Rocky The Italian Stallion:
Competitor in the 2018 Robolympics

Amy Phung and Everardo Gonzalez

November 2018

Abstract

In this paper, we review the dynamics, math, controls, and code implementation of controlling a segway robot. The "Athlete Demographics" section runs through our process for modeling our robot as well as creating our controls system. We also reveal the pole diagram of our "Survivor" (making the robot stand in one spot for as long as possible) implementation. In the "Athlete Performance Information" section, we explain how we determined the parameters of our system as well as what our final parameters were. We follow this with an extended analysis and explanation of our implementation beyond making the robot stand. Video evidence of our robot practicing for its events can be found under the "Qualification Video" section. All code can be found in the Github Repository under "Detailed Strategy".

1 Athlete Demographics

Figure 1: The world famous Italian Stallion standing in the halls.

1.1 Our Champion As A Moving Inverted Pendulum

We have to begin by creating an understanding of the system we want to control. For the sake of simplicity, we can model Rocky as an inverted pendulum. We know this is a justified model because Rocky acts similarly to a point mass on the end of a rigid rod. We take a more in depth look at this abstraction in Figure 2.

Knowing this, we can create a free body diagram of the forces acting on Rocky to begin understanding how our system behaves. We know Rocky is being acted on primarily by two external forces: gravity pulling it

Figure 2: Here we can see further how Rocky is abstracted to a simple inverted pendulum.

downwards from its own weight, and a normal force pushing outwards from the center of its pivot. These forces are illustrated in Figure 3.

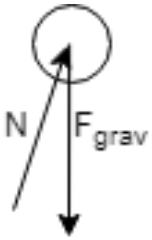

Figure 3: Free Body Diagram of all major forces acting on Rocky.

We can go one step further and illustrate the entire system, including forces and where the different parts of the pendulum are. This is illustrated in Figure 4

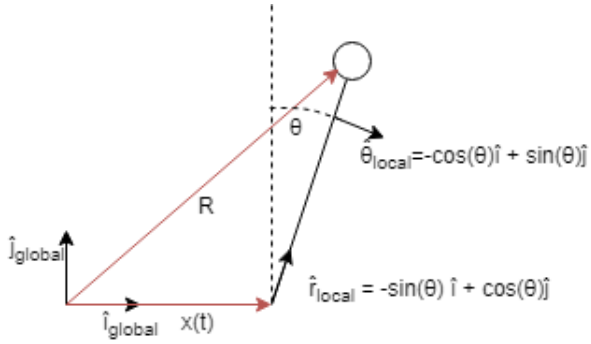

Figure 4: Illustration of the entire system, including all relevant variables that are explained below..

Now we can begin creating a mathematical model of Rocky's behaviour. Since we know the total forces acting on Rocky, we can write:

$$\sum \vec{F}_{\text{total}} = N\hat{r}_{\text{local}} - mg\hat{j}_{\text{global}} \tag{1}$$

where N is the normal force acting along the $\hat{r}$ axis, m is Rocky's mass, g is acceleration due to gravity on Earth, and $\hat{j}$ is perpendicular to $\hat{r}$. So far, we haven't taken into account the fact that Rocky is a *moving* inverted pendulum. This is important for deriving the position of Rocky's center of mass because while the

changing position of the pivot point does not affect what forces act on Rocky, the changing position does affect the position of Rocky. We can write the position of Rocky's center of mass as:

$$\sum \vec{R}_{\text{com}} = x(t)\hat{i}_{\text{global}} - l\hat{r}_{\text{local}} \tag{2}$$

where $\vec{R}_{\text{com}}$ is the position of the point mass, $x(t)$ is the distance of the base of the inverted pendulum from its starting point, and l is the distance from the base to the center of mass. We can take the second derivative of this equation to derive the acceleration of the point mass at any given time. Multiplying this by Rocky's mass gives us the net force acting on the system, which we can set equal to all of the forces acting on Rocky.

$$m\sum \vec{R}''_{\text{com}} = N\hat{r}_{\text{local}} - mg\hat{j}_{\text{global}} \tag{3}$$

We can now solve for Rocky's angular acceleration, which gives us a governing ODE (Ordinary Differential Equation) that explains how Rocky's angle changes as a function of its current angle and base acceleration.

$$\theta''(t) = \cos\theta(t)\frac{(\cos\theta(t))(g(\tan\theta(t)) + x''(t))}{l} \tag{4}$$

From here, we can use the Laplace Transform to turn our ODE into a transfer function that relates the angle of the pendulum to the velocity of its base. The Laplace Transform is an extremely useful tool in controls that allows us to take a time domain ODE and transform it to an s domain transfer function. You can read more about the Laplace Transform here[1]. Our transfer function is shown below in Equation 5

$$\frac{\theta(s)}{v(s)} = \frac{-s}{ls^2 - g} \tag{5}$$

where $\theta(s)$ is the output, $v(s)$ is the input, l is the length of the pendulum, and g is acceleration due to gravity. Equipped with an understanding of Rocky as an inverted pendulum, we can move on to understanding other parts of our system.

1.2 Motors Aren't Magic

In an ideal world, we could simply send a velocity command straight to Rocky's motor to control it. Since we don't live an ideal world, this isn't the case. We have to make sure our model takes into account the relationship between the PWM (Pulse Width Modulation) signal sent to motors and the motors' actual velocity responses. For our purposes, we abstract out the complexities of PWM, and use a value to represent the duty cycle of the signal sent to the motors. You can read more about PWM here.[2]

We know the relationship between our PWM value and velocity can be approximated by a first order ODE. We show this in Equation 6 below:

$$v'(t) = \alpha\beta p(t) - av(t) \tag{6}$$

where $v'(t)$ is the changing velocity, $v(t)$ is the current velocity, and $p(t)$ is the PWM value. α and β are both constants. Utilizing the Laplace transform and some rearranging, we can rewrite this as shown in Equation 7

$$\frac{V(s)}{P(s)} = \frac{\alpha\beta}{s + \alpha} \tag{7}$$

[1] http://mathworld.wolfram.com/LaplaceTransform.html
[2] https://learn.sparkfun.com/tutorials/pulse-width-modulation/all

We were given experimentally derived α and β values based on a best fit curve for the motors' velocity responses. Now that we can accurately represent our system, we can begin controlling it.

1.3 Taking Control

Our implementation involves several control loops working together to keep Rocky stable. So that we aren't overwhelmed trying to understand the entire system at once, we can start by looking at the entire system and analyze it piece by piece. We reveal our entire control system in Figure 5.

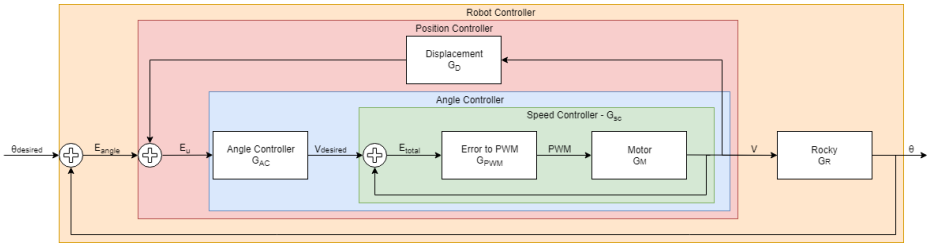

Figure 5: This block diagram illustrates all the complexities of our controlled system as simply as possible. Each block contains a transfer function that directly relates its output to its input.

We've already derived transfer functions for the "Motor" and "Rocky" blocks; we now have to define transfer functions for all of our controllers. We will start with our Speed Controller.

1.3.1 Speed Controller

The Speed Controller makes it so that the motors move at precisely the velocity we want them to move. Our Speed Controller takes a desired velocity as an input, and attempts to output a matching actual velocity. The controller calculates the difference between the desired and actual velocities of each wheel, and feeds that into the "Error to PWM" block. The "Error to PWM" block runs PI control on this error to calculate the PWM value to feed into the motor. We see the transfer function for this below in Equation 8.

$$G_{\text{PWM}} = \frac{E(s)_{\text{total}}}{P(s)} = J_p + \frac{J_i}{s} \tag{8}$$

Where J_p is a proportional control constant, and J_i is a proportional integral control constant. J_p amplifies error in the velocities, and J_i amplifies the accumulated error. Combined, this allows our motors to reach appropriate steady state velocities. With that understanding, we can abstract out what exactly is happening in our Speed Controller and wrap into one big transfer function, with a desired velocity as input, and the actual velocity as output. You see this below in Equation 9

$$G_{\text{SC}} = \frac{G_{\text{PWM}} G_{\text{M}}}{1 + G_{\text{PWM}} G_{\text{M}}} \tag{9}$$

where G_{SC} is the overall transfer function for the controller, G_{PWM} is the transfer function for "Error to PWM", and G_{M} is the transfer function for the motor we defined earlier. We can now abstract this out of our overall transfer function to create a simpler diagram as seen in Figure 6.

1.3.2 Angle Controller

The overall Angle Controller is one of the most important parts of our system. The overall Angle Controller takes in the difference between the error in the angle of the inverted pendulum and the displacement from

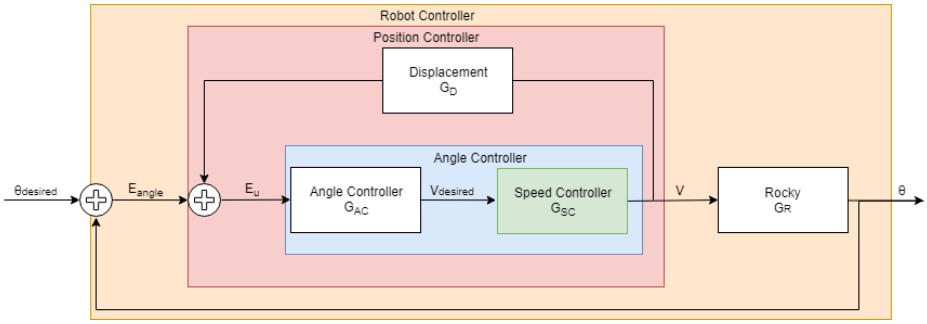

Figure 6: This block diagram illustrates our controller with the complexities of the Speed Controller hidden.

where the base began. The overall Angle Controller takes this difference, E_u and turns it into a desired velocity to feed into the Speed Controller, which outputs the actual velocity. We can see the transfer function for the inner Angle Controller below in Equation 10.

$$G_{\text{AC}} = \frac{E(s)_{\text{u}}}{V(s)_{\text{desired}}} = K_p + \frac{K_i}{s} \tag{10}$$

where K_p is a proportional control constant, and K_i is a proportional integral control constant. This inner Angle Controller makes it possible for our system to take into account both the error from the desired angle required for standing upright as well as how far our Rocky has been displaced from its starting position. We can combine this functionality with the Speed Controller to write an overall transfer function for the overall Angle Controller as seen below in Equation 11.

$$G_{\text{overallAC}} = G_{\text{AC}} G_{\text{SC}} \tag{11}$$

where $G_{\text{overallAC}}$ captures the transfer function for the overall Angle Controller. This lets us abstract the complexities of our overall controller one step further as shown in Figure 7.

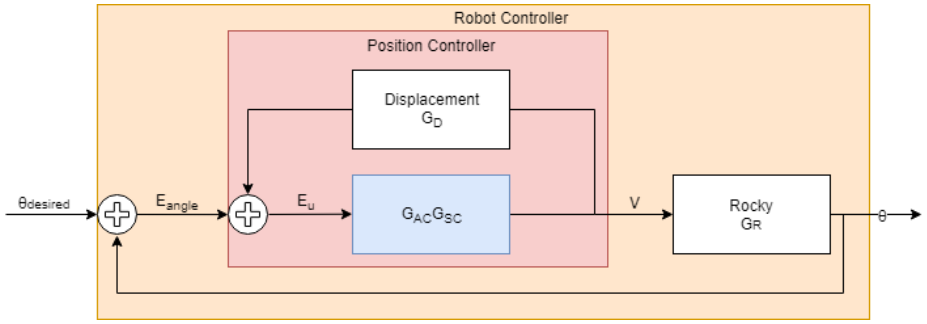

Figure 7: This block diagram goes one step further and hides the complexites of the Overall Angle Controller.

1.3.3 Position Controller

This part of the system is what makes it possible for Rocky to account for drift away from starting position. The "Displacement" block takes the velocity output from the overall Speed Controller and integrates it to calculate Rocky's total displacement from its starting position. We can write this as a transfer function as shown below in Equation 13.

$$G_{\text{D}} = \frac{Z_p}{s} \tag{12}$$

where Z_p is a constant we multiply this displacement by. This gives us control over how much power we want the displacement of the Rocky to have over how the angle changes. We can combine the interaction between this Displacement block and the overall Angle Controller block into a single transfer function as seen below in Equation 13.

$$G_{\text{Pos}} = \frac{G_{\text{overallAC}}}{1 + G_{\text{overallAC}} G_{\text{D}}} \tag{13}$$

We can now visualize our entire system simply as a Robot Controller as seen in Figure 8.

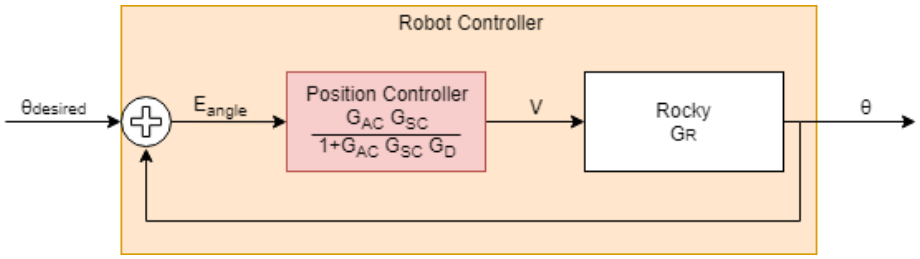

Figure 8: This block diagram takes us further into the abstract and removes the complexites of the entire Position Controller.

1.3.4 Overall Robot Controller

Since we've found ways to abstract out all of the complexities of our system, we can now easily write a transfer function for our Overall Robot Controller. In essence, the Overall Robot Controller calculates the error between our desired angle and the actual angle of the Rocky, and feeds this into our Position Controller. From there, a velocity is sent to Rocky, which in turn affects Rocky's actual angle. We've reached a point in this where we can now write a simple transfer function for the entire system, as shown below in Equation 14.

$$G_{\text{RC}} = \frac{G_{\text{Pos}} G_{\text{R}}}{1 + G_{\text{Pos}} G_{\text{R}}} \tag{14}$$

where G_{RC} is our overall controller. We can also create a simple control diagram that abstracts our entire controller to a single block, as shown in Figure 9.

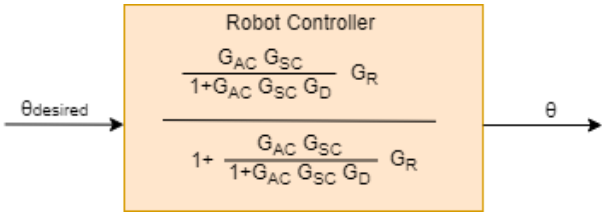

Figure 9: This block diagram illustrates our entire system as simply as possible as a single block.

1.4 Pole Diagram

We used the overall transfer function to determine parameters for all of the abstract constants inside of our controller. Before diving into the specifics of these constants, we are going to take a look at a pole diagram that reflects how our system behaves. We needed parameters that would make our system stable, and we

know whether or not our system will be stable based on our pole diagram. The pole diagram for our system is shown in Figure 10.

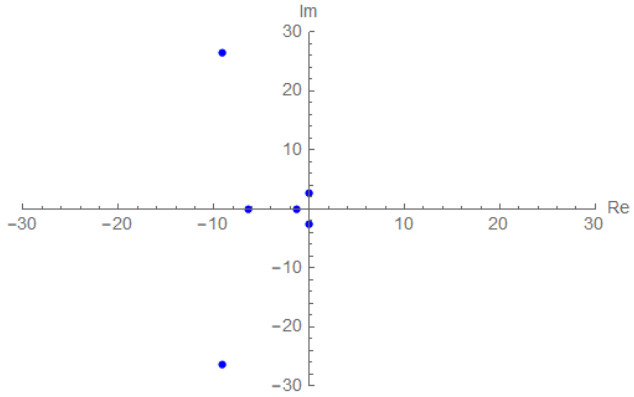

Figure 10: We can see in our pole diagram that our system has no positive poles along the real axis, and that some poles are part real and part imaginary.

We wanted our poles to be as negative as possible while staying true to the constraints of the physical system. For example, we know that the Rocky can't move faster than 0.7 meters per second. The more negative our poles are on the real axis, the faster our system converges to its desired angle. If we had any poles on the positive side of the real axis, we would have an unstable system. We have some poles that are both real and imaginary. The imaginary part of our poles makes it so that the system can overshoot its desired angle and oscillate back to it over time. Overshooting our desired angle does make our system somewhat inefficient. However, this also makes it so that if our desired angle isn't set properly, our system overshoots it and oscillates around it instead of optimizing for an unsustainable desired angle. No matter what we do, we can never calibrate our desired angle to be exactly zero degrees to the vertical, and that makes this oscillation extremely important.

2 Athlete Performance Information

Now that we have the transfer function for the overall system, we can use this to find values for the constants that would make the system stable. By definition, the overall system is stable when the real component of the poles (i.e. the roots of the transfer function of the system) are all negative. We can use a computational tool like Mathematica to sweep through different values for each of the constants and visualize them to see which values work best. We have an example shown in Figure 11.

Theoretically, constants that lead to negative poles with large magnitudes would be immensely stable since they decrease the amount of time it takes for the system to reach it's steady-state. In practice, the system is limited by the power and response time of the motors, which limits the usable range of constants. However, in our analysis thus far, we have not taken into account characteristics of the platform including things like the max speed of the motors, the max PWM signals, or the momentum of the robot. Without taking this into account, our tool for sweeping values can find an infinite number of constants that appear to be stable but might not be physically feasible. We can see this explored further in Figure 12.

To narrow down the possible range of constants that will work, we need to first provide reasonable approximations for the magnitude and accuracy of each of the constants.

2.1 Estimating Reasonable Constants

Notice that K_p and K_i relate the error in angle to velocity. The usage of these constants in our implementation is show below in Equation 15.

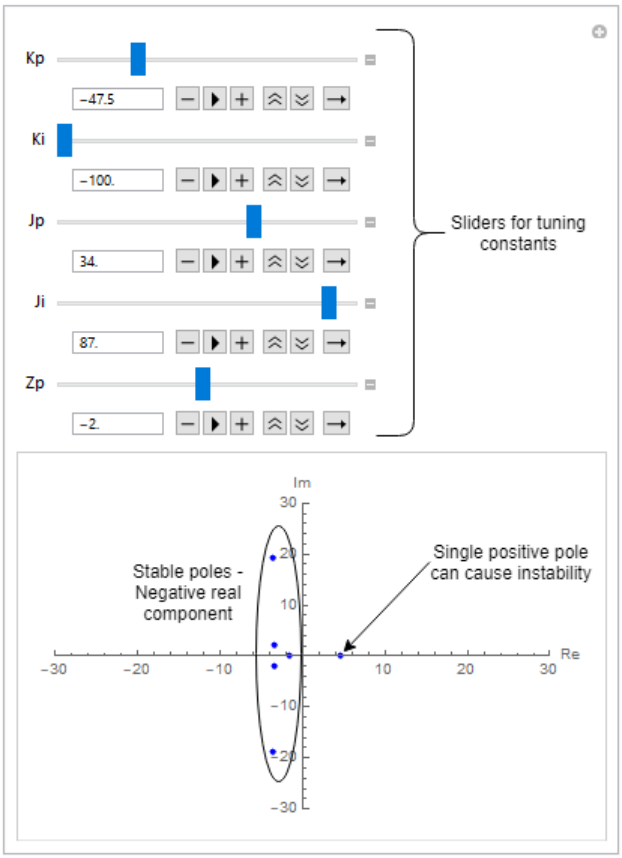

Figure 11: Mathematica tool can be used for quickly visualizing poles with particular constants

$$V_{\text{desired}} = K_{\text{p}} * E_{\text{angle}} + K_{\text{i}} * AccumulatedAngleError \tag{15}$$

For now, let's ignore the accumulated angle error since it contributes less to the calculation of V_{desired} and we're currently just approximating. Now we have Equation 16.

$$V_{\text{desired}} \approx K_{\text{p}} * E_{\text{angle}} \tag{16}$$

We expect the maximum angle the robot will be able to recover from to be around 10 degrees, which is approximately 0.2 radians. The maximum speed the robot can travel at is approximately 0.7 meters per second, so if we substitute those values for E_{angle} and V_{desired} we get Equation 17.

$$0.7 \approx K_{\text{p}} * 0.2 \tag{17}$$

So our result here is Equation 18.

$$K_{\text{p}} \approx 0.35 \tag{18}$$

We can repeat this process for J_{p} and Z_{p} and choose to sweep K_{i} and J_{i} later.

Using this same process:

- J_{p} relates V_{desired} to PWM, which are at most 0.7 meters per second and 300 respectively. Therefore, we expect $J_{\text{p}} \approx 400$.

- Z_{p} relates P_{desired} (desired position) to V_{desired}, which are at most around 1 meter and 0.7 meters per second respectively. Therefore, we expect $Z_{\mathrm{p}} \approx 0.7$.

Now that we know the expected values of each of the constants, we can put these into our Mathematica tool and sweep only K_{i} and J_{i}, which is a lot easier than sweeping through all 5 values. Once we found values that appeared to be stable and similar to our estimates, we tested them on the robot and took note of their behavior. If the robot responded violently, then we decreased the constants related to the violent behavior. If the robots were slow to respond, we increased the constants. Our experiments yielded the values shown in Table 1.

Kp	Proportional Constant for Angle Control	-6.78
Ki	Integral Constant for Angle Control	-50
Jp	Proportional Constant for Speed Control	314
Ji	Integral Constant for Speed Control	800
Zp	Proportional Constant for Position Control	0.86

Table 1: Final Constants Used To Make Robot Stand

We can see that the magnitude of these values are similar to our estimates.

3 Training Session Description

Our original Rocky code made it possible for Rocky to stand up, accelerate for a short distance, and flop pathetically. This control system simply would not stand for any event, so we decided to improve our control system in various ways.

3.1 Extensions

Though our focus so far has been an in-depth analysis of the "Survivor" event, this analysis could be extended in various ways to attempt different events. Some of our successful extended behaviors include:

- Velocity control - Implemented using a speed controller (See Section 1.3.1 for details)
- Position control - Implemented using a position controller (See Section 1.3.3 for details)
- Independent wheel velocity control - Implemented by incorporating separate desired velocities for each wheel
- Choosing constants that made robot resistant to external forces - Implemented by decreasing magnitudes of proportional constants, which resulted in a robust system that was slightly more under-damped

These behaviors allowed us to participate in these additional events:

- Survivor (robot stands as long as possible)
  - Used position and velocity control
- Sprint (robot drives as fast as possible without falling over)
  - Used velocity control
- Spin (robot spins as many times as possible in one minute)
  - Used independent wheel velocity control
- Battle Bots (robot attempts to be the last one standing)

  - Used improved constants
- Bridge of Death (robot drives along a parametric curve)
  - Used independent wheel velocity control

3.2 Process - The Good, The Bad, and The Ugly

In general, our process involved making block diagrams to fully understand the system, estimating the constants for control system, then using those estimates in a tool like Mathematica to fine-tune those values before testing. We also developed our code incrementally, making sure each portion of it worked before implementing more complex controllers/behaviors.

3.2.1 The Good

- Understanding the system well before resorting to "guess-and-check" methods on the robot
- Put learning first - got a solid understanding of dynamics, controls, Mathematica, and code implementation of this project!
- Taking breaks - featuring boomerangs and RC sailboats!

3.2.2 The Bad

- Asking for help - We thought we knew what our errors were but ended up spending a combined 22 hours in one day working on the robot while this one error was stumping progress. Was resolved in 5 minutes after asking Paul (thanks Paul, you're a lifesaver!)

3.2.3 The Ugly

- Our consistency with naming and defining variables in our code made it difficult to read at times, making implementation difficult.
- Our GitHub repository was pretty messy for most of the project. Doing a better job with version control would have made it easier for us to to keep track of what code corresponded to what control system.

4 Qualification Videos

Here we have videos of our Rocky practicing for all of its events!

- Survivor - https://www.youtube.com/watch?v=Lp10CplbuaEfeature=youtu.be
- Sprint - https://youtu.be/p1YsiRUwfDM
- Spin - https://youtu.be/5j-c7CFjFqE
- Battle Bots - (same code as Survivor) https://www.youtube.com/watch?v=Lp10CplbuaEfeature=youtu.be
- Bridge of Death - https://photos.app.goo.gl/fkHDGmiQr6ArEZtM6

5 Detailed Strategy

Code is available online here: https://github.com/AmyPhung/QEA-Rocky

Code Structure:

- Code - Where the code is stored
  - Archive - All the code that we wrote
    - Good Backups - Backups for code that works but is missing documentation
    - Sandbox - Test folder for trying new code
  - Events - Where the final code is stored
- Deliverables - Documents to turn in for class
  - DiagramsPNG - PNG files for diagrams
  - DiagramsXML - XML files for diagrams in paper
- Mathematica Notebooks - Notebooks for computing
  - Final Calculations - model we ended up using
  - Sandbox - Folder for trying new models

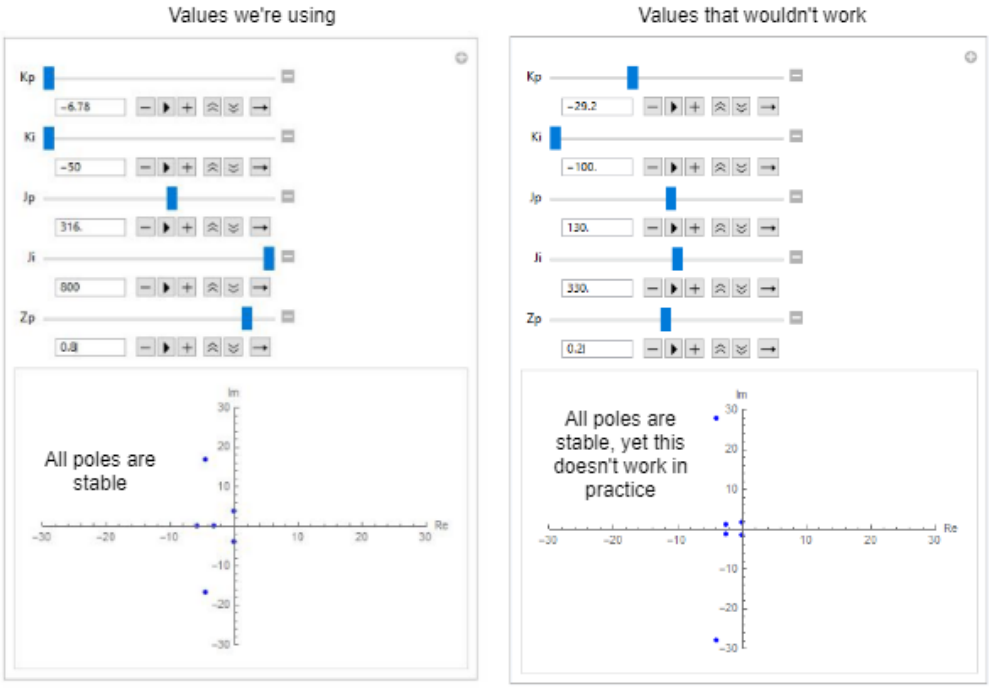

Figure 12: There are an infinite range of values that would make our system theoretically stable but wouldn't work in practice. Notice here that only a single positive pole can cause instability within our system.