Inverted PIDulum

Carl Moser and Serena Chen

December 15, 2016

Introduction

PID control stands for *Proportional, Integral, Derivative* control. PID is a very common control loop, often used to control joints, temperature, and various precise positioning systems. In this project, we hope to use a PID controller to control the angle on an inverted pendulum, applying torque to our simulated pendulum to hopefully get it to balance in the upright, inverted state.

PID works in the example of joint control (which we are doing in our project) by having three constants: a proportional constant ($pconst$), a derivative constant ($dconst$), and an integral constant ($iconst$). We first have to abstract the state of the system into one "error" term. In the case of joint control, it is the difference between the joint's current position and the joint's desired position. That is the feedback part of the loop. Then we pass it through the control part. First, determine the following:

- **The proportional term** or $pterm$
  - the current error (which should be given)
- **The derivative term** or $dterm$
  - the change in error (the error from the last loop iteration minus the current error)
- **The integral term** or $iterm$
  - the summation of all the errors that have been passed into the loop thus far

Using these terms and the above constants, solve:
$(pconst) * (pterm) + (dconst) * (dterm) + (iconst) * (iterm)$

The result of the above equation should be the additional torque or angular acceleration to be applied to the joint at that time step.

Manual PID training often involves a lot of trial and error on the part of the programmer. There are algorithms to train PID controllers. Our approach is one that applies a little bit of machine learning. Our basic procedure is to test many random PID constants and record their results, do a regression on the results, and then perform gradient descent on the regression to find the set of constants that will most quickly and effectively move the pendulum to its inverted state.

Code Walkthrough

We decided to write our code in Java, since we wanted to visualize the pendulum (Java has a nice GUI library). We anticipated a large code architecture and Java makes organizing software architecture very easy.

Pendulum

To simulate the pendulum, we kept track of the coordinates of its pivot as well as the coordinates of the center of its weight. We used a Lagrangian to model the physics of the pendulum[1].

PID

For a PID controller, we created a simple class that stored *pconst*, *dconst*, and *iconst*, as well as a few other constants that were essential to the PID function, such as *ibound*, which serves as the upper and lower bounds of the *iterm*, to prevent over-saturation of the *iterm*. The *getCorrection()* method solves the equation outlined in the introduction, and returns the externally induced acceleration, in this specific case[2].

PID Tuner

We created yet another class to train the PID controller. First we simulated multiple random points for the *pterm*, *dterm*, *iterm*, and *ibound* constants, for 5 seconds each. The score for an individual set of points is determined by adding up the absolute values of all the errors for each time step, with a lower score being better (since it, on average, was closer to the stable position of the inverted pendulum). The ranges of *pterm*, *dterm*, and *iterm* were experimentally determined, by starting with a large range one constant at a time, writing the constants and the resulting error into a file, and using Python's *MatPlotLib* to compare the constant's value vs. the resulting error. By testing, we were able to continually lower our ranges for *pterm*, *dterm*, and *iterm* to [0, 1.5] for both *pterm* and *dterm*, and [0.0, 0.0005] for *iterm*.

[1]See our code for the pendulum model here: `https://github.com/cmoser96/PIDTuner/blob/master/src/InvertedPendulum.java`

[2]See our code for PID here: `https://github.com/cmoser96/PIDTuner/blob/master/src/PID.java`

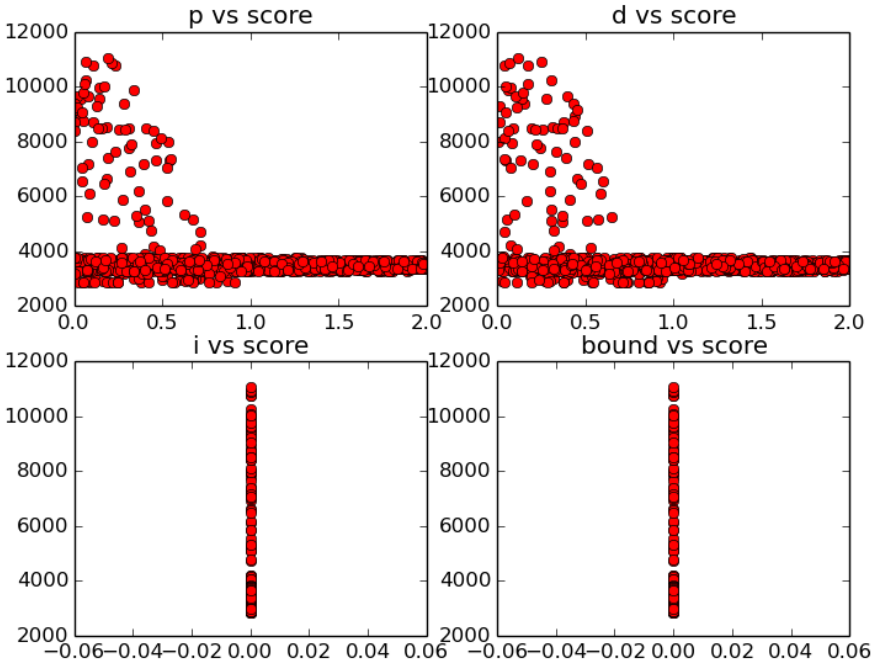

Figure 1: With *iconst* and *ibound* set to 0, you can easily see the ranges of *pconst* and *dconst* that are relevant

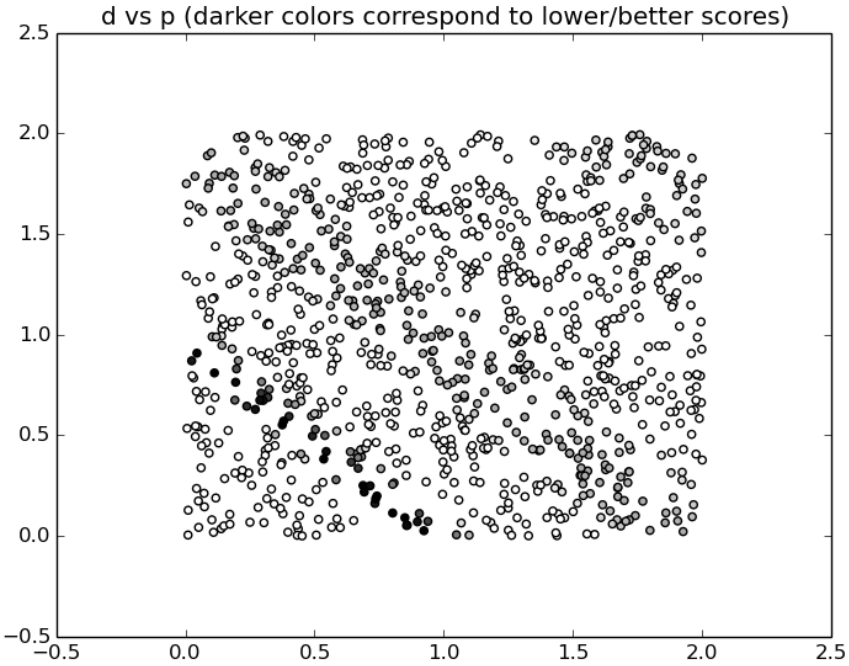

Figure 2: Showing the correlation between *dconst* and *pconst*

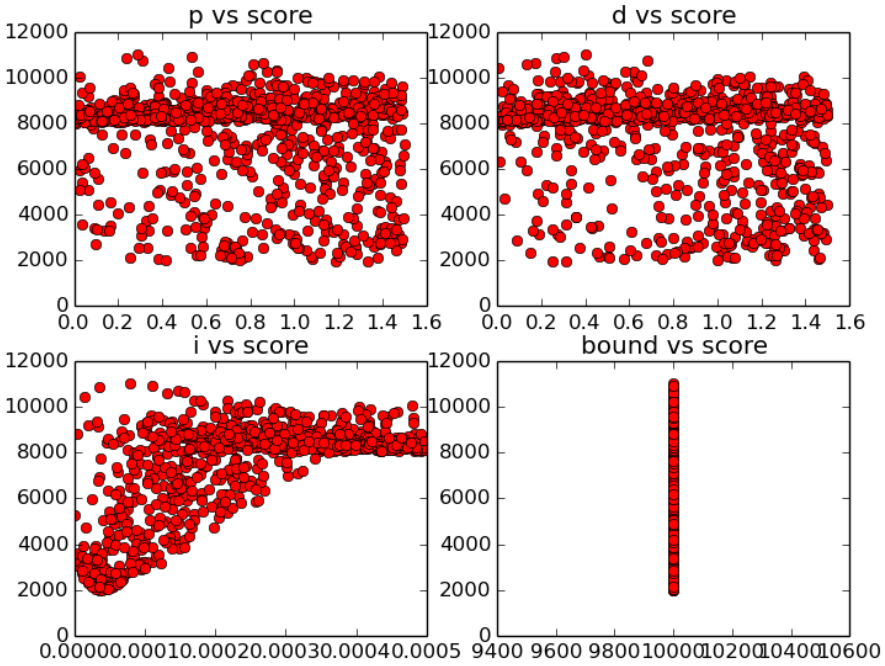

Figure 3: Showing the correlation of *iconst* and the error/score of the system

We then took the data points from the random tests, and did a regression to the third order. We used Michael Flanagan's library[3] to do multiple regression. Then, using the regression coefficients, we performed gradient descent on the function to find the constants at which the score was the lowest.

Results

YouTube video of 40 simulations (in succession) of training the PID, and then running through the optimized solution: `https://youtu.be/QWLUMeKPONc`.

Each time the simulation is run, the test *pconst*, *dconst*, and *iconst* are randomly generated, as well as the starting *pconst*, *dconst*, and *iconst* for the gradient descent. This is why the results of each run vary so wildly. In general, there are two kinds of results for the gradient descent: one in which the pendulum oscillates around or near the unstable equilibrium (inverted) position, and one in which the pendulum accelerates in a circle infinitely.

Running the simulation 100 times and recording the output, we can analyze a little more closely what is going on[4]. For our purposes, we defined a *stability* characteristic, where a stable pendulum is one in which the pendulum oscillates, and an unstable pendulum is one in which the pendulum

[3]Library here: `http://www.ee.ucl.ac.uk/~mflanaga/java/`

[4]Raw data here: `https://docs.google.com/document/d/1vDe5IeXX2opcWtOBDCifznroUp_qnUo3f4koUPY5GmY/edit?usp=sharing`

accelerates in one direction infinitely. We defined these terms as such, since the stable pendulum will attempt to correct itself, whereas the unstable pendulum will stray farther and farther from the desired position. In our 100 simulations, we discovered that 50 of the pendulums had stable movement, while the other 50 had unstable movement.

Of the 50 pendulums that were unstable, 2 of them had timed out of the gradient descent, 43 of them failed due to a problem with either the gradient descent or the regression, which caused *pconst*, *dconst*, and *iconst* to be ridiculously large numbers, and the score to be NaN (not a number), which is apparently treated as a 0 in Java when comparing numbers. Either the step size is too large, or the regression has very steep slopes. The last 5 pendulum simulations were unstable for reasons specific to the PID constants themselves.

Of the pendulums that were *stable*, there were a few whose oscillations that did not cross the unstable equilibrium position and almost none of them were perfectly centered about the unstable equilibrium. This is an empirical observation on the 100 pendulum simulations, as we did not record data on the pendulums' oscillation positions. Our hypothesis is that the asymmetry is due to the *iterm*, since the value of *iterm* would be 0 at the start, but as it moves, the *iterm* would change, affecting the acceleration and velocity of the pendulum in perhaps an unbalanced way. Thus, one hypothesis is that PIDs with larger *iterm*s would stray farther from the unstable equilibrium. Another, similar hypothesis is that *ibound* bounds *iterm*, thus causing an asymmetrical effect on angular acceleration.

Conclusion

Although we were able to generate PID constants that would oscillate about the unstable equilibrium, those simulations were few and far between, due to many factors such as regression inaccuracies, step size with gradient descent, and possible kinks to work out with the integral term. We have learned a lot about patterns in PID and experimenting with gradient descent when there are so many variables involved.