# Inverted Pendulum Project Report

Olivia Jo Bradley and Nabih Estefan
Engineering System Analysis
Olin College of Engineering
3/20/2021

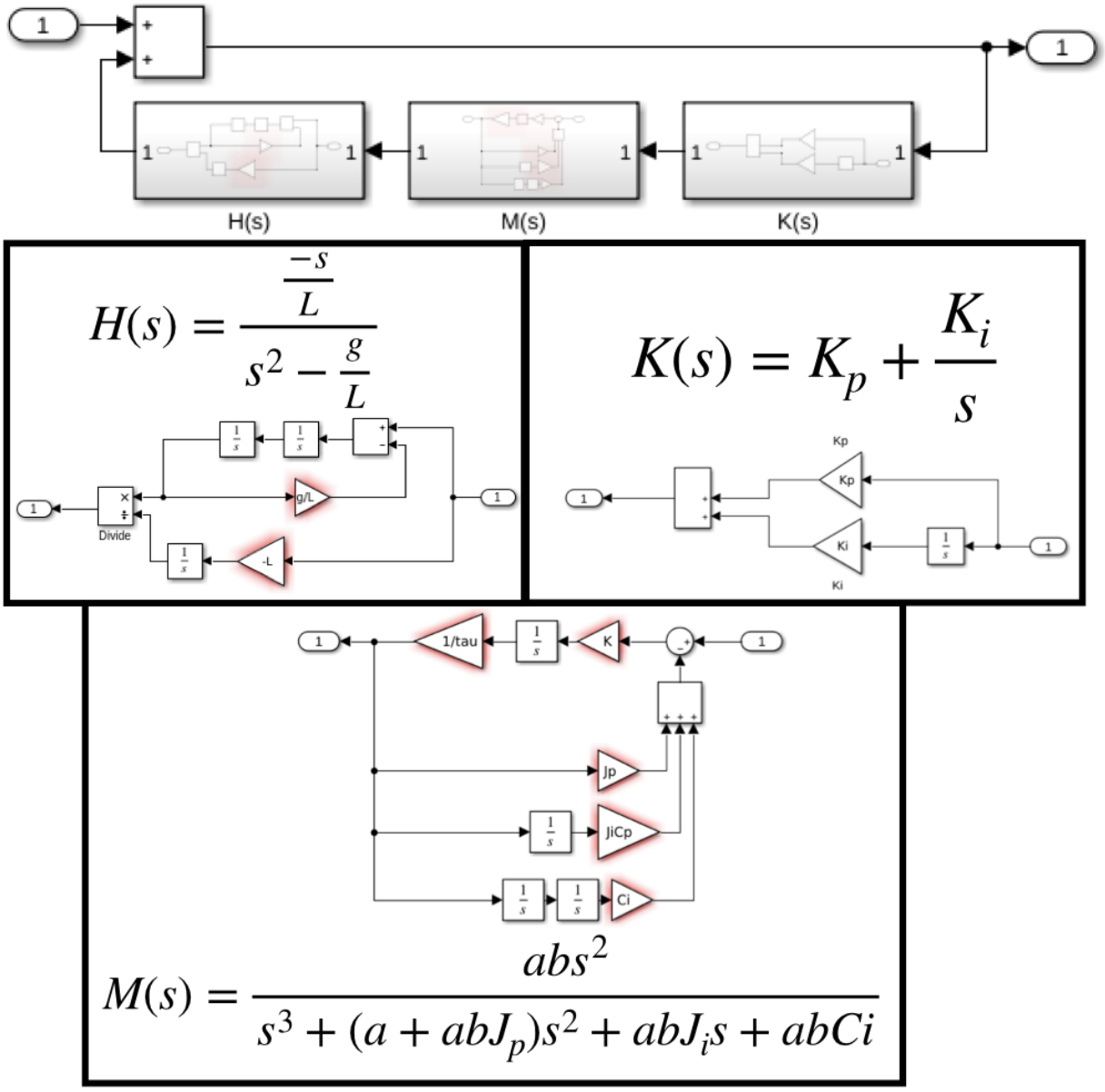
## Introduction

For this project, we used a 32U4 Balancing Robot Kit from Pololu Robotics and Electronics, also known as Rocky, to mimic a balancing inverted pendulum (seen above). Below, we will explain how we created our control system to make our Rocky balance in a single location.

## Base model & Block Diagram

Below you can see the base model, and the zoom into the specific parts we abstracted. As you can see there are 3distinct parts of the model:

- H(s) which is the main transfer function between angle and velocity.
- K(s) which is the main steady-state error function.
- M(s) which is the Motor transfer function with a feedback loop to change expected velocity to actual velocity.




$$H(s) = \frac{\frac{-s}{L}}{s^2 - \frac{g}{L}}$$

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

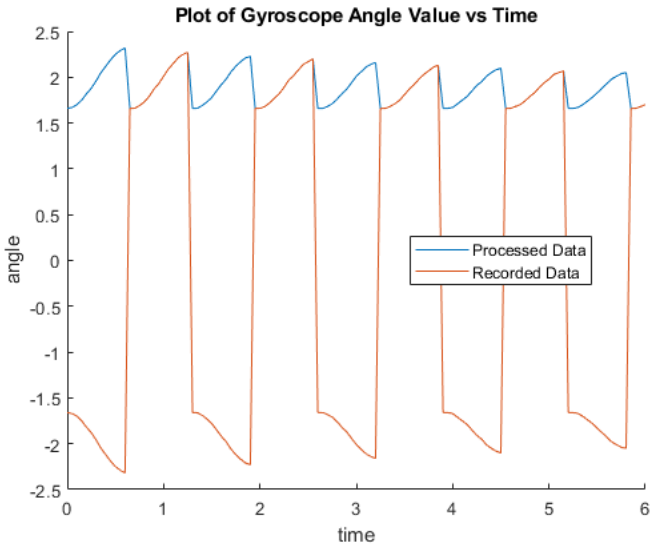
$$M(s) = \frac{abs^2}{s^3 + (a + abJ_p)s^2 + abJ_i s + abCi}$$

As you can see above, there are multiple variables which are: g, L, tau, K, Kp, Ki, Jp, Ji, and Ci. g is the value of gravity, which is 9.81 m/s. The other values we find throughout the process of our calculations which you will see below. L is the effective length, and the process to get it can be seen in the “Natural Frequency and Effective Length” section. K and tau are Motor Constants, and you can see the process to get them in the “Motor Constants” section. Finally, Kp, Ki, Jp, Ji, and Ci are the steady-state variables which are calculated in the “Controlled Parameters” section.

Finally, each of the main sections (H(s), K(s), and M(s)) has a transfer function equation defined for each. These can be seen in the image above in each of the breakdown for each subsystem.

## Natural Frequency and Effective Length

To find effective length, we first had to collect frequency data using the provided Rocky_Gyro_Calibration code. We carried our Rocky by the axles, and recorded the gyroscope data as it rocked back and forth.
To make the data look more like a sine curve, we absolute valued the data. Below is a graph of the data before and after the absolute value.




From this data, we are able to find the natural frequency using Fast Fourier Transforms (FFT), which plots the data we collected in the frequency domain.

With the equation $\omega_n = \sqrt{\frac{g}{l_{eff}}}$ we are able to find the effective length of our inverted pendulum, which we then subtracted from the total length, to get about 17 inches.

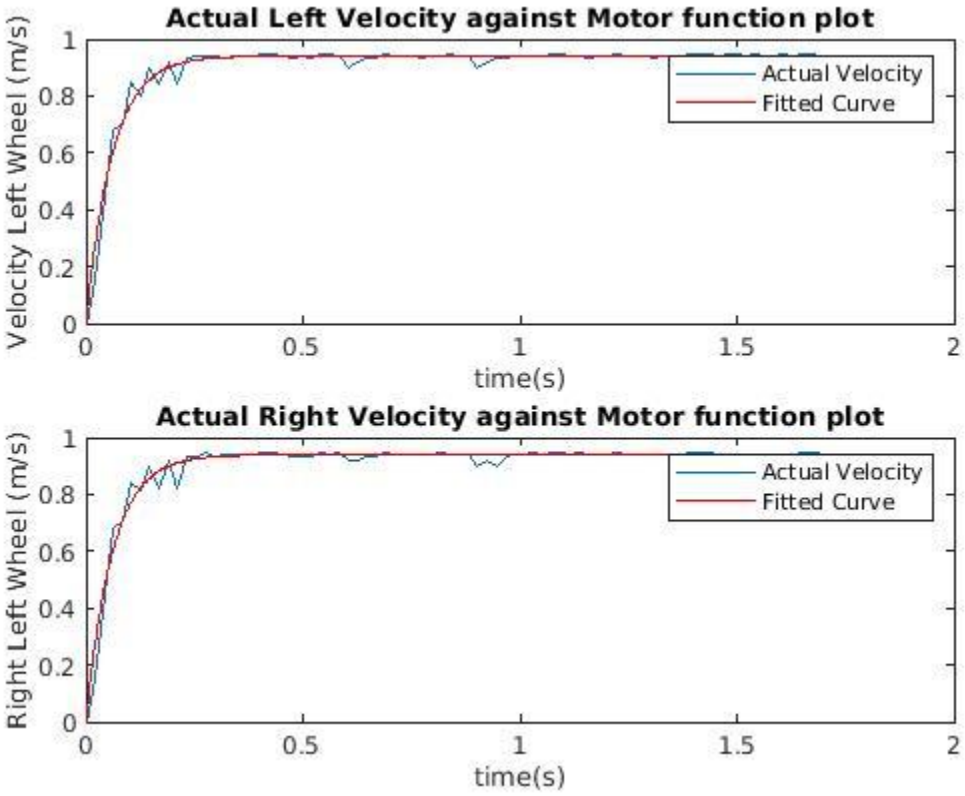
## Motor Constants

For this section, we need to calculate two separate motor constants: K and tau. To calculate these, we measured the actual values of the left and right wheels when we give them a 300 input and plotted how those wheels reached that speed. Then, we used MATLAB's fit() function to find a function that would fit this graph. For the base of this function, we used the motor

$$M_{base}(s) = \frac{\frac{K}{tau}}{s + \frac{1}{tau}}$$

function we have which is . Which. By inverse laplacing, we can turn into a function of time: $M_{base}(t) = 300K(1 - e^{\frac{-t}{tau}})$ where 300 is the value of our input when testing, and K and tau are the values we are fitting for.

The graphs for these fits look like this:




And the K and tau values (when averaged between both wheels) end up being:
K = 0.0031, tau = 0.0602.

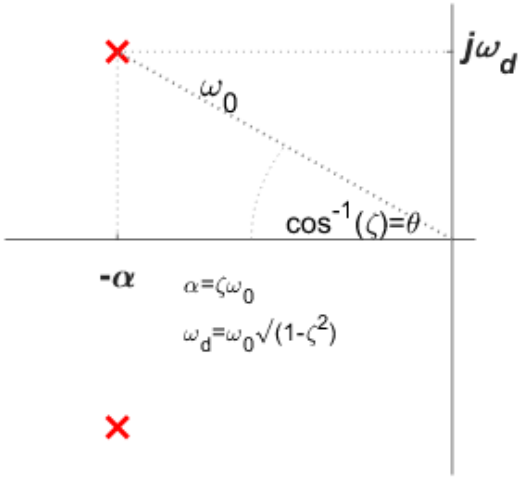
## Performance Specifications

Enhanced model and the control system performance specifications and why you chose them. We are planning to create a slightly underdamped system. We want dampening values (zeta) of around .90 and .85. We have two values because it is a fifth degree equation, so we are separating it into two values for more stability. This means we have a high decay of angles and a frequency of oscillations relatively close to the natural frequency. The disturbance we are working against is any type of push towards the rocky, specifically pushes that change his angle

of tilt but do not turn him. This could theoretically be represented as a constant step input with boundaries in the time domain.

## Poles of the System

For the poles of the system, we decided to use simple calculations based on the graphic below:




For our five poles, we chose one pole to be a constant value of $-\ w$ (where w is the natural frequency of the system), which is a critically dampening system. Then for the other 4 poles, we separate them into 2 pairs: for both of them, the points are: $w(-\ cos(\Theta)\ \pm\ sin(\Theta))$ where the angle is defined by the terms we said above, which lead to us using angles of 25° and 30°. This gives us the pole values of:

Constant: -9.3469
Angle of 25°: - 4.5210 & -12.4213
Angle of 30°: -3.4212 & -12.7681

## Controller Parameters

For determining the controller parameters (Kp, Ki, Jp, Ji, and Ci) we use a very simple script. This script uses MATLAB's syms function to create a simulation of the system using the values we already calculated. It also creates a target characteristic polynomial using the poles we gave the system, and using both of these it calculates the values of Kp, Ki, Jp, Ji, and Ci. Using the values we showed to calculate above, we get controller parameters of:
Kp = 13046, Ki = 61176, Jp = 497.7807, Ji = -15643, Ci = -20073.

# Rocky Balance Base System


## Nabih Estefan and Olivia Jo Bradley


```
clear all
```

### Determine tau and K

```
syms s K_sys tau
desiredVel300 = 1;

values300 = load('actualVel300.mat').actualVel300;
final = 80;

Vin = 300;

leftV = values300(1:80,1);
rightV = values300(1:80,2);
time = values300(1:80,3)/1000;
time = time - time(1);

Ms = ((K_sys/tau) / (s + 1/tau)) * Vin/s; %for final value theroem
Mt = ilaplace(Ms)
```

```
Mt =
```

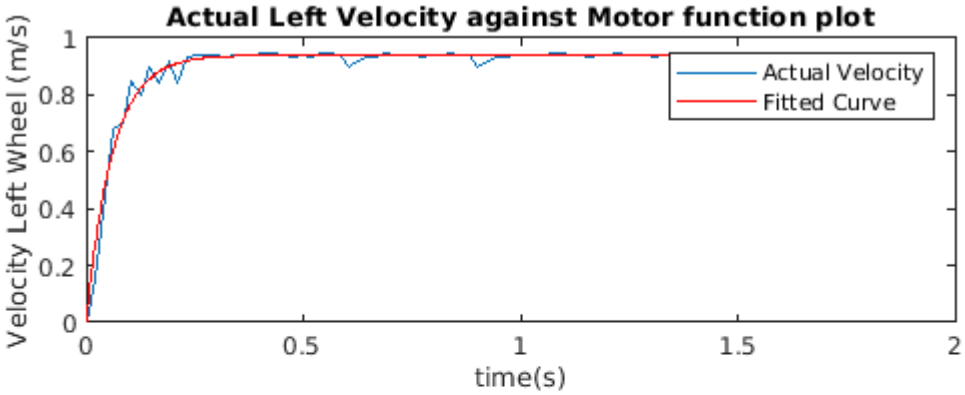
$$300\,K_{\mathrm{sys}} - 300\,K_{\mathrm{sys}}\,\mathrm{e}^{-\frac{t}{\tau}}$$

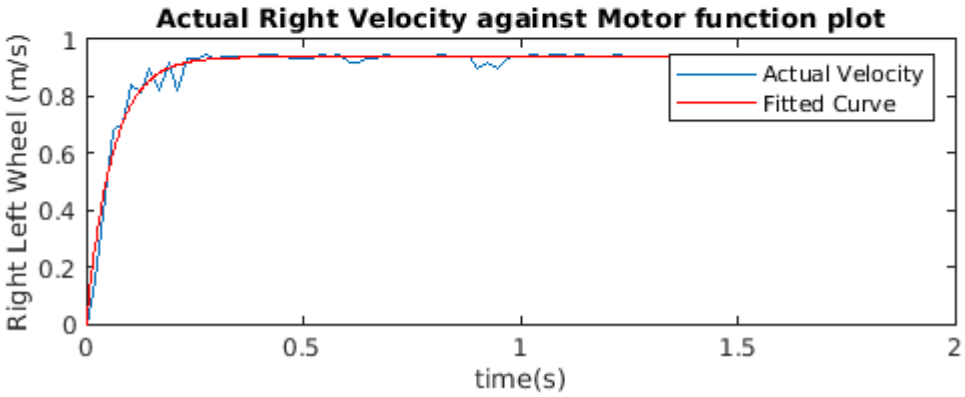
```
fittypeLeft = fittype(char(Mt), 'independent', 't');
fLeft = fit(time, leftV, fittypeLeft, 'Start', [time(2), leftV(2)]);
fRight = fit(time, rightV, fittypeLeft, 'Start', [time(5), rightV(5)]);

figure()
subplot(2,1,1)
plot(time, leftV)
hold on
plot(fLeft)
title('Actual Left Velocity against Motor function plot')
xlabel('time(s)')
ylabel('Velocity Left Wheel (m/s)')
legend('Actual Velocity', 'Fitted Curve')
hold off

subplot(2,1,2)
plot(time, rightV)
hold on
plot(fRight)
title('Actual Right Velocity against Motor function plot')
xlabel('time(s)')
ylabel('Right Left Wheel (m/s)')
legend('Actual Velocity', 'Fitted Curve')
```

```
hold off
```




```
tauL = fLeft.tau;
kL = fLeft.K_sys;
tauR = fRight.tau;
kR = fRight.K_sys;

tau = (tauL + tauR)/2 % 16.5993
```

```
tau = 0.0602
```

```
K_Sys = (kL + kR)/2 % 0.0031
```

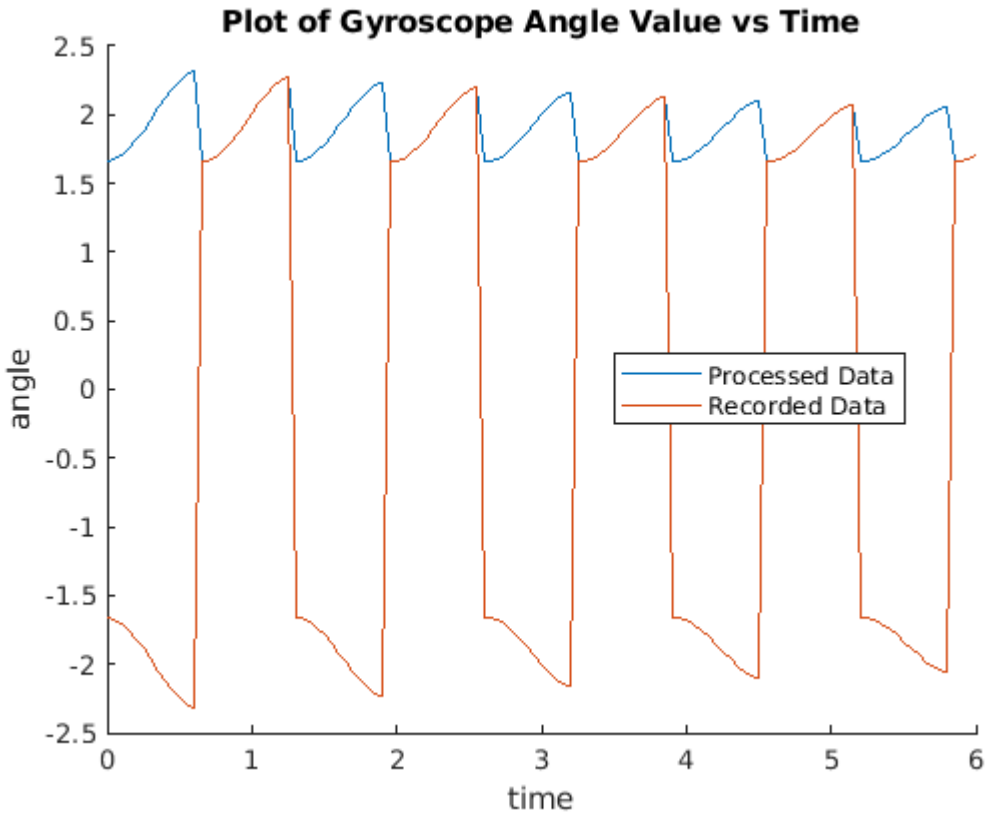
```
K_Sys = 0.0031
```

### Determining the natural frequency and effective length

```
load('natFreqData.mat')
absNatFreqData = abs(natFreqData); % not abs valued
timeForFreq = linspace(0,6,length(absNatFreqData))';

figure()
hold on
plot(timeForFreq, absNatFreqData)
plot(timeForFreq, natFreqData)
hold off
title("Plot of Gyroscope Angle Value vs Time")
```

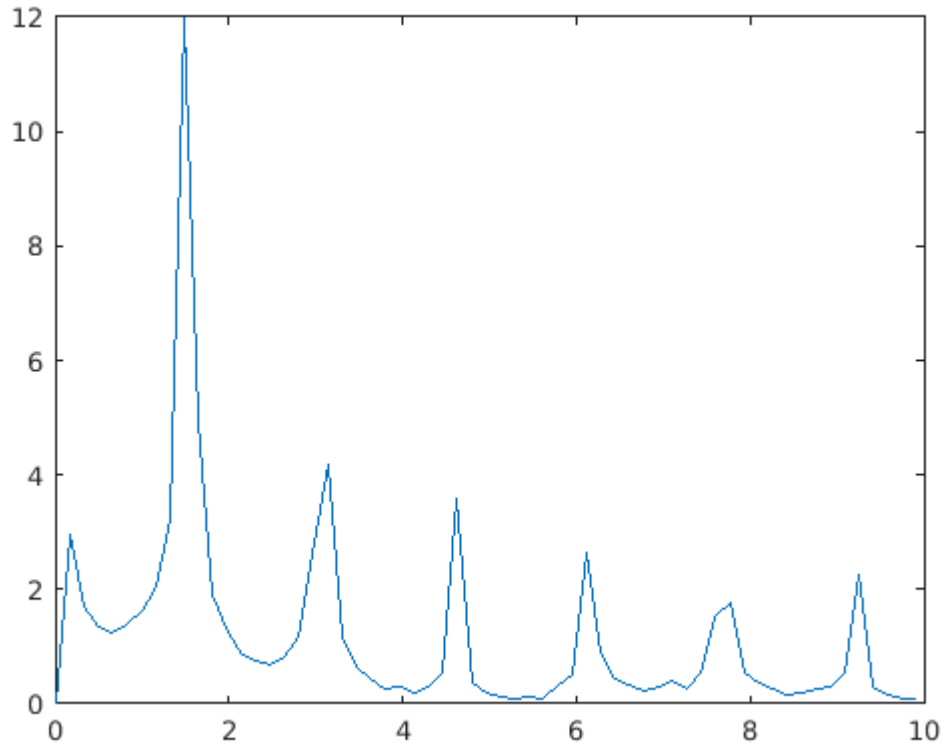
```
xlabel('time'); ylabel('angle')
legend("Processed Data", "Recorded Data", "location", "best")
```




```
t = timeForFreq; %0:.001:1-0.001;
Fs = 20; %1e3;
x = absNatFreqData;
x = detrend(x,0);
xdft = fft(x);
freq = 0:Fs/length(x):Fs/2;
xdft = xdft(1:length(x)/2+1);
```

```
Warning: Integer operands are required for colon operator when used as index.
```

```
figure()
plot(freq,abs(xdft));
```




```
[~,I] = max(abs(xdft));
fprintf('Maximum occurs at %d Hz.\n',freq(I));
```

```
Maximum occurs at 1.487603e+00 Hz.
```

```
wn = freq(I) * 2 * pi
```

```
wn = 9.3469
```

```
g = 9.81; % m/s
leff = 0.5588 - g/(wn^2)% in meters
```

```
leff = 0.4465
```

**Performance Specifications**

We are planning to create a slightly underdamped system. We want dampening values (zeta) of around .90 and .85. We have two values because it is a fifth degree equation, so we are separating it into two values for more stability. This means we have a high decay of angles and a frequency of oscillations relatively close to the natural frequency. The disturbance we are working against is any type of push towards the rocky, specifically pushes that change his angle of tilt but do not turn him. This could theoretically be represented as a consant step input with boundaries in the time domain.

**Identify locations for the poles of our system**

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

Hv = -s/L/(s^2-g/L);                     % TF from velocity to angle of pendulum
K = Kp + Ki/s;                           % TF of the angle controller
J = Jp + Ji/s + Ci/s^2;                  % TF of the controller around the motor
M = a*b/(s+a);                           % TF of motor
Md = M/(1+M*J);                          % TF of motor + feedback controller around it
                                         % J is applied on the feedback path

% pretty(collect(Md)) % display Md(s)

Htot = 1/(1-Hv*Md*K);                     % this is the total transfer function from dis

(simplify(Htot)) % display the total transfer function
```

```
ans =
```

$$-\frac{1}{\frac{a\,b\,s^2\,(\mathrm{Ki}+\mathrm{Kp}\,s)}{(g-L\,s^2)\,(a\,s^2+s^3+\mathrm{Ci}\,a\,b+\mathrm{Jp}\,a\,b\,s^2+\mathrm{Ji}\,a\,b\,s)}-1}$$

```
% Substitute parameters and solve

% system parameters
g = 9.81;                                % gravitational acceleration
L = leff;                                 % motor time constant
a = 1/tau;
b = K_Sys;

Htot_subbed = subs(Htot);                % substitutes parameters defined above into Hto
```

**Kp and Ki values that meet our specifications**

```
% define the target poles
angle1 = 25;
angle2 = 30;

p1 = wn*(-cosd(angle1) + sind(angle1))
```

```
p1 = -4.5210
```

```
p2 = wn*(-cosd(angle1) - sind(angle1))
```

```
p2 = -12.4213
```

```
p3 = -wn
```

```
p3 = -9.3469
```

```
p4 = wn*(-cosd(angle2) + sind(angle2))
```

```
p4 = -3.4212
```

```
p5 = wn*(-cosd(angle2) - sind(angle2))
```

```
p5 = -12.7681
```

```
% this is the target characteristic polynomial
tgt_char_poly = (s-p1)*(s-p2)*(s-p3)*(s-p4)*(s-p5);

% get the denominator from Htot_subbed
[num, den] = numden(Htot_subbed);

% find the coefficients of the denominator polynomial TF
cDenom = coeffs(den, s);

% divide out the coefficient of the highest power term
cDenom = coeffs(den, s)/(cDenom(end));

% find coefficients of the target charecteristic polynomial
cTarget = coeffs(tgt_char_poly, s);

% solve the system of equations setting the coefficients of the
% polynomial in the target to the actual polynomials
solutions = solve(cDenom == cTarget, Jp, Ji, Kp, Ki, Ci);

% display the solutions as double precision numbers

Jp = double(solutions.Jp)
```

```
Jp = 497.7807
```

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

```
Ji = -1.5643e+04
```

```
Kp = double(solutions.Kp)
```

```
Kp = 1.3046e+04
```

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

```
Ki = 6.1176e+04
```

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

```
Ci = -2.0073e+04
```

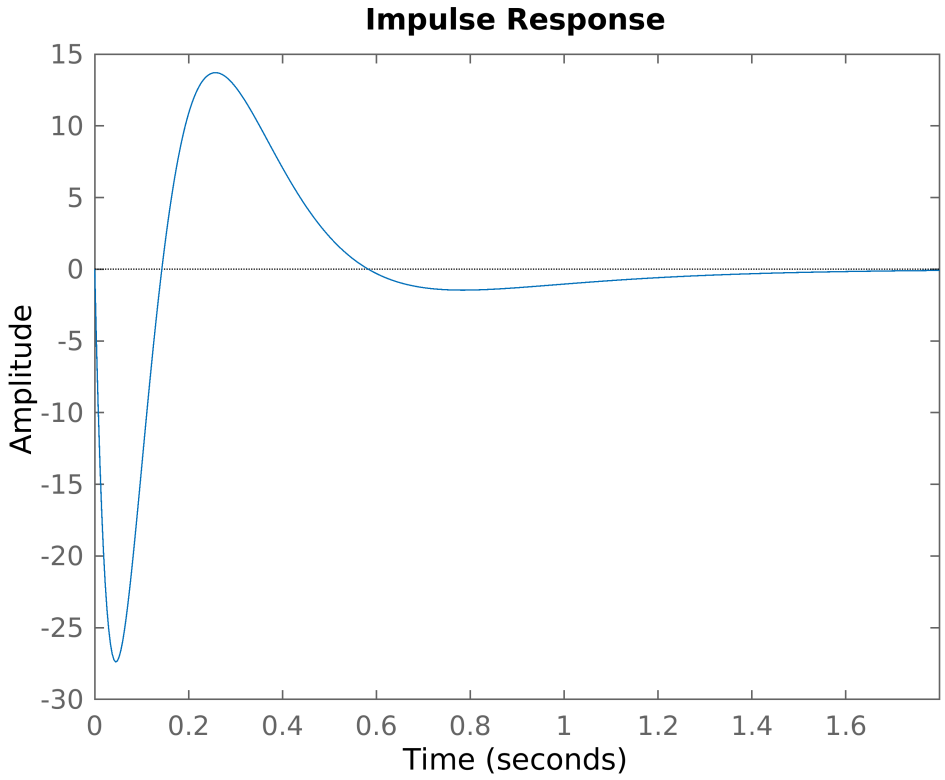
```
TF = char(subs(Htot));
% Define 's' as transfer function variable
s = tf('s');
eval(['TFH = ',TF]);
```

```
TFH =

  2.466e96 s^8 + 1.457e98 s^7 - 3.21e98 s^6 - 3.907e100 s^5 - 3.686e100 s^4 + 7.88e101 s^3 + 9.386e101 s^2
  ---------------------------------------------------------------------------------------------------------
  2.466e96 s^8 + 1.457e98 s^7 + 3.425e99 s^6 + 4.068e100 s^5 + 2.547e101 s^4 + 7.88e101 s^3 + 9.386e101 s^
```

```
Continuous-time transfer function.
```

```
figure()
impulse(TFH);
```

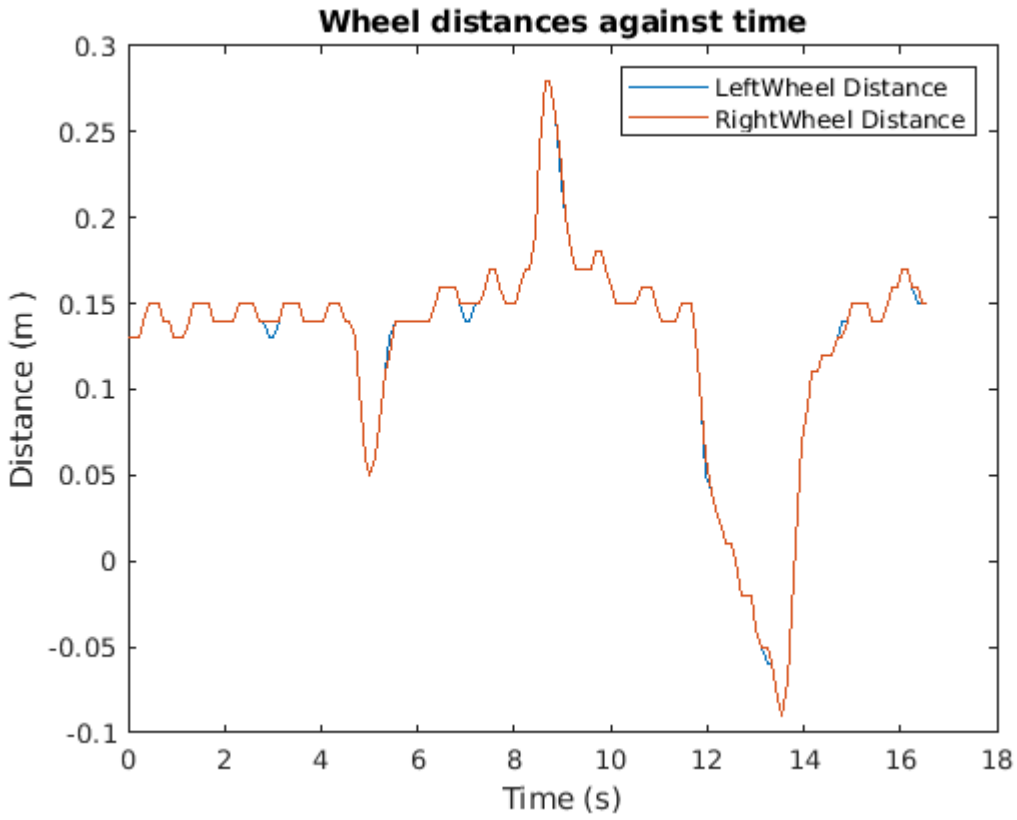



### x(t) and v(t) responses

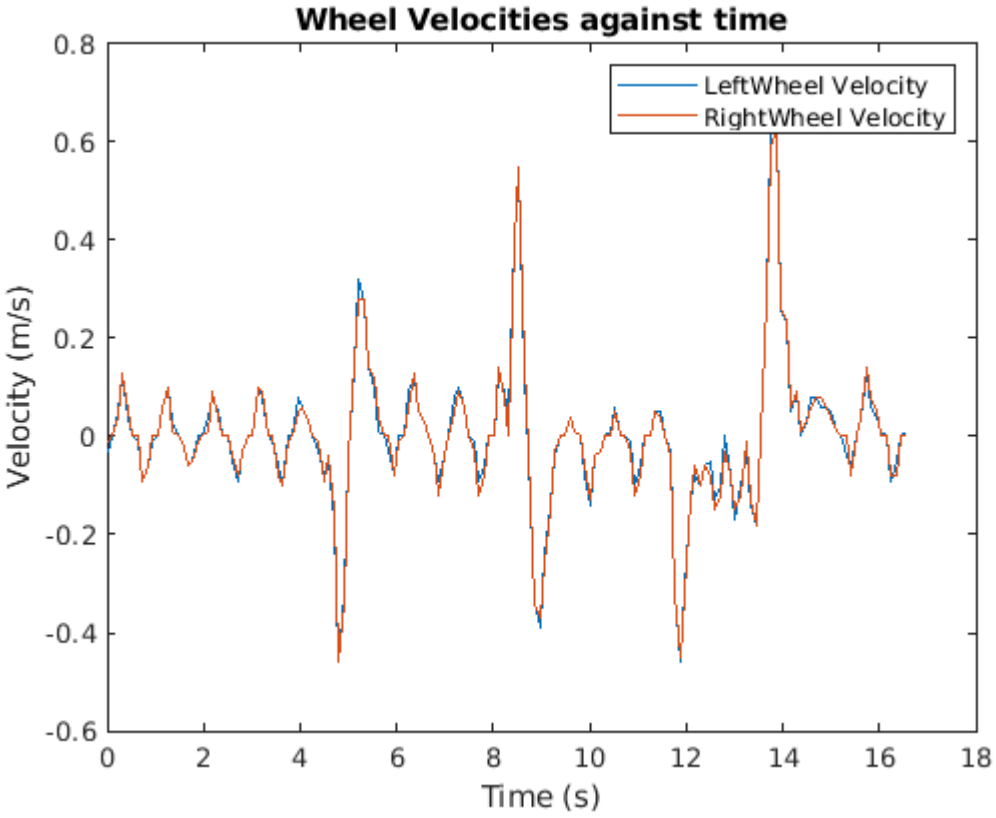
words

```
load('rockyNormalData.mat');
time = rockyNormalData(:,1);
time = time - time(1);
time = time /1000;
distLeft = rockyNormalData(:,2);
distRight = rockyNormalData(:,3);
velLeft = rockyNormalData(:,4);
velRight = rockyNormalData(:,5);

figure()
plot(time, distLeft)
hold on
plot(time, distRight)
legend('LeftWheel Distance', 'RightWheel Distance')
title("Wheel distances against time")
xlabel("Time (s)")
ylabel("Distance (m )")
hold off
```




```
figure()
plot(time, velLeft)
hold on
plot(time, velRight)
legend('LeftWheel Velocity', 'RightWheel Velocity')
title("Wheel Velocities against time")
xlabel("Time (s)")
ylabel("Velocity (m/s)")
hold off
```




In the graphs above, you can see the distance and velocities of the wheels in a normal run. You can also see the external inputs we gave the Rocky robot at times t = 5, 9, and 12, with the one at t = 12 being a much bigger than the other two. Besides, you can also see how the velocity reacts to the pushes to get the rocky to return to its original spot.

```
// 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;
int16_t circleCounter = 0;
int16_t balanceCounter = 0;

#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

////////////////////////////////////////////////////////////////////////////
// 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 circleRocky()
{
    motors.setSpeeds((int16_t)(0), (int16_t)(0));
    delay(UPDATE_TIME_MS);
    Serial.print("TURN");
    Serial.print("\t");
}


void BalanceRocky()
{
  // Enter the control parameters here
  float Jp = 497.7807;
  float Ji = -15643;
  float Kp = 13046;
  float Ki = 61176;
  float Ci = -20073;


  // these are the control velocities to be sent to the motors
  float v_c_L, v_c_R;
  // this is the desired speed produced by the angle controller
  float v_d = 0;


 // 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 (integral of vel)
 // 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 balance

  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));
  //Serial.print("BALANCE");
  //Serial.print("\t");
  // circleRocky();


}



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;
  // stores difference in distance in encoder clicks traversed by the wheels
  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 angle calibration procedure
    // del_theta corrects for long-term drift
    angle_rad = ((float)angle)/1000/180*3.14159 - FIXED_ANGLE_CORRECTION - del_theta;
```

```
  // If angle is not within +- 6 degrees, reset counter that waits for start
  if(angle_rad > 0.1 || angle_rad < -0.1)
  {
    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();
    circleCounter ++;
    //delay(10);
    if (circleCounter >= 25)
    {
      circleRocky();
      balanceCounter ++;
      if (balanceCounter >= 5)
      {
        circleCounter = 0;
        balanceCounter = 0;
        balanceResetEncoders();
      }
    }
  }
  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());
  }
  // do the printing every 105 ms.
  // Don't want to do it for integer multiple of 10ms to not hog the processor
  if(cur_time - prev_print_time > 103)
  {
//    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);
      Serial.println(circleCounter);
    prev_print_time = cur_time;
  }
}
```