IEEE Micromouse Spring 2018

Lab 6: Controls II

- Introduction
- Expanding on Proportional Control
- Arduino PID Library
- Wall Following

Introduction

Our mice should be driving reasonably straight using only proportional control. This week, we will learn how to correct for long-term error, use the Arduino PID library, and have our mouse follow walls.

Expanding on Proportional Control

Recall our implementation of a **proportional controller** from last week's lab:

$$error(t) = setpoint(t) - input(t)$$

$$u(t) = K_p \cdot error(t)$$

$$\texttt{applyPowerLeft}(u_{linear}(t) + u_{angular}(t))$$
$$\texttt{applyPowerRight}(u_{linear}(t) - u_{angular}(t))$$

Proportional control works pretty well for making our mouse drive straight, but it doesn't account for long-term drift. Even if our mouse manages to correct most of the error every timestep, there's still a little bit of uncorrected error that builds up. What if our mouse kept track of the total error so far?

$$u(t) = K_p \cdot error(t) + K_i \cdot \int_0^t error(t')dt'$$

This is known as a **proportional-integral (PI)** controller. The integral sums up the total error since we started the controller. As small errors build up, they'll cause the controller to apply a stronger correction to u(t) and get our mouse back on track.

It might also help to add a term that's proportional to the derivative of error:

$$u(t) = K_p \cdot error(t) + K_i \cdot \int_0^t error(t')dt' + K_d \cdot \frac{d}{dt}error(t)$$

The derivative of error represents how fast the error is changing. If the error is increasing rapidly, we might want to apply a stronger control input u(t). Likewise, we might want to reduce the amount of u(t) we apply if the error is approaching zero quickly.

When we combine these three terms, we get a **proportional-integral-derivative (PID)** controller.

Kp, Ki and Kd are tunable **gains** that determine how much each term affects the final output power.

Let's break down the terms from left to right:

- The **P** term (**proportional** term) increases our correction proportionally to the error: the higher our error, the more power we should apply to correct for it. This is generally considered the most important term.
- The **I** term (**integral** term) compensates for long-term error & drift by integrating the error over time. While a small tendency for our mouse to veer to the right might have little effect on the P term, it'll cause the I term to build up until the system reaches the setpoint perfectly.
- The **D** term (**derivative** term) takes the derivative of the error with respect to time. Systems often have inertia: the faster our error is already decreasing, the less power we need to apply to correct for it. This dampens our controller and decreases overshooting.

Checkoff #1

1. We only have individual measurements of the error, and not a continuous function. How could we estimate the integral of the error? (You don't need to write code for this part.)
2. How could we estimate the derivative of the error? (You don't need to write code for this part.)
3. One potential issue with PID control is **integral windup**. You can read about it on Wikipedia. What are some ways of preventing integral windup?

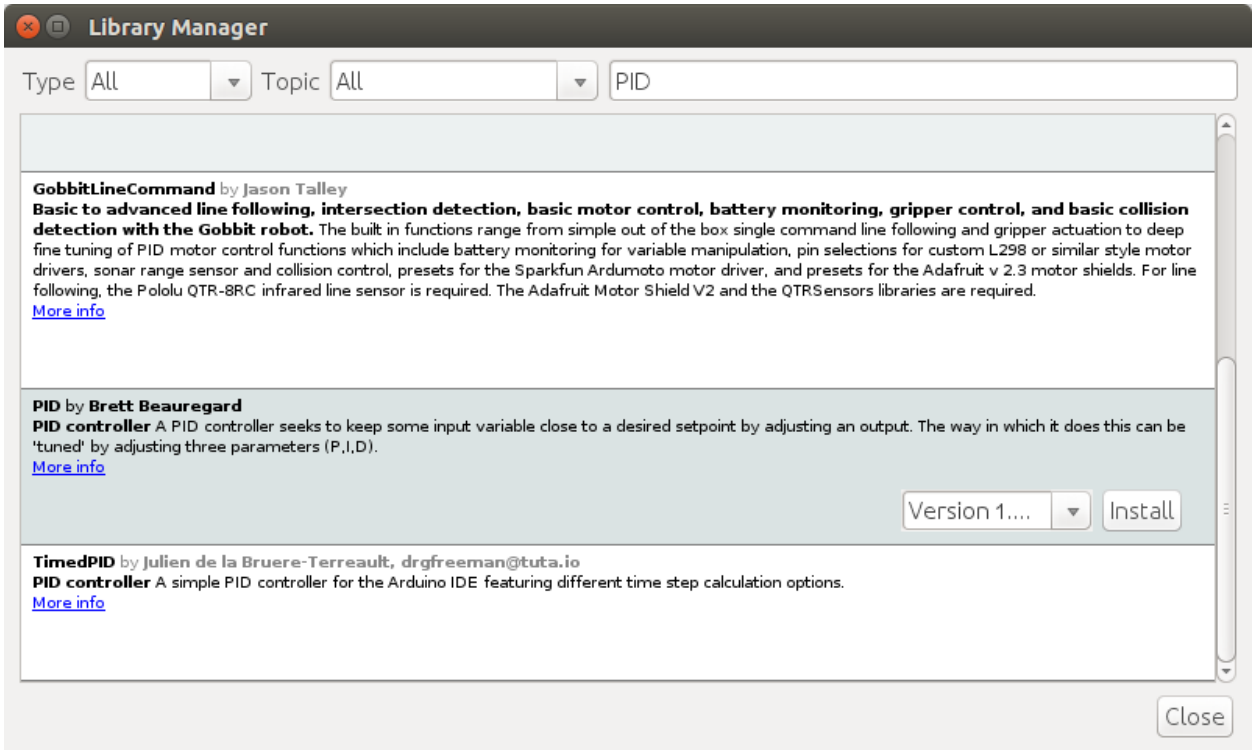
Arduino PID Library

Implementing a proportional controller manually like we did last week is fairly straightforward. Adding integral and derivative terms gets messier, especially as you add more and more PID loops. There are also some potential instabilities to watch out for, especially with the integral term.

Luckily, there's an Arduino library that we can use to simplify this process! Libraries are basically pre-made code that somebody decided to share with us. We will be using Brett Beauregard's PID library, which takes many of the potential issues into account.

You can search for and install this through Arduino's library manager:

Sketch > Include Library > Manage Libraries



Here's an example of how you might use the library!
Note that **this code is not intended to run**, it is only meant for illustration purposes.

```
#include <PID_v1.h>

// Supporting variables
double velocity_linear_setpoint;
double velocity_linear;
double velocity_linear_power;

// Specify the links and PID tuning parameters
// Here, Kp = 0.006, Ki = 0.0005, and Kd = 0
PID velocity_linear_pid(&velocity_linear, &velocity_linear_power,
&velocity_linear_setpoint, 0.006, 0.0005, 0, DIRECT);

void setup()
{
  velocity_linear = 0;
  velocity_linear_setpoint = 50.0; // We want to go 50 mm/sec
  velocity_linear_pid.SetOutputLimits(-1.0, 1.0); //
  velocity_linear_pid.SetSampleTime(10); // Run control loop at 100Hz
(10ms period)

  // Turn the PID loop on
  velocity_linear_pid.SetMode(AUTOMATIC);

  [other setup code]
}

void loop()
{
  velocity_linear = getLinearVelocity();
  velocity_linear_pid.Compute();

  applyPowerLeft(velocity_linear_power);
  applyPowerRight(velocity_linear_power);
}
```

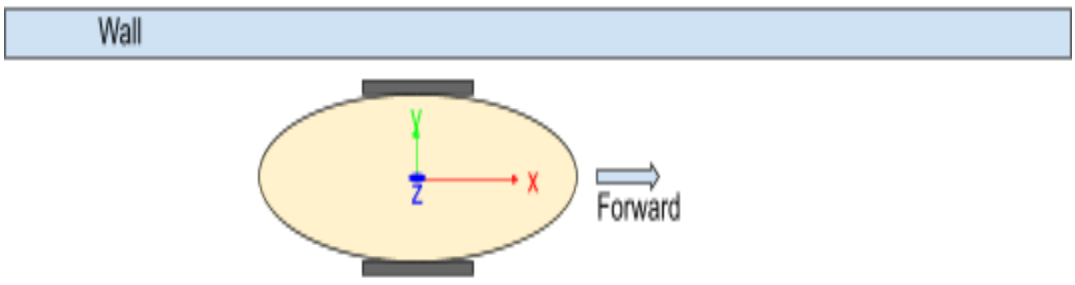
You can also find all the documentation for the library here:
https://playground.arduino.cc/Code/PIDLibrary

Checkoff #2

1. Re-implement the linear and angular velocity controllers from Lab 5 using the PID library. Leave the Ki and Kd parameters at 0.0 for now.
    a. Some gains to get you started:
        i. Angular Kp = 0.4
        ii. Linear Kp = 0.004
2. Now try adding the Ki parameter[1].
    a. Some gains to get you started:
        i. Angular Ki = 0.05
        ii. Linear Ki = 0.0005
    b. What effect does increasing Ki have? What happens if Ki is too large?

---

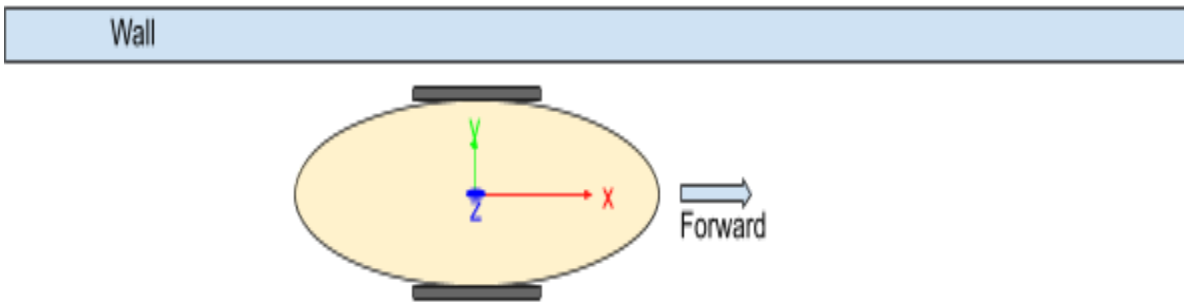
3. To avoid collisions while navigating through a maze, we'll often want to do "wall following" -- that is, driving forward while maintaining a fixed distance from a wall parallel to our movement. Imagine writing a PID loop used to maintain this distance:
    a. What might the input to our control loop be?
        i. Hint: What sensors do we have available for this?
    b. What might the setpoint be?
        i. Hint: we want to center our mouse in any corridors it drives down.
    c. What might the output be?
        i. Hint 1: If we're too close to a wall, how do we get further away from it?
        ii. Hint 2: It's common to have nested controllers, where the output of one controller is the setpoint of another.



[1] We won't be using the derivative term, since the inertia of our system is relatively low.

Wall Following

No matter how well our mouse can drive straight, there will always be some drift. Since our mouse will be driving through a maze with straight walls, we can use the walls as a reference to correct any remaining error.



Mouse following a wall to its left.

Checkoff #3

1. **Implement the wall-following controller that you designed in checkoff #2.**
2. **How would you deal with gaps in the walls, or only one wall being present?**
    a. Bonus: Implement this!