Rocky Project

Sydney Chung, Amit Kumar-Hermosillo, Austin Cline

March 2024

Link to final Rocky Demonstration video: `https://drive.google.com/file/d/1nDenw_emLLbFz2iOxh1ZBsGk58QRZF7p/view?usp=sharing`

1 Parameter Identification

1.1 Motor Constants

Per the project instructions, the motor has a transfer function (from the controlled input velocity to the actual output) of

$$M(s) = \frac{\frac{K}{\tau}}{s + \frac{1}{\tau}} \tag{1}$$

with K and τ being constants unique to each motor, and identifying those constants is critical for a well performing Rocky. To estimate those parameters, we held our Rocky upright and ran the motors for three seconds along the carpeted floor of MAC 128 while recording the measured motor speed using the motor test calibration sketch found in the appendix.

With the data in hand, we needed to now convert the motor transfer function into an equation for velocity in the time domain, since this the domain our measured data exists in. The procedure for that is below.

$$M(s) = \frac{V(s)}{V_c(s)} = \frac{\frac{K}{\tau}}{s + \frac{1}{\tau}} \tag{2}$$

$$V_c(s) = \frac{300}{s}$$

$$V(s) = \frac{300\frac{K}{\tau}}{s(s + \frac{1}{\tau})}$$

Now we take the inverse Laplace transform using tables:

$$v(t) = 300k(1 - e^{\frac{-t}{\tau}})$$

To obtain estimates for our two motor parameters, we used MATLAB's curve-fitting toolbox to fit this curve onto our measured motor data (shown in Figure 1) and got $K = 0.0031$ and $\tau = 0.1044$

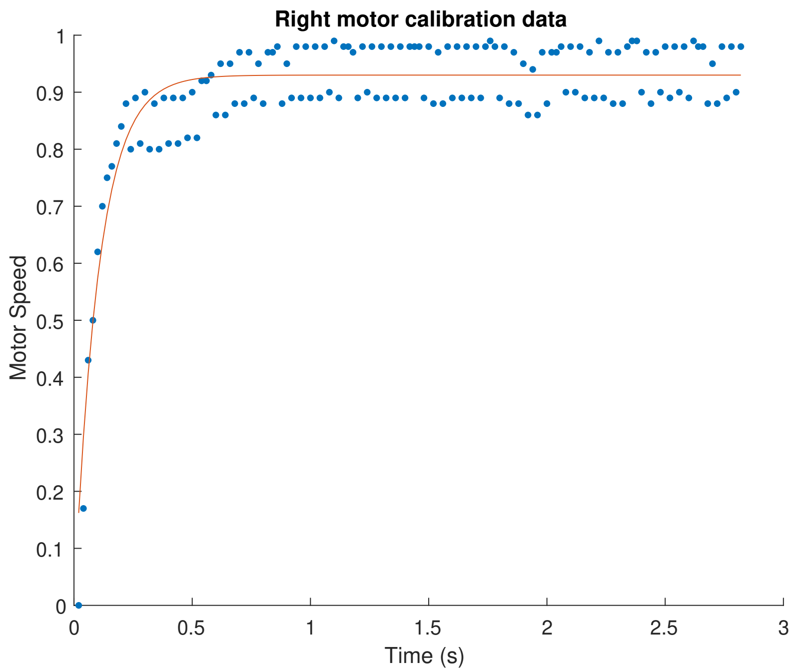

Figure 1: Right Motor Speed

1.2 Natural Frequency & Pendulum Length

In order to find our Rocky's natural frequency and the effective length of the pendulum body, we needed to calibrate its gyroscope. To do so, we started with the Rocky laying down horizontally on the table, and then picking it up and holding it vertically in the air as if it were balancing. This was done while running the gyro calibration sketch found in the appendix to track angle measurement offset.

After accounting for the angle correction needed, we could collect more gyro data to measure natural frequency. We compiled and loaded the gyro calibration sketch again with the Rocky (wheels off) laying flat. After loading, we opened the serial monitor and held the Rocky by two points at the top, allowing it to swing freely for a few seconds to collect angle measurement data. This data output was then copied into MATLAB and plotted to find the period (found experimentally by inspecting the data), which we plugged into the following equation to find natural frequency:

$$\omega_n = \frac{2\pi}{\text{period}} \; [rad/s] \tag{3}$$

$$\omega_n = \frac{2\pi}{1.3} = 4.8332\ [rad/s]$$

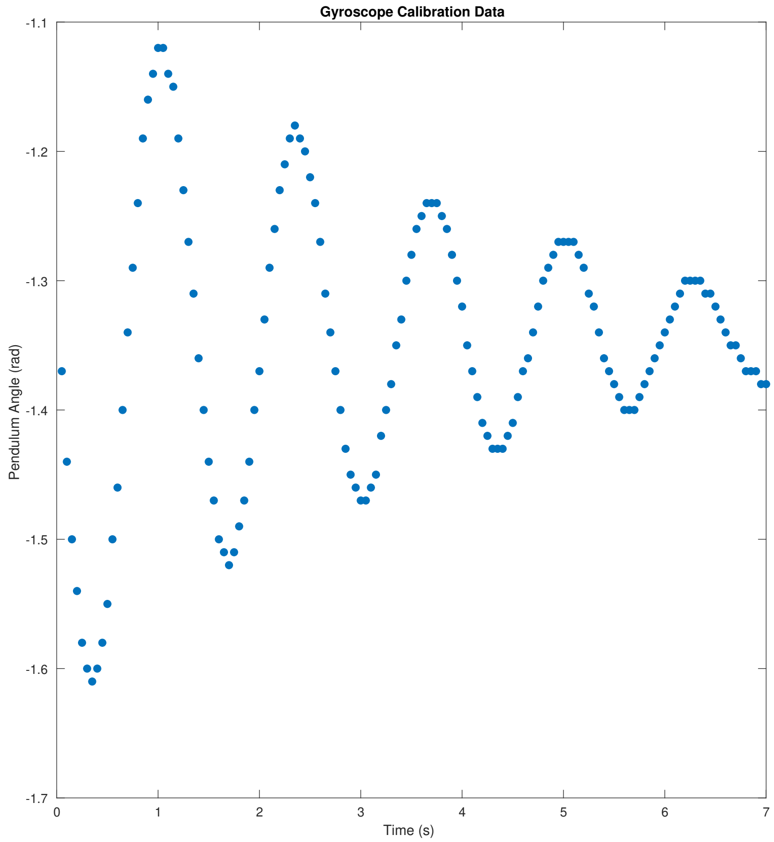

Figure 2: Gyroscope Calibration Data

Now that we have natural frequency, we can solve backwards for pendulum length using the provided approximation.

$$\omega_n = \sqrt{g/l_{\text{eff}}} \tag{4}$$

This gives us an effective length $l_{\text{eff}} = 0.4195$.

2 Initial (3 pole) System

2.1 Pole Values

For our Initial System, the poles we chose were $-1 \pm 2\pi i$ and -10. The imaginary poles were selected due to their rapidly decaying nature without the presence of large initial oscillations. The real pole was selected to negative be negative to influence the system to have a decaying response to return to steady state. After having a general idea of what our poles should look like we determined the ones we used through a few iterations of trial and error. We found that the larger the negative value of the real pole, the Rocky would return back to steady state quicker. This was juxtaposed with the limitations of the physical Rocky system that were only able to respond to input so fast.

2.2 Calculating Control Constants

From our selected poles we were able to derive our control constants: K_p and K_i. We did this by expanding our poles into a third order polynomial, find the target coefficients of the polynomial, reorder the coefficients to match the denominator polynomial, which is found from the closed loop transfer function:

$$H_{\text{Cloop}} = \frac{1}{1 - (\Theta)MK} \tag{5}$$

Where Θ represents the transfer function from velocity to angle of a pendulum $\frac{\frac{-s}{l}}{s^2 - g/l}$, M represents the transfer function of a 1st order model of the motor $\frac{a*b}{s+a}$, and $K = K_p + \frac{K_i}{s}$ represents a transfer function of the PI angle controller. Lastly, the system of equations that set the coefficients of the target polynomial into actual polynomials is solved, outputting K_p and K_i as shown below. We found that the this model failed to correct itself effectively. Despite the angle returning to zero radians, this outcome resulted in our Rocky continuously moving forward or backwards faster and faster, as well as driving away from its original location. This seemed to be due to it not make the necessary over correction, just barely making it back to zero before falling in the same direction it corrected itself from. This phenomenon is also visible in our Simulink plots, shown below.

Parameter	Value
K_p	1185
K_i	8883

2.3 Simulink model

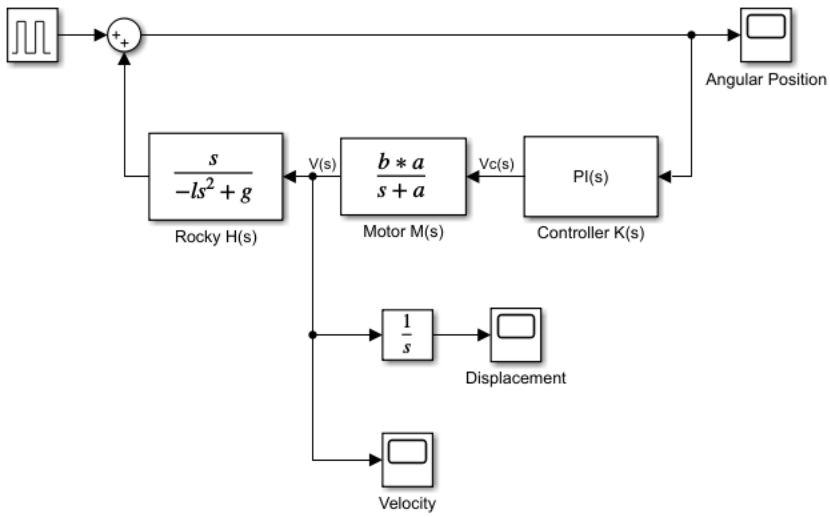

Figure 3: Block Diagram for 3 Pole System

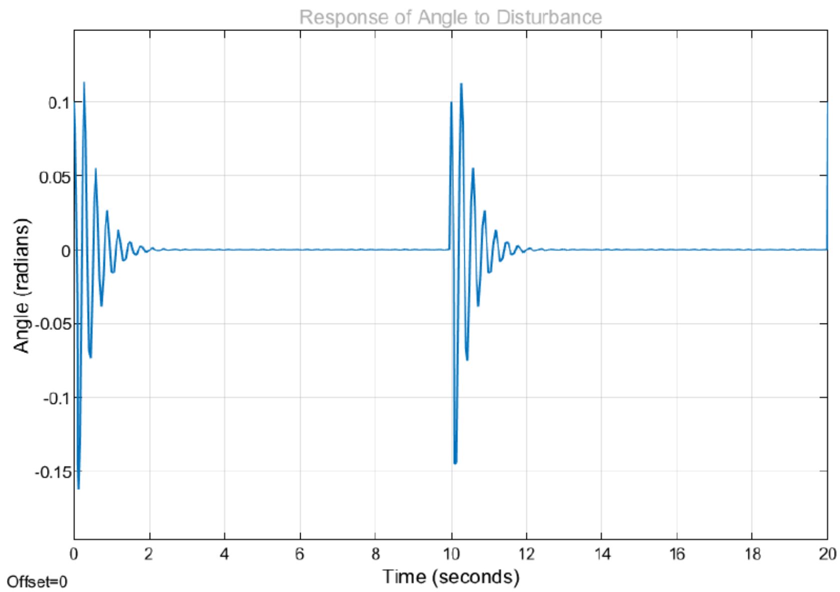

Figure 4: Angle Impulse Responses for 3 Pole System

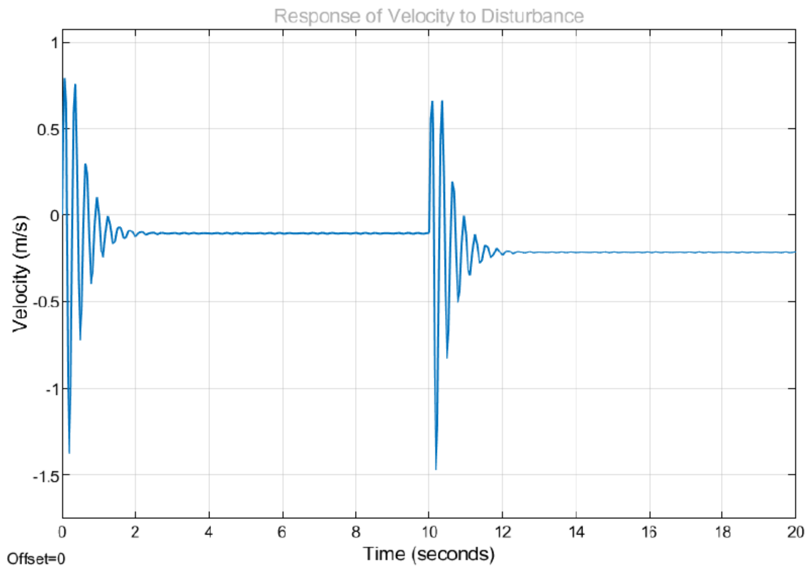

Figure 5: Velocity Impulse Responses for 3 Pole System

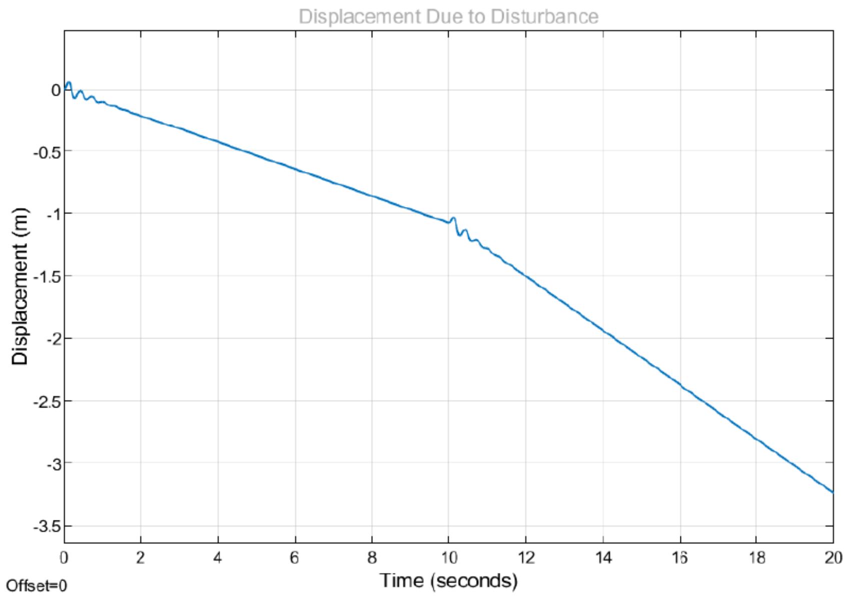

Figure 6: Displacement Impulse Responses for 3 Pole System

3 Balancing System

3.1 Pole Values

For our Revised System, the poles we chose were $-3.38 \pm 3.45i$, -10, -4.8332, and -4.8332. The imaginary poles were selected due to their rapidly decaying nature without the presence of large initial oscillations. The real poles were selected to be negative to influence the system to have a decaying response to return to steady state. We combined trial and error with a bit of rough hand calculations to determine our poles this. time we calculated our first and second (imaginary poles) and our fourth and fifth (real poles) with dampening values.

Those calculations were just choosing a damping ratio that seemed appropriate for both the angular position control loop and the motor velocity control loop (both of them being very slightly underdamped) and applying the following formula from Day 7 to find the corresponsing poles assuming this was a 2 pole system.

$$p_{1,2} = -\zeta\omega_n \pm i\omega_n\sqrt{1-\zeta^2}$$

After that, we just performed a bit of trial-and-error tuning in order to find exact values that worked well for us.

3.2 Calculating Control Constants

From our selected poles we were able to derive our control constants: K_p, K_i, J_p, J_i, and C_i. We did this by expanding our poles into a fifth order polynomial, finding the target coefficients of the polynomial, reordering the coefficients to match the denominator polynomial, which is found from the closed loop transfer function:

$$H_{\text{Cloop}} = \frac{1}{1 - (\Theta) M_{fb} K} \tag{6}$$

Where Θ represents the transfer function from velocity to angle of a pendulum $\frac{\frac{-s}{l}}{s^2 - g/l}$, M_{fb} represents Black's formula of a 1st order model of the motor with a PI feedback controller (in order to ensure the Rocky balances in place) $\frac{a*b}{s+a+J(a+b)}$ and $J = J_p + \frac{J_i}{s} + \frac{C_i}{s^2}$, and $K = K_p + \frac{K_i}{s}$ represents a transfer function of the PI angle controller. Lastly, the system of equations that set the coefficients of the target polynomial into actual polynomials is solved, outputting our control parameters as given in the table below. We found this system was far more effective at balancing the Rocky, as it was able to maneuver back and forth in order to keep the Rocky balancing in one spot.

Parameter	Value
K_p	33819
K_i	172920
J_p	2169
J_i	-27530
C_i	-43550

3.3 Simulink model

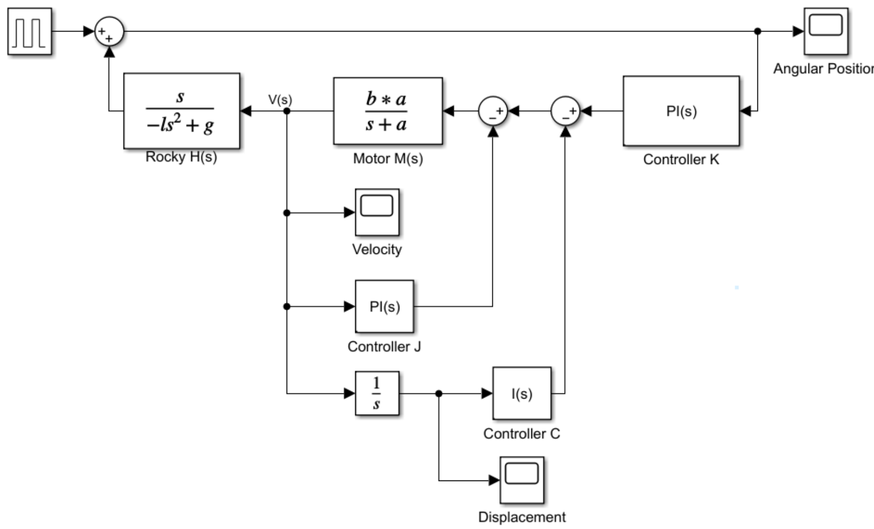

Figure 7: Block Diagram for 5 Pole System

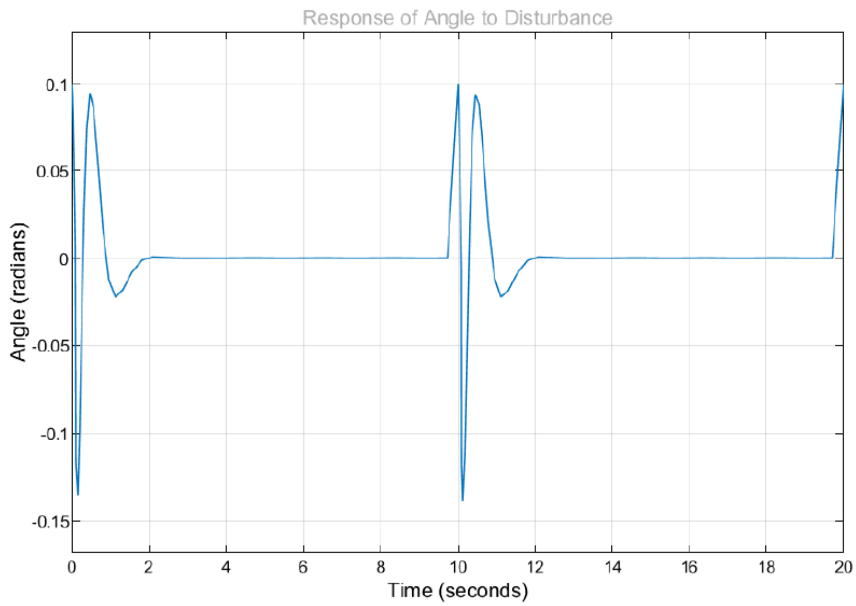

Figure 8: Angle Impulse Responses for 5 Pole System

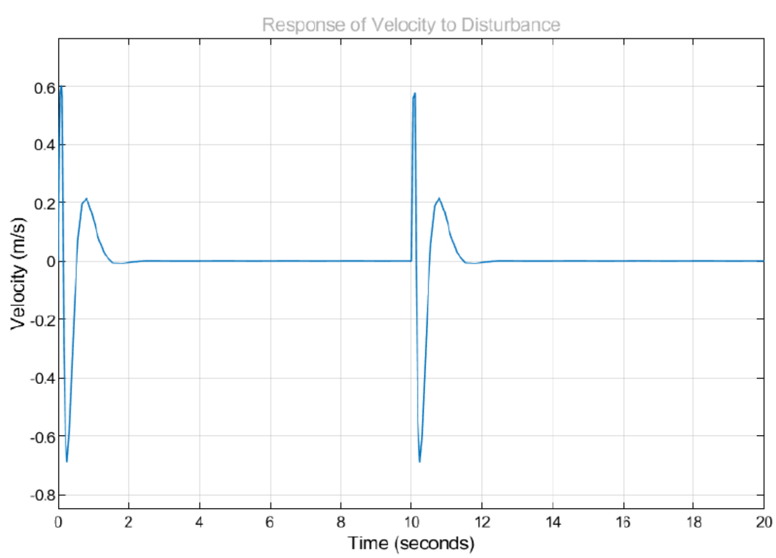

Figure 9: Velocity Impulse Responses for 5 Pole System

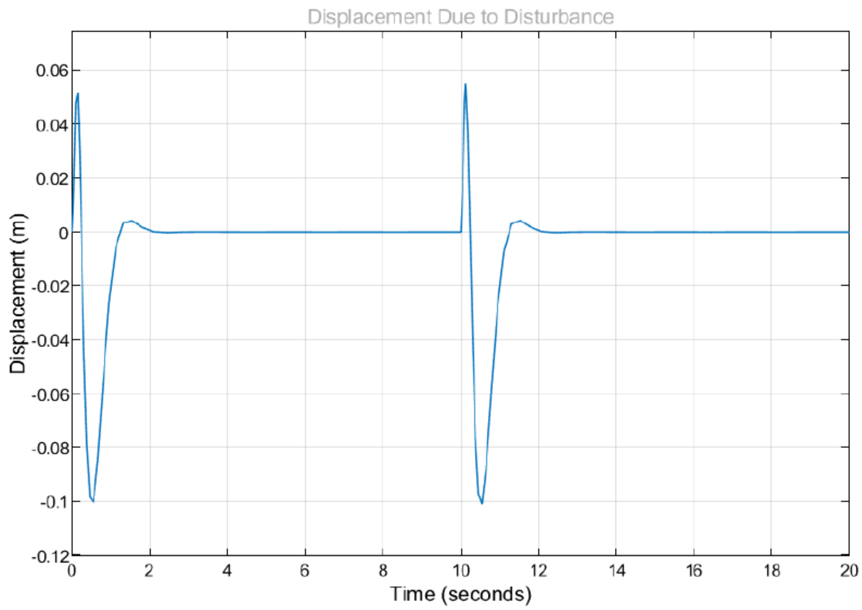

Figure 10: Displacement Impulse Responses for 5 Pole System

4 Appendix

4.1 Miscellaneous Math/Parameter Estimation - MATLAB

```
%% MOTOR PARAMETER ESTIMATION
t = .02:.02:.02*141;
l = data(:, 1)';
r = data(:, 1)';

% from cftool with custom equation
K = 0.0031
tau = 0.1044
hold on
plot(t, r, '.', 'MarkerSize', 10)
plot(t, 300*K*(1-exp(-t/tau)))
xlabel("Time (s)")
ylabel("Motor Speed")
title("Right motor calibration data")

```

```
%% GYROSCOPE CALIBRATION
t = .05:.05:.05*140;
figure
plot(t, gyro, '.')
title("Gyroscope Calibration Data")
xlabel("Time (s)")
ylabel("Pendulum Angle")

wn = 2*pi * (1/1.3)
leff = 9.8 / wn^2
```

4.2 3 Pole System Parameter Calculation - MATLAB

```
% 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 = 0.4195;   %effective length
a = 1/0.1044;           %nominal motor parameters
b = 0.0031;         %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


% 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)

% solve the system of equations setting the coefficients of the
% 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)
```

```matlab
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
```

4.3 5 Pole System Parameter Calculation - MATLAB

```matlab
% 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-combinedz 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 = .4195;  %effective length
a = 1/.1044;           %nominal motor parameters
b = .0031;        %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 = -1 + 2*i   % dominant pole pair
p2 = -1 -2*i    % dominant pole pair
p3 = -6
p4 = -42    % dominant pole pair
```

```
p5 = -24     % 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
```

4.4 Balancing Rocky Sketch - ARDUINO

```
// 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 (0.27)  // ***** 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 = 30370;
  // float Ki = 154230;
  // float Ci = -8709.9;
  // float Jp = 2169.5;
  // float Ji = -19442;
  float Kp = 6879.2;
  float Ki = 33267;
  float Ci = -7847.4;
  float Jp = 567.3923;
  float Ji = -6306.2;




    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 =  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.
  //   v_j =
   //
   // 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 = 0.84 * v_d; // CHANGE HERE TO FIX MOTORS
     //v_c_L = 1 * v_d;
     v_c_R = v_d - (Jp*measured_speedR + Ji*distRight_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;
     Serial.println(v_d);

     // 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;
  }


```

```
}
```