PID Controller

Brent Wylie, August 29th, 2017

The goals / steps of this project are the following:

- Create a PID controller in C++ using class structures
- Make a successful lap around the track using the PID controlled steering, without leaving the track surface
- Understand and explain how each hyper parameter affects the behavior of the controlled system itself

Implementation

I built one PID controller for controlling steering angle in main.cpp. The definition of the PID controller is in the code in PID.cpp. I did not make any changes to PID.h or the JSON.h file. The PID is initialized with a Kp, Kd, and Ki coefficients with PID::Init(), to set the proportional, derivative and integral component gains respectively. The main program interacted with the simulator environment, receiving track error, observed vehicle speed, and observed steering angle in the form of a JSON object. This simulated reading actual sensor measurements and allowed me to focus on using the PID itself.

The overall idea of the process is to take in measurements/observations, update the errors, and issue a steering value to correct the angle of the car and return to the center of the lane. I accomplish the first step, the error update, with PID:UpdateError(), which takes the track error value and breaks it into P, I and D components and stores that state in the PID object. The P component is just the error itself: “how far off am i?”, the derivative component is the difference between the current error and the previous amount of error (which is conveniently the past proportional component), the integral component of the error is an accumulation of past error measurements. Note that I provide an upper and lower bound for the integral component, to avoid it dominating the response of the system. This is a common problem in PI and PID controllers, called “integrator windup”, where an error builds up so significantly that the integral component of the controller vastly overshoots the correction. My car was actually falling off the on subsequent laps before I made this fix. This fix ensured my car could go around the track more than once without failing.

Once this process is finished, I total up the errors and component gains in PID::TotalError(), which outputs a steering value needed to minimize/eliminate the error. The steering value is then sent to the simulator for the car to take the corrective action. In my main program, I effectively established an upper limit to the speed at +35Mph, as faster speeds are more unpredictable due to the not necessarily deterministic time of socket based communication between the simulator and the PID controller. In a real application this controller would probably run as a periodic ISR.

I chose each of the Kp, Ki and Kd gains manually, starting with all at 0. I increased the P gain until there was a steady oscillation on the track, then I increased the D term until the system became critically damped and these stopped. I repeated with just these two terms until I saw the error correction to be pretty good. I brought in the I term to help it respond quicker. Usually a PD is good enough for these sorts of applications, but I wanted it to quickly recover and make a true PID controller. Plus dealing with the integrator windup was a fun educational experience with no consequence in a simulated environment.

In conclusion, here is a video of a PID based robot I built in college a few years ago: It measures distance to the walls with angled IR sensors and tries to minimize the difference between the two readings. It was programmed in C.
https://www.youtube.com/watch?v=ocOtLYbkpjw