# Rocky Report

Audrey Abraham, Maya Lum, Aydin O'Leary

March 2022

## 1 Motor Parameter Identification

We found the motor constants $\tau$ (time constant) and K using the following steps:

1. Upload the provided code `Rocky_Motor_Test_Calibration.m`. This code turns the motors on at a constant input signal of 300 (maximum power, forward direction – this is a step input) and measures the motor speeds at 20 millisecond intervals.
2. Hold the robot vertically with its wheels on the ground and run the code. Allow the robot to move forward when the wheels start to spin. The motors will run for 3 seconds then stop.
3. Record the values shown in the serial monitor. These values are the motor speeds at each time step.
4. Use the MATLAB curve-fitting tool (code in appendix) to fit the data using the equation.

$$y = K \cdot \left(1 - e^{-b \cdot x}\right)$$

with

$$\tau = \frac{1}{b}$$

Using this method, as seen in fig. 1, we determined K to be 0.654 for the left motor and 0.690 for the right, and $\tau$ to be 0.0325 for the left motor and 0.0371 for the right.

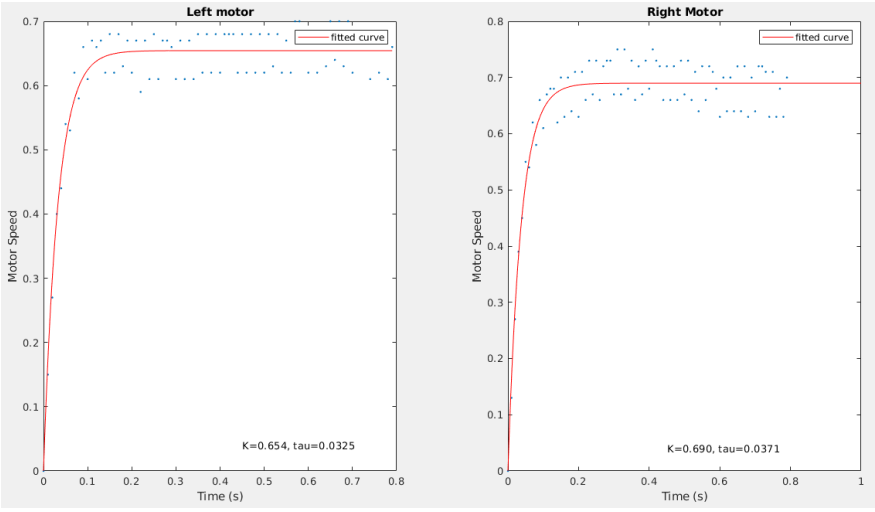


Figure 1: The motor calibration data with the fitted line and extracted constants.

## 2 Pendulum Parameter Identification

To find the natural frequency of the Rocky robot as an inverted pendulum, we began by calibrating the gyro: finding the gyro output value for a known angle, in this case, when the Rocky robot was fully upright. We took the following steps:

1. Place the robot flat on the ground, so that the gyro has a baseline value to start from.
2. Compile and load the provided code `Rocky_Gyro_Calibration.m`, which measures the angle using the gyroscope every 50 milliseconds and prints the gyro value to the Arduino's serial monitor.
3. Hold the robot upright in the air in the same position it would be in if it were balancing.
4. Record the gyro value shown on the Serial Monitor. This value corresponds to a constant offset of the angle measurement, which we subtracted from the angle measurements in our main control loop using the `FIXED_ANGLE_CORRECTION` constant. The value we found was 0.27.

With our gyro values calibrated, we then followed the procedure below to measure the Rocky robot's natural frequency:

1. Remove the wheels of the robot to directly access its axis of rotation.
2. Place the robot on the ground and upload `Rocky_Gyro_Calibration.m`.
3. Once the code has uploaded, hold the robot upside down by the axles. Apply a reaction force to the axles, but not a reaction moment, so the axle is in a constant position but can swing as a pendulum.
4. Swing the robot freely from left to right by giving it an initial velocity. Once the pendulum starts to slow, click "Stop AutoScroll" in the Arduino Serial Monitor to separate the pendulum data from the data where the robot isn't moving.
5. Copy and paste the angle measurements into MATLAB.

In MATLAB, we found the natural frequency of the robot using the `findpeaks()` function. `findpeaks()` gave us the indexes for the peaks, so, by multiplying the peak indexes by the constant sampling rate of 0.05 seconds, we were able to find the times of each peak. The differences between these peak times, which we averaged, gave us the period of the robot as an inverted pendulum, which was 1.35 seconds. As the frequency (in Hertz) is the inverse of the period in seconds, we then divided 1 by the period value to find the natural frequency of the Rocky robot, which was 0.741 Hz. We also converted this value to the frequency in radians per second by multiplying the Hertz frequency by $2\pi$, to get a natural frequency (rad/s) of 4.65.

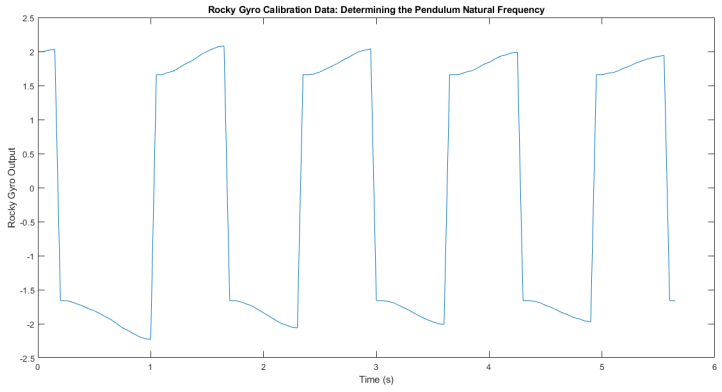


Figure 2: MATLAB plot of Rocky gyro output vs. time. We used this graph to visualize our gyro calibration data so we could double-check that our period and frequency values made sense.

We then used the natural frequency in radians per second to find the effective length of the robot. The natural frequency and effective length are related by the equation

$$\omega = \sqrt{\frac{g}{l}}$$
$$l = \frac{g}{\omega^2}$$

With our natural frequency of 4.65 rad/s and gravity being 9.81 m/s$^2$, we found that the effective length was 0.45 meters.

# 3 Initial Model

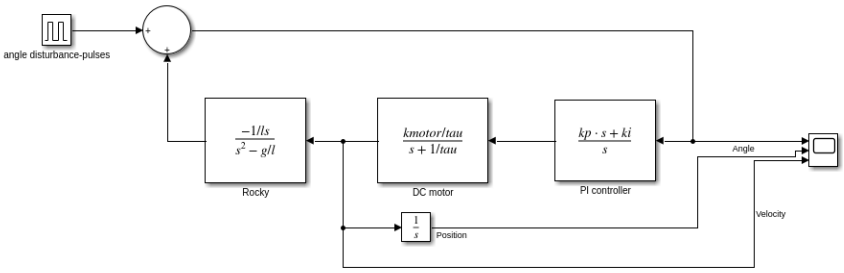


Figure 3: The block diagram of our initial model.

For the initial model as described in fig. 3, we picked arbitrary values of $k_p = 500$ and $k_i = 30$. This turned out to have a stable-looking result, judging by the plot seen in fig. 4 of the modeled position, velocity, and angle of Rocky.

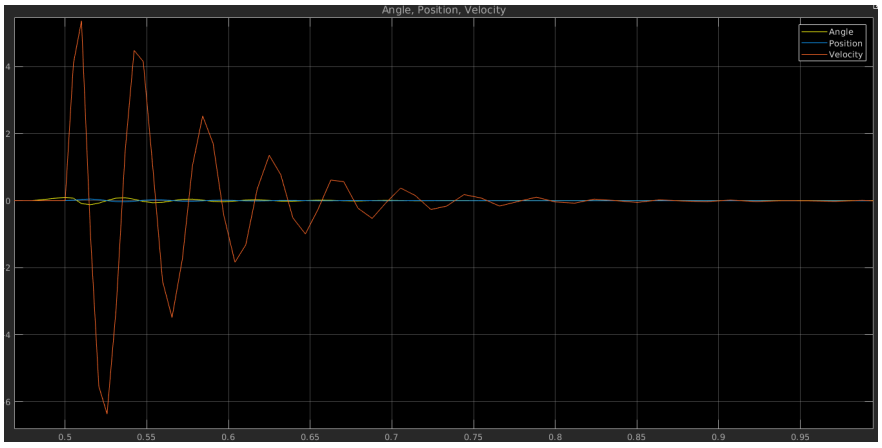


Figure 4: Plot of position, velocity, and angle of the modelled Rocky's reaction to an input disturbance.

This section was mostly used as an excuse to figure out how to use Simulink, hence the arbitrary constant selection. Real thought was put into determining poles for the base and enhanced models.

# 4 Base Model

To create the base model, we followed the instructions in `ESA_Sp22_Guide_Stationary_Balance_Invert_Pend_on_Cart.pdf`, which provided a diagram of the full control loop of a balancing Rocky. We created this control loop in Simulink to test out how well our control values matched the ideal behavior, shown in fig. 5.

In our basic control loop, there are five PI values we must set:

1. $K_p$, the proportional constant for controlling the desired overall velocity of the Rocky
2. $K_i$, the integral constant for controlling the desired overall velocity of the Rocky
3. $J_p$, the proportional constant for controlling the motor inputs such that they reach the desired overall velocity
4. $J_iC_p$ (referred to $J_i$ in our Arduino and MATLAB code), the integral constant for controlling the motor inputs such that they reach the desired overall velocity. As $J_i$ only appears in the control loop in combination with the effects of $C_p$, the two were combined into one constant equal to the sum of the two.
5. $C_i$, the integrating constant that controls the position of the Rocky.

To figure out what to set these constants to, we started by determining pole locations for the ideal response-to-disturbance behavior. We set the magnitude of the poles (or distance from the origin) to be at 5, close to the Rocky's natural frequency of 4.65 rad/s. We sketched out the ideal step response in fig. 6, wanting the system to quickly approach the desired value before the robot tips past the range of the small angle approximation, but not having it go so fast that the motor wouldn't be able to provide those speeds.

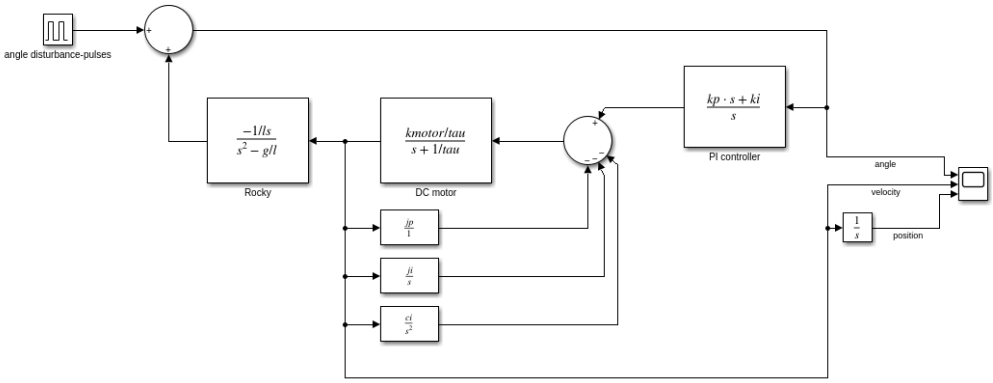


Figure 5: Our Simulink model of the full Rocky disturbance-balancing system.

Through prior experience, we determined that the ideal response corresponded to a damping constant of approximately 0.7 for a good balance of a quick response and a minimum of oscillation, which puts the poles at an angle above the real axis of 45 degrees. For the fourth and fifth poles, we set them at the natural frequency with an angle above the real axis of 55 degrees. We used the MATLAB code provided by the teaching team, `Rocky_closed_loop_poles.m`, to find the PI control values that would place the poles at these locations. However, we found that the balancing algorithm tended to over-correct for disturbances when using poles at 45 and 55 degrees, so we instead placed poles at 25 and 35 degrees from the real axis. These design decisions resulted in the following constants:

| | |
|---|---|
| $K_i$ | 22699 |
| $K_p$ | 4539.7 |
| $J_p$ | 765.19 |
| $J_i$ | -3833.4 |
| $C_i$ | -5345.2 |

When these control system constants were plugged into our Simulink model, the Rocky exhibited the following angle vs. time behavior plotted in fig. 7. Despite not working consistently due to low battery and multiple hardware failures, these poles and resulting PI constants were by far the most successful out of our tests. Using these constants, our Rocky was able to balance for up to 10 seconds before falling or losing power.

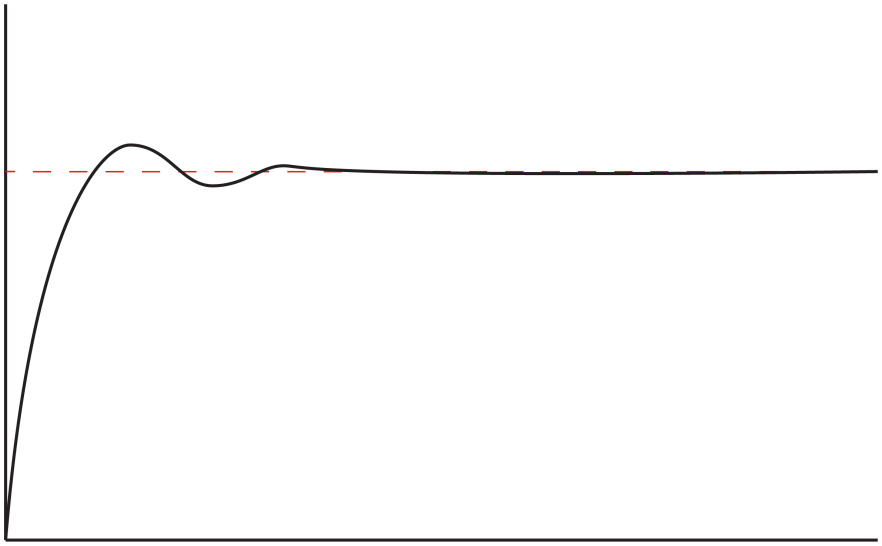

Figure 6: Ideal error correction vs. time behavior for the base model Rocky system.

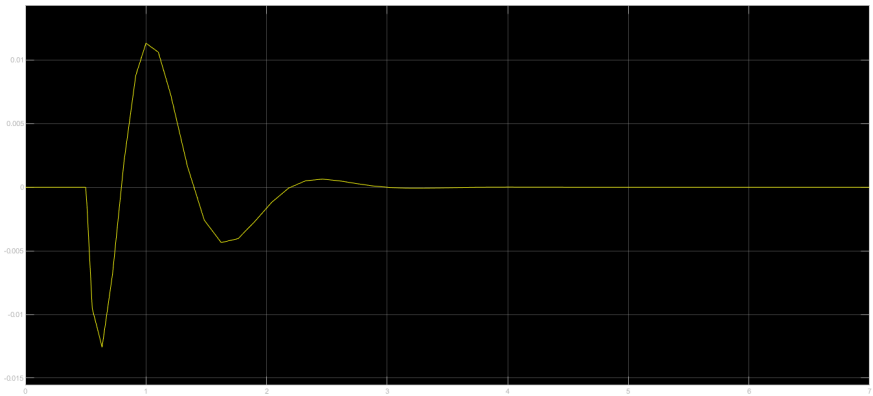

Figure 7: Simulink results for the Rocky's angle vs. time in the basic self-balancing model.

## 5 Enhanced Model

For the enhanced model, we modified our model to make Rocky move forwards while balancing. We realized that by using the same control loop as the base system, shown in fig. 5, but with the integral control constants set to 0, we could introduce a steady-state position error back into the system. Rocky is still able to self-correct in angle, as shown in fig. 9, but with $J_i$ and $C_i$ set to 0, it drifts in position. With this control loop and control constants, Rocky was able to move forward while balancing for about 5 seconds, stopping due to lack of battery power rather than instability in the control system.

## 6 Video

https://youtu.be/XMYJeberOak

## 7 Code & Reference

Our code and the `.pdf` files we used can be found at https://github.com/zbwrm/balance-control.

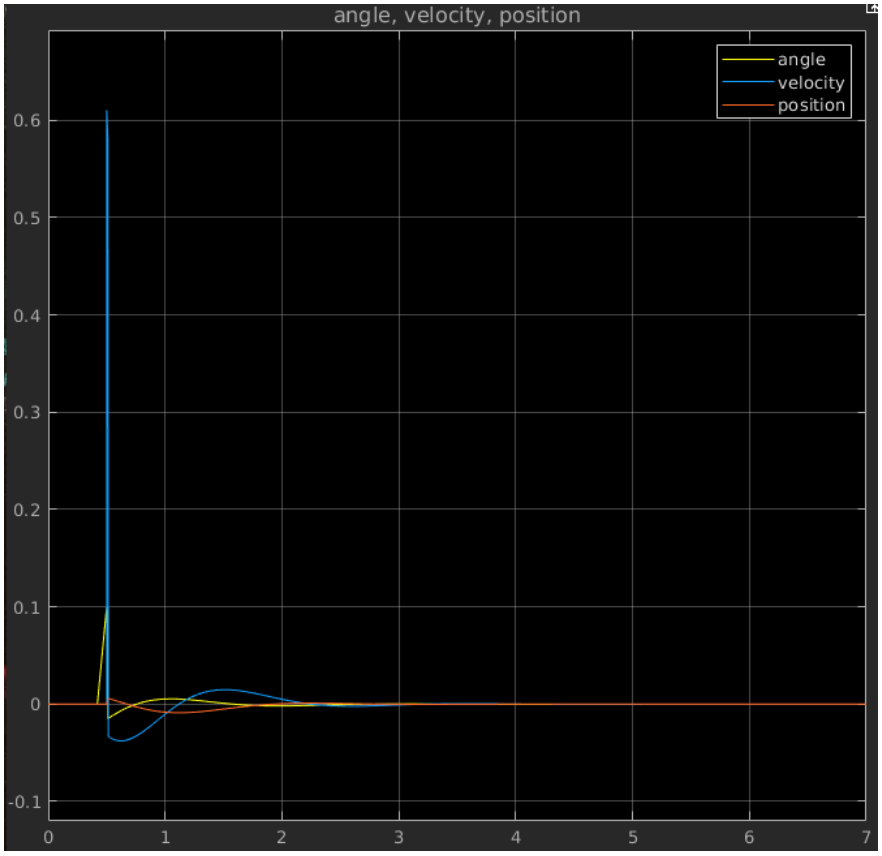


Figure 8: The angle, position, and velocity behavior of our modeled base system.

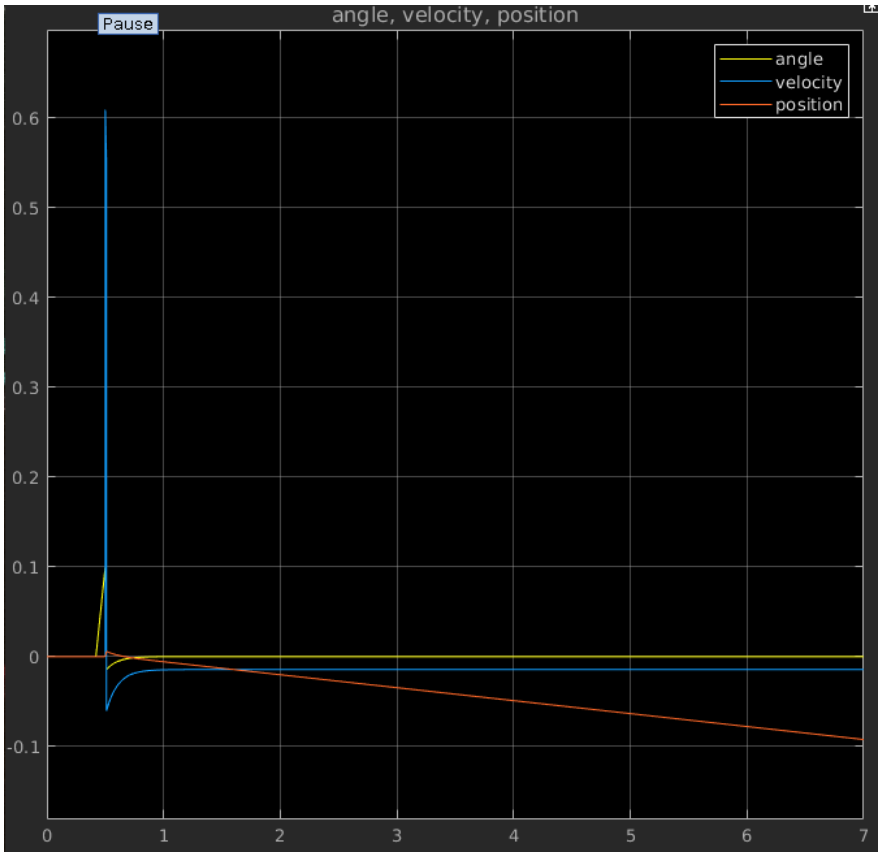


Figure 9: The angle, position, and velocity behavior of our modeled enhanced system. Note the steadily changing position and the steady-state "error" in the velocity.