Rocky Project Report

Hong Zhang, Mateo Otero-Diaz, Zaraius Bilimoria

Overall Project Scope

The tasked project consists of a classic “self-balancing” pendulum mechanism, which uses a motor platform to compensate for pendulum disturbances which are cause for unstable angular evolution. This “balancing” process consists of a five step looping process where the desired angle undergoes three differing transformations, a controller change, a motor translation, and a CartPole evolution.

To begin with, this process is a closed loop, where even before the first transformation, a desired angle is input into the independent variable, θ, and summed with the feedback from the end of the cycle. After the final transformation, the output from the CartPole evolution is paired with a disturbance, which represents our perturbation to the actual mechanism. This output is exactly what is summed to the desired angle in order for the system to adjust when the loop recommences.

This process is better detailed in *figure 1*, which represents the symbolic values and variables that govern the evolution and stabilization of the system at hand.

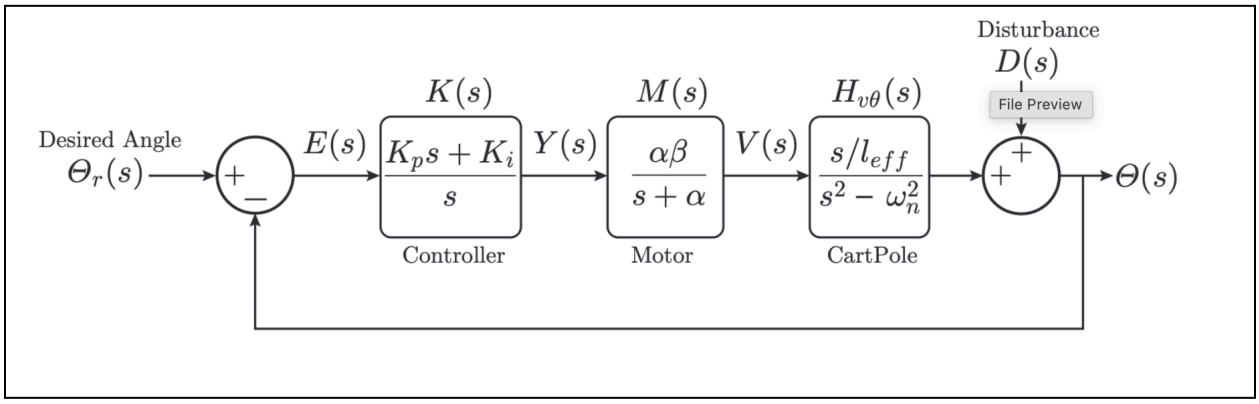


Figure 1: Macro-block representation of the inverted pendulum system with system parameters

In terms of how this system is translated into the actual mechanism, the inverted pendulum is best represented by our Pololu Balboa paired with laser cut side-facts that carry the weight on the crown of the pendulum. *Figure 2* gives a best representation of how the system actually represents the inverted pendulum along with its functionality in terms of self-stabilizing capabilities.

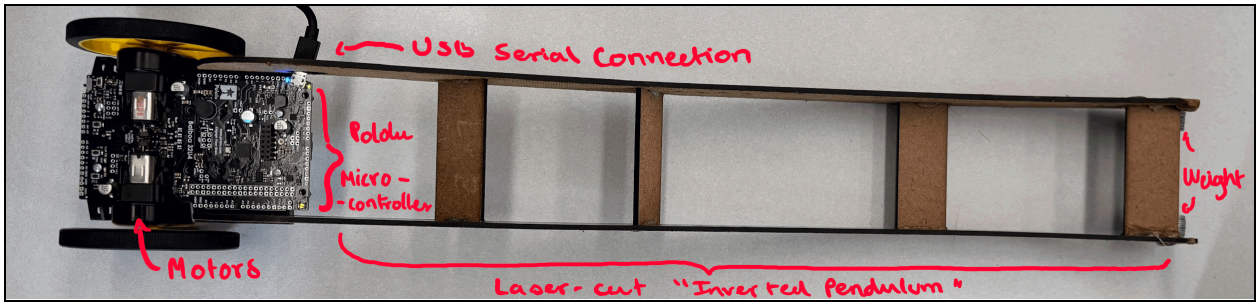
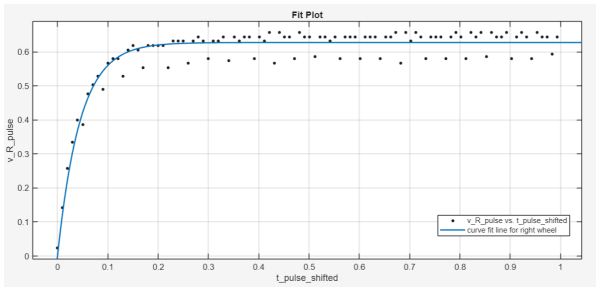


Figure 2: Physical representation of "Inverted Pendulum" including system components

Motor Constant Quantification

The process to identify the underlying constants, which govern the motors' translation of an electrical signal to actual wheel velocity outputs, comprises a calibration process where square wave test input signals are processed by the motors to output left and right wheel velocities(*figure 3a and 3b*). By aggregating these paired input-output data points, for a single cycle, we can fit their behavior to an exponential decay function with the form $a(1 - e^{-ct})$.



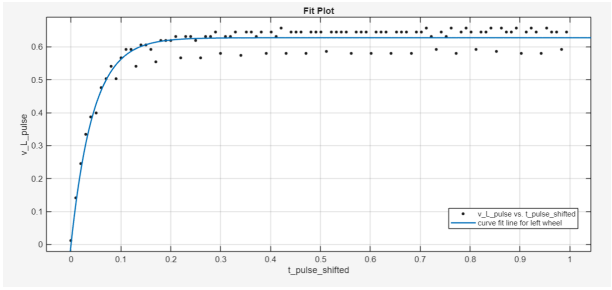


Figure 3a and 3b: *Left and Right motor data aggregation based on one cycle of square wave inputs*

Now to relate the values within the motor transfer function to the time-domain fit computed, one must derive the time domain version of the transfer function:

$$\text{Laplace-Domain: } \frac{Y(s)}{V(s)} = \frac{\alpha\beta}{s+\alpha}$$

$$\frac{Y(s)}{\mathcal{L}\{u(t)\}} = \frac{\alpha\beta}{s+\alpha}$$

$$\frac{Y(s)}{\frac{300}{s}} = \frac{\alpha\beta}{s+\alpha}$$

$$\text{Time-Domain: } \mathcal{L}^{-1}\{Y(s) = 300s\frac{\alpha\beta}{s+\alpha}\}$$

$$y(t) = 300\alpha\beta(1 - e^{-\alpha t})$$

With the values found in *figure 4a and b*, we can now match the fit constants to α and □, which are 22.3146 and 0.0021 for the left motor as well as 22.5964 and 0.0021 for the right motor.

Coefficients and 95% Confidence Bounds

	Value	Lower	Upper
a	22.3146	20.2277	24.4015
b	0.0021	0.0021	0.0021

Coefficients and 95% Confidence Bounds

	Value	Lower	Upper
a	22.5964	20.6429	24.5499
b	0.0021	0.0021	0.0021

Figure 4a and 4b: *Left and Right fit α and □ constants*

Angular Frequency Determination

In the guise of value identification, to identify the natural frequency, decay constant, damped frequency, damping coefficient, we followed a combination of methodologies found in the solutions for "Homework 2".

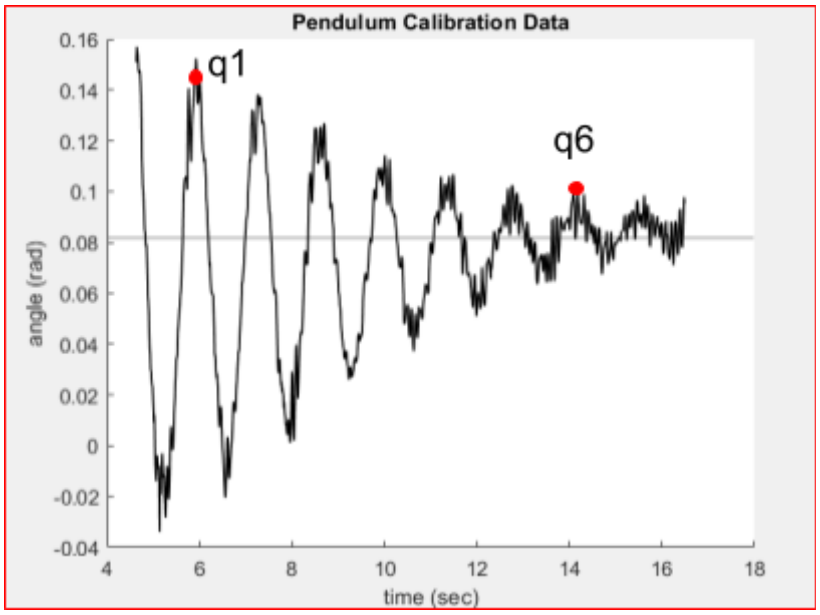


Figure 5: *Pendulum Calibration Data(Pendulum Angle vs. Time)*

Starting with the graph of the pendulum calibration data, *figure 5*, "the offset (steady-state) value was estimated by plotting different horizontal lines in MATLAB" and by using *equation 1*, the ratio between consecutive peaks could be computed:

$$q = \left(\frac{\theta_2 - \theta_{ss}}{\theta_1 - \theta_{ss}}\right)^{\frac{1}{6}}$$

Equation 1: Ratio between Consecutive Peaks

With that value and *equation 2*, the decay rate is obtainable.

$$\sigma = -\frac{1}{\tau}\ln(q)$$

Equation 2: Decay Rate

The time constant can be computed as well through *equation 3.*

$$\tau = \frac{t_{p_n} - t_{p_m}}{m - n}$$

Equation 3: Time Constant

Where m and n are two real and positive integers that are lesser than the maximum number of peaks in the function. Or in other words, take the time between any two peaks and divide it by the distance between the order of the peaks, and that results in the time constant.

The damped frequency could also be concurrently computed through *equation 4.*

$$\omega_d = \frac{2\pi}{\tau}$$

Equation 4: Damped Frequency

Using both the damped frequency and the decay rate, *equation 5* represents the derivation of natural frequency.

$$\omega_n = \sqrt{\sigma^2 + \omega_d^2}$$

Equation 5: Natural Frequency

With both decay rate and natural frequency, the damping coefficient is obtainable through *equation 6*.

$$\zeta = \frac{\sigma}{\omega_n} = \frac{\sigma}{\sqrt{\sigma^2 + \omega_d^2}}$$

Equation 6: Damping Coefficient

Finally, using the acceleration due to gravity, or approximately 9.81 N/kg, and the natural frequency, it is possible to compute the characteristic length of the pendulum using *equation 7*.

$$w_n = \sqrt{\frac{l}{g}}$$

Equation 7: Natural Frequency in relation to length

$$l = \frac{g}{w_n^2}$$

Equation 8: Length

Table 1 sums up all the obtained values for the pendulum calibration data.

τ (Time Constant) (s)	1.39216666667
q (Ratio between Consecutive Peaks)	0.826955394539
w_d (Damped Frequency) (rad/s)	4.51324216954
σ (Decay Rate)	0.136481160218
w_n (Natural Frequency) (rad/s)	4.51530530396
ζ (Damping Coefficient)	0.0302263415274
l_{eff} (Characteristic Pendulum Length) (m)	0.481165816499

Table 1: *Pendulum Calibration Data Values*

Initial System (3-Pole Balancing) Documentation

To establish a baseline for our PI angle controller, we first needed to define our system's physical constraints. To account for slight manufacturing variations between the two motors and simplify our calculations, we averaged our empirically determined motor constants, yielding:

$$\alpha_{avg} = \frac{22.3146 + 22.5964}{2} = 22.4555$$
$$\beta_{avg} = \frac{0.0021 + 0.0021}{2} = 0.0021$$

With the motor parameters established, we determined our target closed-loop poles. Based on the system's characteristic equation, the sum of the three poles is constrained by the motor's time constant, permanently locking the average of the three poles at $-\frac{\alpha}{3}$. Because system stability requires the poles to be placed as far into the left-half of the complex s-plane as possible, we set the real component of all three poles to exactly $-\frac{\alpha}{3}$.

To finalize the pole locations, we had to determine the value for q, which represents the imaginary component of the dominant pole pair. As outlined in the project resources, introducing a non-zero q only serves to increase the required control gains. To minimize the control effort and keep the gains as low as possible while maintaining stability, we set $q = 0$. This configuration places all three closed-loop poles purely on the real axis.

Theoretical Control Constant Calculations

Using our chosen pole locations, we analytically derived our Proportional K_p and Integral K_i control gains. The relationship between the target poles and the gains is defined by the following equations:

$$K_P = \frac{l_{eff}}{\alpha\beta}\left(\frac{\alpha^2}{3} + q^2 + \omega_n^2\right)$$
$$K_i = \frac{l_{eff}}{\alpha\beta}\left(\frac{\alpha^3}{27} + \frac{\alpha q^2}{3} + \alpha\omega_n^2\right)$$

By substituting our measured physical parameters ($\alpha = 22.4555$, $\beta = 0.0021$, $l_{eff} = 0.481\,m$, $\omega_n = 4.515\,rad/s$) alongside our chosen value of $q = 0$ into the equations, our MATLAB script calculated the theoretical gains necessary to stabilize the idealized system.

$$K_P = 1923.1$$
$$K_i = 8950.6$$

As expected from our target polynomial, applying these gains placed all three closed-loop poles exactly at:

$$pole\ 1:\ s\ =\ -7.49$$
$$pole\ 2:\ s\ =\ -7.49$$
$$pole\ 3:\ s\ =\ -7.49$$

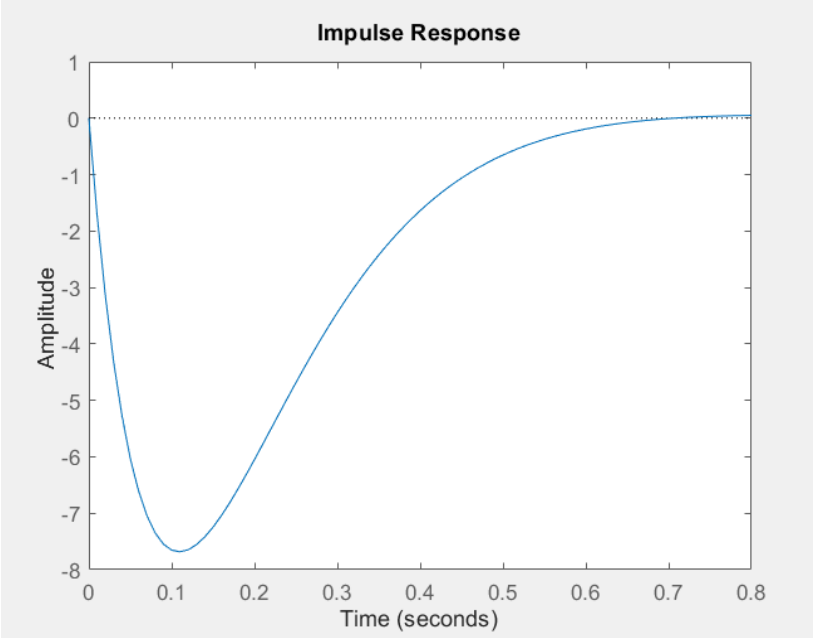


Figure 6: *Impulse Response for Three-Pole System*

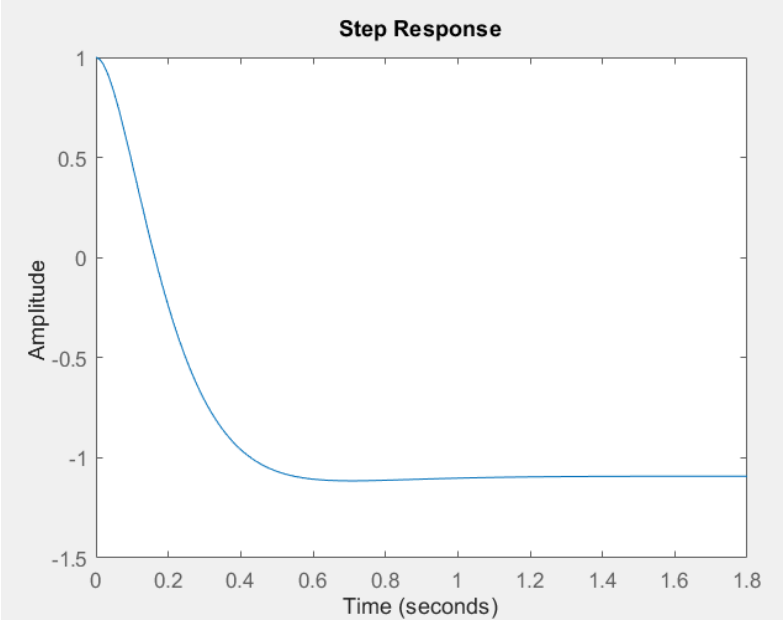


Figure 7: *Step Response for Three-Pole System*

We created a Simulink model to evaluate our Kp and Ki values and determine whether the system is stable. We have our desired angle at 0 and are putting that into our PI Controller and a

motor Controller. We then connect a block that measures velocity, a block that measures position, a block that goes into our cart-pole transfer function to measure the output, and a disturbance.

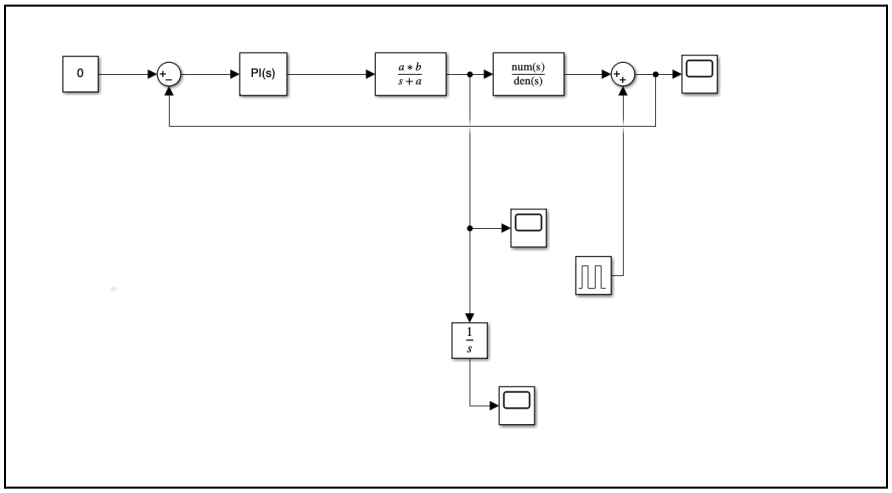


Figure 8: Block Diagram for 3-Pole Simulink Model

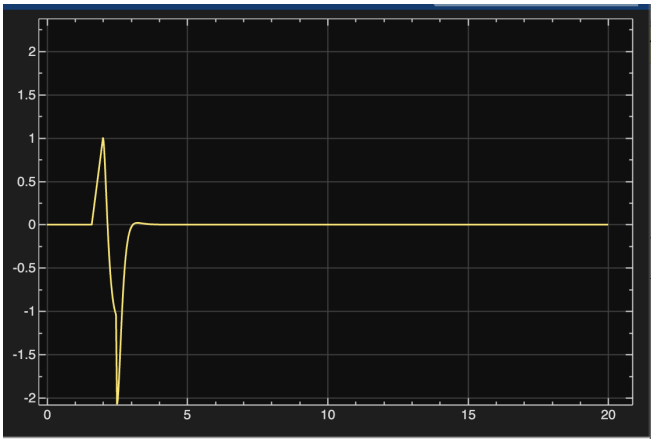


Figure 9: Simulink Plot for Angle Output

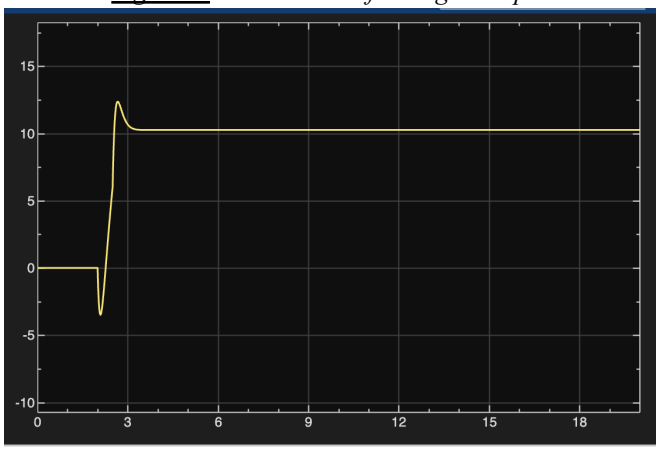


Figure 10: Simulink Plot for Velocity Output

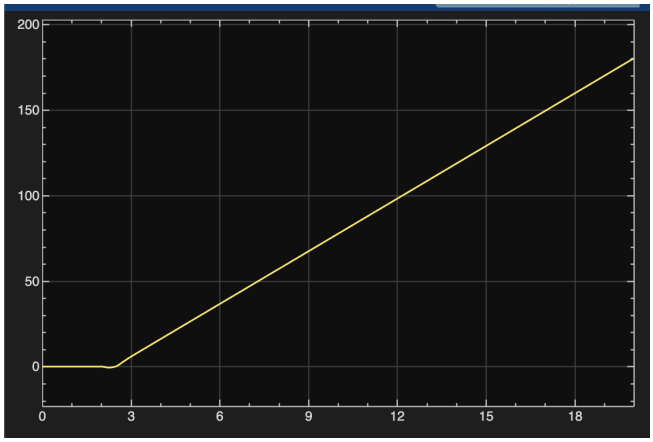


Figure 11: Simulink Plot for Position Output

In terms of the output angle, velocity and position, they all expect as predicted compared to the actual pendulum behavior. In other words, the angle will stabilize again at 0º since the pendulum will attempt to redirect itself to an upright position. On the other hand, since the Rocky does not care about positioning, the position will keep increasing at a constant rate, and that "constant rate" is equal to a constant velocity.

Hardware Implementation and Empirical Tuning

While the theoretical gains successfully stabilized the idealized continuous-time Simulink model, testing them on the physical Rocky robot resulted in severe instability. This discrepancy is primarily due to unmodeled real-world dynamics, specifically the 10-millisecond digital delay of the Arduino's control loop, motor backlash, and the hard voltage limit (motor saturation cap of $\pm\ 300$). The idealized high gains caused the physical robot to violently overcorrect, vibrate, and trip its 45-degree safety shutoff. To achieve empirical stability on the hardware, we scaled down the theoretical gains through iterative tuning to prevent motor saturation. This resulted in our final empirically tuned control constants:

$$K_{P,\,tuned} = 1334.53$$

$$K_{i,\,tuned} = 4642.05$$

Stationary Balancing System (5-Pole) Documentation

While the 3-pole PI controller successfully maintained Rocky's upright angle, it lacked spatial awareness, causing the robot to drift across the floor over time. To achieve a true stationary balancing, an outer control loop was added with the inner angle loop. This outer loop acts as a position and velocity controller with the transfer function:

$$J(s) = J_p + \frac{J_i}{s} + \frac{C_i}{s^2}$$

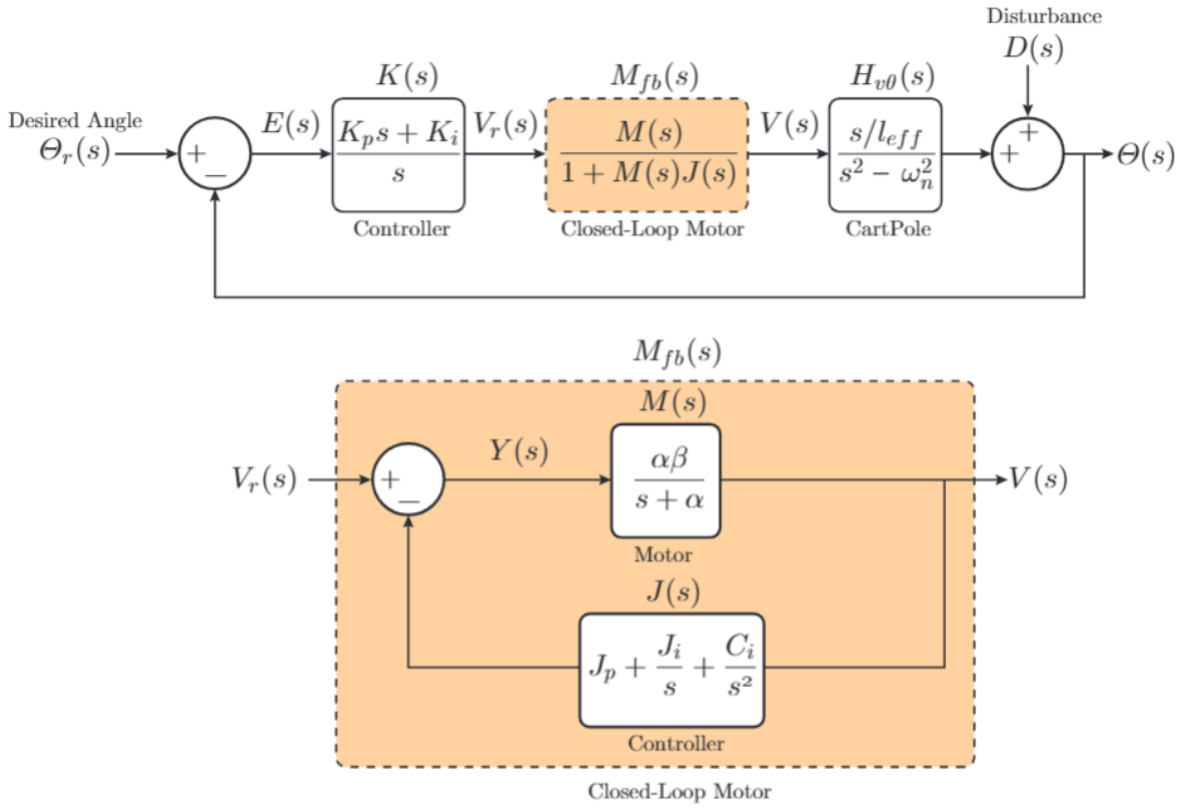


Figure 12: 5 Pole Block Diagram

Adding this controller increases the overall system to a 5th order system, requiring the placement of five closed-loop poles. Rather than arbitrarily guessing all five pole locations, we used the same formula to calculate poles 1, 2, and 3. For the remaining two poles, which dictate the outer-loop position control, we placed them very close to the origin at -0.01. This makes sure that the robot's drive to return to its starting coordinate is slow and gradual, preventing aggressive acceleration that would otherwise destabilize the fast inner balancing loop. Our target poles were:

$$pole\ 1\text{: } s = -7.49$$
$$pole\ 2\text{: } s = -7.49$$
$$pole\ 3\text{: } s = -7.49$$
$$pole\ 4\text{: } s = -.01$$
$$pole\ 5\text{: } s = -.01$$

This is the same transfer function as the three-pole system except we replace the motor transfer function with a closed-loop motor transfer function. We use a controller of the form:

$$Y(s) = (J_p + \frac{J_i}{s} + \frac{C_i}{s^2})(V_r(s) - V(s))$$

We can replace $V_r(s) - V(s))$ with $\Delta V(s)$ and implement it in the time domain as:

$$y(t) = -J_p \Delta v(t) - J_i \Delta x(t) - C_i \int \Delta x(t) dt$$

Solving these gave us theoretical values for K_p, K_i, J_i, J_p, C_i. These are used for the Proportional (Kp) and Integral (Ki) control of the robot's angle (with Ki specifically helping to eliminate steady-state angular error), and for the Proportional (Jp), Integral (Ji), and Double-Integral (Ci) control in the inner loop to manage the robot's position and velocity, ensuring returns back to its starting position after a disturbance.

$$K_P = 2011.8$$
$$K_i = 9339.6$$
$$J_i = -88.9888$$
$$J_p = 4.2412$$
$$C_i = -4.3620$$

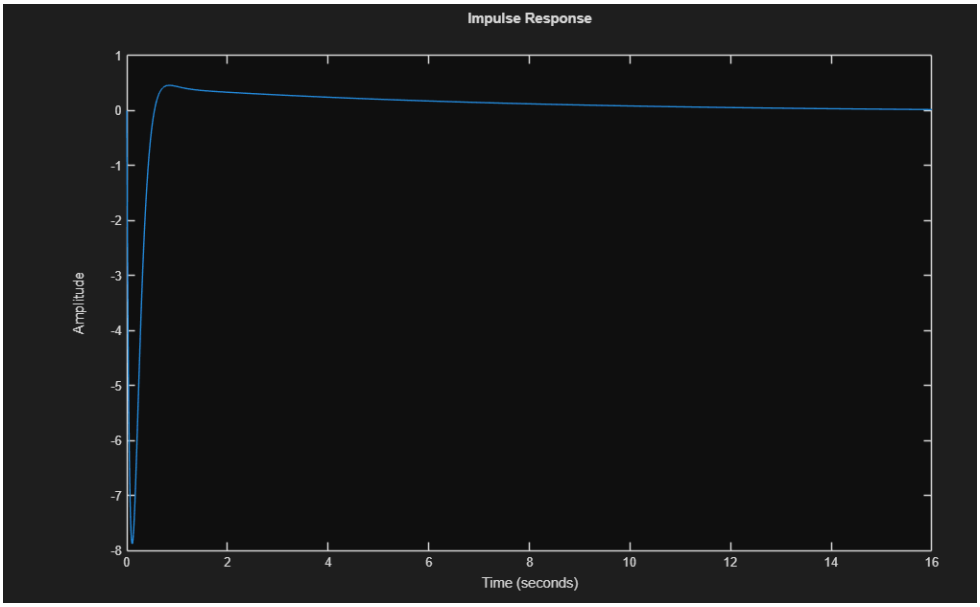


Figure 13: *Impulse Response for Five-Pole System*

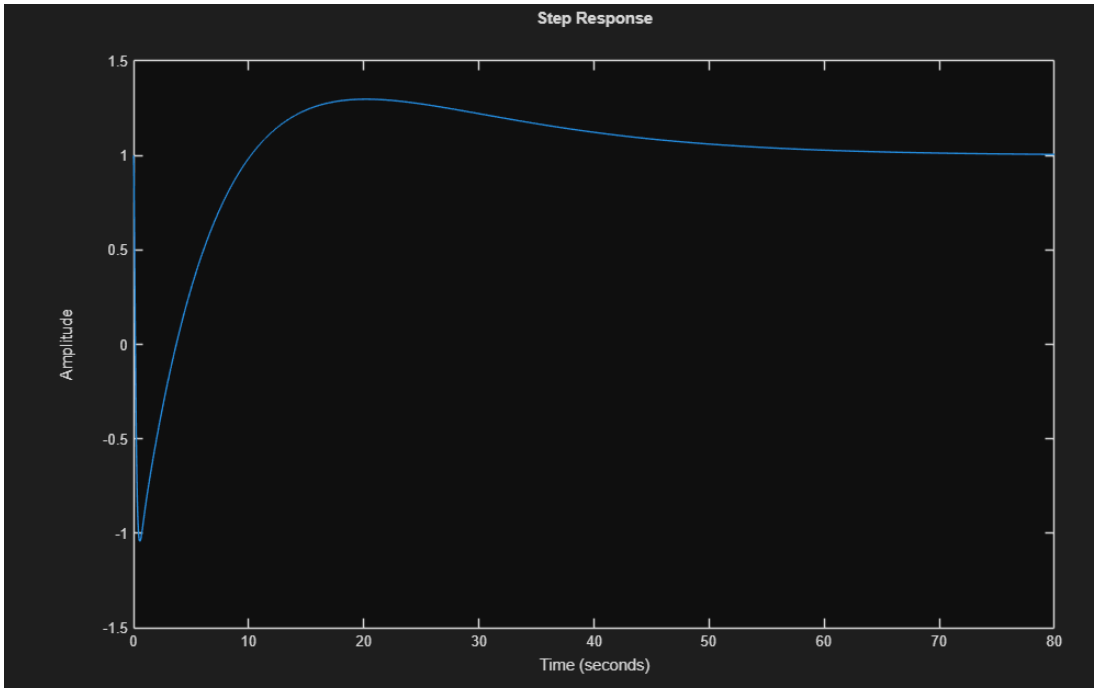


Figure 14: *Step Response for Five-Pole System*

We also created a Simulink model for our five-pole system. This is the same system as the three-pole system, except we replaced the open-loop motor controller with the closed-loop motor control. We are showing the output of angle as well as the position and velocity of the rocky after balancing out a disturbance.

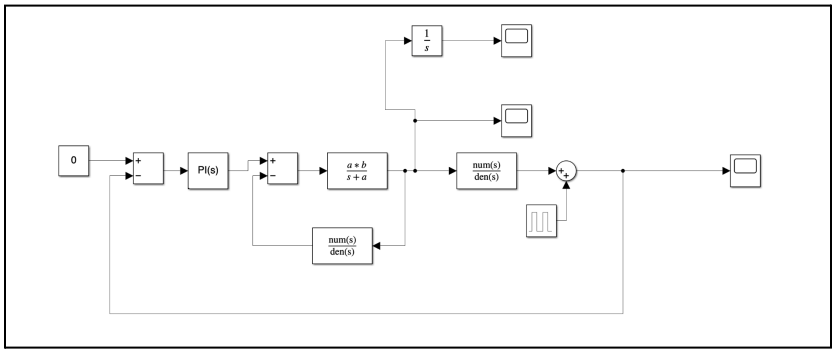


Figure 15: Block Diagram for five pole in Simulink

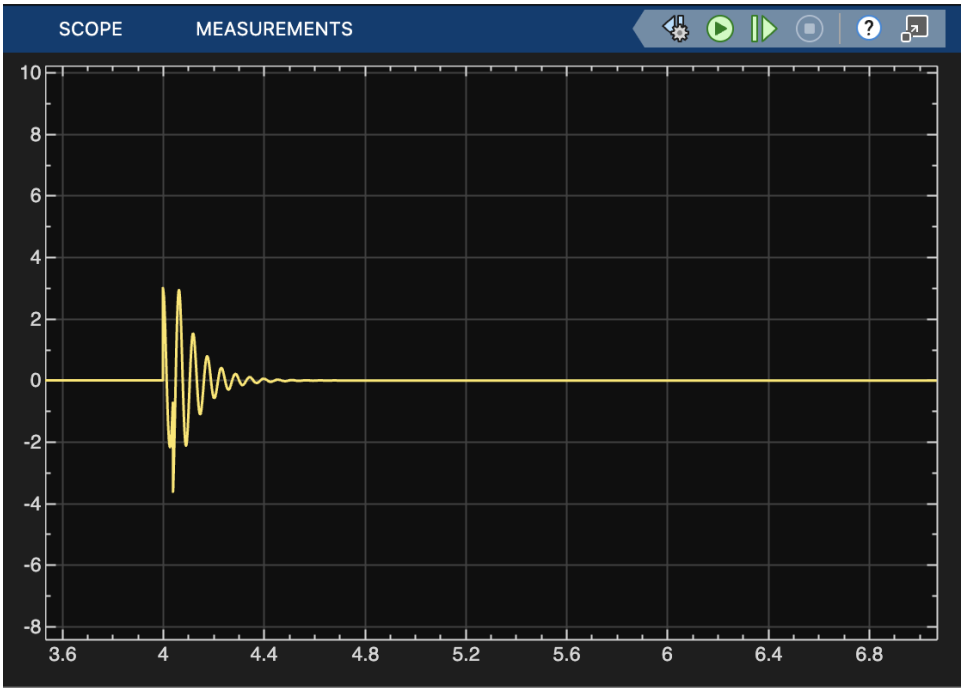


Figure 16: Angle Output for five pole system

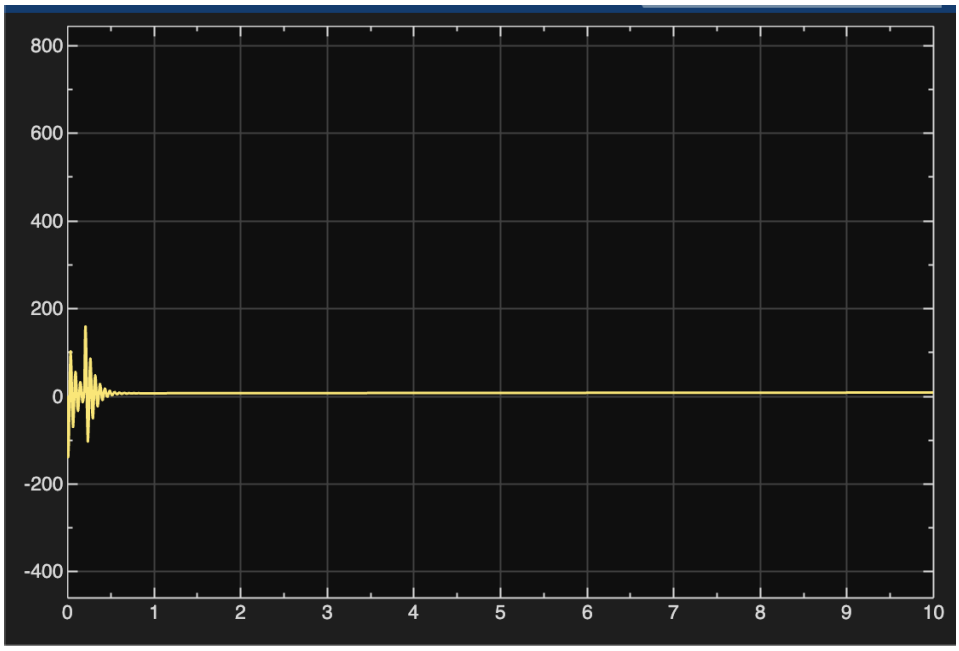


Figure 17: Velocity Output for five pole system

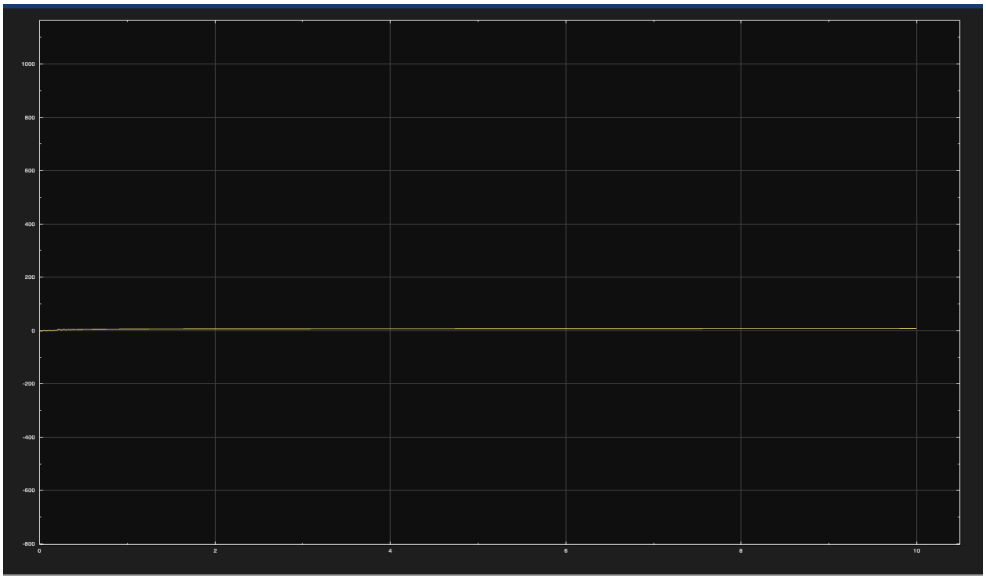


Figure 18: Position Output for five pole system

As expected for the behavior of the five pole system, which finished at an upright angle, with velocity null and net displacement being zero, or in other words, the “rocky” returns to its original position at steady state. The results didn’t converge in the way we expected, with the final velocity not being exactly zero, but very close and approximately zero, and when that small area between 0 and the steady state velocity is integrated for position, the position also is not at exactly 0.

While the theoretical gains successfully stabilized the idealized continuous-time Simulink model, testing them on the physical Rocky robot resulted in severe instability. To achieve a stable stationary hover on the hardware, we had to abandon the strict theoretical numbers and transition to a "guess and check" empirical tuning method based on our real-time physical observations. We observed that the outer-loop position demands were causing integral windup, so we aggressively scaled down J_p, J_i, C_i to soften its return to the origin. We then iteratively adjusted the K_p and K_i inner-loop gains until the physical jitter was eliminated. This real-world guess-and-check process resulted in our final, hardware-stable control constants:

$$K_{P,\,tuned} = 1334.53$$
$$K_{i,\,tuned} = 4642.05$$
$$J_{i,\,tuned} = -8.78864$$
$$J_{p,\,tuned} = 21.34$$
$$C_{i,\,tuned} = -2.15375$$

Youtube Video in action:
https://youtube.com/shorts/h7tqY11KXBA?si=5MnYEz_azDOXNGJj

Code:

Rocky_closed_loop_poles23:

```
% Rocky_closed_loop_poles_23.m
%
% 1) Symbolically calculates closed loop transfer function of a disturbannce
% rejection PI control system for Rocky.
% No motor model (M =1). With motor model (1st order TF)
%
% 2) Specify location of (target)poles based on desired reponse. The number of
% poles = denominator polynomial of closed loop TF
%
% 3) Extract the closed loop denomiator poly and set = polynomial of target
% poles
%
% 4) Solve for Ki, Kp  to match coefficients of polynomials. In general,
% this will be underdefined and will not be able to place poles in exact
% locations. In this, the control constants can be found exactly
%
% 5) Plot impulse and step response to see closed-loop behavior.
%
% based on code by SG. last modified 2/25/23 CL
clear all;
close all;
syms s a b l g Kp Ki    % define symbolic variables
```

```
Hvtheta = -s/l/(s^2-g/l);       % TF from velocity to angle of pendulum
K = Kp + Ki/s;                  % TF of the PI angle controller
M = a*b/(s+a)                   % TF of motor (1st order model)
%  M = 1;                       % TF without motor
%
%
% closed loop transfer function from disturbance d(t)totheta(t)
Hcloop = 1/(1-Hvtheta*M*K)    % use this for no motor feedback
pretty(simplify(Hcloop))       % to display the total transfer function
% Substitute parameters and solve
% system parameters
g = 9.81;
l = 0.481165816499;   %effective length
a = 22.4555;           %nominal motor parameters
b = 0.0021;        %nominal motor parameters
Hcloop_sub = subs(Hcloop) % sub parameter values into Hcloop
% specify locations of the target poles,
% choose # based on order of Htot denominator
% e.g., want some oscillations, want fast decay, etc.
p1 = -a/3;   % dominant pole pair
p2 = -a/3;     % dominant pole pair
p3 = -a/3;
% target characteristic polynomial
% if motor model (TF) is added, order of polynomial will increases
tgt_char_poly = (s-p1)*(s-p2)*(s-p3)
npoly = 3
```

```
% get the denominator from Hcloop_sub
[n d] = numden(Hcloop_sub)
% find the coefficients of the denominator polynomial TF
coeffs_denom = coeffs(d, s)
% divide though the coefficient of the highest power term
coeffs_denom = coeffs(d, s)/(coeffs_denom(end))
% find coefficients of the target charecteristic polynomial
coeffs_tgt = coeffs(tgt_char_poly, s)
% polynomial in the target to the actual polynomials
solutions = solve(coeffs_denom(1:npoly-1) == coeffs_tgt(1:npoly-1),  Kp, Ki)
% display the solutions as double precision numbers
Kp = double(solutions.Kp)
Ki = double(solutions.Ki)
% reorder coefficients for the check polynomial
for ii = 1:length(coeffs_denom)
   chk_coeffs_denom(ii) = coeffs_denom(length(coeffs_denom) + 1 - ii);
end
closed_loop_poles = vpa (roots(subs(chk_coeffs_denom)), npoly )
% Plot impulse response of closed-loop system
   TFstring = char(subs(Hcloop));
   % Define 's' as transfer function variable
   s = tf('s');
   % Evaluate the expression
   eval(['TFH = ',TFstring]);
   figure (1)
   impulse(TFH);   %plot the impulse reponse
```

```
figure(2)
step(TFH)       %plot the step response
```

Rocky_5_closed_poles:

```
% Rocky_5_closed_loop_poles.m
%
% 1) Symbolically calculates closed loop transfer function of PI disturbannce
% rejection control system for Rocky.
% No motor model (M =1). With motor model (1st order TF)
%
% 2) Specify location of (target)poles based on desired reponse. The number of
% poles = denominator polynomial of closed loop TF
%
% 3) Extract the closed loop denomiator poly and set = polynomial of target
% poles
%
% 4) Solve for Ki, Kp, Ji, Jp, Ci to match coefficients of polynomials. In
general,
% this will be underdefined and will not be able to place poles in exact
% locations. In this case (5th order), the control constants can be found
exactly
%
% 5) Plot impulse response to see closed-loop behavior.
%
% based on code by SG. last modified 3/8/22 CL
clear all;
close all;
```

```
syms s a b l g Kp Ki Jp Ji Ci    % define symbolic variables

Hvtheta = -s/l/(s^2-g/l);        % TF from velocity to angle of pendulum

K = Kp + Ki/s;                   % TF of the PI angle controller

M = a*b/(s+a);                   % TF of motor (1st order model)

% M = 1;                         % TF without motor

%

J = Jp + Ji/s + Ci/s^2;          % TF of controller around motor-combined PI of
x and v

Mfb = M/(1+M*J);                 % Black's formula to get tf for motor with PI
feedback control

%

% closed loop transfer function from disturbance d(t)totheta(t)

% Hcloop = 1/(1-Hvtheta*M*K)     % use this for no motor feedback

% with motor feedback

Hcloop = 1/(1-Hvtheta*Mfb*K)     % use this for motor with feedback

pretty(simplify(Hcloop))        % to display the total transfer function

% Substitute parameters and solve

% system parameters

g = 9.81;

l = 0.481165816499;   %effective length

a = 22.4555;           %nominal motor parameters

b = 0.0021;         %nominal motor parameters

Hcloop_sub = subs(Hcloop) % sub parameter values into Hcloop

% specify locations of the target poles,

% choose # based on order of Htot denominator

% e.g., want some oscillations, want fast decay, etc.

% p1 = -1 + 2*pi*i    % dominant pole pair
```

```
% p2 = -1 - 2*pi*i    % dominant pole pair

% p3 = -10

% p4 = -8

% p5 = -8.

p1 = -a/3   % dominant pole pair

p2 = -a/3    % dominant pole pair

p3 = -a/3

p4 = -0.1    % dominant pole pair

p5 = -0.1    % dominant pole pair

% target characteristic polynomial

% if motor model (TF) is added, order of polynomial will increases

% tgt_char_poly = (s-p1)*(s-p2)*(s-p3)

% check polynomial-expand to fifth order

tgt_char_poly = (s-p1)*(s-p2)*(s-p3)*(s-p4)*(s-p5)

exp_tgt_char_poly = expand(tgt_char_poly)

% get the denominator from Hcloop_sub

[n d] = numden(Hcloop_sub)

% find the coefficients of the denominator polynomial TF

coeffs_denom = coeffs(d, s)

% divide though the coefficient of the highest power term

coeffs_denom = coeffs(d, s)/(coeffs_denom(end))

% num_coeff_denom = length(coeffs_denom)

% find coefficients of the target charecteristic polynomial

coeffs_tgt = coeffs(tgt_char_poly, s)

% num_coeff_tgt = length(coeffs_tgt)

% for check. reorder the coefficients to match the denomimator polynomial
```

```
for ii = 1:length(coeffs_denom)
   reord_coeffs_tgt(ii) = coeffs_tgt(length(coeffs_tgt) + 1 - ii);
end
% check roots of target polynomial-should be same as selected poles
roots_target = vpa(roots(reord_coeffs_tgt),4)
% solve the system of equations setting the coefficients of the
% polynomial in the target to the actual polynomials
solutions = solve(coeffs_denom(1:5) == coeffs_tgt(1:5), Jp, Ji,  Kp, Ki, Ci);
% display the solutions as double precision numbers
Kp = double(solutions.Kp)
Ki = double(solutions.Ki)
Ji = double(solutions.Ji)
Jp = double(solutions.Jp)
Ci = double(solutions.Ci)
%write out denominator polynomial
aaa = vpa(subs(coeffs_denom),4)
% reorder coefficients for the check polynomial
for ii = 1:length(coeffs_denom)
   chk_coeffs_denom(ii) = coeffs_denom(length(coeffs_denom) + 1 - ii);
end
% check poles should be same as chosen input poles
check_closed_loop_poles = vpa (roots(subs(chk_coeffs_denom)), 4)
% write out target polynomial
% bbb = vpa( expand(
(s-check_closed_loop_poles(1))*(s-check_closed_loop_poles(2)) ...
%     *(s-check_closed_loop_poles(3))*(s-check_closed_loop_poles(4)) ...
```

```
%     *(s-check_closed_loop_poles(5)) ) )
% Plot impulse and step responses of closed-loop system
   TFstring = char(subs(Hcloop));
   % Define 's' as transfer function variable
   s = tf('s');
   % Evaluate the expression
   eval(['TFH = ',TFstring]);
   figure (1)
   impulse(TFH);   %plot the impulse reponse
   figure(2)
   step(TFH)       %plot the step response
```

Rocky_Balance_Starter_Code25:

```
// Start the robot flat on the ground
// compile and load the code
// wait for code to load (look for "done uploading" in the Arduino IDE)
// wait for red LED to flash on board
// gently lift body of rocky to upright position
// this will enable the balancing algorithm
// wait for the buzzer
// let go
//
// The balancing algorithm is implemented in BalanceRocky()
// which you should modify to get the balancing to work
//
```

```
#include <Balboa32U4.h>
#include <Wire.h>
#include <LSM6.h>
#include "Balance.h"



extern int32_t angle_accum;
extern int32_t speedLeft;
extern int32_t driveLeft;
extern int32_t distanceRight;
extern int32_t speedRight;
extern int32_t distanceLeft;
extern int32_t distanceRight;
float speedCont = 0;
float displacement_m = 0;
int16_t limitCount = 0;
uint32_t cur_time = 0;
float distLeft_m;
float distRight_m;
```

```
extern uint32_t delta_ms;

float measured_speedL = 0;

float measured_speedR = 0;

float desSpeedL=0;

float desSpeedR =0;

float dist_accumL_m = 0;

float dist_accumR_m = 0;

float dist_accum = 0;

float speed_err_left = 0;

float speed_err_right = 0;

float speed_err_left_acc = 0;

float speed_err_right_acc = 0;

float errAccumRight_m = 0;

float errAccumLeft_m = 0;

float prevDistLeft_m = 0;

float prevDistRight_m = 0;

float angle_rad_diff = 0;

float angle_rad;              // this is the angle in radians

float angle_rad_accum = 0;    // this is the accumulated angle in radians

float angle_prev_rad = 0; // previous angle measurement

extern int32_t displacement;

int32_t prev_displacement=0;

uint32_t prev_time;
```

```
#define G_RATIO (162.5)



LSM6 imu;

Balboa32U4Motors motors;

Balboa32U4Encoders encoders;

Balboa32U4Buzzer buzzer;

Balboa32U4ButtonA buttonA;



#define FIXED_ANGLE_CORRECTION (0.262)  // ***** Replace the value 0.25 with the value you obtained from the Gyro calibration procedure



////////////////////////////////////////////////////////////////////////////////////////

// This is the main function that performs the balancing

// It gets called approximately once every 10 ms  by the code in loop()

// You should make modifications to this function to perform your
```

```
// balancing

////////////////////////////////////////////////////////////////////////////////////////////

void BalanceRocky()

{


   // ***************Enter the control parameters here


  float Kp = 2001.8 / 1.5;

  float Ki = 9284.1 / 2.0;

  float Ci = -4.3075 / 2.0;

  float Jp = 4.268 * 5.0;

  float Ji = -87.8864 / 10.0;









   float v_c_L, v_c_R; // these are the control velocities to be sent to the motors

   float v_d = 0; // this is the desired speed produced by the angle controller



   // Variables available to you are:

   // angle_rad  - angle in radians
```

```
  // angle_rad_accum - integral of angle
  // measured_speedR - right wheel speed (m/s)
  // measured_speedL - left wheel speed (m/s)
  // distLeft_m - distance traveled by left wheel in meters
  // distRight_m - distance traveled by right wheel in meters  (this is the integral of the velocities)
  // dist_accum - integral of the distance

  // *** enter an equation for v_d in terms of the variables available ****
   float avg_speed = (measured_speedL + measured_speedR) / 2.0;
   float avg_dist = (distLeft_m + distRight_m) / 2.0;

   v_d = (Kp * angle_rad) + (Ki * angle_rad_accum) + (Jp * avg_speed) + (Ji * avg_dist) + (Ci * dist_accum);
// this is the desired velocity from the angle controller


  // The next two lines implement the feedback controller for the motor. Two separate velocities are calculated.
  //
  //
  // We use a trick here by criss-crossing the distance from left to right and
  // right to left. This helps ensure that the Left and Right motors are balanced

  // *** enter equations for input signals for v_c (left and right) in terms of the variables available ****
   float sync_gain = 1000.0;
```

```
v_c_R = v_d + (sync_gain * (distLeft_m - distRight_m));

v_c_L = v_d + (sync_gain * (distRight_m - distLeft_m));









// save desired speed for debugging

desSpeedL = v_c_L;

desSpeedR = v_c_R;


// the motor control signal has to be between +- 300. So clip the values to be within that range

// here

if(v_c_L > 300) v_c_L = 300;

if(v_c_R > 300) v_c_R = 300;

if(v_c_L < -300) v_c_L = -300;

if(v_c_R < -300) v_c_R = -300;


// Set the motor speeds

motors.setSpeeds((int16_t) (v_c_L), (int16_t)(v_c_R));
```

```
}



void setup()
{
  // Uncomment these lines if your motors are reversed.
  // motors.flipLeftMotor(true);
  // motors.flipRightMotor(true);

  Serial.begin(9600);
  prev_time = 0;
  displacement = 0;
  ledYellow(0);
  ledRed(1);
  balanceSetup();
  ledRed(0);
  angle_accum = 0;

  ledGreen(0);
  ledYellow(0);
}
```

```
int16_t time_count = 0;
extern int16_t angle_prev;
int16_t start_flag = 0;
int16_t start_counter = 0;
void lyingDown();
extern bool isBalancingStatus;
extern bool balanceUpdateDelayedStatus;

void UpdateSensors()
{
  static uint16_t lastMillis;
  uint16_t ms = millis();

  // Perform the balance updates at 100 Hz.
  balanceUpdateDelayedStatus = ms - lastMillis > UPDATE_TIME_MS + 1;
  lastMillis = ms;

  // call functions to integrate encoders and gyros
  balanceUpdateSensors();

  if (imu.a.x < 0)
```

```
  {
    lyingDown();
    isBalancingStatus = false;
  }
  else
  {
    isBalancingStatus = true;
  }
}



void GetMotorAndAngleMeasurements()
{
    // convert distance calculation into meters
    // and integrate distance
    distLeft_m = ((float)distanceLeft)/((float)G_RATIO)/12.0*80.0/1000.0*3.14159;
    distRight_m = ((float)distanceRight)/((float)G_RATIO)/12.0*80.0/1000.0*3.14159;
    dist_accum += (distLeft_m+distRight_m)*0.01/2.0;

    // compute left and right wheel speed in meters/s
    measured_speedL = speedLeft/((float)G_RATIO)/12.0*80.0/1000.0*3.14159*100.0;
    measured_speedR = speedRight/((float)G_RATIO)/12.0*80.0/1000.0*3.14159*100.0;
```

```
    prevDistLeft_m = distLeft_m;
    prevDistRight_m = distRight_m;



    // this integrates the angle
    angle_rad_accum += angle_rad*0.01;
    // this is the derivative of the angle
    angle_rad_diff = (angle_rad-angle_prev_rad)/0.01;
    angle_prev_rad  = angle_rad;

}

void balanceResetAccumulators()
{
    errAccumLeft_m = 0.0;
    errAccumRight_m = 0.0;
    speed_err_left_acc = 0.0;
    speed_err_right_acc = 0.0;
}
```

```
void loop()

{

  static uint32_t prev_print_time = 0;   // this variable is to control how often we print on the serial monitor

  int16_t distanceDiff;    // this stores the difference in distance in encoder clicks that was traversed by the right vs the left wheel

  static float del_theta = 0;

  char enableLongTermGyroCorrection = 1;



  cur_time = millis();                  // get the current time in miliseconds



  if((cur_time - prev_time) > UPDATE_TIME_MS){

    UpdateSensors();                    // run the sensor updates.



    // calculate the angle in radians. The FIXED_ANGLE_CORRECTION term comes from the angle calibration procedure (separate sketch available for this)

    // del_theta corrects for long-term drift

    angle_rad = ((float)angle)/1000/180*3.14159 - FIXED_ANGLE_CORRECTION - del_theta;



    if(angle_rad > 0.1 || angle_rad < -0.1)      // If angle is not within +- 6 degrees, reset counter that waits for start

    {

      start_counter = 0;

    }
```

```
  if(angle_rad > -0.1 && angle_rad < 0.1 && ! start_flag)
  {
    // increment the start counter
    start_counter++;
    // If the start counter is greater than 30, this means that the angle has been within +- 6 degrees for 0.3 seconds, then set the start_flag
    if(start_counter > 30)
    {
      balanceResetEncoders();
      start_flag = 1;
      buzzer.playFrequency(DIV_BY_10 | 445, 1000, 15);
      Serial.println("Starting");
      ledYellow(1);
    }
  }


  // every UPDATE_TIME_MS, if the start_flag has been set, do the balancing
  if(start_flag)
  {
    GetMotorAndAngleMeasurements();
```

```
    if(enableLongTermGyroCorrection)
      del_theta = 0.999*del_theta + 0.001*angle_rad;  // assume that the robot is standing. Smooth out the angle to correct for long-term gyro drift

    // Control the robot
    BalanceRocky();
  }
  prev_time = cur_time;
  }
// if the robot is more than 45 degrees, shut down the motor
  if(start_flag && angle_rad > .78)
  {
    motors.setSpeeds(0,0);
    start_flag = 0;
  }
  else if(start_flag && angle_rad < -0.78)
  {
    motors.setSpeeds(0,0);
    start_flag = 0;
  }

// kill switch
  if(buttonA.getSingleDebouncedPress())
```

```
  {
    motors.setSpeeds(0,0);
    while(!buttonA.getSingleDebouncedPress());
  }

if(cur_time - prev_print_time > 103)   // do the printing every 105 ms. Don't want to do it for an integer multiple of 10ms to not hog the processor
  {
      Serial.print(angle_rad);
      Serial.print("\t");
      Serial.print(distLeft_m);
      Serial.print("\t");
      Serial.print(measured_speedL);
      Serial.print("\t");
      Serial.print(measured_speedR);
      Serial.print("\t");
     Serial.println(speedCont);
     prev_print_time = cur_time;
  }


}
```