IEEE Micromouse Spring 2018

Lab 5: Controls I

Introduction

Last time, we learned how to determine how fast our mouse is driving and how fast it's turning. This week, we'll use these values to make our mouse drive straight!

Setup

Start by picking up a battery and downloading the skeleton code, which includes all of the functionality we've implemented in our previous labs.

The directory structure should look like:

```
mouse/
    mouse.ino
    mouse_helpers.ino
    pins.h
```

By default, the code will apply 20% power to each motor:

```
////////////////////////////////////
// Your changes should start here //
////////////////////////////////////
float left_power = 0.2;
float right_power = 0.2;

applyPowerLeft(left_power);
applyPowerRight(right_power);
```

`applyPowerLeft()` and `applyPowerRight()` should turn their corresponding motors at full power forwards when power is `1.0` and full power backwards when power is `-1.0`.

Checkoff #1

1. Run the skeleton code.
    a. Verify that each wheel is spinning in the forwards direction.
        i. Flip the INVERT_MOTOR_LEFT and INVERT_MOTOR_RIGHT flags if needed
    b. Verify that the measured linear velocity is positive.
        i. Flip the INVERT_ENCODER_LEFT and INVERT_ENCODER_RIGHT flags if needed

Driving Straight

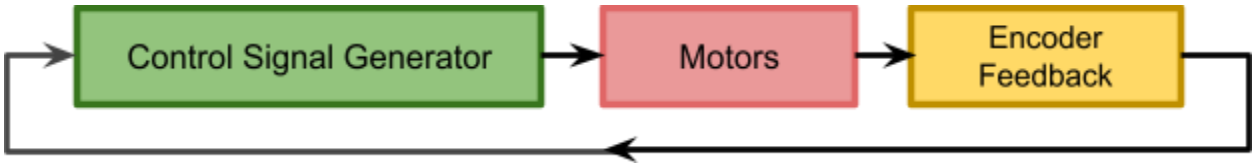
The current driving implementation in the skeleton code is an example of **open loop control**, which uses knowledge of the system (our mouse) to estimate the required control input (motor voltage). We know that applying the same voltage to both motors will make them spin in the same direction at roughly the same speed.

This is great! That is, of course, under the assumption that we are in a vacuum with no air resistance. And the mouse is perfectly balanced. And the ground is perfectly flat, with no defects. And our motors are exactly the same, to the atom.

Unfortunately, encasing the room in a vacuum violates the competition specs, so we will have to find some other way to deal with these problems.

In reality, **applying the same voltage to each motor will not give us identical motor behaviors.** The mouse will veer to one side.

To fix this, we can use something called **closed-loop control** or **feedback control**, which utilizes readings from the encoders to correct the amount of power we apply to the motors.



The encoder readings can tell us how far we are from driving straight, and the feedback control corrects the error by, for instance, applying extra power to the slower motor and reducing power to the fast one.

Proportional Control for Driving Straight

One common way to do feedback control is with a proportional (P) controller. P controllers try to minimize an **error** — the difference between a measured value (**input**) and its desired value (**setpoint**).

This is accomplished by increasing a correction power *proportionally* to the error: the higher our error, the more power we should apply to correct for it.

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

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

Let's try driving forward at 20% power again.
This time, however, we'll apply a proportional correction value *u* as shown:

$$\texttt{applyPowerLeft}(0.2 + u_{angular}(t))$$
$$\texttt{applyPowerRight}(0.2 - u_{angular}(t))$$

...where *Kp* is a gain that scales our corrective power. There are ways to calculate the right value of Kp, but guessing and checking also works.

Checkoff #2

1. If we use our mouse's angular velocity as the input to a control loop that tries to make our mouse drive **straight**, what should the setpoint be?
2. If the error is an angular velocity and *u(t)* is a duty cycle, what units is Kp in?
3. If we want to our applied correction power *u(t)* to be maximized (ie 1.0) when the error is 2 radians/second, what should Kp be?
4. Implement the angular velocity P controller described above.
5. Try a few different Kp values. Some values to start with: 0.1, 0.5, 5
    a. What do you think (or see) happens when Kp is too low? Why?
    b. What do you think (or see) happens when Kp is too high? Why?
    c. What seems to be a good value for Kp?
6. Plot your angular velocity error. Does it converge to zero?

Linear Velocity Control

Instead of blindly adding 0.2 to our wheel duty cycles to command our mouse to drive forward, we can add a controller for our linear velocity as well:

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

This allows our mouse to drive forwards and backwards as well as stop. Being able to control the speed of our mouse is really useful. Just as imbalances in the motors can prevent our mouse from driving straight, they can also cause our mouse to drift when turning in place.

Checkoff #3

1. What should the input and setpoint be for our linear velocity controller?
2. If the error is a linear velocity and *u(t)* is a duty cycle, what units is Kp in?
3. If we want to our applied correction power *u(t)* to be maximized (ie 1.0) when the error is 140 mm/second, what should Kp be?
4. Implement the linear velocity P controller described above.
    a. We recommend that you start by disabling the angular velocity control -- get them both working separately before combining them.
5. Try a few different Kp values. Some values to start with: 0.001, 0.01, 0.1
    a. What do you think (or see) happens when Kp is too low? Why?
    b. What do you think (or see) happens when Kp is too high? Why?
    c. What seems to be a good value for Kp?
6. Plot your linear velocity error. Does it converge to zero?

(Bonus) Integral Control

Recall our implementation of a **proportional controller**:

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

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

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

Proportional control works better than nothing 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 we could have our mouse keep track of the total error so far?

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

The integral sums up the total error since we started the controller. As the error builds up, the controller will apply a stronger correction to u(t) to get our mouse back on track. This is known as a **proportional-integral (PI)** controller.

Checkoff #4

1. Add an integral term to the your angular velocity controller.
    a. We recommend doing so with the Arduino PID library.
    b. Does your error converge to zero?
2. Add an integral term to your linear velocity controller.
    a. Does your error converge to zero?