Homework 6: Velocity Control

Due: October 23, 2014

You must hand in your code on Owlspace, bring a printout to class, and hand in your written report in class. All of your code must be in a single python file called PS06_*netid*.py. Be sure to comment your code. so that the graders will know what you are trying to do. You can't do this in one night. Start now! You will need to demonstrate your velocity controller in class.

Velocity Control

For this assignment you will implement a velocity controller for your robot's motors. For the first part of this assignment, we will characterize the motors on the robot, and write a *open loop* controller to move a fixed distance. This won't work very well, but that's ok, because in the second part of the assignment, you will write a *closed loop* controller to keep the velocity of each wheel close to a specified value. This should give you much better results.

The key to a closed-loop controller is the *feedback* from the sensor to the computer, then the *control* back to the motors. By measuring the actual velocity and comparing that to the desired velocity, you can adjust the motor PWM and make the robot move very close to the desired velocity.

1 Open-Loop Controller

In this section you will build and measure an open-loop motor controller. An open-loop controller does not use sensor feedback to control the motors. Since by now you must realize that feedback is critical in all engineering systems, you might imagine that an open-loop controller won't work very well. You would be right. Let's measure it. Remember, put your robot on its block while you are working on the code, and run it on the ground, not your desk, for tests.

1.1 Characterize the motors

We know the motors don't move when given low PWM values, so we need to characterize them to see exactly how they perform. Your job is to complete the `characterize_motors()` function. This function will ramp the PWM from 0-100 for each wheel and measure the resulting wheel velocity. Every second, print the current PWM, the elapsed encoder ticks, and velocity. Note that the upper bound in the Python `range()` function is not inclusive, so you will need to use `for pwm in range(0, 101, 10):`

First, you will need two helper functions for this: `compute_distance()` and `compute_velocity()`. These functions use encoder values. The encoders can only count between 0 and 65,535. So, if the count is increasing, when it would have hit 65,536, it instead wraps to 0. Similarly, if the count is decreasing, when it would have hit -1, it wraps to 65,535. We have provided a function, `encoder_delta_ticks` for you that takes a `new` and an `old` encoder value and returns the difference, accounting for this wrapping behavior. [1] Note that your robot will not actually travel at these velocities when it is on the ground, because friction and other effects will slow it down. However, this is not important, as the feedback controller will correct these errors. Be careful not to divide-by-zero in `compute_velocity()`. **Hand-in: Write `compute_distance()` and `compute_velocity()` (2 pts each).**

[1] This function will only return the correct value if the distance the robot traveled between encoder readings was around 1.5 meters or less.

1.2 Plot and Analyze the data

Finish the `characterize_motors()` function and use it to measure velocity vs. PWM for both motors. This is nice, but not directly useful for our controller, because when we are driving our robot, we want to specify velocity and determine the PWM value to set the motors to. In order to do this, we need the *inverse* of the data we just collected. This is far easier than it sounds, just switch the axis and re-plot. **Hand-in: Write `characterize_motors()` (3 pts), collect data, and make a plot of velocity vs. PWM and PWM vs. velocity for each motor (3 pts each).**

1.3 Compute Open-Loop Control Gains

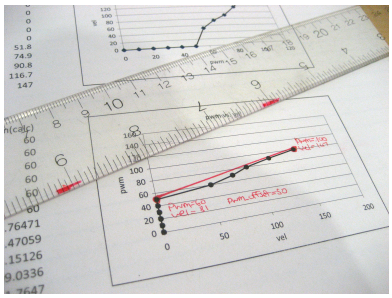
We want to write the `feedforward_compute()` function that will take a goal velocity as an input, and return the required PWM. So, we need to create an equation that models the PWM vs velocity data from above. We will use the form:

$$pwm = K_{ff\_offset} + K_{ff} \cdot vel_{goal}$$

Where K_{ff_offset} is the largest PWM value that *does not* produce any motion, vel_{goal} is the desired velocity, and K_{ff} is the *feed-forward gain* — the relationship between velocity and PWM from the data. Compute K_{ff} from your data by computing the slope of the line:

$$K_{ff} = \frac{\text{change in y-values}}{\text{change in x-values}} = \frac{pwm_{max} - pwm_{min}}{vel_{max} - vel_{min}}$$

This is easy to see if you take a ruler to your data, you just need to find the slope of the line between the points. Use the line from your best motor — the higher line. My data looked like this:



...but my robot had a low battery and one sticky motor. Your data will be different. Set the two constants at the top of your file, `K_FEEDFORWARD` $= K_{ff}$ and `K_FEEDFORWARD_OFFSET` $= K_{ff\_offset}$, with your calculated values.

Finally, we can write the `feedforward_compute(vel)` function. This function should take a velocity and return the appropriate PWM value. It must always return a PWM value between -100 and 100, whether the robot can operate at the goal velocity or not. As the controller tries to increase or decrease speed (especially if the goal velocity is outside the controllable range of the robot), the computed PWM values may be outside of the useable range, $[-100, 100]$. You should write a function `bound` which takes a PWM value and returns a PWM value that is bounded to the valid range, being careful to handle negative values. **Hand-in: Compute K_{ff_offset} and k_{ff}. (2 pts each) Write `bound(val,val_max)` and `feedforward_compute(vel)` (3 pts each)**

1.4 Measure Open-loop controller performance

Use the `controller_test()` function to test your controllers. Write `open_loop_motion()` to test the open-loop controller. This function should compute the velocities for each motor, run them for the correct time, then stop. Do not use the `sys.sleep()` function to measure time, it is too inaccurate. Instead, use the `sys.time()` function. Get the start time, then make a while loop, and compare the current time to the start time to determine when to stop.

Run 10 trials to drive the robot 10 seconds at 100 mm/sec. Measure the error between the calculated position and the actual position. Also, determine the lowest velocity before your robot doesn't drive properly. **Hand-in: One table with 10 error measurements, and the lowest controllable velocity of your robot (4 pts)**

2 Closed-Loop (Feedback) Control

In order to drive straight, you must command each wheel to travel at the same velocity. This will require different PWM values for each wheel and those values will change depending on the battery voltage, the unevenness of the floor, and other factors. You cannot manually account for these issues. This is the job of a feedback control loop.

We start by breaking time up into fixed periods, 30ms in this problem set. For each period, you measure the current velocity of each wheel, and increase or decrease the PWM value to each motor according to the difference between the measured velocity and the desired velocity. We will use the `feedforward_compute()` function to make a good guess of PWM, then use our controller to fine-tune the values. The type of controller we will be building is called an *proportional-integral controller*. The following equations show how the controller operates. First, you must initialize the encoder values, time step, and integral term:

$$\begin{aligned} ticks_0 &= rone.encoder\_get\_ticks() \\ time_0 &= sys.time() \\ iterm_0 &= 0 \end{aligned}$$

Then, at each period, you measure the velocity error and update the PWM sent to the motor. The following equations show how the controller is updated at each timestep, n (for $n > 0$):

$$velocity_n = \frac{distance_n}{\Delta t} \tag{1}$$

$$ffterm_n = \text{ feedforward\_compute}(goal\_vel_n) \tag{2}$$

$$error_n = goal\_vel_n - velocity_n \tag{3}$$

$$pterm_n = (K_p * error_n) \tag{4}$$

$$iterm_n = iterm_{n-1} + (K_i * error_n) \tag{5}$$

$$pwm_n = ffterm_n + pterm_n + iterm_n \tag{6}$$

In these equations, Δt is the duration of the last time period, which you must calculate in your code. First, compute the feedforward term based on the current goal velocity (Eqn. 2). This does not use any feedback. Then you measure the actual velocity (Eqn. 1), calculate the error in the velocity (Eqn. 3) and update the feedforward term(Eqn. 2), the proportional term(Eqn. 4), and the integral term (Eqn. 5). The new PWM value to drive the motors (Eqn. 6) In the code, `K_INTEGRAL`

$=K_i$ and `K_PROPORTIONAL` $=K_P$. Note that the PWM value given to `rone.motor_set_pwm()` must be an int. Use `PWM = int(PWM)` to cast the variable as an int.

For this problem set, the goal velocities for both wheels will remain the same, but you can build programs that command different velocities. Because each motor is different, they will require different PWM values, even for the same goal velocity. So you will need to calculate these equations separately for each motor, and then give each motor its own PWM value.

2.1 Computing the Proportional and Integral terms

First, write the `proportional_compute()` function. This takes as input the goal velocity, and the current velocity, and implements Eqn. 3 and Eqn. 4. It returns the `p_term`. Next, write the `integral_compute()` function. This takes as input the goal velocity, current velocity, and current `i_term`, and implements Eqn. 3 and Eqn. 5. It returns the new `i_term`. **Hand-in: your `proportional_compute()` function (6 pts), and your `integral_compute()` function (6 pts)**

2.2 Implementing the Velocity Controller

We've given you the `closed_loop_motion()` function, which does the task of keeping all the variables organized. It calls the `velocity_controller()` function, which is where all the magic happens. This function should implement the integral controller from Section 2. Use `integral_compute()` from the previous section. Whenever you read the encoder, you should immediately also read the time. You should not rely on `sleep` or other mechanism to keep track of when the encoders are being read. We've put comments in the code for you to use to get started.

You will need to select K_p and K_i by experimentation. First, write a program that commands a constant velocity. Set $K_i = 0$ and increase K_p until the robot's velocity becomes unstable, then back down a bit. Once you have set K_p, use this value and repeat the same process to determine K_i. For K_i, start small, around 0.01, and increase this parameter until the robot stutters. Note that setting $K_p = 0$ and $K_i = 0$ transforms the feedback controller to a feed-forward controller. **Hand-in: A wonderful `velocity_controller()` function (6 pts).**

2.3 Measure Closed-loop Controller Performance

Use `controller_test()` again to test your closed-loop controller. We've given you the `closed_loop_motion()` function, because you've done enough work. Again, run 10 trials to drive the robot 10 seconds at 100 mm/sec. Measure the error between the calculated position and the actual position. Determine the lowest velocity before your robot doesn't drive properly. **Hand-in: One table with the 10 error measurements, and the lowest controllable velocity of your robot (4 pts)**

3 Design Challenge 4: Robot Croquet

On the due date, you will demonstrate your robot driving straight in a Robot Croquet Design Challenge.[2] The first wicket on the croquet course is the check-off for this lab, and is mandatory. Completing more advanced hoops will earn you fabulous prizes. You must have your code working by the start of class or you will not be able to complete this challenge.

[2]See `http://en.wikipedia.org/wiki/Croquet` for more information on Croquet.