Rocky Project:

The Stabilization of a Wheeled Inverted Pendulum

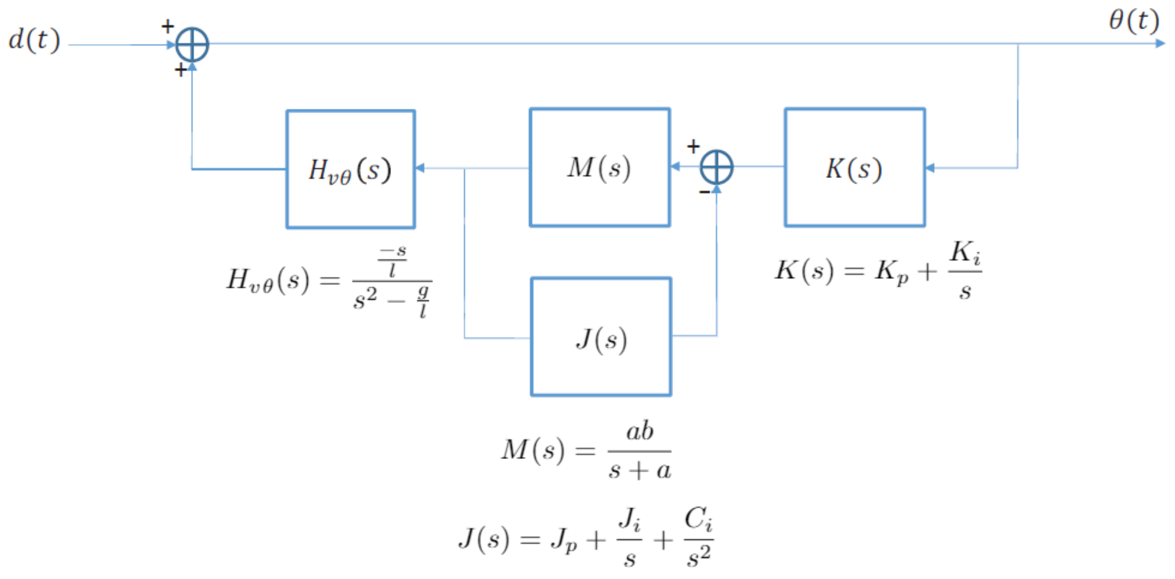
Isabella Abilheira, Chris Allum, and Maya Sivanandan

March 22, 2021

1 Introduction

For this project, we designed and implemented a control system that maintains the balance of a wheeled inverted pendulum and corrects for angular disturbance.

2 Block Diagram


$d(t)$
$\theta(t)$
+
+
$H_{v\theta}(s)$
$M(s)$
$K(s)$
$J(s)$
+
-
$H_{v\theta}(s) = \frac{\frac{-s}{l}}{s^2 - \frac{g}{l}}$
$K(s) = K_p + \frac{K_i}{s}$
$M(s) = \frac{ab}{s+a}$
$J(s) = J_p + \frac{J_i}{s} + \frac{C_i}{s^2}$

3 Control System

An inverted pendulum is an unstable system. In order to balance Rocky, we needed to create a feedback system that corrects for disturbances in angle through horizontal motion of its base. Such a system would need to have a transfer function with negative poles so that it is stable.

The entire system transfer function is represented by the block diagram above (see Section 2). The feedback section of the block diagram consists of three independent transfer functions: the PI controller transfer function, the motor controller transfer function, and the transfer function for Rocky itself. The transfer function of the PI controller is the following:

$$K(s) = K_p + \frac{K_i}{s} \tag{1}$$

The transfer function relating the motor control signal of the Pololu Balboa board, and the actual velocity of the robot is modeled by the equation

$$M(s) = \frac{ab}{s+a} \tag{2}$$

where $a = \frac{1}{\tau}$, where τ is the time constant, and $b = K$, where K is the steady state value. Lastly, the transfer function of Rocky relating its velocity to the angle of the pendulum is the following:

$$H_v(s) = \frac{\frac{-s}{l}}{s^2 - \frac{g}{l}} \tag{3}$$

If one were to combine all of the transfer functions listed above, the rocky would, in theory, balance; however, it might drift significantly from its starting location. In other words, Rocky would have a steady state angle and velocity, but it would not have a steady state x-position. To resolve this issue, we modified the motor transfer function with a feedback loop with the following transfer function:

$$J(s) = J_p + \frac{J_i}{s} + \frac{C_i}{s^2} \tag{4}$$

Using Black's formula, the overarching motor transfer function for the motors becomes the following:

$$M_c(s) = \frac{M(s)}{1 + M(s)J(s)} \tag{5}$$

In order to find the transfer function of the whole closed-loop system, we again used Black's formula. As shown by the block diagram above, the feedback signal is added to the summing junction. This changes our general form equation to

$$\frac{1}{1 - G(s)} \tag{6}$$

where G(s) is the feedback signal. As a result, our transfer function for the entire system system is the following:

$$H_{Total}(s) = \frac{1}{1 - H_v(s)M_c(s)K(s)} \tag{7}$$

By combining these functions in a feedback loop, we ensure that the angle, horizontal velocity, and position of Rocky all approach zero in response to a disturbance in angle.

4 Parameter Identification

To complete our system wide transfer function, we needed to first identify the following characteristics of Rocky: the pendulum's natural frequency, its effective length, and the motor constants. Each Rocky system is built slightly differently, so each of these parameters differ from system to system.

4.1 Natural Frequency

In order to find the natural frequency of our robot, we held Rocky as a pendulum (wheels up) and measured the change in angle over time as Rocky oscillated. The data we collected is shown in Figure 1 below.

We then used a Fourier transformation of the calibration data to find the natural frequency. As seen in Figure 2, Rocky's natural frequency is roughly 0.72 Hz or 4.52 rad/sec.

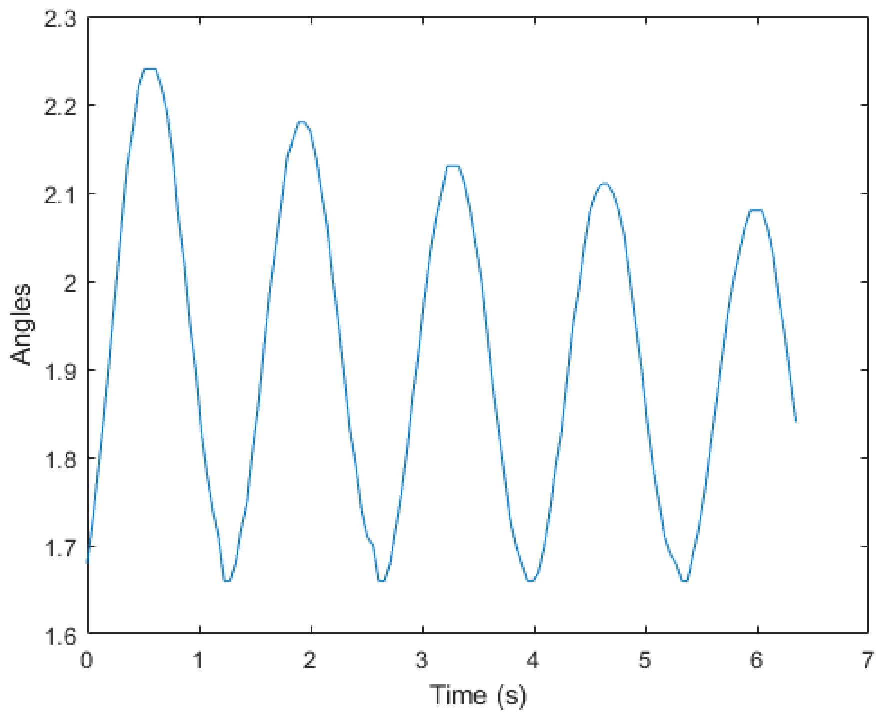


Figure 1: Gyroscopic data of Rocky's angle over time

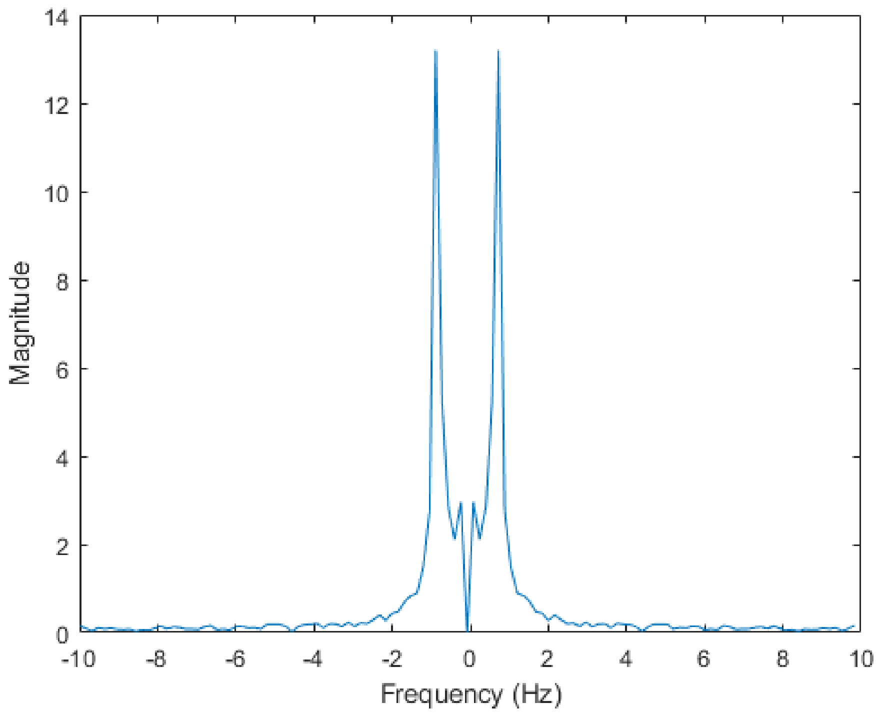


Figure 2: Fourier Transformation of gyroscopic data

4.2 Effective Length

Next, we used the natural frequency to find the effective length of Rocky. Natural frequency, ω_n, is defined by the following equation

$$\omega_n = \sqrt{\frac{g}{l_{eff}}} \tag{8}$$

where $g = 9.8m/s^2$, the acceleration due to gravity and l_{eff} is the effective length of the pendulum. We can then use our two known values, g and ω_n, to solve for an effective length of 18.85 in.

4.3 Motor Constants

In order to determine the motor constants, we collected calibration data from Rocky by running the wheels at maximum speed for 3 seconds. As mentioned above, the transfer function relating the motor control signal of the Pololu Balboa board, and the actual velocity of the robot is modeled by the equation

$$M(s) = \frac{abK_{i_n}}{s + a} \tag{9}$$

where $a = \tau^-1$, where τ is the time constant, and $b = K$, where K the steady state value. An inverse Laplace transformation shows that the analytical solution to the transfer function with a motor control signal of 300 is

$$M(t) = 300K - 300Ke^{\frac{-t}{\tau}} \tag{10}$$

With that, we fitted our data from each wheel to this equation to get values for K and τ which we then averaged. Knowing K and τ, we were able to calculate a and b to be 16.6711 and $\frac{1}{299.9}$, respectively. The comparison of our experimental data and analytical model is shown below in Figure 3.

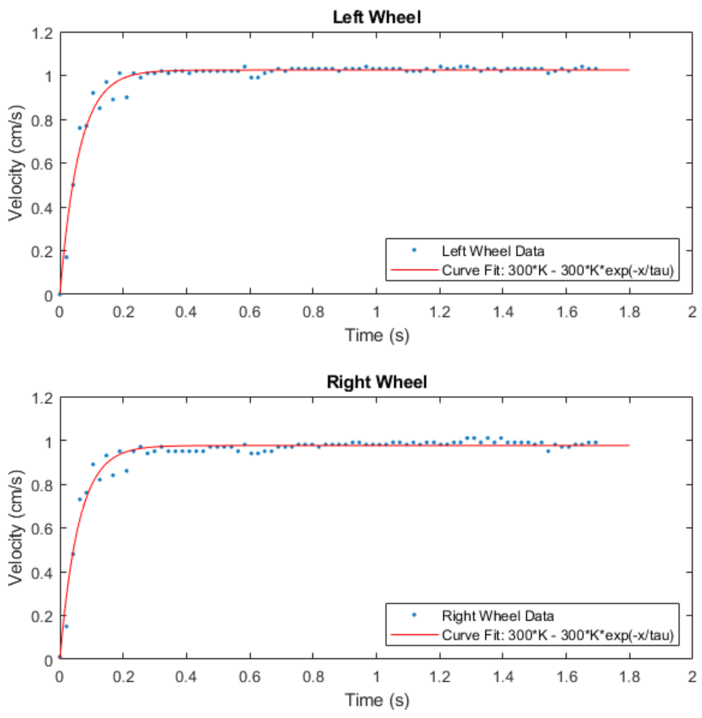


Figure 3: Rocky wheel velocity vs. time

5 Poles

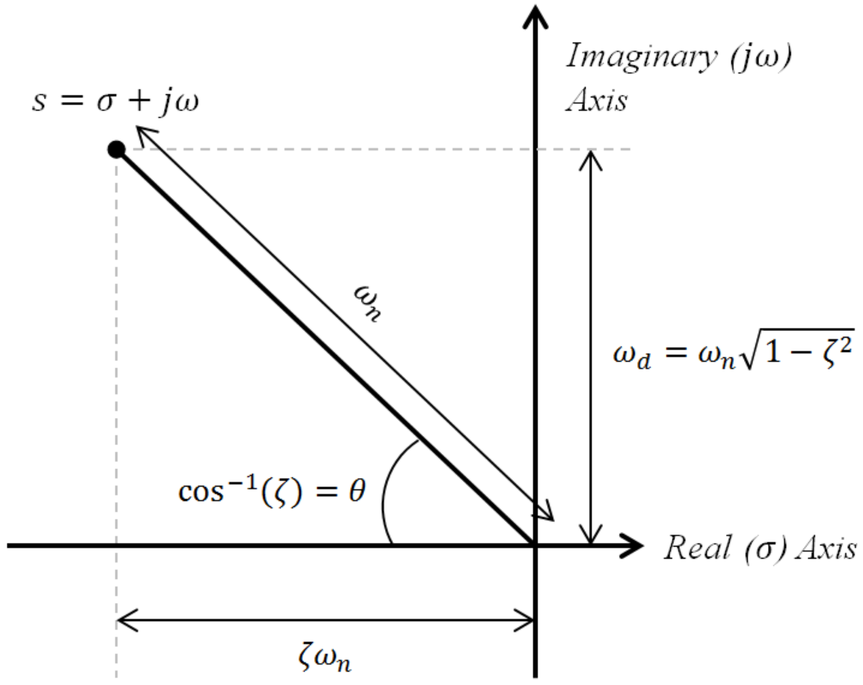

$s = \sigma + j\omega$
Imaginary ($j\omega$) Axis
ω_n
$\omega_d = \omega_n\sqrt{1-\zeta^2}$
$\cos^{-1}(\zeta) = \theta$
Real (σ) Axis
$\zeta\omega_n$

Figure 4: Pole location in the complex plane

The last step in creating the system wide transfer function is deciding where to place the poles. For a system to be stable, all of its poles need to be negative. Therefore, we made sure that all of Rocky's five poles were first and foremost negative. Secondly, we knew that the magnitude of the pole is equivalent to the natural frequency of the system (see Figure 4). Given that we knew Rocky's natural frequency, and therefore the magnitude of its poles, we just needed to define the angle of each pole in the imaginary axis with respect to the real axis. As seen in Figure 4, cos(angle) is equal to zeta, the damping ratio of the system. We knew that the system was slightly under-dampened, and therefore, some of the poles had to be complex. Furthermore, the complex poles had to be in conjugate pairs. Therefore, we decided on two conjugate pairs and one real pole. After a bit of trial and error, we decided on the following poles

1. $-\omega_n(\cos(25°) + i\sin(25°))$
2. $-\omega_n(\cos(25°) - i\sin(25°))$
3. $-\omega_n(\cos(35°) + i\sin(35°))$
4. $-\omega_n(\cos(35°) - i\sin(35°))$
5. $-\omega_n$

We decided on these poles because it resulted in a system that was stable and just slightly under-dampened.

Knowing the effective length of the pendulum, the motor constants, and the poles of the system, we were then able to calculate the entire system wide transfer function. The transfer function is

$$H_{Total}(s) = \frac{1.23\times10^{63}s^5 + 2.48\times10^{64}s^4 - 7.29\times10^{64}s^3 - 5.33\times10^{65}s^2 + 9.76\times10^{65}s + 5.13\times10^{65}}{1.23\times10^{63}s^5 + 2.48\times10^{64}s^4 + 1.87\times10^{6}s^3 + 6.42\times10^{65}s^2 + 9.76\times10^{65}s + 5.13\times10^{65}} \tag{11}$$

Furthermore, as seen in Figure 5 below, the transfer function is indeed stable.

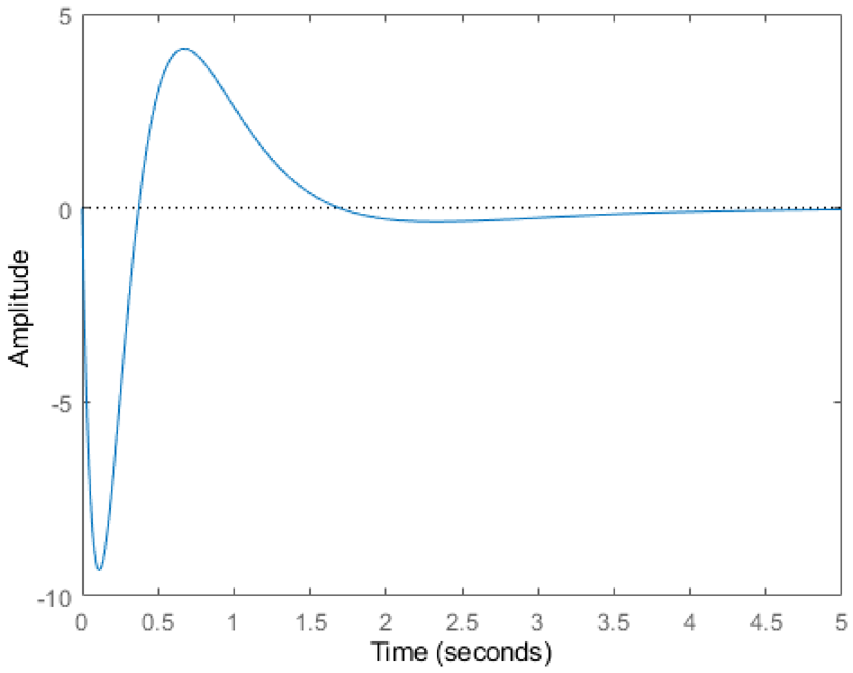


Figure 5: Impulse response of transfer function

6 MATLAB and Arduino Code

See attached pages.

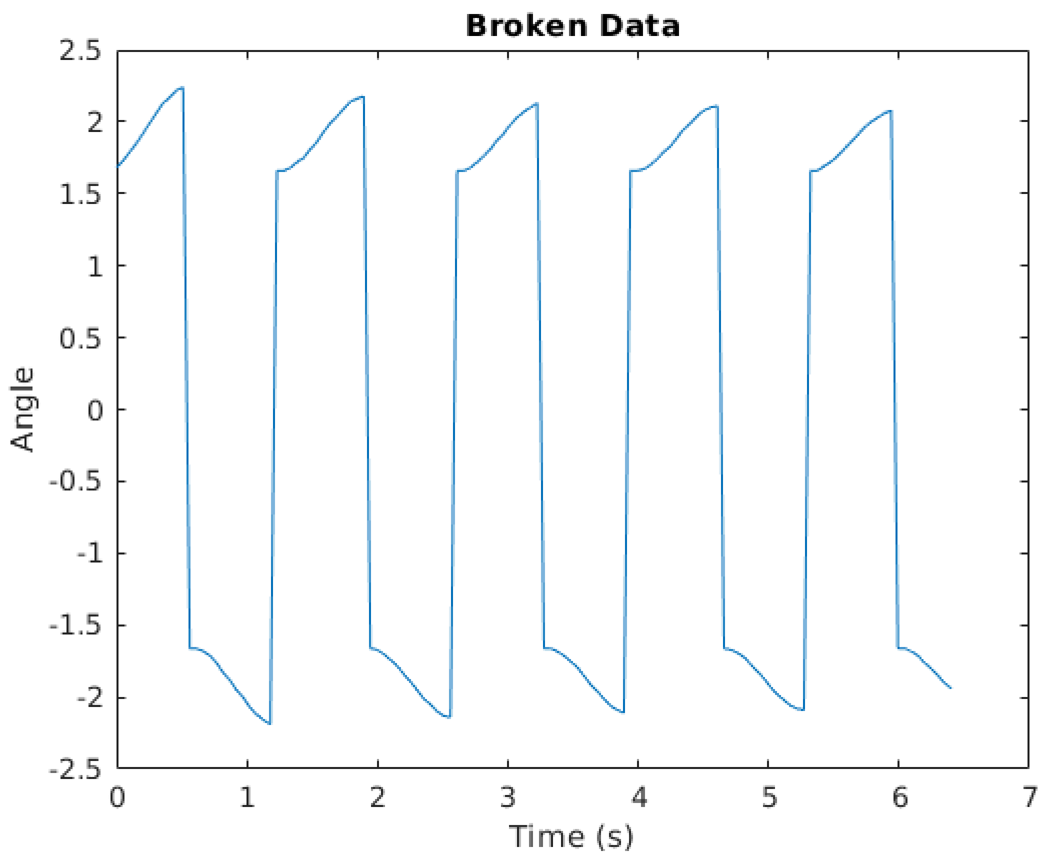
ESA Rocky Project

Authors: Chris A, Maya S, Izzie A

```
clc
clearvars
```

Load and graph gyro calibration data

```
load data.mat

figure
plot(time,angle);
title("Broken Data")
xlabel("Time (s)")
ylabel("Angle")
```

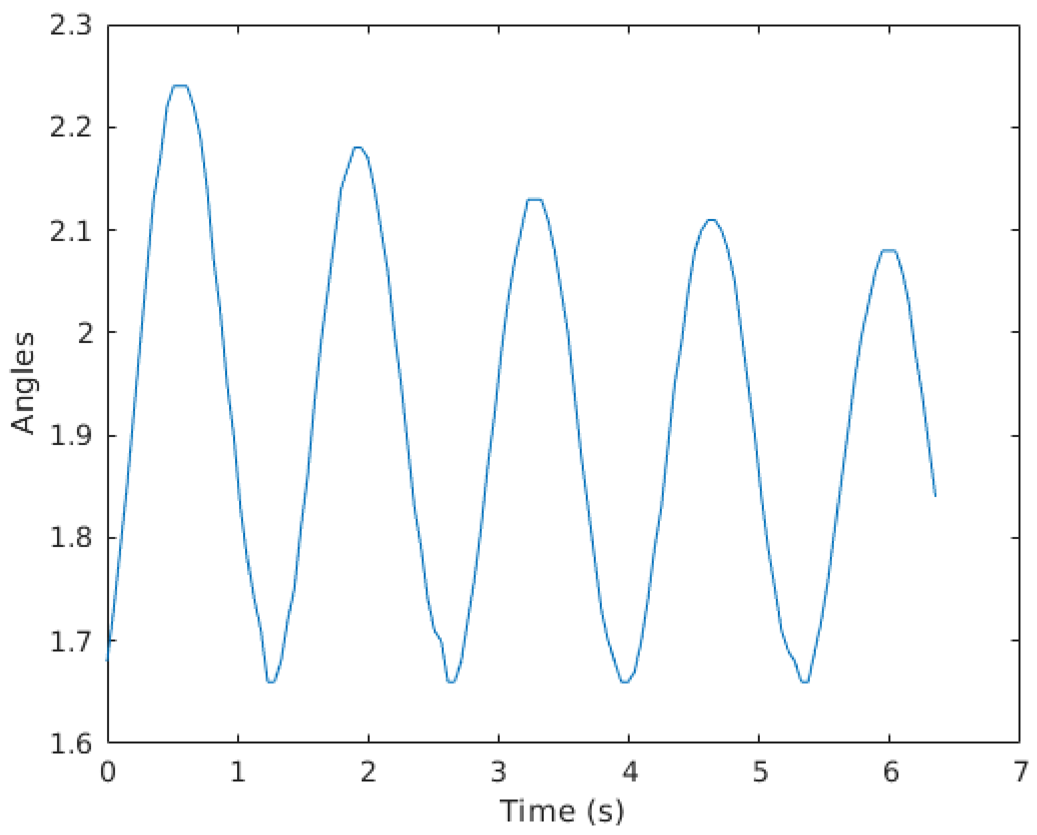


Clearly, something is wrong with the data. In the next section, we edit the data so that it forms a sinusoidal curve.

```
% create new array, each row alternates between positive data points and
% negative data points
new_data = [0];
row = 1;
col = 1;
for q = 1:length(angle)-1
    new_data(row,col) = angle(q);
    col = col + 1;
    if angle(q)*angle(q+1) < 0
        row = row + 1;
        col = 1;
    end
end

% isolate negative data points and convert zeros to NaN
negative_data = new_data(2:2:end,:);
negative_data(negative_data == 0) = NaN;

% find offset between positive and negative data
offset = max(new_data(1:2:end,:),[],2)-max(negative_data,[],2);
% shift negative data up match up with positive data
new_data(2:2:end,:) = new_data(2:2:end,:) + offset;

% reshape data back into a column vector
new_data = reshape(new_data',[],1);
% delete outlier values
new_data = new_data(1<abs(new_data) & abs(new_data)<3);

% trim time vector to match new angle data
time = time(1:length(new_data));
angle = new_data;
```

Now we graph the new data and save the modified data

```
figure
plot(time,angle)
%title("Fixed Data")
xlabel("Time (s)")
ylabel("Angles")
```

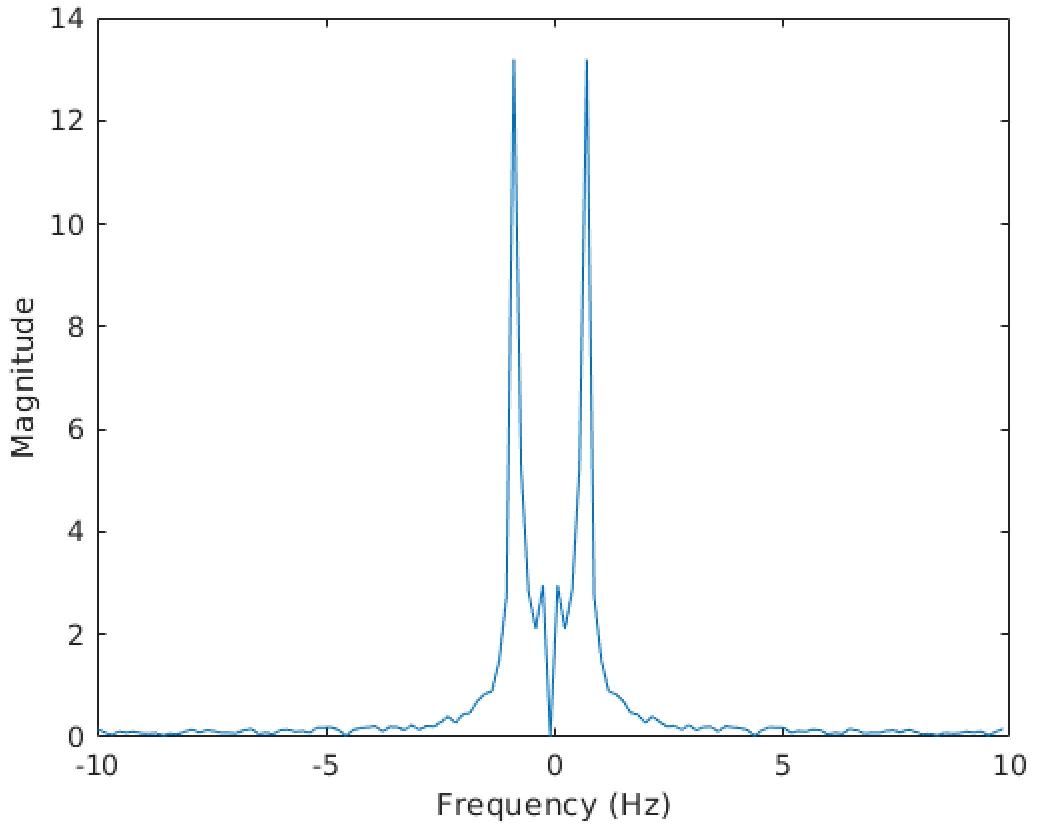


```
save modified_data time angle
```

Now we load the modified data and calculate the effective length of the Rocky and its natural frequency

```
clc
clearvars

load modified_data.mat

TS = 0.05; % time step

y = fft(angle);

[~, sort_index] = sort(abs(y), "descend");
y(sort_index(1:1)) = 0;

fs = 1/TS;
f = (0:length(y)-1)*fs/length(y);

n = length(angle);
fshift = (-n/2:n/2-1)*(fs/n);
yshift = fftshift(y);

figure
plot(fshift,abs(yshift))
```

```
xlabel('Frequency (Hz)')
ylabel('Magnitude')
```



```
%title("FFT of Rocky swinging freely by top")

[~, sort_index] = sort(abs(yshift), "descend");
wn = fshift(sort_index(2))
```

```
wn = 0.7200
```

```
wn = wn*2*pi;
gravity = 9.8;
l_eff = gravity/wn^2    % units meter
```

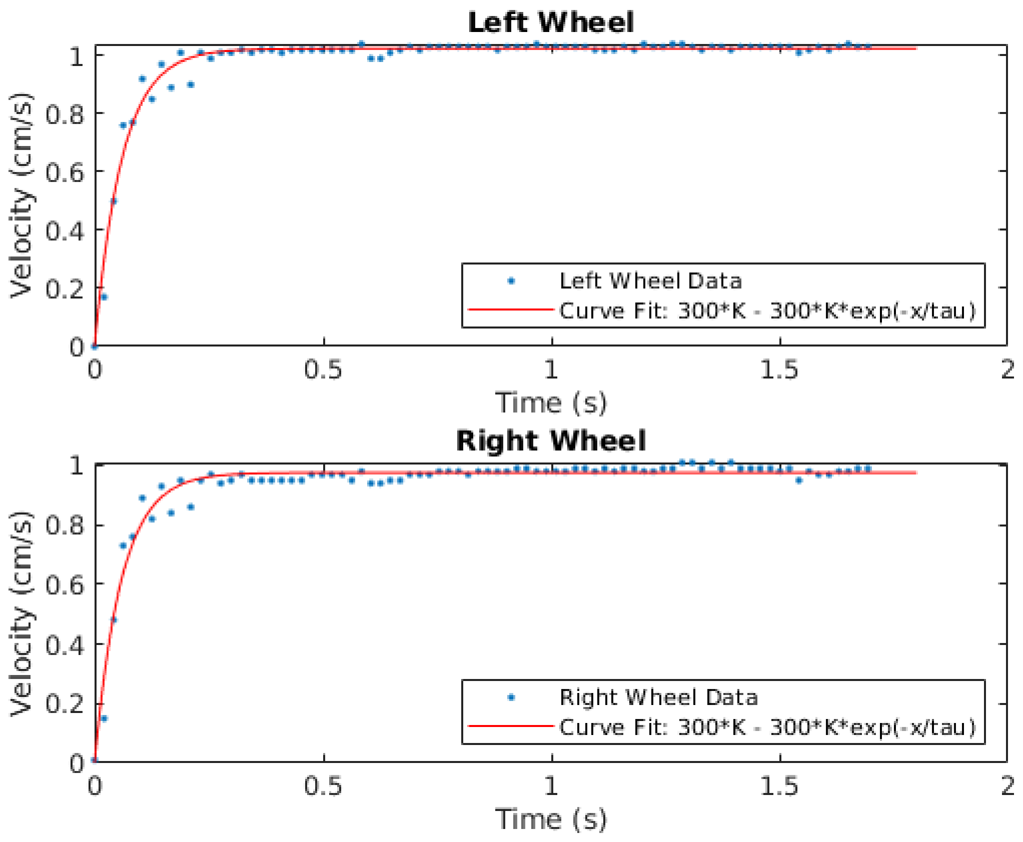
```
l_eff = 0.4789
```

```
l_eff_in = l_eff*39.3701;   % units inch
```

Now we will load in wheel velocity data and calculate the motor constants and transfer function

```
clc
clearvars -except wn l_eff
load wheelVel.mat;

% create symbolic vars
syms a b s tau K x K_in
% define TF of motor
```

```
eqn = (K_in*a*b)/(s*(s+a));
% find analytic solution to TF
soln = ilaplace(eqn);
a = 1/tau;
b = K;
t = x;
% define motor input
K_in = 300;
soln = subs(soln);
% convert solution into fittype
exp_eqn = fittype(char(soln));
% curve fit data to analytical solution
fitLeft = fit(time,leftV,exp_eqn,'Start',[1,1]);
fitRight = fit(time,rightV,exp_eqn,'Start',[1,1]);

% calculate A and B from the tau and K values
l_A = 1/fitLeft.tau;
l_B = fitLeft.K;
r_A = 1/fitRight.tau;
r_B = fitRight.K;

% plot left motor data and curve fit
subplot(2,1,1)
plot(time, leftV,'.')
hold on
plot(fitLeft)
xlabel('Time (s)')
ylabel('Velocity (cm/s)')
title('Left Wheel')
legend('Left Wheel Data', sprintf('Curve Fit: %s', char(soln)),'Location','southeast')
hold off

% plot right motor data and curve fit
subplot(2,1,2)
plot(time, rightV,'.')
hold on
plot(fitRight)
xlabel('Time (s)')
ylabel('Velocity (cm/s)')
title('Right Wheel')
legend('Right Wheel Data',sprintf('Curve Fit: %s', char(soln)),'Location','southeast')
hold off
```



```
% calculate average A and B terms for motors
motor_a = (l_A+r_A)/2
```

```
motor_a = 16.6711
```

```
motor_b = 2/(l_B+r_B)   % b is actuall 1/motor_b
```

```
motor_b = 299.9725
```

Now we will use the natural frequency, effective length, motor A and B values to calculate the transfer function for Rocky

```
clearvars -except wn l_eff motor_a motor_b
clc

% defind syms
% values
syms s a b l g Kp Ki Jp Ji Ci

% TF from velocity to angle of inverted pendlum
Hvtheta = -(s/l)/(s^2-(g/l));
% TF of angle controller
K = Kp + Ki/s;
% TF of the controller around the motor
J = Jp + Ji/s + Ci/(s^2);
% TF of motor
```

```
M = (a*b)/(s+a);
% TF of motor and feedback controller around it
Md = M/(1+M*J);

% Total TF function
Htot = 1/(1-Hvtheta*Md*K);

% system parameters
g = 9.81;
l = l_eff;
a = motor_a;
b = 1/motor_b;

% sub in parameter values
Htot_subbed = simplify(subs(Htot));
```

In this section, we define the poles of the system. From these poles, we calculate the system coefficients

```
% define target poles
angle1 = 25;
angle2 = 35;
p1 = wn*(-cosd(angle1) + sind(angle1));
p2 = wn*(-cosd(angle1) - sind(angle1));
p3 = wn*(-cosd(angle2) + sind(angle2));
p4 = wn*(-cosd(angle2) - sind(angle2));
p5 = -wn;

% define target characterisitic polynomial
char_poly = (s-p1)*(s-p2)*(s-p3)*(s-p4)*(s-p5);

% find coeffs of char polynomial denom
coeffs_char = coeffs(char_poly,s);

% get denominator of Htot_subbed
[~, denom] = numden(Htot_subbed);

% find coefficients of the denom
coeffs_denom = coeffs(denom,s);

% divide out coeff of highest power term
coeffs_denom = coeffs_denom/coeffs_denom(end);

% solve the system of equations setting the coefficients of the polunomial
% in the target to the actual polynomial
solutions = solve(coeffs_denom == coeffs_char,[Kp,Ki,Jp,Ji,Ci]);

% display the solutions as doubles
Kp = double(solutions.Kp)
```

```
Kp = 1.8158e+03
```

```
Ki = double(solutions.Ki)
```

```
Ki = 8.2186e+03
```

```
Jp = double(solutions.Jp)
```

```
Jp = 62.3368
```

```
Ji = double(solutions.Ji)
```

```
Ji = -696.3410
```

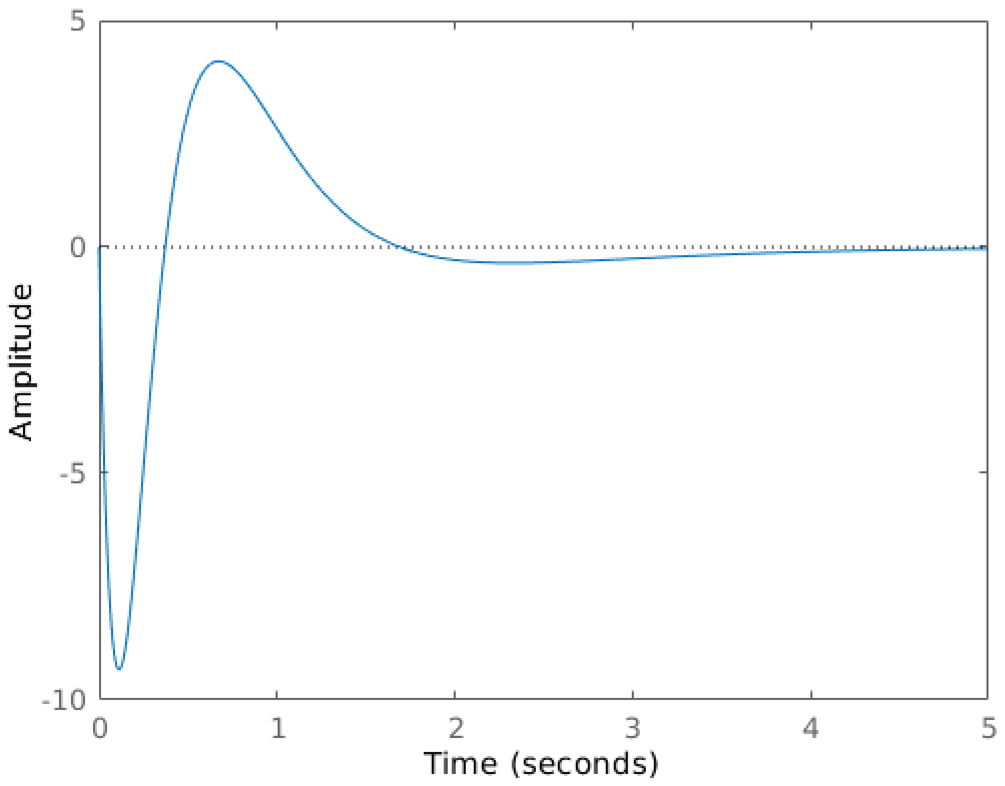
```
Ci = double(solutions.Ci)
```

```
Ci = -365.8732
```

```

%[Kp Ki Ci Jp Ji]
```

Lastly, we will graph the entier system transfer function

```
eqn = collect(simplify(subs(Htot_subbed)));

ExpFun = matlabFunction(eqn);
ExpFun = str2func(regexprep(func2str(ExpFun), '\.([/^\\*])', '$1'));
TF = ExpFun(tf('s'));

figure
impulse(TF)
title('')
```



```
// 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 (0.28)  // 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

// 1.8158    8.2186   -0.3659    0.0623   -0.6963



    float Kp = 1815.8;
    float Ki = 8218.6;

    float Ci = -365.9;

    float Jp = 62.3;
    float Ji = -696.3;


    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


    v_d = Kp*angle_rad + Ki*angle_rad_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

    v_c_R = v_d - Jp*measured_speedR - Ji*distLeft_m  - dist_accum*Ci;
    v_c_L = v_d - Jp*measured_speedL - Ji*distRight_m - dist_accum*Ci;

    // 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 < -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;
  }


}
```