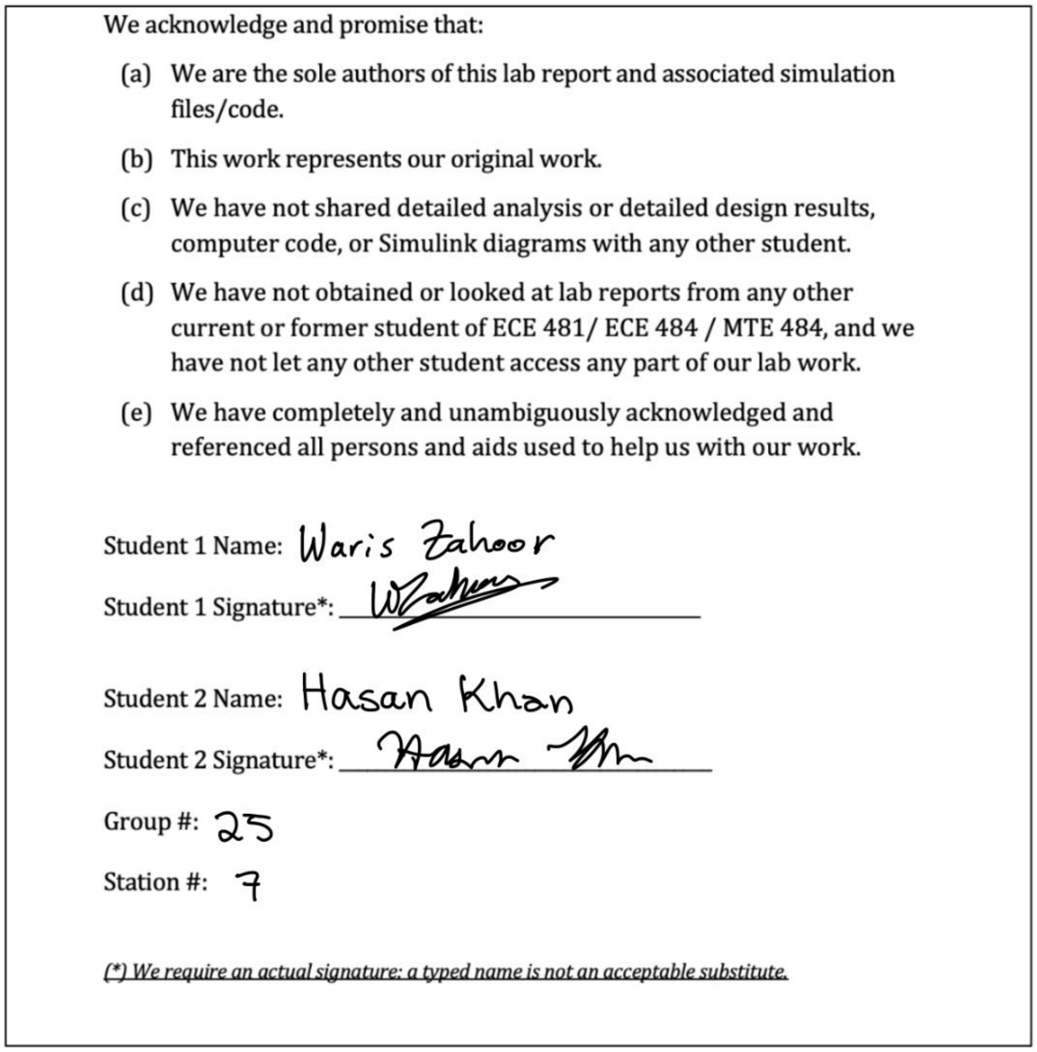
Lab 2:
Design of the Inner-Loop Controller & Modelling of the Ball & Beam

MTE 484: Control Applications

Hasan Khan, 20928382

Waris Zahoor, 20951829

November 9th, 2025



We acknowledge and promise that:

(a) We are the sole authors of this lab report and associated simulation files/code.

(b) This work represents our original work.

(c) We have not shared detailed analysis or detailed design results, computer code, or Simulink diagrams with any other student.

(d) We have not obtained or looked at lab reports from any other current or former student of ECE 481/ ECE 484 / MTE 484, and we have not let any other student access any part of our lab work.

(e) We have completely and unambiguously acknowledged and referenced all persons and aids used to help us with our work.

Student 1 Name: Waris Zahoor

Student 1 Signature*: [illegible]

Student 2 Name: Hasan Khan

Student 2 Signature*: [illegible]

Group #: 25

Station #: 7

(*) We require an actual signature: a typed name is not an acceptable substitute.

Sampling Time: 0.02764604299 seconds

Gear angle scaling :

- Slope (m): -0.012727 rad/count
- Intercept (b): 5.128981 rad

Stiction offsets:

- Positive direction: -0.3 V
- Negative direction: 0.2 V

K1 = 1.998

Tau = 0.022

The DC gain value was cahnged due to issues with a proper controller being able to work on the real-world motor. This issue was faced when moving from Simulink & Matlab to the real controller. To compat this the accurate measurement of the plant was taken to better model a controller that would be more accurate to the plant.

Part A

Before creating the discrete-time controller $D_1[z]$, we must first determine a proper sampling to properly capture the motor's dynamics. We will base the sampling time off the τ_m as seen from Equation 1 below.

$$Equation\ 1: \quad G(s) = \frac{K_1}{s(\tau_m s + 1)}$$

This τ_m value represents how quickly the system responds to changes in the input. Using the K_1 and τ_m that was found in Lab 1 we get the following:

$$G(s) = \frac{1.998}{s(0.022s + 1)}$$

After doing partial fraction decomposition we then find the following:

$$G(s) = \frac{1.998}{s} + \frac{1.998}{s + 45.4545}$$

From here the maximum bandwidth is taken which is the largest real plant pole that exists.

$$w_{bw} = \max|Re(plant\ poles)|$$
$$w_{bw} = 45.4545$$

To get the get the optimal sampling frequency we must multiply the w_{bw} as seen in Equation 2 below.

$$Equation\ 2: \quad f_s = 5 * w_{bw}$$
$$f_s = 5 * 45.4545 = 227.2725$$

The to get sampling time we use Equation 3 to get the optimal sampling time used for the controller $D_1[z]$

$$Equation\ 3: \quad T_{sampling} = \frac{2\pi}{f_s}$$
$$T_{sampling} = 0.02764\ s$$

Next, G(s) was taken from continuous to discrete time and the partial fraction decomposition was taken. This resulted in the following function below:

$$G[z] = \frac{0.0552367}{z - 1.0000000} - \frac{0.0314457}{z - 0.2846091}$$

After getting the plant transfer function in discrete time, the function *generate_poles()* was used for approximating $W[z]$. From here 30 poles within a radius of 0.85, centered at 0 (within the unit circle of radius 1) were used to generate the respective poles for the system. Generating these random poles allowed for different poles of varying strengths to be generated. These values were then used for the IOP with SPA, which resulted in a primal and dual feasible system after applying the constraints outlined in the simulation part of the lab instructions. From here the 6 poles with the highest strength were chosen to simplify the system and decrease means of instability that could be caused when implementing the controller. These 6 poles can be seen in Figure 1 below.

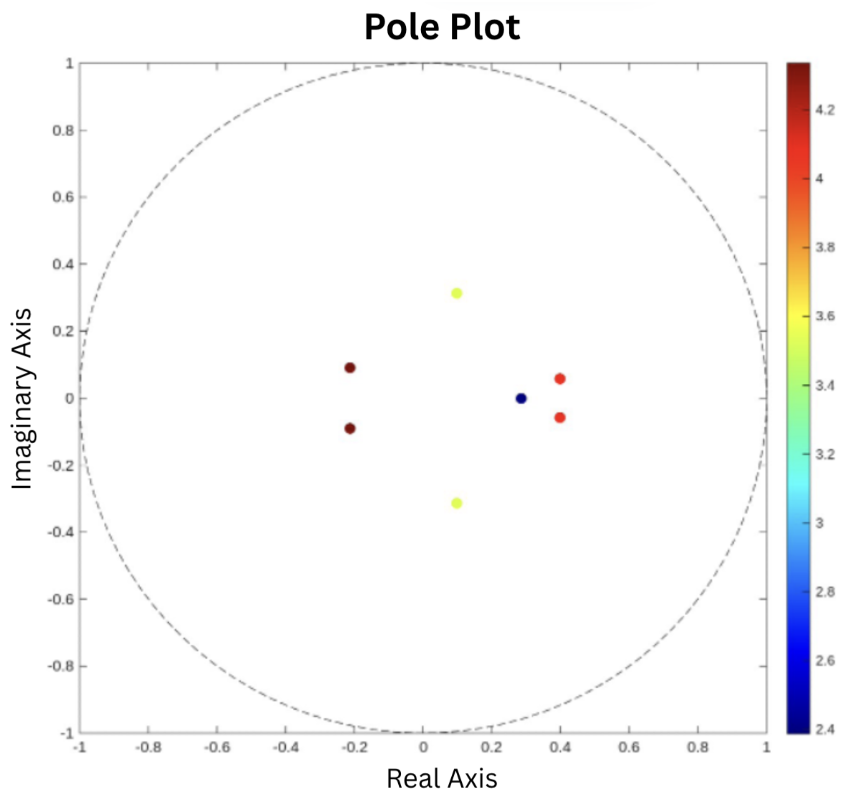


Figure 1. Pole plot diagram using chosen poles.

The poles were specifically:

0.398371042488927 + 0.057450087083452i

0.398371042488927 - 0.057450087083452i

0.097390191562406 + 0.313871232493950i

0.097390191562406 - 0.313871232493950i

-0.213734070429800 + 0.091201683852363i

-0.213734070429800 - 0.091201683852363i

This was specifically done to decrease the settling time so that it would fall within the expected range. The code for IOP with SPA, along with full poles can be found in the Appendix.

The system was able to meet the constraints of

- $-6 < V_{input} < 6$
- $e_{ss} = 0$
- $OS\% = 0$
- $T_{settling} \leq 0.25\ seconds$

The system was verified by plotting the appropriate control signal and position output plot in MATLAB seen in Figure 2 and 3 below.

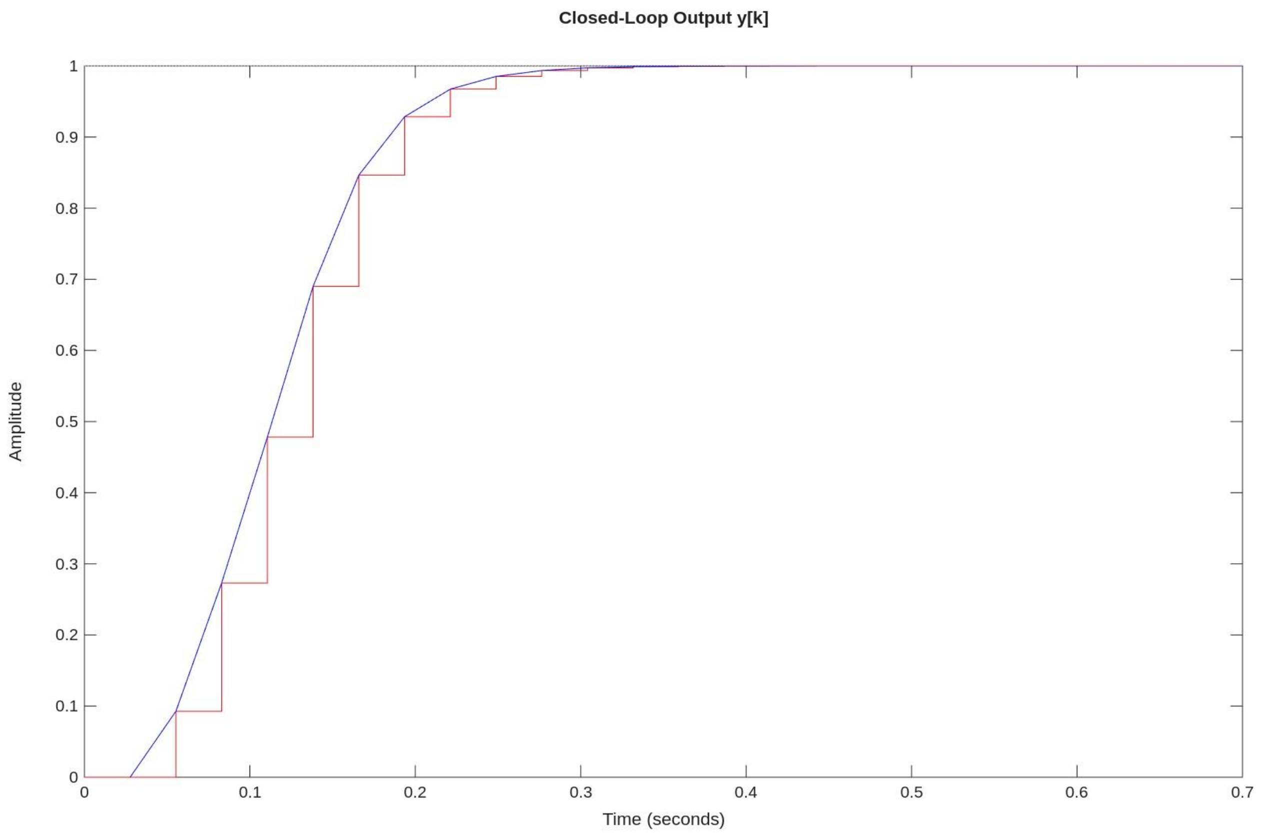


Figure 2. MATLAB generated closed loop output.

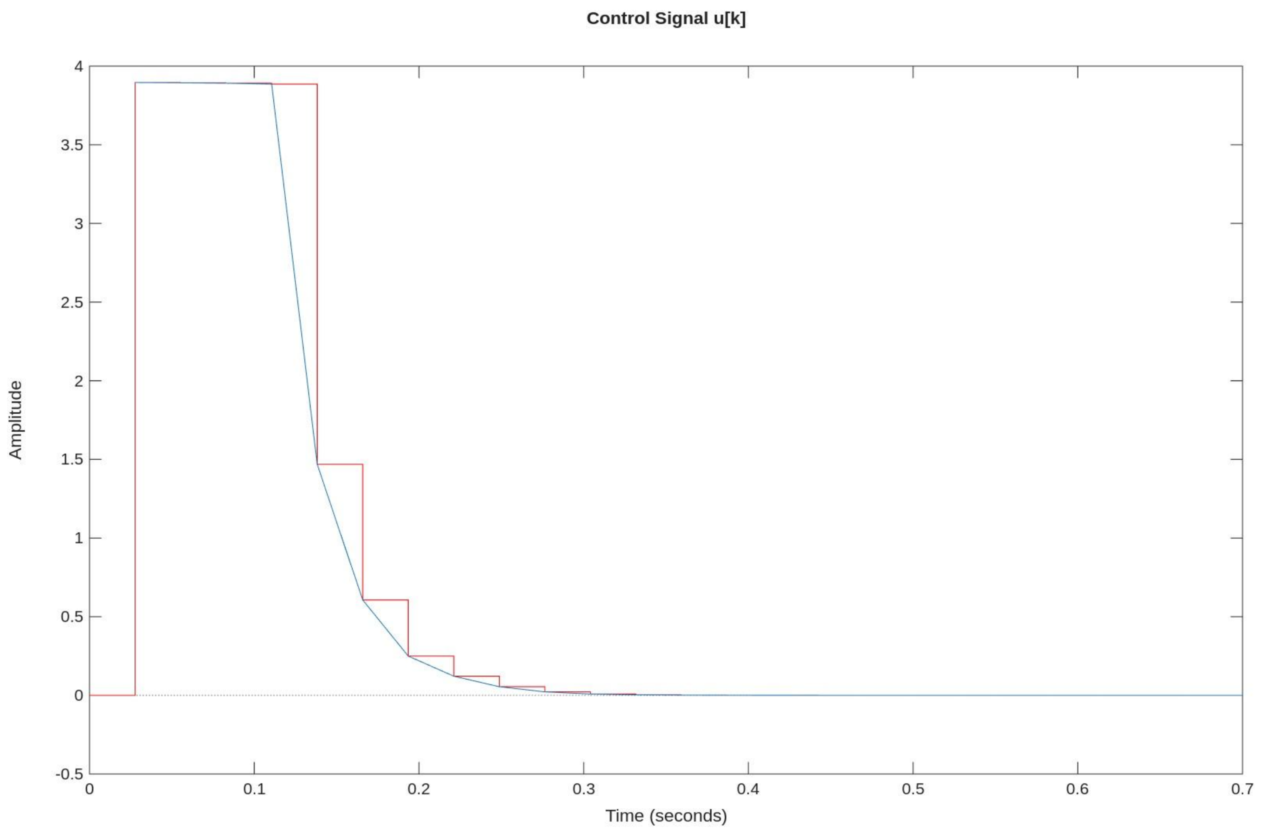


Figure 3. MATLAB generated controller signal with step input.

The block diagram used in the Simulink can be seen in Figure 4. From Figures 5 and 6, it can be seen that the $e_{ss} = 0$, the control signal output does not exceed ±6V, and the system settles within 2% of the unit step input. From here since all plots looked appropriate the discretized controller now needed to be tested on Simulink.

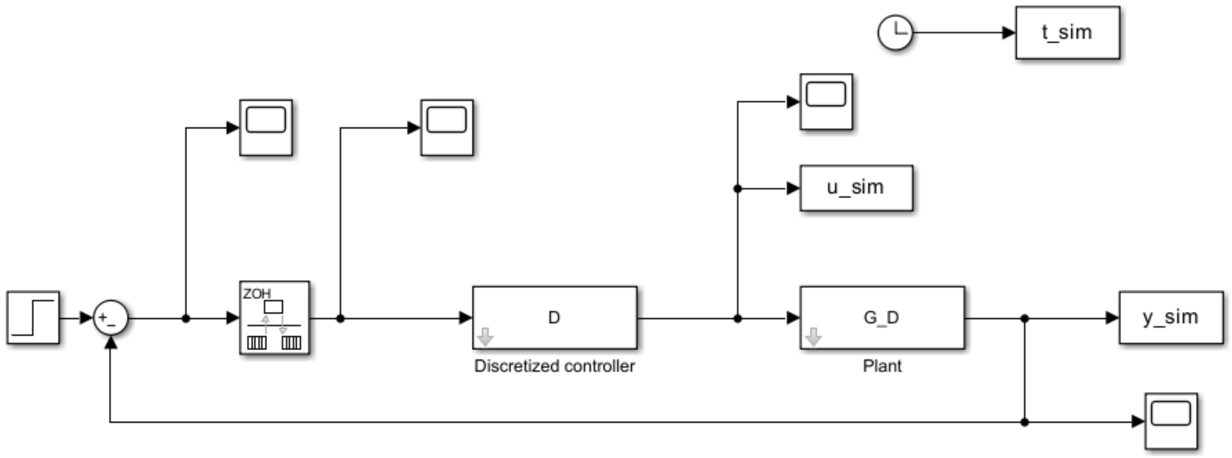


Figure 4. Simulink experiment simulation.

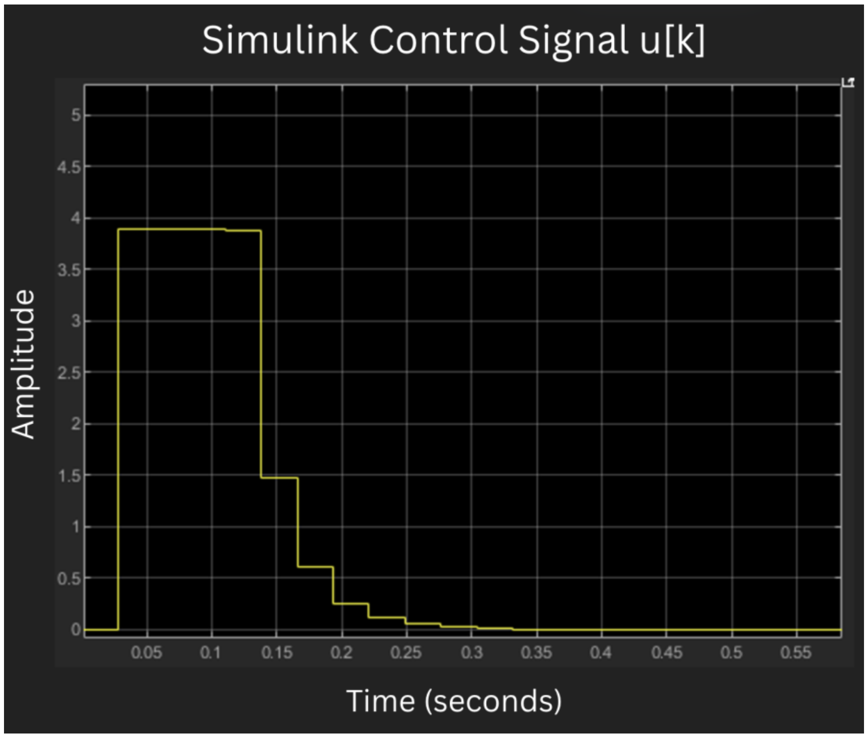


Figure 5. Simulink scope output for control signal.

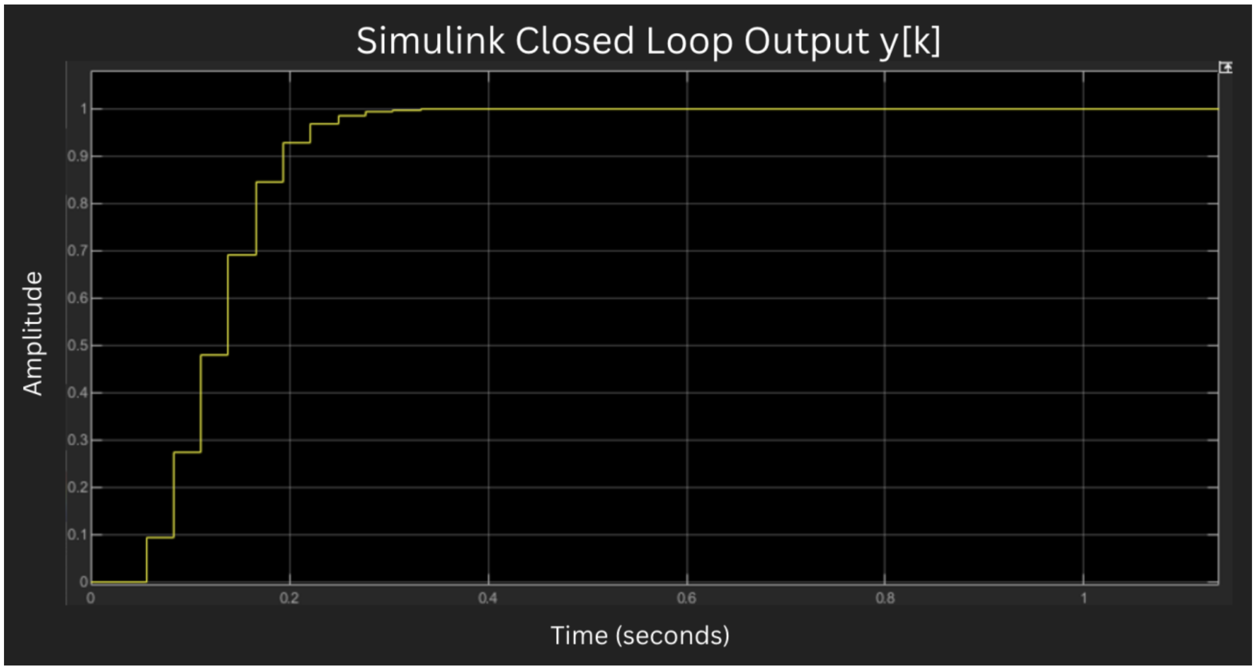


Figure 6. Simulink scope for output y[k].

Looking at Figures 2, 3, 5, and 6, the MATLAB and Simulink match one another. The upcoming step is to now test the controller on the physical system.

Part B

Once our controller started working in Matlab and Simulink, the controller *D[z]* was implemented on the real experiment. The beam was kept detached and the discrete controller was implemented in Arduino (code can be found in Appendix). The discrete controller transfer function is $T[z] = \frac{U[z]}{\mathrm{E[z]}}$, where $U[z]$ is the control output and $E[z]$ is the error signal. Knowing that z^{-k} evaluates to a time delay of k sampling periods, we could represent the current controller output (found from MATLAB) as a combination of previous control outputs and current/previous error signals. Applying this transformation to T(z) with 4 previous control outputs and 3 previous error outputs as an example, yields the discrete time controller in Equation 4 below:

$$u[k] \;=\; \frac{1}{b_0}(-b_1u_1 - b_2u_2 - b_3u_3 - b_4u_4 + a_0e + a_1e_1 + a_2e_2 + a_3e_3)$$

Equation 4: Controller output example provided in manual [1]

The controller output for our system was as follows below, where terms continue to expand as a summation. Coefficients have been written with 3 decimals, though the actual code used the `long` format precision.

$$\begin{aligned} u[k] \;=\; (1/1.0)(&-0.151\cdot u[k-1] \;-\; 0.275\cdot u[k-2] \;-\; 0.168\cdot u[k-3] \;-\; 0.083\cdot u[k-4] \\ &-\; 0.010\cdot u[k-5] \;+\; 0.008\cdot u[k-6] \;+\; 0\cdot e[k] \;+\; 3.896\cdot e[k-1] \;+\; 0.588\cdot e[k \\ &-2] \;+\; 1.428\cdot e[k-3] \;+\; 1.404\cdot e[k-4] \;-\; 1.059\cdot e[k-5] \;+\; 0.144\cdot e[k-6]) \end{aligned}$$

As a transfer function, this closed loop inner controller could also be modelled as:

$$D[z] = \frac{3.896z^{-1} + 0.588z^{-2} + 1.428z^{-3} + 1.404z^{-4} - 1.059z^{-5} + 0.144z^{-6}}{1.000 + 0.151z^{-1} + 0.275z^{-2} + 0.168z^{-3} + 0.083z^{-3} + 0.010z^{-5} + 0.008z^{-6}}$$

Implementing this in Arduino (code found in Appendix), resulted in the following output in Figure 7 using the CoolTerm chart feature. The setpoint was changed from 0.7 rad to -0.7 rad, and it was clear that our controller was working as expected.

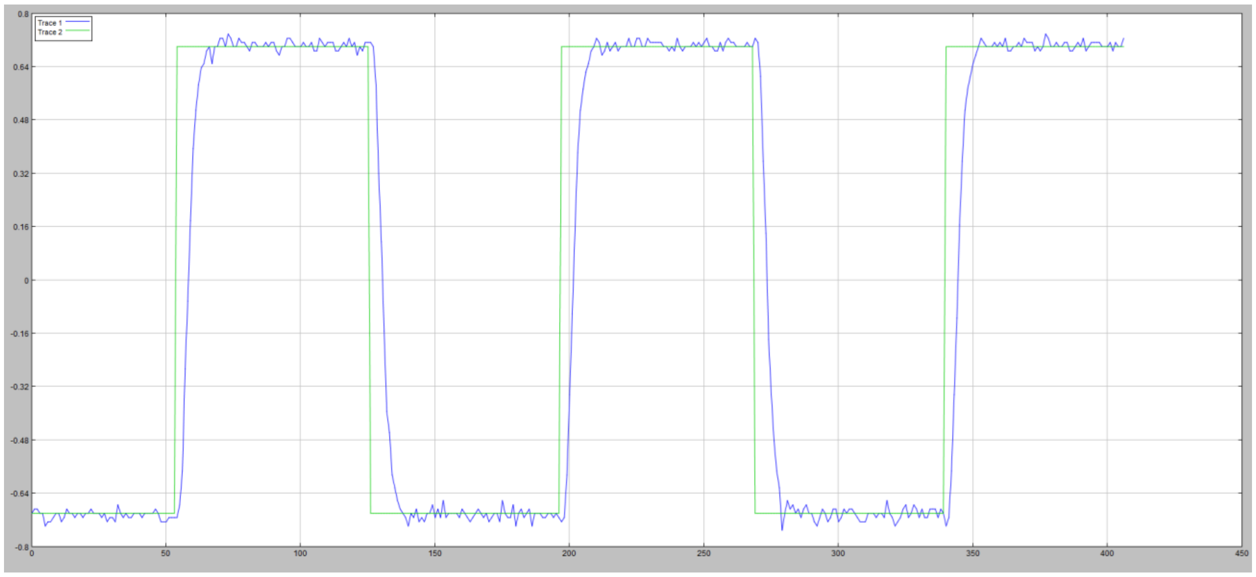


Figure 7. CoolTerm raw output for measured angle (blue) and setpoint (green) for changing setpoint.

Part C

The inner loop controller designed in part (a) was successfully implemented on the experimental apparatus using a sampling period of 0.02764604299 seconds, with step inputs tested from θ_{ref} = -0.7 rad to θ_{ref} = +0.7 rad. Figures 8, 9, and 10 demonstrate that all specifications were met. The steady-state error is effectively zero as the measured angle (orange) converges to and tracks the setpoint angle (blue dashed) with decent accuracy. The system itself even when not moving contains a high level of noise and for this reason the jittering in potentiometer values can be observed.

The 2% settling time for both negative to positive and positive to negative steps is approximately 0.3 seconds, which falls within the specified 0.40 seconds. The overshoot in both directions is approximately zero percent as the system does not experience signifigant overshoot of any kind, falling lower than the 5% specification. Figure 1 shows the motor voltage remains within the ±6V limit throughout operation, with peak voltages of approximately ±6V occurring at small intervals during step changes.

The experimental results closely match the simulated performance from part (a), with minor deviations in attributes due to unmodeled friction, mechanical backlash in the gear train, sensor noise in potentiometer readings, and quantization effects in ADC/DAC conversion. The saturator was successfully implemented to limit $\theta_{ref,\mathrm{sat}}$ to ±0.7 rad, and the controller demonstrates stable closed-loop operation, making it ready for integration with the beam attachment in the next phase of the lab.

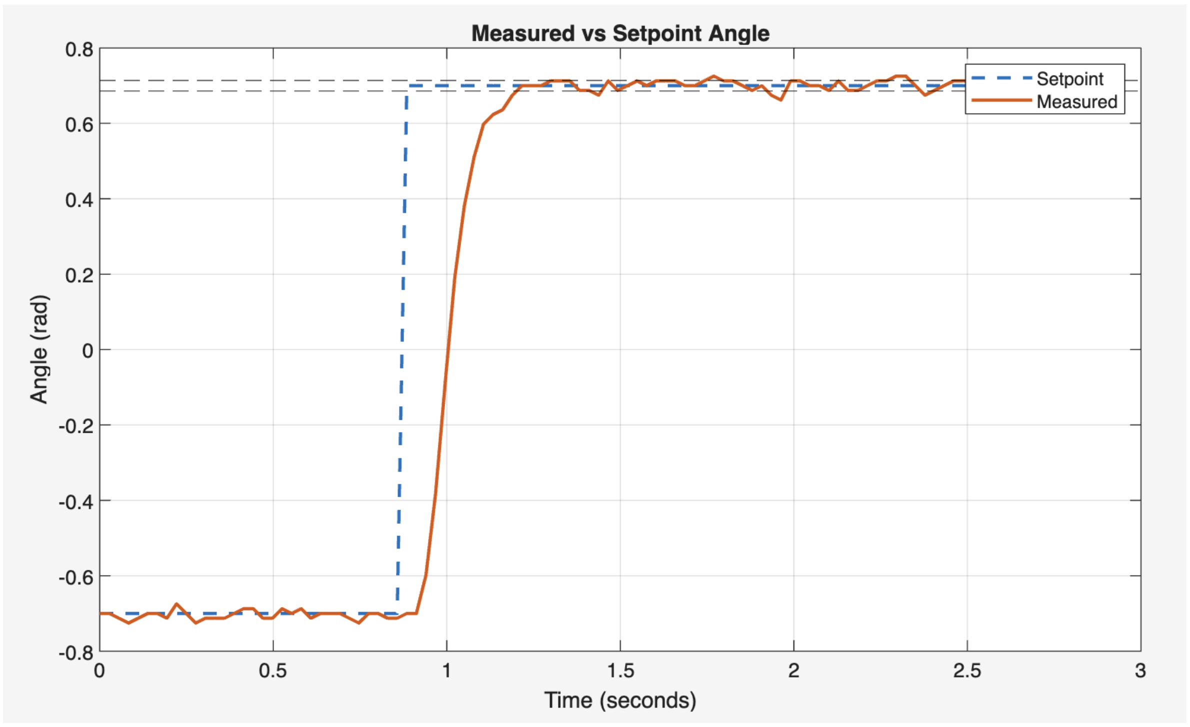


Figure 8. Setpoint angle set to 0.7 radians and associated measured response.

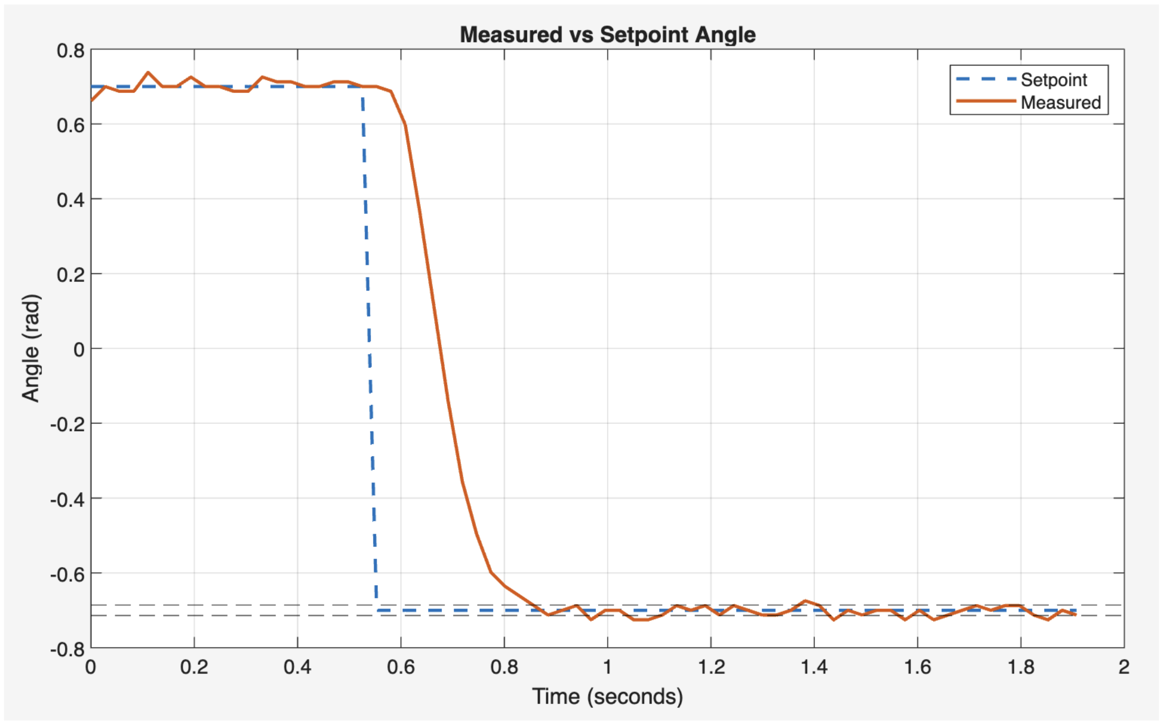


Figure 9. Setpoint angle set to -0.7 radians and associated measured response.

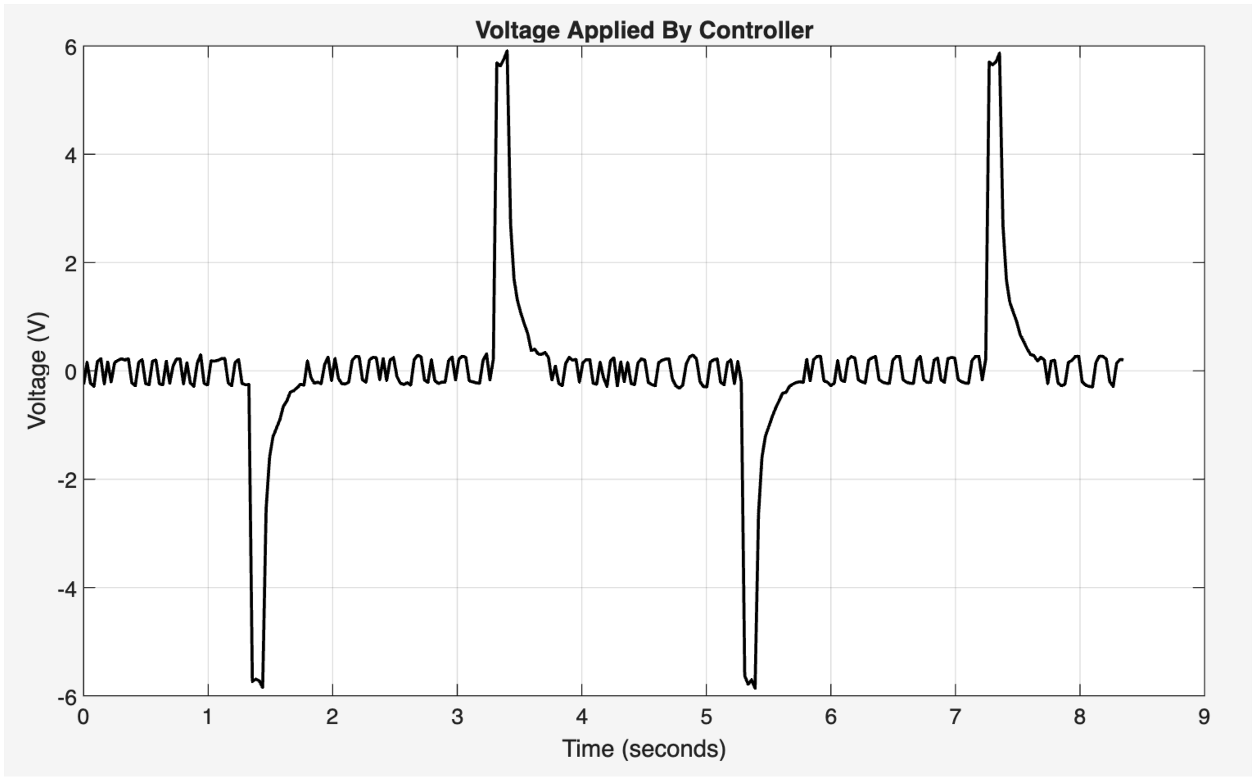


Figure 10. Voltage applied by controller over time, not exceeding 6V constraint.

Part D

The new motor stiction values were then found with the beam attached. Although we can assume that the inner-loop model remains the same, the new stiction offsets needed to be found in both directions in order to meet the specifications. We decided to not put the ball on the beam while doing this calculation, as the ball would roll on the beam, exerting a changing torque which would make it difficult to find an accurate stiction value. Furthermore, it was important to determine the stiction value in both directions as it was more difficult to move the beam upwards than downwards as it was acting against gravity.

This was done by using binary search between 0V to 1V to find a voltage value where the motor moved, and then gradually decreasing this value by 0.01V increments until it no longer did. This allowed us to systematically find the correct stiction offset such that applying this voltage would barely move the motor. By doing this for both the clockwise and counter-clockwise directions, we found that the stiction offsets were 0.3 for positive and 0.2 for negative direction.

Part E

In order to make the outer-loop control work the raw ball position sensor measurement needed to be converted into a physical unit of meters from the ADC value from the potentiometer.

We can take the expression of $y(t)$ to be the position relative to the left end of the beam. The left-most side of the beam is the origin. Since the sensor outputs an ADC value and we want a physical conversion, a linear calibration was done. This was in the form of Equation 5.

$$Equation\ 5: \quad y(t) = m * sensor\ reading + offset$$

$$y(t) = mx + b$$

In this case y is the relative position, m is the scale factor in meters per sensor unit output, b is the offset corresponding to when the physical ball is at y(t) =0, and x is the sensor reading (in ADC counts). The calibration process was done by placing the ball on the furthest point to the left and right of the beam ($y = 0m$, $y = 0.471m$). The length of the beam was found in Table A1 in the lab manual [1]. The corresponding ADC value at either end was recorded. To reduce sensor noise that may occur multiple samples were collected at each position and averaged. The resulting sensor reading are denoted $x_1$at y_1 and x_2 at y_2. Assuming that the potential is linear as it moves downwards towards the end the calibration can be done using the two points we have as seen from Equation 6.

$$Equation\ 6: \quad m = \frac{(\mathrm{y}_2 - y_1)}{(\mathrm{x}_2 - \mathrm{x}_1)}$$

$$m = \frac{(0.417 - 0)}{(708 - 398)}$$

$$m = 0.00104773869$$

The offset, b was obtained using the following formula

$$b = \mathrm{y}_1 - (m * x_1)$$

$$b = 0 - (0.00104773869 * 310)$$

$$b = -0.32479899252$$

After substituting these numbers into Equation 5 we get:

$$y(t) = 0.00104773869x \ - 0.32479899252$$

This will be the equation that will be run when the sensors convert its ADC value into a length in meters.

Part F

The model for the beam arm can be modelled as follows:

$$\frac{\phi(s)}{\Theta(s)} = K_2$$

$$\frac{Y(s)}{\phi(s)} = \frac{K_3}{s^2}$$

To solve for K_2, we used geometry and small angle approximation. Assuming that ϕ is usually small, we can approximate the arc lengths and equate them as follows:

$$[Arc\ Length\ at\ radius\ (r)] = r \cdot \Delta\theta$$

$$[Arc\ Length\ with\ Full\ Beam] = L \cdot \Delta\varphi$$

$$L \cdot \Delta\varphi = r \cdot \Delta\theta$$

$$K_2 = \frac{\Delta\varphi}{\Delta\theta} = \frac{r}{L} = \frac{2.54[cm]}{41.7[cm]} = 0.060911$$

To solve for K_3, we can use the Laplace transform property for derivatives:

$$\frac{Y(s)}{\phi(s)} = \frac{K_3}{s^2}$$

$$\frac{s^2 Y(s)}{\phi(s)} = K_3$$

Taking the Laplace transform yields:

$$\frac{\ddot{y}}{\phi} = K_3$$

Therefore,

$$Equation\ 7\text{:} \quad K_3 = \frac{a}{\phi}$$

To determine acceleration (a), we recorded the position of the ball over time with different beam angles. We then used the following kinematic equation to determine acceleration:

$$d = v_i t + \frac{1}{2}at^2$$

Where d is the position of the ball and v_i is the initial velocity. Since we began from rest, v_i was 0, and the equation became:

$$d = \frac{1}{2}at^2$$

$$Equation\ 8: \quad a = 2d/t^2$$

The position of the ball over time was tested with θ angles of 0.7, 0.35, -0.35, and -0.7 radians. The results were plotted in Figures 11-14 below, to determine the total time taken for the ball to roll down the beam.

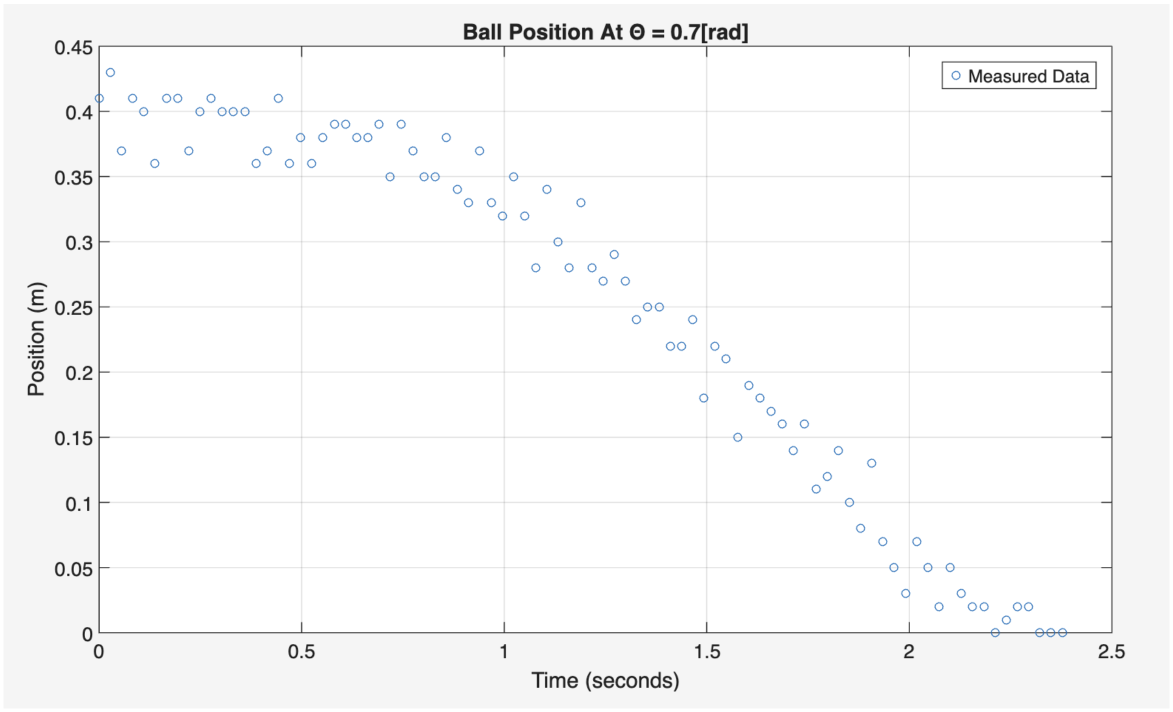


Figure 1111. Ball position over time with theta at 0.7 radians

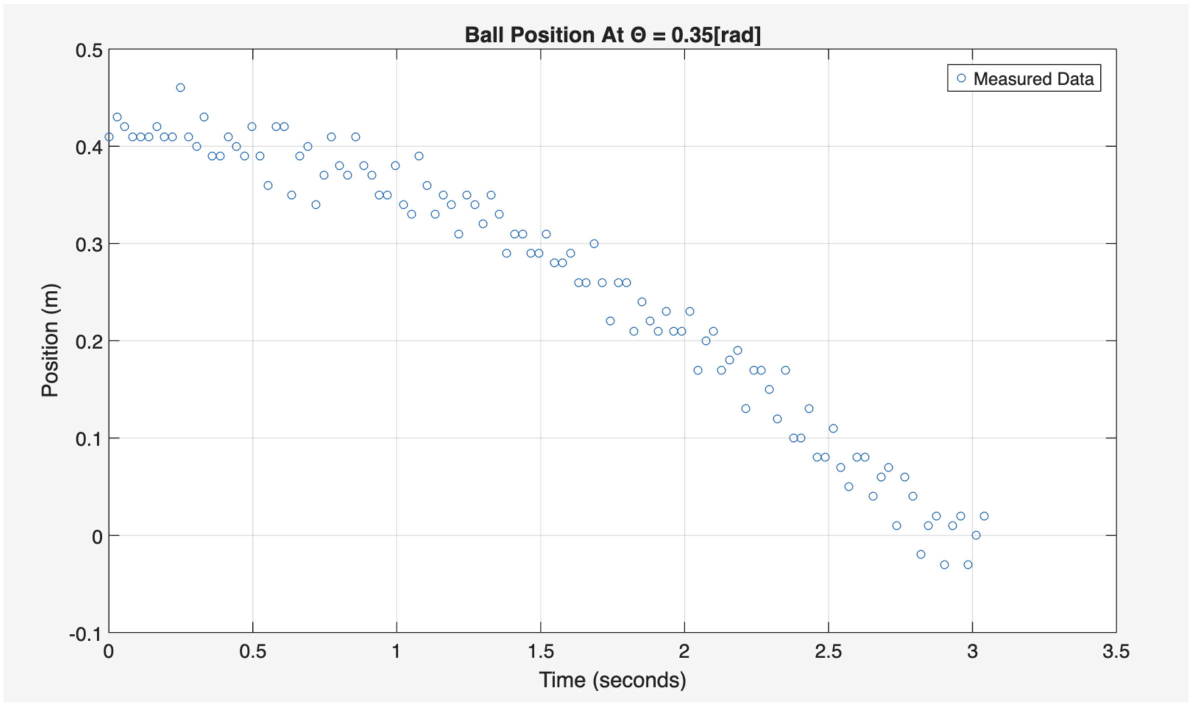


Figure 1212. Ball position over time with theta at 0.35 radians

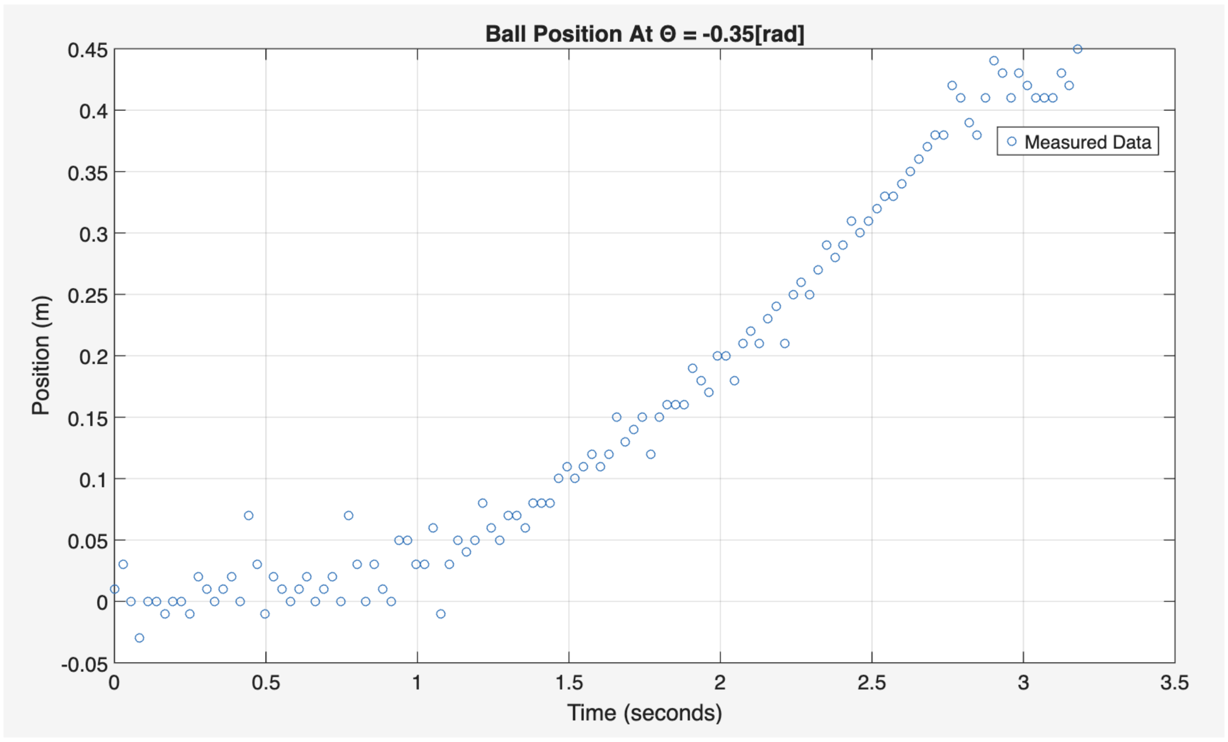


Figure 1313. Ball position over time with theta at -0.35 radians

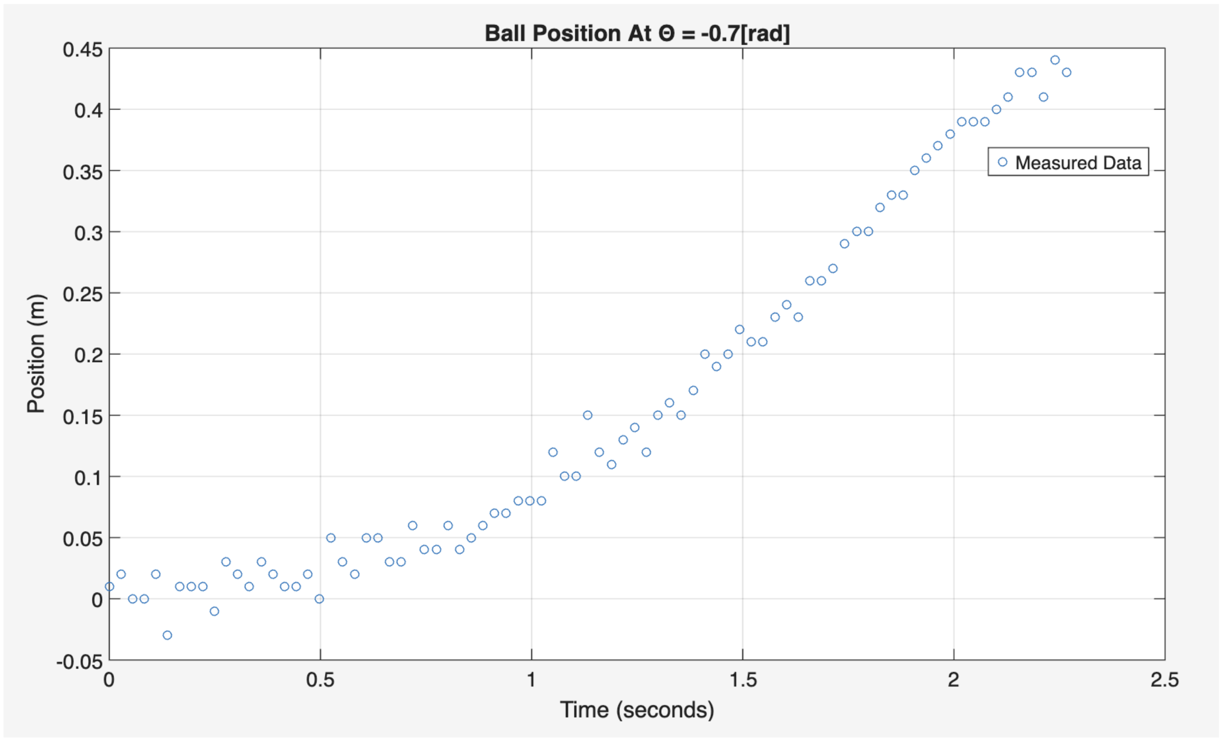


Figure 1414. Ball position over time with theta at -0.7 radians

Using the results above, we calculated the acceleration values at different angles, as seen in Table 1 below.

Table 1. Calculation of acceleration using θ, ϕ , and total time.

θ [rad]	ϕ [rad]	Total time (s)	Acceleration (m/s^2) using Equation 8
0.7	0.0426377	2.405	-0.1441902
0.35	0.02131885	3.068	-0.0886044
-0.35	-0.02131885	3.124	0.0854562
-0.7	-0.0426377	2.322	0.1546826

Using the acceleration values, the following K_3 values were found using Equation 8:

K_3 = -3.3818, -4.1562, -4.0085, -3.6278 for θ = 0.7, 0.35, -0.35, and -0.7 radians respectively. Therefore, our average value was:

$$K_3 = -3.793$$

Part G

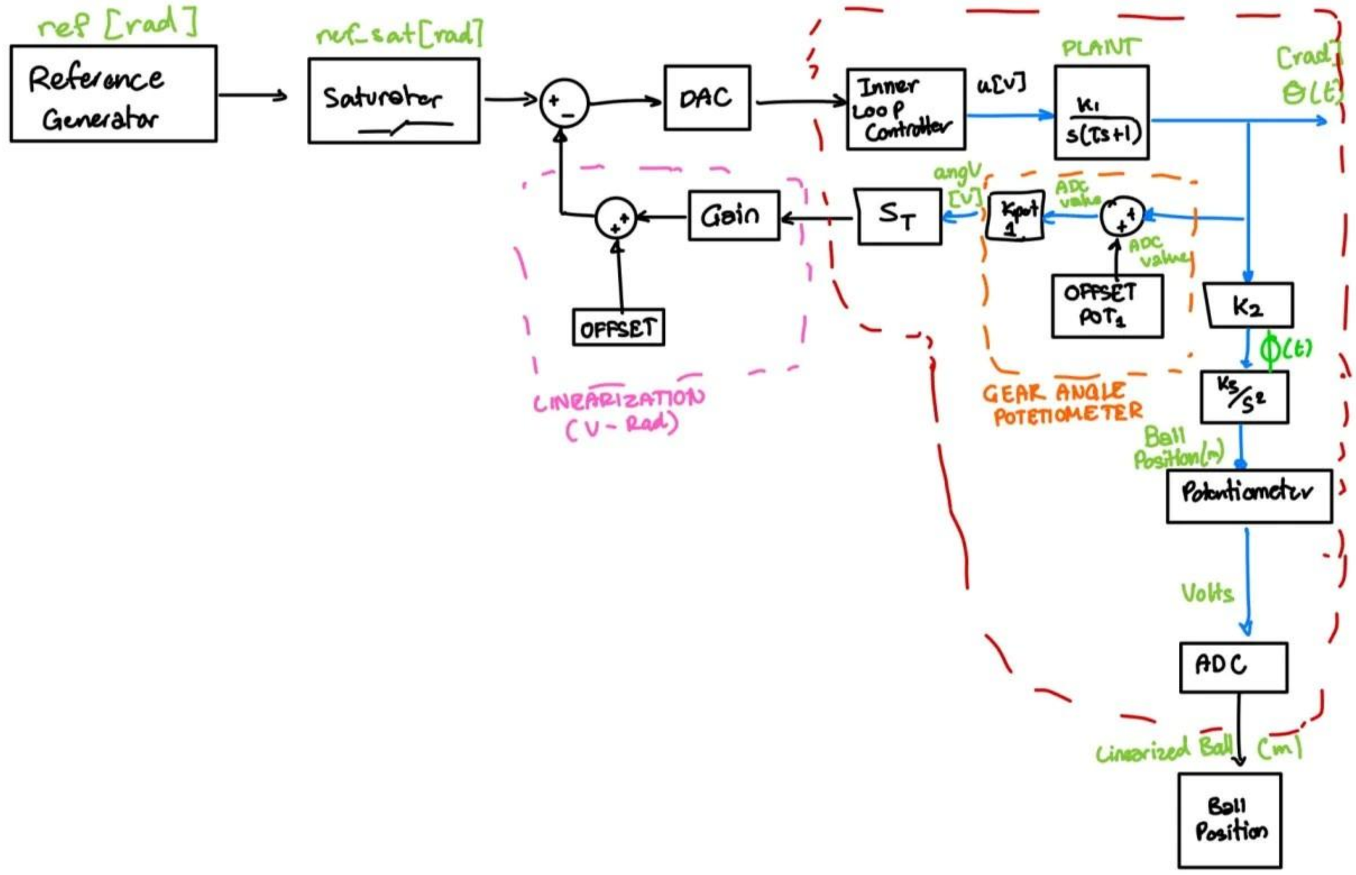


Figure 15. Block diagram for experiment.

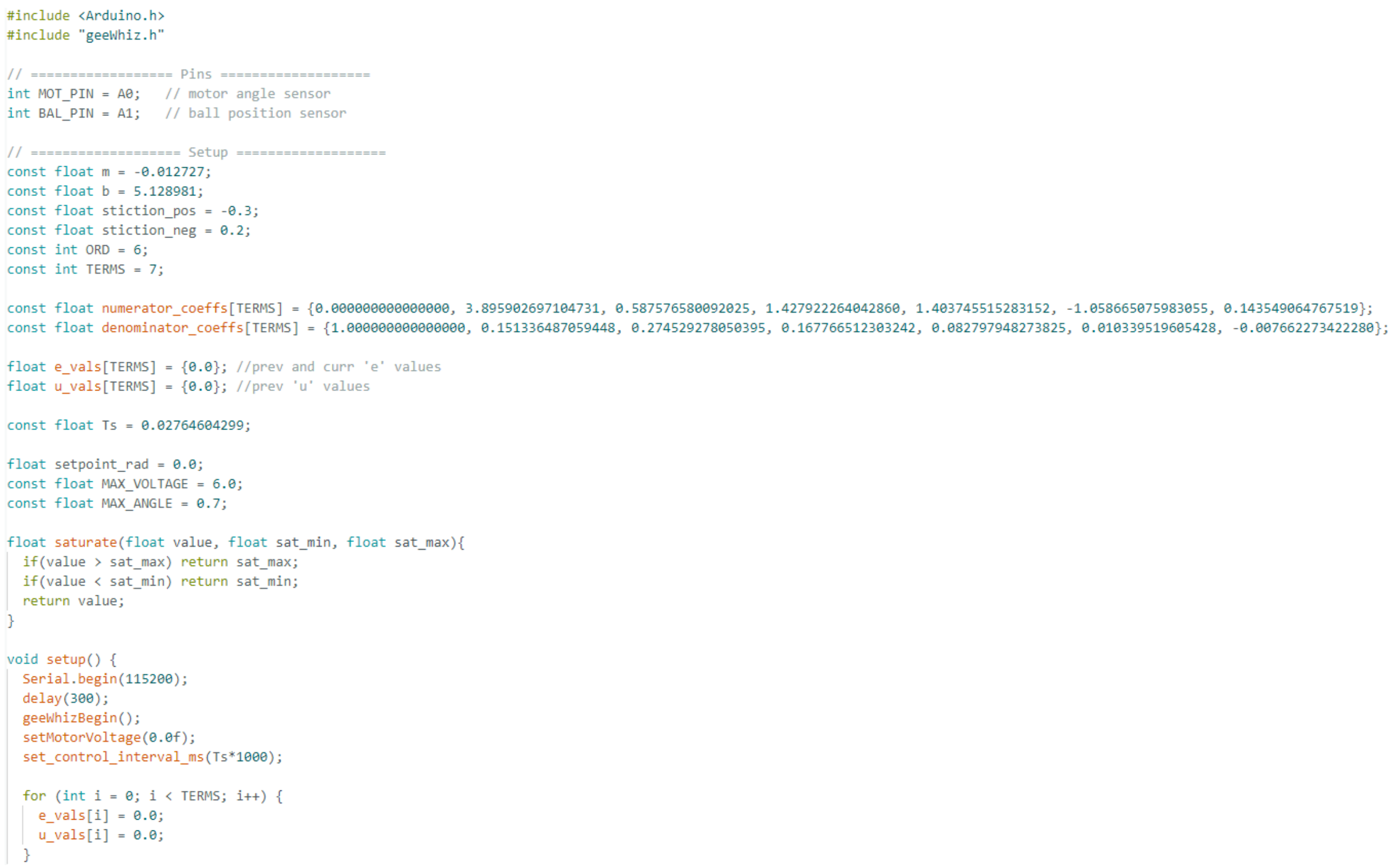
References

[1] Department of Mechatronics Engineering, "Ball and Beam Lab Project: Student Handout," MTE 484 - Control Applications, University of Waterloo, Waterloo, ON, 2025.

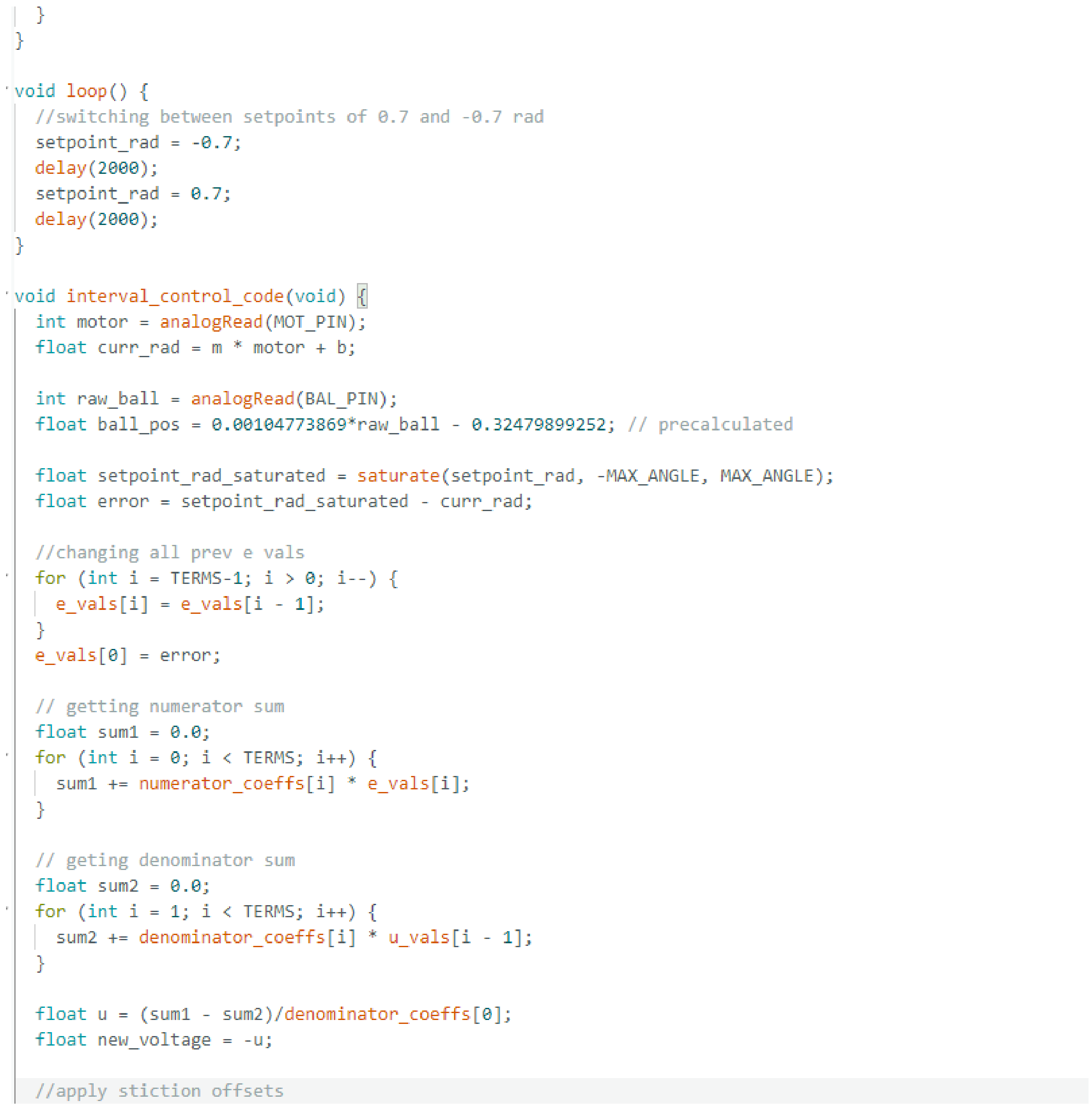
[2] N. S. Nise, *Control Systems Engineering*, 3rd ed. Pomona, CA, USA: John Wiley & Sons, Inc., 2000.

Appendix

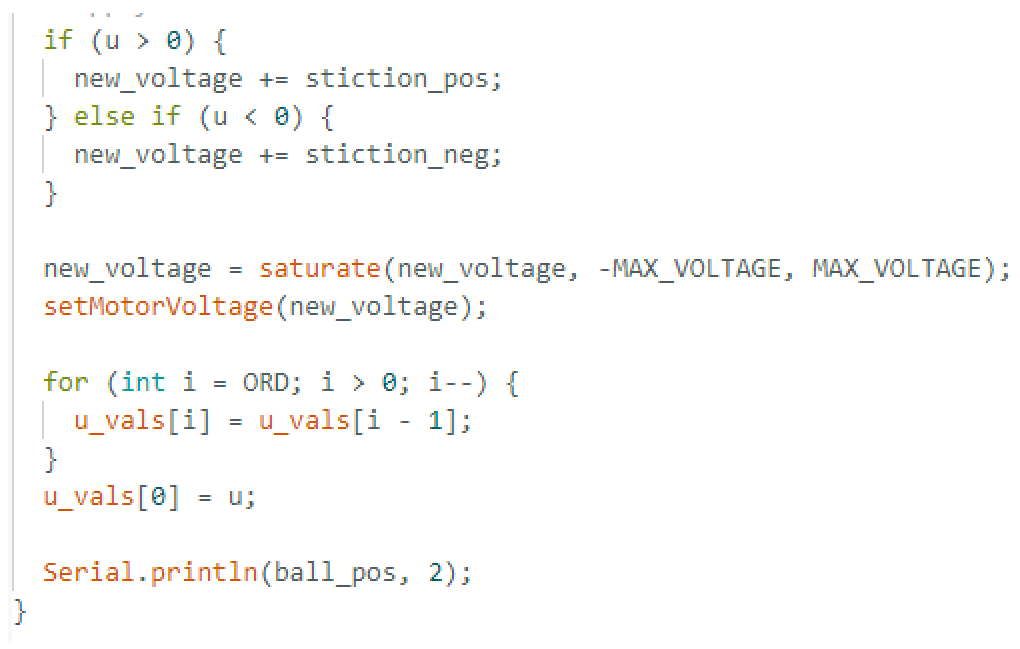
Arduino Code



```
#include <Arduino.h>
#include "geeWhiz.h"

// ================== Pins ===================
int MOT_PIN = A0;   // motor angle sensor
int BAL_PIN = A1;   // ball position sensor

// =================== Setup ===================
const float m = -0.012727;
const float b = 5.128981;
const float stiction_pos = -0.3;
const float stiction_neg = 0.2;
const int ORD = 6;
const int TERMS = 7;

const float numerator_coeffs[TERMS] = {0.000000000000000, 3.895902697104731, 0.587576580092025, 1.427922264042860, 1.403745515283152, -1.058665075983055, 0.143549064767519};
const float denominator_coeffs[TERMS] = {1.000000000000000, 0.151336487059448, 0.274529278050395, 0.167766512303242, 0.082797948273825, 0.010339519605428, -0.007662273422280};

float e_vals[TERMS] = {0.0}; //prev and curr 'e' values
float u_vals[TERMS] = {0.0}; //prev 'u' values

const float Ts = 0.02764604299;

float setpoint_rad = 0.0;
const float MAX_VOLTAGE = 6.0;
const float MAX_ANGLE = 0.7;

float saturate(float value, float sat_min, float sat_max){
  if(value > sat_max) return sat_max;
  if(value < sat_min) return sat_min;
  return value;
}

void setup() {
  Serial.begin(115200);
  delay(300);
  geeWhizBegin();
  setMotorVoltage(0.0f);
  set_control_interval_ms(Ts*1000);

  for (int i = 0; i < TERMS; i++) {
    e_vals[i] = 0.0;
    u_vals[i] = 0.0;
  }
```



```
  }
}

void loop() {
  //switching between setpoints of 0.7 and -0.7 rad
  setpoint_rad = -0.7;
  delay(2000);
  setpoint_rad = 0.7;
  delay(2000);
}

void interval_control_code(void) {
  int motor = analogRead(MOT_PIN);
  float curr_rad = m * motor + b;

  int raw_ball = analogRead(BAL_PIN);
  float ball_pos = 0.00104773869*raw_ball - 0.32479899252; // precalculated

  float setpoint_rad_saturated = saturate(setpoint_rad, -MAX_ANGLE, MAX_ANGLE);
  float error = setpoint_rad_saturated - curr_rad;

  //changing all prev e vals
  for (int i = TERMS-1; i > 0; i--) {
    e_vals[i] = e_vals[i - 1];
  }
  e_vals[0] = error;

  // getting numerator sum
  float sum1 = 0.0;
  for (int i = 0; i < TERMS; i++) {
    sum1 += numerator_coeffs[i] * e_vals[i];
  }

  // geting denominator sum
  float sum2 = 0.0;
  for (int i = 1; i < TERMS; i++) {
    sum2 += denominator_coeffs[i] * u_vals[i - 1];
  }

  float u = (sum1 - sum2)/denominator_coeffs[0];
  float new_voltage = -u;

  //apply stiction offsets
```



```
  if (u > 0) {
    new_voltage += stiction_pos;
  } else if (u < 0) {
    new_voltage += stiction_neg;
  }

  new_voltage = saturate(new_voltage, -MAX_VOLTAGE, MAX_VOLTAGE);
  setMotorVoltage(new_voltage);

  for (int i = ORD; i > 0; i--) {
    u_vals[i] = u_vals[i - 1];
  }
  u_vals[0] = u;

  Serial.println(ball_pos, 2);
}
```

MATLAB Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This template is for the IOP control design with SPA, and is incomplete.
% You need to complete it by replacing every * with the correct code.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% set time step
T = 0.02764604299;
tfinal =3;
s = tf('s');
z = tf('z', T);
G_C = 1.998/(0.022*s^2 +s);
G_D = c2d(G_C, T);
G_D
%% Plant Poles and Coefficients in its Partial Fraction Decomposition
stableRealPlantPoles = [0.284609185783745];
stableComplexPlantPoles = [];
unstablePlantPoles = [1.000000000000000];
stablePlantPoles = [stableRealPlantPoles stableComplexPlantPoles];
qs = [stablePlantPoles unstablePlantPoles];
% coefficents go in order of the poles
cs = [-0.031445718629690 0.055236793894020];
n = length(qs);
nhat = length(stablePlantPoles);
nreal = length(stableRealPlantPoles);
ncomplex = length(stableComplexPlantPoles);
% verify that your plant is correct!
G = 0;
for k=1:n
G = G + cs(k)/(z-qs(k));
```

```
end
G
%% Poles Chosen in the Simple Pole Approximation of W[z]
realWPoles = [];
complexWPoles = [
% 0.356930347157456 + 0.826196542765724i
% 0.356930347157456 - 0.826196542765724i
% 0.666437870114671 + 0.558444773703814i
% 0.666437870114671 - 0.558444773703814i
% 0.818568286012092 + 0.178734331160040i
% 0.818568286012092 - 0.178734331160040i
% 0.772183714907377 + 0.227447379480407i
% 0.772183714907377 - 0.227447379480407i
% 0.531829123858141 + 0.557815187151022i
% 0.531829123858141 - 0.557815187151022i
% 0.156352764579849 + 0.718020760847653i
% 0.156352764579849 - 0.718020760847653i
% -0.246138666287763 + 0.652239033605081i
% -0.246138666287763 - 0.652239033605081i
% -0.541821158388287 + 0.372061597484514i
% -0.541821158388287 - 0.372061597484514i
% -0.614171375570681 + 0.028169512412883i
% -0.614171375570681 - 0.028169512412883i

% -0.419743298364956 + 0.384467896550295i
% -0.419743298364956 - 0.384467896550295i
% -0.037728748040347 + 0.518243708665676i
% -0.037728748040347 - 0.518243708665676i
% 0.321582463138287 + 0.335536465085832i
% 0.321582463138287 - 0.335536465085832i
0.398371042488927 + 0.057450087083452i,
0.398371042488927 - 0.057450087083452i,
0.097390191562406 + 0.313871232493950i,
0.097390191562406 - 0.313871232493950i,
-0.213734070429800 + 0.091201683852363i,
-0.213734070429800 - 0.091201683852363i
];
ps = [realWPoles complexWPoles];
% ps = generate_poles(30,0.9,0);
mreal = sum(imag(ps) == 0); % number of purely real poles
mcomplex = sum(imag(ps) > 0)*2; % count each conjugate pair twice
m = length(ps);
%% Calculation of alpha, beta, gamma, and gamma hat
alpha = zeros(m);
for i=1:m
for k=1:n
alpha(i,i) = alpha(i,i) + cs(k)/(ps(i)-qs(k));
```

```
end
end
beta = zeros(n,m);
for i=1:m
for k=1:n
beta(k,i) = cs(k)/(qs(k)-ps(i));
end
end
gamma = zeros(n-nhat,m);
for i=1:m
for j=(nhat+1):n
gamma(j-nhat,i) = cs(j)/(qs(j)-ps(i));
end
end
gammaHat = zeros(n-nhat,nhat);
for k=1:nhat
for j=(nhat+1):n
gammaHat(j-nhat,k) = cs(j)/(qs(j)-qs(k));
end
end
% % verify on a simple example that alpha, beta, gamma, and gammahat are correct!
% alpha
% beta
% gamma
% gammaHat
%% Determination of A and b matrices for IOP equations
A = [alpha eye(m) zeros(m,nhat);
beta [zeros(nhat,m) eye(nhat);
zeros(size(beta,1)-nhat,m+nhat)];
zeros(size(gamma)) gamma gammaHat];
b = [zeros(m+size(beta,1),1);
-cs((nhat+1):end)'];
%% Determination of step response matrices
% time horizon
K = 100;
stepry = zeros(K,m+nhat);
for k=1:K
for i=1:m
stepry(k,i) = -(1-ps(i)^k)/(1-ps(i));
end
for j=1:nhat
stepry(k,m+j) = -(1-qs(j)^k)/(1-qs(j));
end
end
stepru = zeros(K,m);
for k=1:K
for i=1:m
```

```
stepru(k,i) = (1-ps(i)^k)/(1-ps(i));
end
end
% verify on a simple example that stepry and stepru are correct!
% stepry
% stepru
%% Determination of steady state vector
steadyState = zeros(1,m+nhat);
for i=1:m
steadyState(i) = 1/(1-ps(i));
end
for k=1:nhat
steadyState(m+k) = 1/(1-qs(k));
end
% verify on a simple example that steadyState is correct!
%steadyState
%% Defining the variables for the optimization
wreal = sdpvar(mreal,1,'full');
wcomplex = sdpvar(mcomplex/2,1,'full','complex');
w = wreal;
for i=1:(mcomplex/2)
w = [w;
wcomplex(i);
conj(wcomplex(i))];
end
xreal = sdpvar(mreal,1,'full');
xcomplex = sdpvar(mcomplex/2,1,'full','complex');
x = xreal;
for i=1:(mcomplex/2)
x = [x;
xcomplex(i);
conj(xcomplex(i))];
end
xhatreal = sdpvar(nreal,1,'full');
xhatcomplex = sdpvar(ncomplex/2,1,'full','complex');
xhat = xhatreal;
for i=1:(ncomplex/2)
xhat = [xhat;
xhatcomplex(i);
conj(xhatcomplex(i))];
end
%% Defining the objective function and constraints for the optimization
Objective = 0;
% IOP constraint
Constraints = [A*[w; x; xhat] == b];
% input saturation constraint
Constraints = [Constraints, max(stepru * w) <= 6/1.4, min(stepru * w) >= -6/1.4];
```

```
% steady state constraint
Constraints = [Constraints,
steadyState*[x; xhat]+1 == 0];
% overshoot constraint
Constraints = [Constraints, max(stepry*[x;xhat]) <= 1.05 * ...
(-steadyState * [x;xhat])];
% settling time constraint
jhat = round(0.25/T);
Constraints = [Constraints,
max(stepry(jhat:end,:)*[x; xhat]) <= ...
1.02*(-steadyState*[x; xhat]),
min(stepry(jhat:end,:)*[x; xhat]) >= ...
0.98*(-steadyState*[x; xhat])];
%% Solving the optimization problem
% set some options for YALMIP and solver
options = sdpsettings('verbose',1,'solver','mosek');
% solve the problem
sol = optimize(Constraints,Objective,options);
% obtain the solution
wsol = value(w);
xsol = value(x);
xhatsol = value(xhat);
%% Plotting the solution
figure(1)
plot(T*(1:K),stepry*[xsol;xhatsol]);
% add green horizontal lines
yline(0.98,'g','LineWidth',1.5);
yline(1.02,'g','LineWidth',1.5);
xlabel('Time [s]');
ylabel('y[k]');
figure(2)
plot(T*(1:K),stepru*wsol);
xlabel('Time [s]');
ylabel('u[k]');
% use log scale for heat map?
log_scale_flag = 1;
% heat map
figure(3)
t = linspace(0,2*pi);
plot(cos(t),sin(t),'k--');
hold on;
if log_scale_flag
scatter(real(ps),imag(ps),50,log(abs(wsol)),'filled');
scatter(real(qs(1:nhat)),imag(qs(1:nhat)),50,log(abs(xhatsol)),'filled');
else
scatter(real(ps),imag(ps),50,abs(wsol),'filled');
scatter(real(qs(1:nhat)),imag(qs(1:nhat)),50,abs(xhatsol),'filled');
```

```
end
hold off;
colormap(jet);
colorbar;
%% Recover the transfer functions
z = tf('z',T);
% calculate W
W = 0;
for i=1:m
W = W + wsol(i)/(z-ps(i));
end
% calculate X
X = 1;
for i=1:m
X = X + xsol(i)/(z-ps(i));
end
for k=1:nhat
X = X + xhatsol(k)/(z-qs(k));
end
% remove the imaginary coefficients in W
[num,den] = tfdata(W);
num{1} = real(num{1});
den{1} = real(den{1});
W = tf(num,den,T);
% remove the imaginary coefficients in X
[num,den] = tfdata(X);
num{1} = real(num{1});
den{1} = real(den{1});
X = tf(num,den,T);
% find the poles and zeros of W and X
zpk(W)
zero(W)
pole(W)
zpk(X)
zero(X)
pole(X)
D = minreal(W / X, 1e-6); % Increase tolerance to remove near-zero poles
[numerator, denominator] = tfdata(D, 'v');
format long
%% Verify the transfer function D
[numD, denD] = tfdata(D, 'v');
fprintf('numD = [%s]\n', num2str(numD, '%.15f '));
fprintf('denD = [%s]\n', num2str(denD, '%.15f '));
%% Partial Fraction Decomposition of D
[r, p, k] = residuez(numD, denD);
disp('========================================');
disp('Partial Fraction Decomposition of D[z]:');
disp('========================================');
disp('Residues (r):');
disp(r);
disp('Poles (p):');
disp(p);
disp('Direct terms (k):');
disp(k);
%% Verify design in discrete time
j = sqrt(-1);
% compute Try and Tru by hand
Try = minreal(G_D * D / (1 + G_D * D), 1e-6);
Tru = minreal(D / (1 + G_D * D), 1e-6);
figure(4)
hold on;
plot(T*(1:K),stepry*[xsol;xhatsol],'Color',[0 0 1]); % bluestep(Try,'r');
title('Closed-Loop Output y[k]')
hold off;
figure(5)
hold on;
plot(T*(1:K),stepru*wsol);
step(Tru,'r');
title('Control Output u[k]')
hold off;
```