MTRE 2610 Engineering Algorithms and Visualization – Dr. Kevin McFall

Laboratory – Proportional/Integral Small Robot Control

Introduction

The goal of this exercise is to control the motion of the wheeled robot in Figure 1 to maintain a constant distance between itself and an object detected by an infrared range finder. This action mimics adaptive cruise control or platooning in an automobile where the self-driving vehicle maintains a safe distance between itself and the preceding vehicle. After learning the basics of setting rotation speed in DC motors and reading distances from the infrared sensor, a proportional and then proportional/integral controller will be implemented to maintain a desired distance.

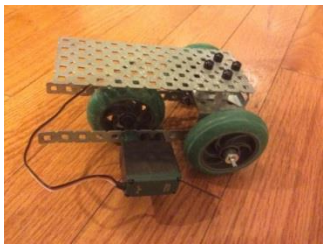

Figure 1: Small wheeled robot to program with a PI controller to maintain distance.

Operating 3-wire VEX motors

The VEX motor depicted in Figure 2 used in this exercise has a built-in motor controller which is driven by a pulse width modulation (PWM) signal. This is different from the 2-wire motor used in the previous laboratory exercises. Recall that motor required additional electronics including resistors, diodes, and transistors or an H-bridge to take the simulated analog output value from the Arduino to move the motor. Those exercises essentially built a rudimentary motor controller. The VEX motor used in this exercise has a built-in motor controller. The PWM signal sent across the white wire is not a simulated analog voltage but rather a control signal where the pulse width determines the motor's speed and direction. The motor controller inside the VEX motor will spin full speed in one direction with a pulse width of 1 ms, full speed in the other direction at 2 ms, and 1.5 ms will cause the motor to be stationary. Operation of the motor is not especially sensitive to the signal period but a value around 20 ms is recommended. Review how PWM signals were generated from the Arduino Basics laboratory. The white signal wire is connected to one of the Arduino digital pins capable of generating a PWM output which are marked with a tilde ~, such as pins 9, 10, and 11.

Figure 2: VEX motor with its three wire connection for ground (black), power (red), and PWM signal (white).

The current required to operate DC motors is generally greater than what an Arduino can supply, and so the red and black wires should be connected to a battery or power supply rather than Arduino output pins. The Arduino is powered through the USB cable connected to a computer, and the motor in this exercise will be driven by a battery

connected to the breadboard using a battery snap or wire leads such as those in Figure 3. The positive + terminal is connected to the motor's red power wire and the negative – terminal to the black motor ground wire. So that all electronics share the same ground, connect a jumper wire from the negative – battery terminal to a ground pin on the Arduino.

Figure 3: 9 V battery and battery snap (left) or the Fischertechnik 8.4 V battery and leads (right) to power the motor.

Laboratory exercise procedure

Task 1: Control motor speed with input from infrared sensor

First connect the motor and demonstrate that speed can be modified by testing various values of pulse width. For this exercise, place some object under the robot or hold it above the tabletop to prevent it from driving off the table. Notice that due to internal and rolling friction, the motor does not turn for pulse widths moderately different from 1.5 ms. Be aware that the `delay` command only takes integer input, so fractional millisecond delays must use `delayMicroseconds`. Combine code for controlling the motor and reading the analog voltage from the infrared range finder. Now, rather than using a static value for the pulse width, make it dependent on the sensor value. Make the motor react by changing directions depending on the sensor input.

Task 2: Proportional control

Consider the task of controlling motor speed to maintain a constant distance between the robot and some object. The larger the measured and desired distances differ, the faster the motor should turn, one direction for positive differences and the other for negative. Motor speed should be determined now not by the sensor value but rather by the difference between the sensor values and a target value. For this exercise, use a target of 220, corresponding to an approximately 1.08 V input or 28 cm distance. The difference

```
diff = sensorValue – 220;
```

will be positive or negative depending on which direction the motor needs to turn. The `pulseWidth` can then be determined by adding 1500 μs to `diff`, so that the motor would not move when `diff` is zero. Notice that `sensorValue` can have a maximum of 1023, and therefore `diff` could exceed 500, causing `pulseWidth` to stray from the 1000 to 2000 μs range. To prevent unpredictable behavior from out of range pulse widths, include `if` statements limiting the `pulseWidth` value. Use the resulting pulseWidth to implement a PWM signal with a 20 ms period. The robot should now automatically adjust the distance between itself and a detected object in front of it. To make the robot react more quickly, multiply `diff` by a constant gain when using it to define the pulse width. Larger gains will make for quicker reactions, but values too large could result in instability. Be sure to define gain variables as `float` data types if fractional values are desired. Adjust the gain value to achieve the best performance. A proportional control scheme has now been implemented with motor speed being proportional to the error in position. While operating the robot, observe the serial monitor and notice that while the robot does react to bring `sensorValue` toward the desired 220 value, it generally stops significantly different from 220. The cause for this is friction preventing the motor from moving for pulses differing small amounts from the 1.5 ms neutral position. The method for eliminating this steady-state error is to add an integral control component.

Task 3: Proportional/integral control

The idea behind proportional/integral control is that motor speed should depend not only proportionally to error, but also depend on the integral of error over time. For small steady-state errors, this integral will continually grow over time until it is sufficiently strong to cause the robot to move. Somewhat counter-intuitively, this integral does not grow larger in perpetuity, but rather will be decreased in magnitude once the robot passes the desired position, error changes sign, and the integral value begins to reduce. Integral control does introduce oscillation about the targeted position, but its magnitude decreases over time until the robot converges on the desired position.

To include an integral control term, declare a new variable `accumulate`, initialize its value to zero in the `setup` function, and add the current value of `diff` to it every time through the `loop`. Update the pulse width now with both the proportional and integral terms, where the integral term has its own gain. Integral gains are typically much smaller than proportional gains as `accumulate` can grow quite large with numerous iterations through the loop. Adjust the integral gain to obtain the best performance, striking a balance between eliminating steady-state error and minimizing oscillation. Display `diff` to show that steady-state error is eliminated by adding integral control.

Grading rubric

1. 33 points: Robot reacts by moving in both directions depending on the distance sensor reading
2. 33 points: Proportional control implemented with observation of steady-state error in the serial monitor.
3. 33 points: Proportional/integral control implemented as evidenced by steady-state error being eliminated quickly with as little overshoot and oscillation as possible.