Rocky Project — Oscar Bao, Danny Kennen, Sam Wisnoski

Section 0: Introduction

Our goal for this project is to design control systems that keep our robot, Rocky, upright when it's subject to an impact disturbance. For this project, we will apply quantitative analysis using analytical, symbolic, and numerical tools, not to an idealization, but to a real system in a real way. Let's get started!

Section 1: System Calibration

In order to ensure Rocky can balance itself, we need to find parameters for the first order motor system. To accomplish this, we recorded the output values of the left and right motors, as well as time elapsed in milliseconds while running the motors on the carpet. Next, we took the output values for both motors and plotted them against the time. We then derived and plotted an equation for the estimated output of the motors, $M(t) = 300K * (1 - e^{-t/\tau})$. In order to find the time constant, τ, and the force constant, K, we chose values that seemed reasonable and adjusted them in order to find an estimated output that worked with our recorded motor outputs, as shown in **Figure 1-1.** The time constant determines exponential decay while the force constant tells us the slope the plot initially has. After testing a few different values, we eventually determined that our $\tau = 0.0625$ and our $K = 0.0027$.

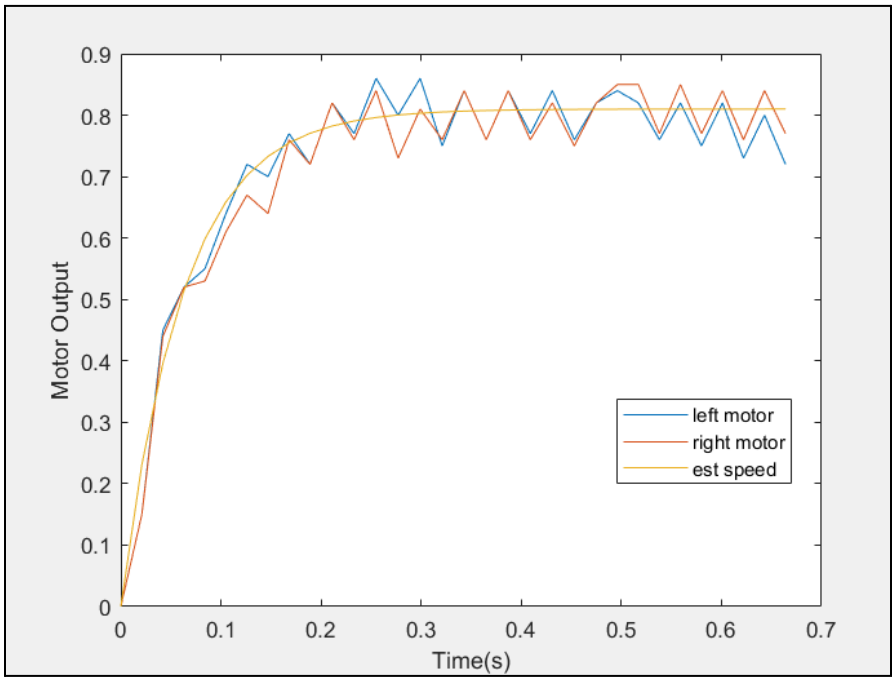

Figure 1-1. Plot of the recorded motor outputs vs. time and the estimated output vs. time.

Once we found our motor parameters, our next step was to determine the natural frequency and effective length of the system. To do this, we used the "Rocky_Gyro_Callibration.ino" script provided by the teaching team to record the angle of Rocky as we swung it back and forth. We then plotted this data and found the peaks, as shown in **Figure 1-2.** Taking the distance between peaks, we determined that the period of the system to be $T = 1.4117\ sec$, and the natural frequency to be $\omega_n = \frac{2\pi}{T} = 4.4509\ (rad/s)$. Lastly, we solved for effective length using the equation $l_{eff} = \frac{g}{\omega_n^2} = 0.4947\ m$.

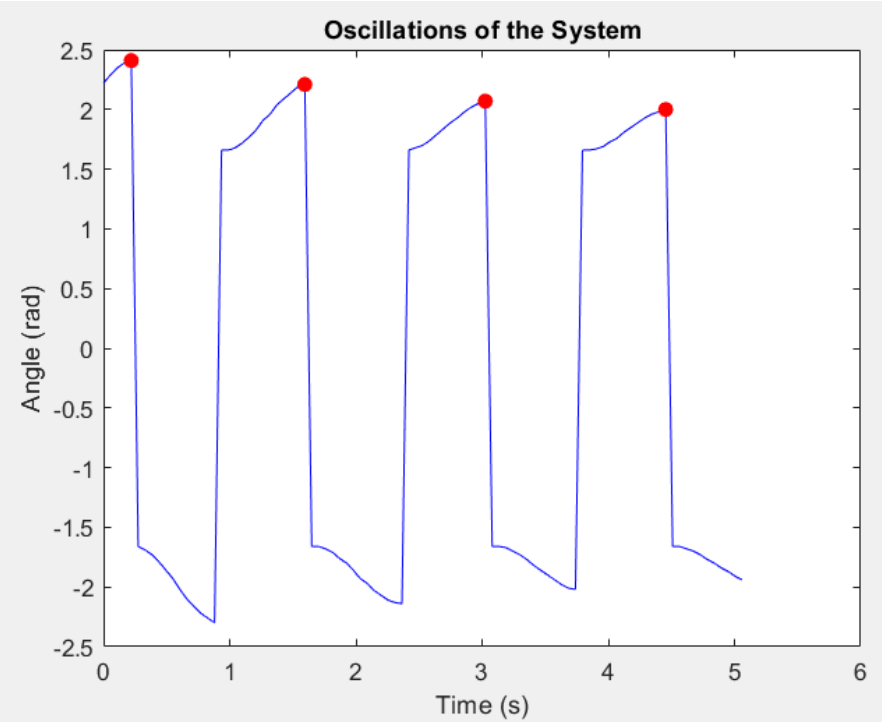

Figure 2-2. Oscillations of Rocky as Pendulum

Section 2: Initial System

The first step of creating a self-balancing robot is to stabilize the angle of the inverse pendulum using a PI controller. Therefore, our next step is to determine the poles and the *Kp* and *Ki* values for our initial system (which contains a total of three poles). As we learned from previous classes, for a complex pole $a + bi$, a denotes the decay rate of oscillations for disturbance rejection, and b denotes the frequency of oscillations after a disturbance. With these in mind, we started placing our poles. We settled our first and second pole—the complex conjugate—at $-5 \pm 5i$, finding both the decay rate (*Re*) and frequency (*Im*) experimentally. The values make sense logically as well — we needed the system to be underdamped in order for the feedback loop to work, but we also wanted the angle response to stabilize as quickly as possible. Additionally, our real components needed to be negative to ensure exponential decay. For our

last pole, we settled on -8, as we wanted to make the first two poles dominant, and a more negative *Re* component (further from the imaginary axis) causes less impact to the system.

Once we had our poles, we needed to actually compute our Kp and Ki values. We did this using the "Rocky_closed_loop_poles_23.m" script provided by the teaching team. We input our poles as $p1 = -5 - 5i, p2 = -5 + 5i,$ and $p3 = -8$ to get $K_p = 4,280$ and $K_i = 19,361$.

Our simulink models and resultant angle, velocity, and position graphs are shown in **Figures 2-1, 2-2, 2-3**, and **2-4**.

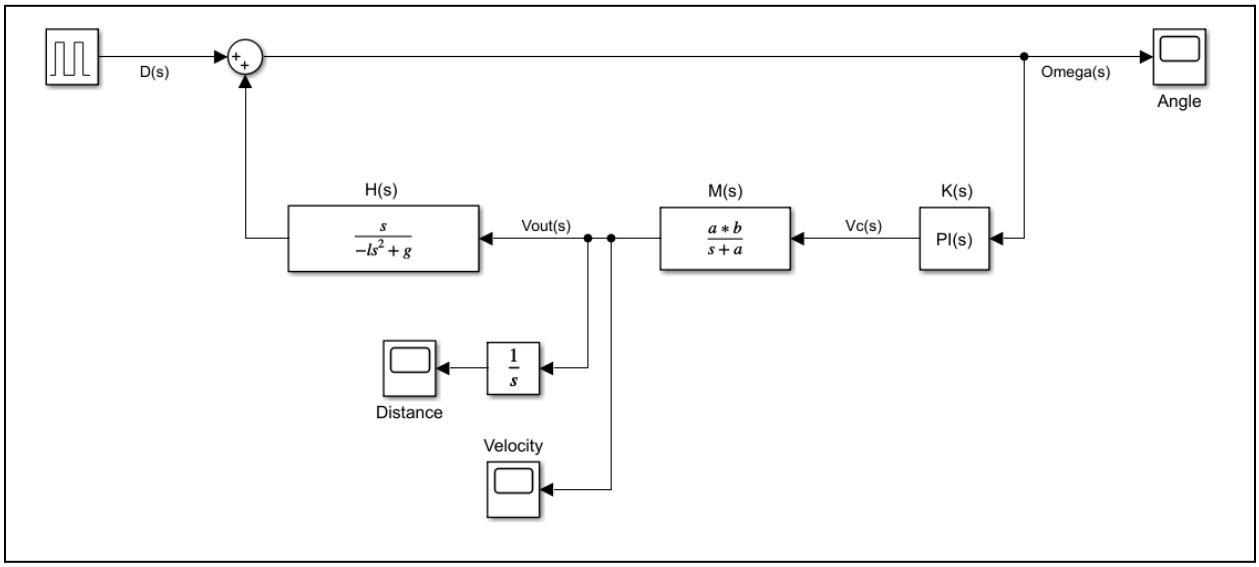

Figure 2-1. Simulink Model of 3 Pole System

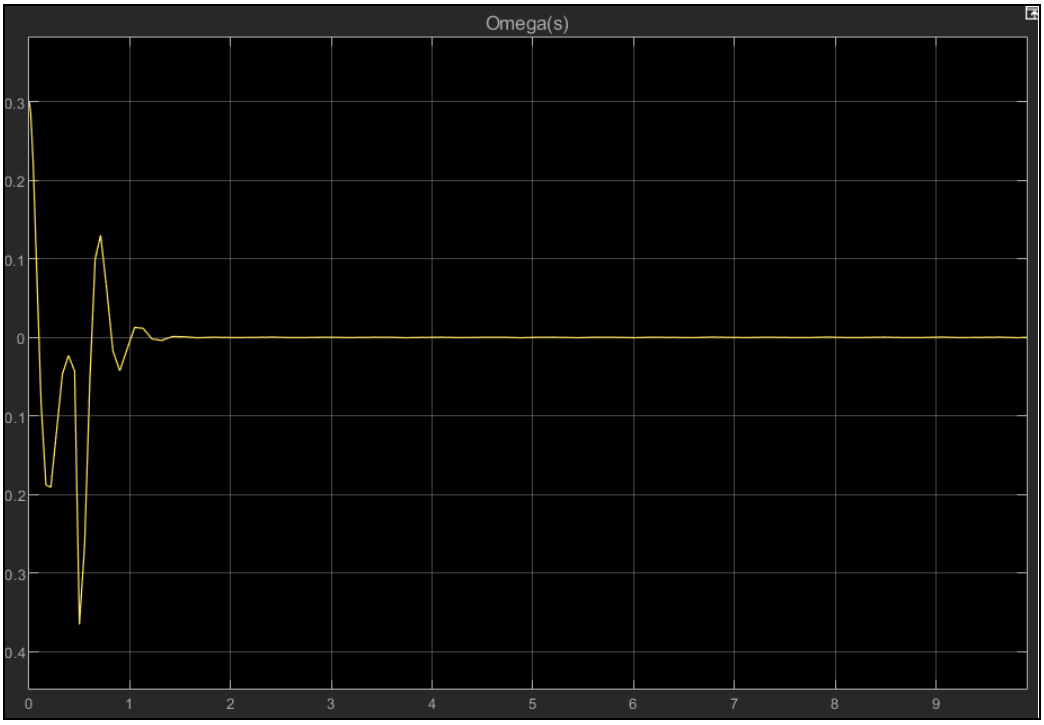

Figure 2-2. Angle Response to Pulse Input for Three Pole System

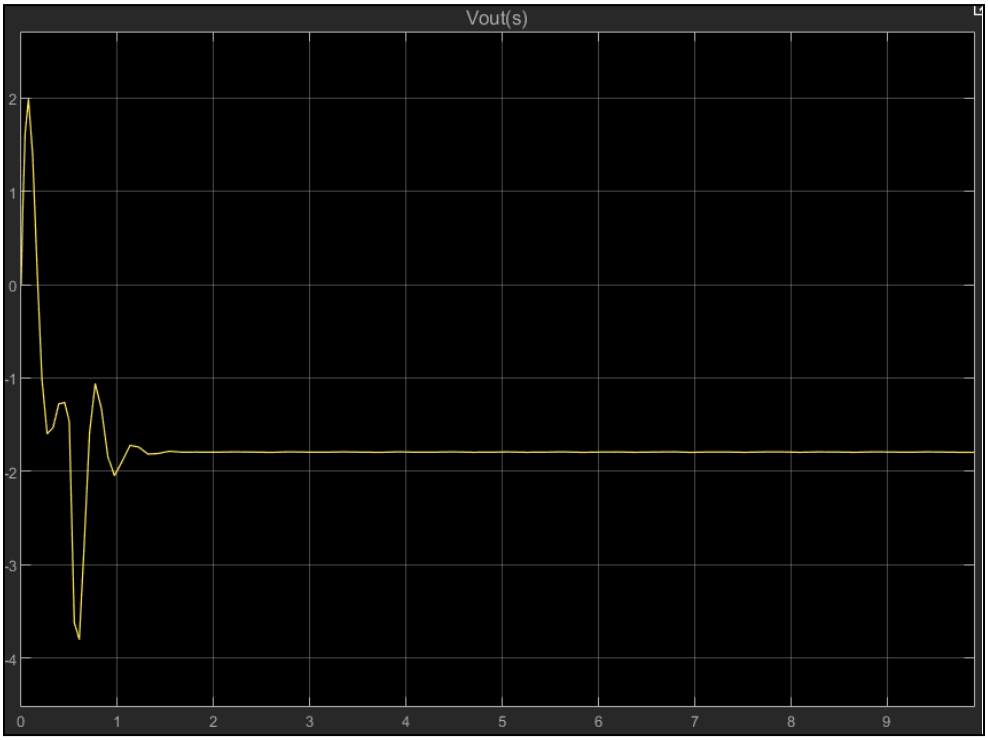

Figure 2-3. Velocity Response to Pulse Input for Three Pole System

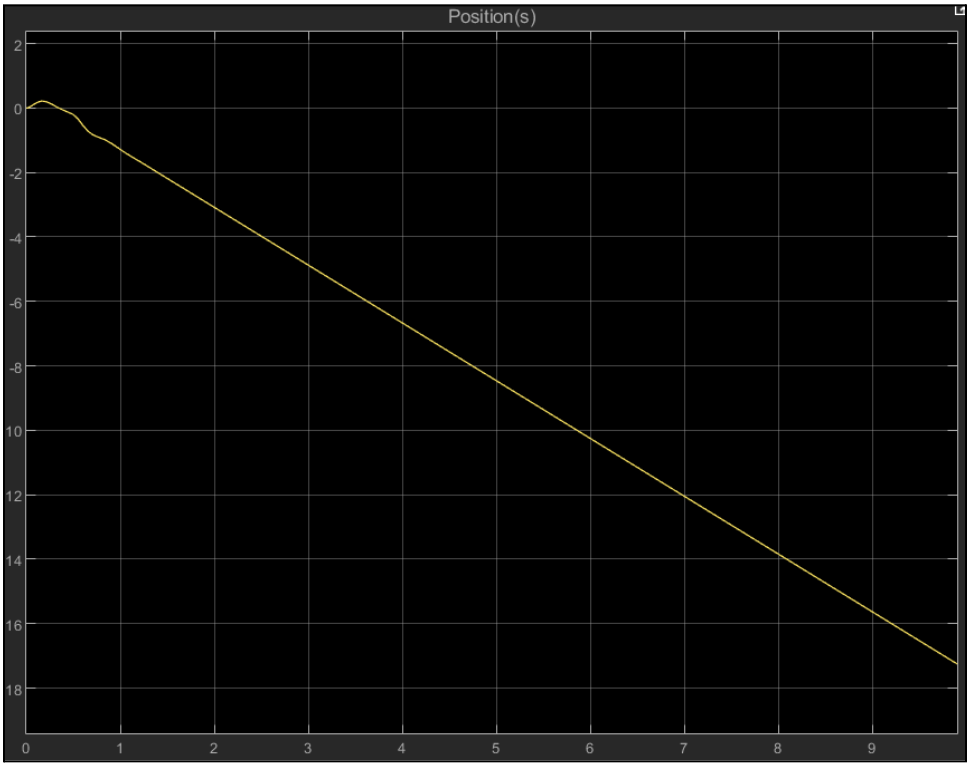

Figure 2-4. Position Response to Pulse Input for Three Pole System

While the angle results from the plots looked very promising, the angle is the only part that stabilizes. While Rocky will balance, it will not be stationary. In order to keep Rocky balanced and stationary, we need to add an additional feedback loop and controller to the system, stabilizing the position and linear velocity of Rocky.

Section 3: Stationary Balancing System

Seeing that the initial system is too oversimplified to apply to real world scenarios, we needed to add an extra controller. From **Figure 2-2** and **2-3**, we can see that while the angle response stabilizes around 0, the velocity and position responses fail to do the same. Therefore, we decided to add a controller around the motor transfer function, so that we can have finer control over the velocity of the system (and therefore the position). We chose to use a PI controller with new variables Ji and Jp to stabilize the velocity of the system, with $J(s) = J_p + \frac{J_i}{s}$. Additionally, we needed to stabilize the position, the integral of velocity, so we multiplied an integral transfer function $\frac{1}{s}$ by a secondary PI controller $C(s)$ such that

$C(s) = \frac{C_p}{s} + \frac{C_i}{s^2}$. When adding these two controllers together, we get a controller of $J_p + \frac{J_i + C_p}{s} + \frac{C_i}{s^2}$. Since Ji and Cp share a term of $\frac{1}{s}$, we can combine them into a single Ji.

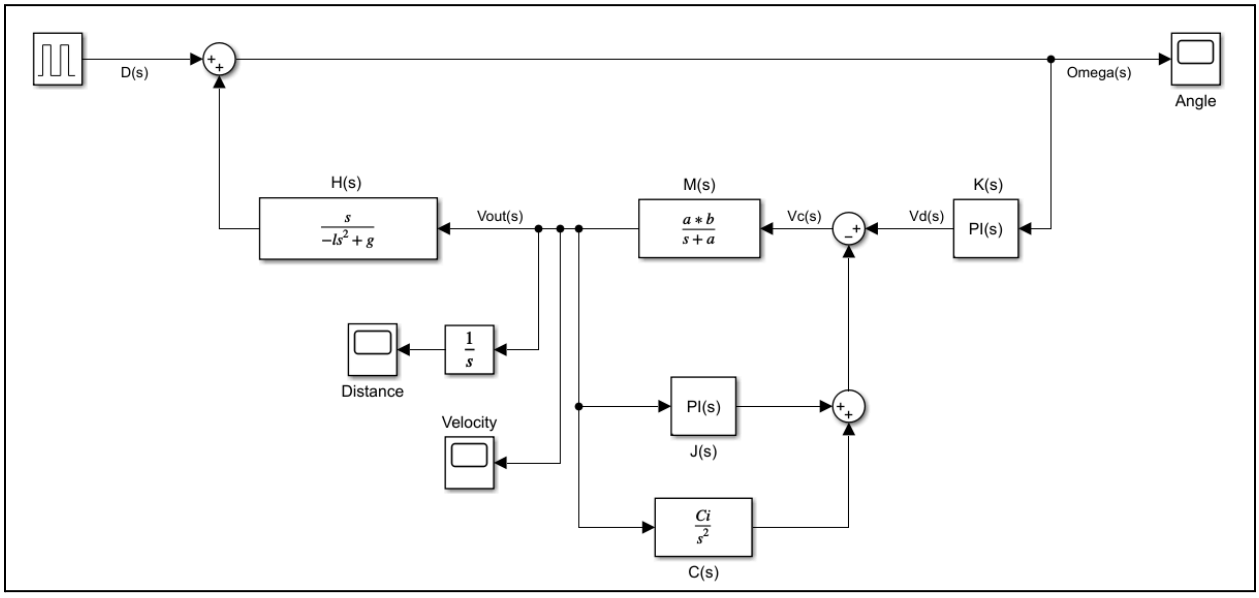

Figure 3-1. Simulink Model of Five Pole System

Now that we have our new model, as shown in **Figure 3-1**, we can place the poles of the system. Since the new controller block is a second-order system, we now have two more poles with a total of five poles. We decided to go with two pairs of complex conjugate poles as our dominant poles, one situated at -4 $\pm$ 2i, and another at -8 $\pm$ 2i. Lastly, we have our fifth pole at -10 so it decays faster than the dominant poles. We chose these poles partly with experience from the initial system, and partly with directly experimenting with the Rocky. After deciding the poles, we modified the MATLAB script provided to solve for the constants, which came out to Kp = 3639, Ki = 17724, Jp = 136, Ji = -2895, Ci = -3067.

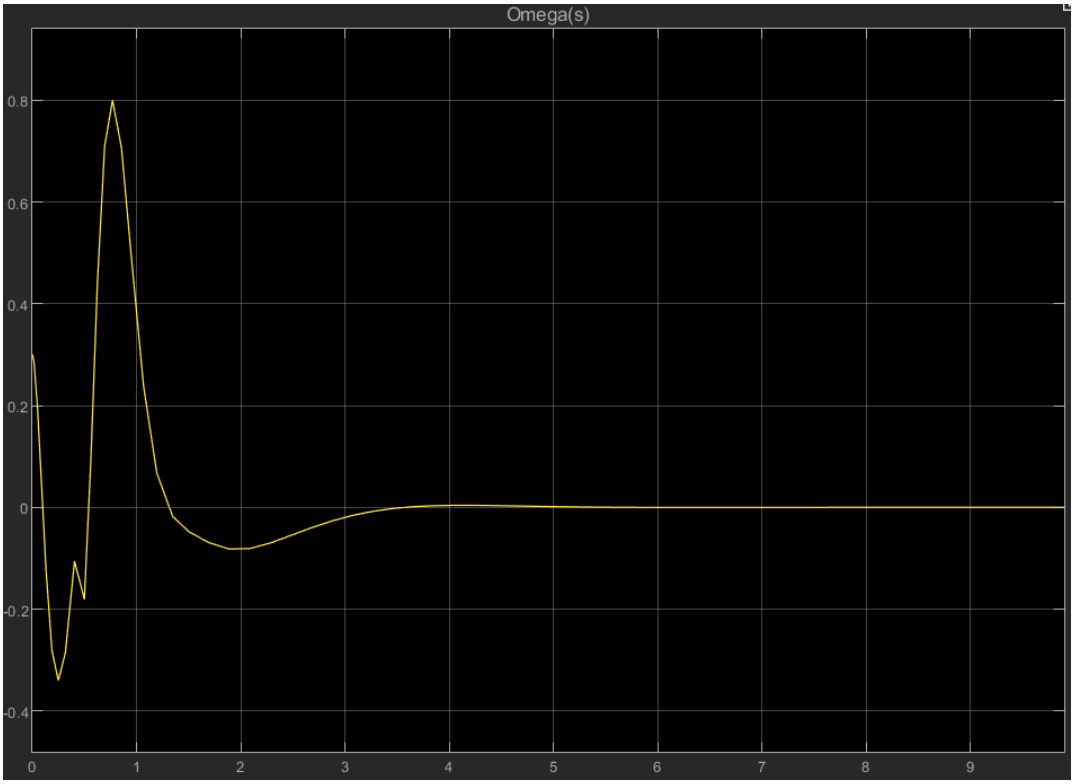

Figure 3-2. Angle Response to Pulse Input for Five Pole System

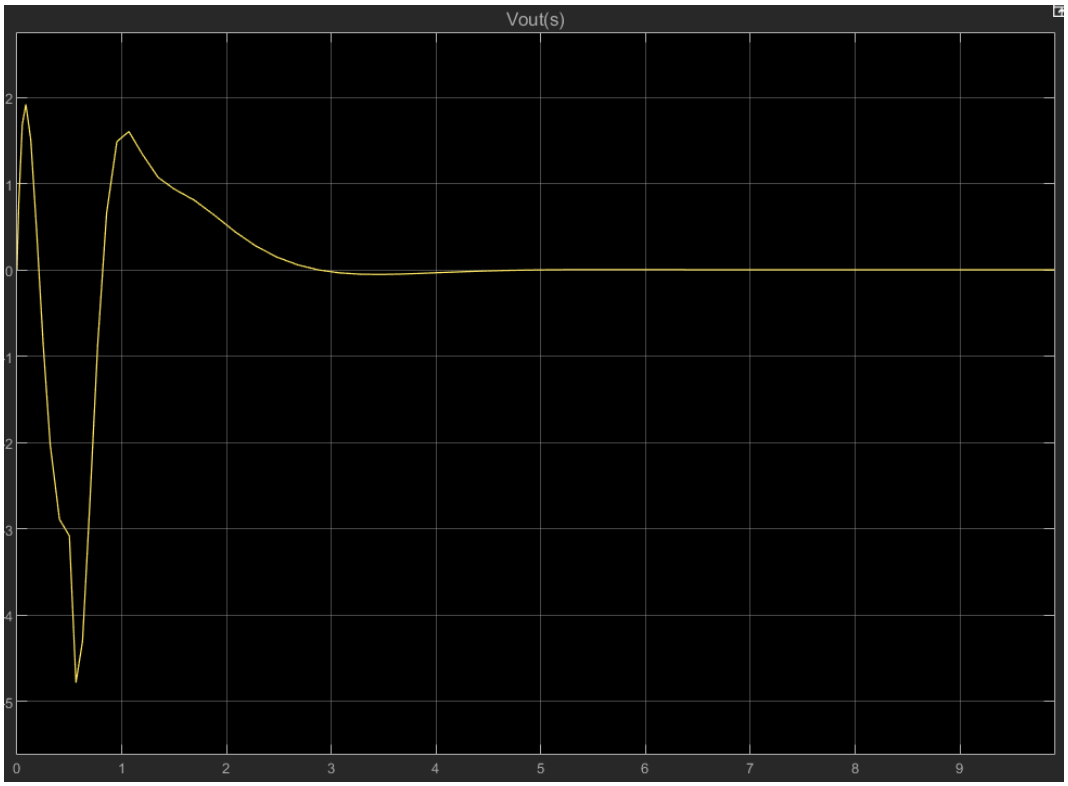

Figure 3-3. Velocity Response to Pulse Input for Five Pole System

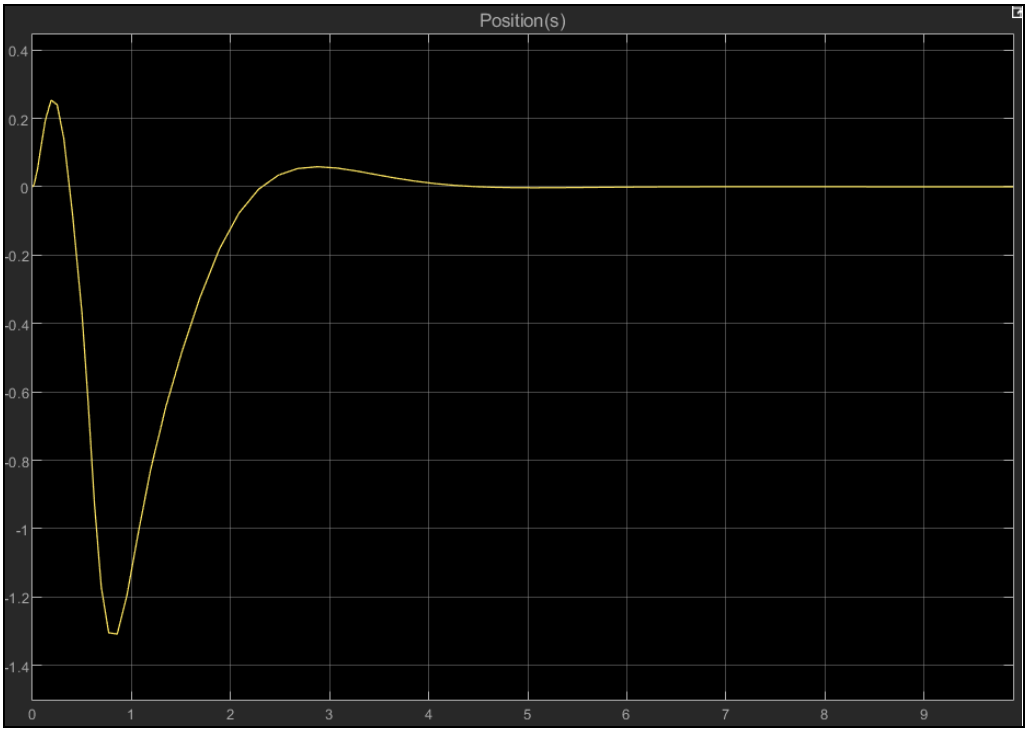

Figure 3-4. Position Response to Pulse Input for Five Pole System

As we can see from the figures, after we implemented the additional feedback loop, angle, velocity, and position response now center around zero after a reasonable reaction time.

Section 4: Implementation, Video and Reflection

The last step of this project is to apply our system to controlling the motor velocities. Using the starting script "Rocky_Balance_Starter_Code_25.ino", we needed to solve for the velocities of the left and right motors to complete balancing. Based off our simulink block diagram, we calculated $V_{cR} = V_d - J_p * speed_L - J_i * position_L - Ci * distance_{accumulated}$ and $V_{cL} = V_d - J_p * speed_R - J_i * position_R - Ci * distance_{accumulated}$, with $V_d = \theta * K_P + \theta_{accumulated} * K_i$.

We applied the updated system to the real life Rocky, and after some tweaking of parameters, we successfully got the Rocky balancing on the floor.

Here is the video showing our Rocky balancing. (Long link: https://olincollege-my.sharepoint.com/:v:/g/personal/xbao_olin_edu/Ed1444VxnWhEka-8b-BuDccB1A3FXZ_3Iah6ywrfzVkENg?e=nSINfU)

As we wrap up the project, we are all happy that as a team, none of the team members lagged significantly behind others. We had a clear division of labor from the start to the end of the project. This is a good end to the first half of ESA. All code for the project is included below.

```
// 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 (0.24)  // ***** 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 = 4154;
  float Ki = 20130;
  float Ci = -3067;
  float Jp = 142;
  float Ji = -2895;


    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 ****
    v_d = angle_rad * Kp + angle_rad_accum * Ki; // 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 ****
    v_c_R = v_d - Jp*measured_speedL - Ji*distLeft_m - Ci*dist_accum;
    v_c_L = v_d - Jp*measured_speedR - Ji*distRight_m - Ci*dist_accum;
```

```
    // 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;
  }


}
```