Introduction to Vision and Robotics
Robotics Practical: Line Follower

Dylan Angus, Matthew Martin

November 23, 2016

1 Introduction

The purpose of this practical is to learn about controlling a four-wheeled robot within a known environment. We used Lego's EV3 Python toolkit, assembling our own robot and developing all the robot code in Python. The robot is meant to accomplish three tasks:

- Follow a curved line from beginning to end
- Follow a set of broken and staggered lines, going from one line to the next
- Complete a lap of a closed circuit while circumventing an object placed in the path of the robot

2 Getting to know our robot

We approached these tasks in a series of steps. First, we tried to gain familiarity with the operation of the robot by performing several tests on it to see its movement based on commands that were sent to it. Then, using this information, we tried developing a system of odometry and dead-reckoning. Finally, we solved the tasks sequentially, as each subsequent task built on some of the methods developed in the prior task.

2.1 Initial Testing

We conducted several tests in order to get consistency in how the commands that were sent to the robot translate to actual distance moved in the world.

First, we ran the motors for a series of durations using `run_to_rel_pos()` keeping the `duty_cycle_sp` parameter constant at 25%. These durations were in the unit of tacho counts, which is how the rotary encoder inside the motor measures turns. We performed tests at 25% power for tacho counts of 100 to 700, incremented by 50. See Figure 1a for this data. From these tests and the slope of the trend line observed, we concluded that for forward commands we can convert from centimeters to tacho counts by performing the following calculation:

$$tachoCounts = \frac{centimeters}{4.807090465}$$

.

Then we performed similar testing on the angle that the robot turned based on a series of commands. We measured the angle by reading the gyro sensor before the command and after the command. The motors were kept at 25% power, and we turned one wheel forward and the other reverse to do an in-place rotation. See Figure 1b for a plot of this data. We observed the following conversion from tacho counts to degrees:

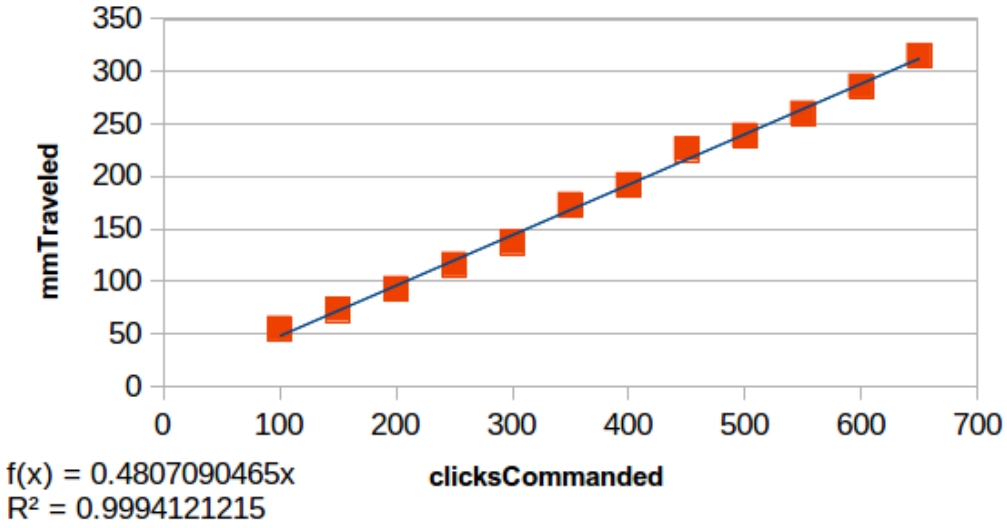
$$tachoCounts = \frac{degrees}{0.4695354523}$$

.

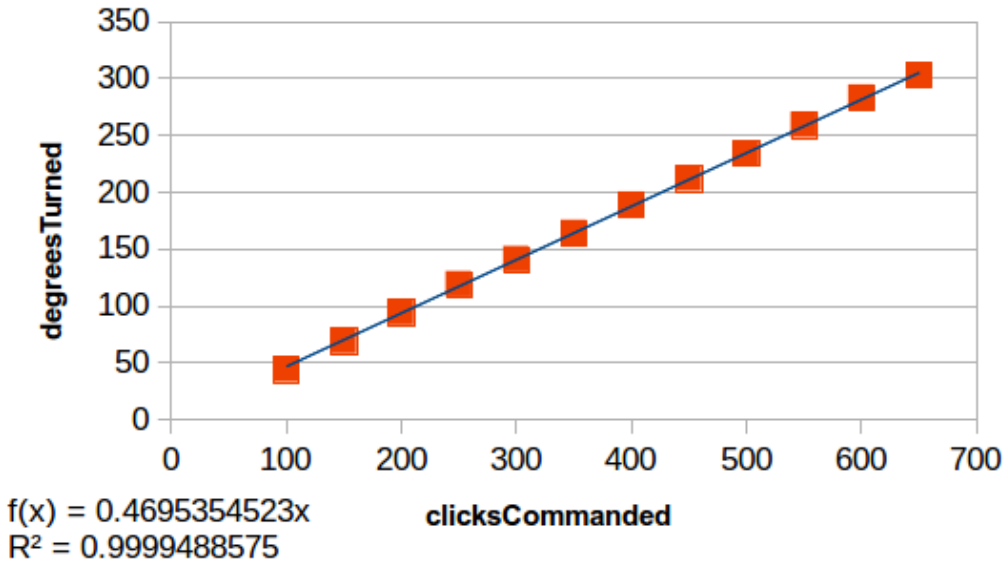
These conversions proved useful in commanding our robot to move set distances or rotate a specific number of degrees, as the robot's performance was quite consistent (as is observable from the very high R^2 values).

2.2 Odometry and Dead Reckoning

Initially, a full dead reckoning system was attempted by following the calculations in ... . The resulting system worked but with too much error to be useful. Though we could've attempted to account for the error and make the system more accurate, the gain from having a working system did not outweigh the cost of testing and developing it as we were able to complete the tasks without it. Instead, it was settled on simply knowing the relation



(a) distance travelled versus tacho counts (clicks) commanded



(b) angle turned versus tacho counts (clicks) commanded

Figure 1: These are two plots generated by a series of commands where a movement metric was recorded as the result of a number of tacho counts (clicks) commanded.

between wheel rotation and real-world distance and angular change. This was achieved through the testing described above.

3 Tasks

3.1 Methods

Each of the three tasks built upon each other, beginning with simple line following.

3.1.1 Line Following

For line following, we used a PID controller. The overall idea was to set the goal of the PID controller to be the brightness value of the edge of the line. Therefore, the controller would constantly try to maintain that goal which would correspond to following the edge of the line. To normalize the readings fed to the PID controller, the raw readings from the color sensor had to undergo some calculations. Firstly, the raw output of the color sensor was capped so that it's values stayed in the range of 10 and 50 (we found the edge of the line to have a value of approximately 30). Those values were then shifted by thirty so that they were centered across 0. Finally, these values were fed to the mathematical function tanh so that the output was between -1 and 1. This information was sent as the current value to the PID each iteration. The goal of the PID was set to 0. Then, we did extensive tuning to achieve smooth results and steady-state accuracy.

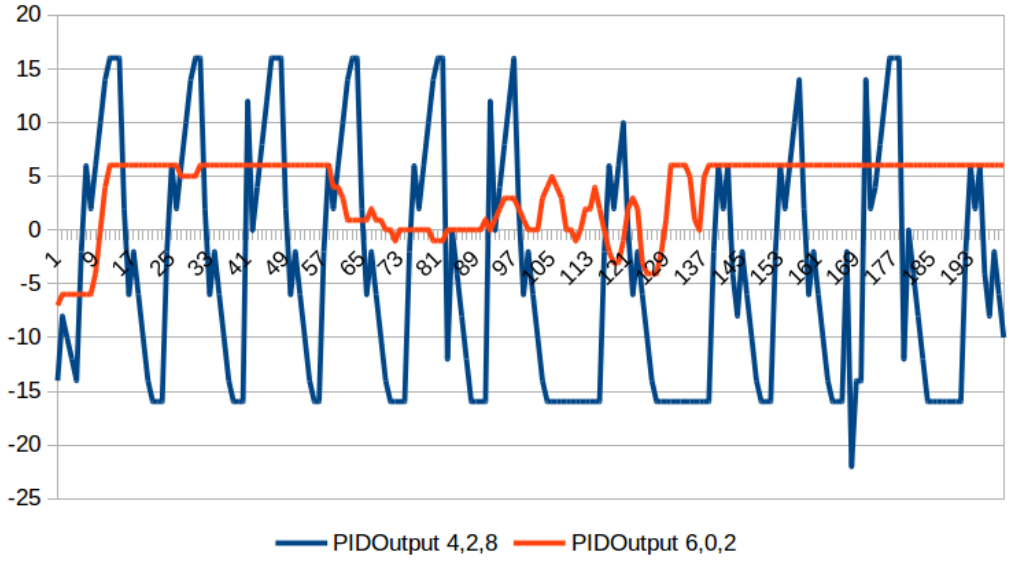
Our tuning process consisted of roughly following the Ziegler-Nichols method, but mostly a lot of try-and-revise. The first set of relatively stable constants we settled on for line following were $K_p = 4, K_i = 2, K_d = 8$. This resulted in our robot having consistent, but somewhat large oscillations, and good performance on turns. This corresponds to the blue line in Figure 2a. It can be seen that its oscillations around the goal (0) are fairly large and it never really settles to steady state. Next, we tried eliminating the integral term with the constants $K_p = 6, K_i = 0, K_d = 2$. The performance of these constants were impressive on the straight part of the line, but disappointing on the curves. You can see the orange line in Figure 2a which plateaus at about 6 at the beginning and end of the plot. These are the periods when the robot was making a turn. However, during the straight section in between

those turns we saw the robot achieve its steady state nicely. With this knowledge, we thought that adding the K_i and increasing the K_p would correct for these errors and maintain some of that stability. We adjusted the constants to be $K_p = 12, K_I = 1, K_d = 3$. Its performance on the straight line is not as good as the second set of constants we tried, but it had less overshoot than the first set as can be seen in the blue line in Figure 2b. We made some adjustments that we thought would account for this. The last iteration is the one we settled on, $K_p = 6.5, K_i = 0.825, K_d = 3.0$. You can see in the red line in Figure 2b that it acheives steady state during the middle section (straight line), and has minimal oscillations on the curve. Its rise time in the curved sections of the line leave room for improvement, but we decided that these constants were the ones we would use for line following.

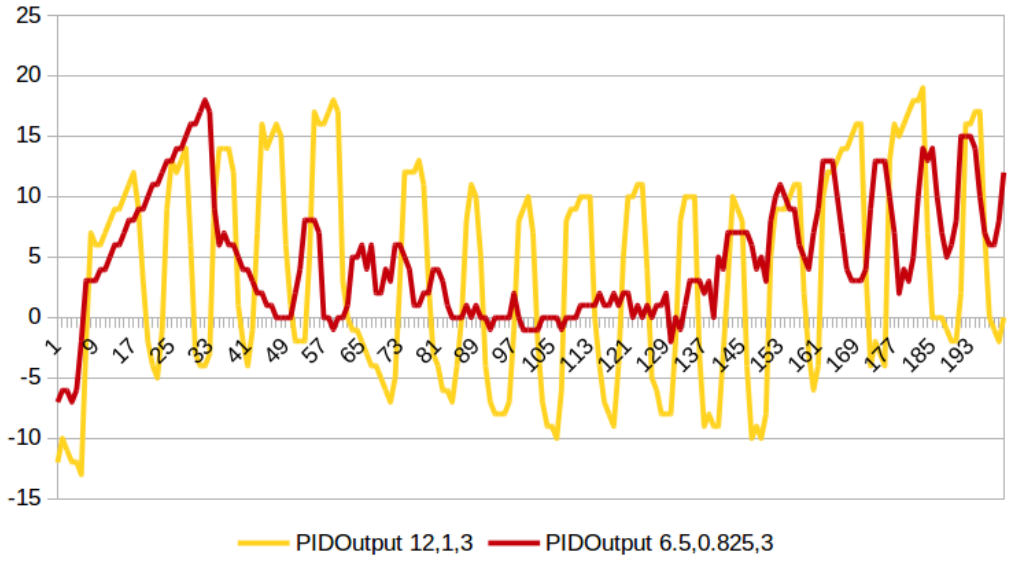
For detecting when the robot has reached the end of the line, we used a combination of two features to make the judgment. We had a variable that incremented each iteration of the following loop if the robot was not on the line, and as soon as it was on the line again, then it was reset to 0. Thus, we could use this as essentially a time variable tracking how long the robot has been looking for the line. If this number gets large it is likely the robot reached the end of the line. We also kept track of the difference in the robot's angle from the last time that it touched the line. If this difference got sufficiently large this also indicated the robot has been looking for the line for a while. Specifying these thresholds allowed us to break out of the loop when the robot was fairly sure it had reached the end of the line.

3.1.2 Staggered line navigation

Following the set of staggered lines proved not to be very difficult after developing a consistent PID controller for the line following. We had to make a few additions to the algorithm used for line following in order to traverse these lines. First, we made a strict breakout case for the line following by having the count and angle discussed above be small values. Then depending on the number of lines it had completed, it turned right or left and drove forward until it found the next line. Then followed that one, and repeated the whole process until it completed all the lines.



(a) the first two sets of constants for PID line following



(b) the second two sets of constants for PID line following

Figure 2: Overlaid plots of the four sets of constants for PID line following. The goal of the controller is 0.

3.1.3 Obstacle avoidance

This task required the robot to follow a line until it sees an obstacle, then go around said obstacle and find the line again. To do this the robot was programmed to use same the line following function that has been discussed until the sensor reading for the ultrasonic sensor fell below a threshold of 75 (7.5 cm). After falling below this threshold it turns ninety degrees on the spot then turns its servo ninety degrees the opposite direction towards the object. It then uses readings from the ultrasonic sensor to send to a PID controller which attempts to maintain a constant distance between the obstacle and the robot until it detects the line again. After seeing the line again the robot then returns to following the line.

The time-consuming aspect of this task was, again, tuning the PID controller for circumventing the object. We set the goal distance as 125 (12.5 cm), and set the robot to maintain that goal around the whole object. In order to remove noise from the sonar reading, which had a range of 2550, we capped those readings at 500. This helped because we wanted behavior to be consistent around the box regardless if the sonar was reading something irrelevant but that was closer than 2550 millimeters away. This improved the performance greatly, and we ended up using mostly the K_p constant, with $K_i = 0$ and K_d equal to a small value. We found that with just K_p, the robot was turning too quickly and a small K_d counteracted this nicely. See Figure 3 for a plot of the error as it rounded the box.

3.2 Results

3.2.1 Line Following

The results of our line following PID controller on the actual path can be seen in Figure **??**. This graph shows that our controller, although it experiences oscillations and never truly finds the steady state, is reliable and robust by not ever letting the error get too large.

3.2.2 Staggered Line Navigation

There are no unique results to discuss in the staggered line, since it uses the PID controller from Task A. We were happy with the consistency of the breakout case, as its accuracy was essential for the robot to be able to find the next line.

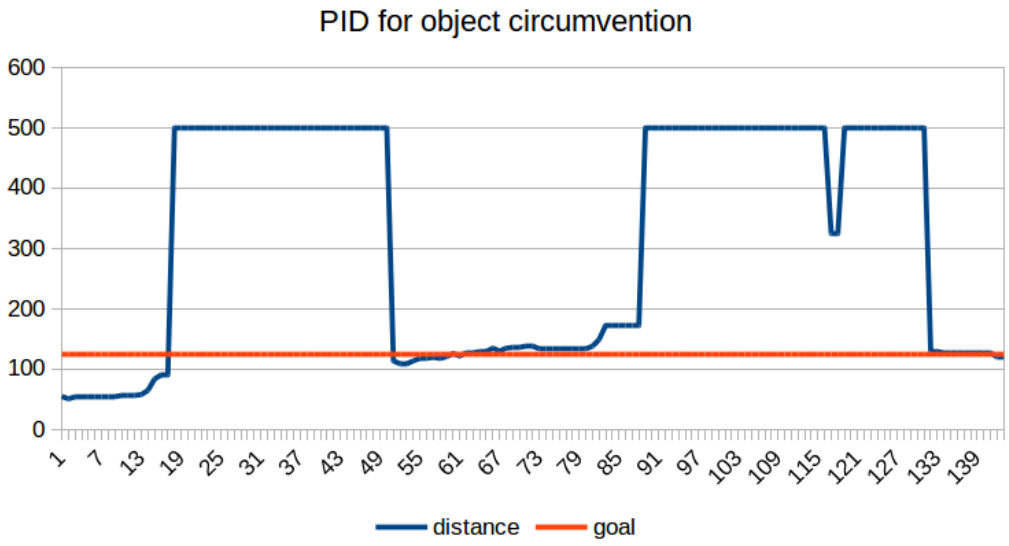


Figure 3: The results from tuning our PID controller for obstacle avoidance.

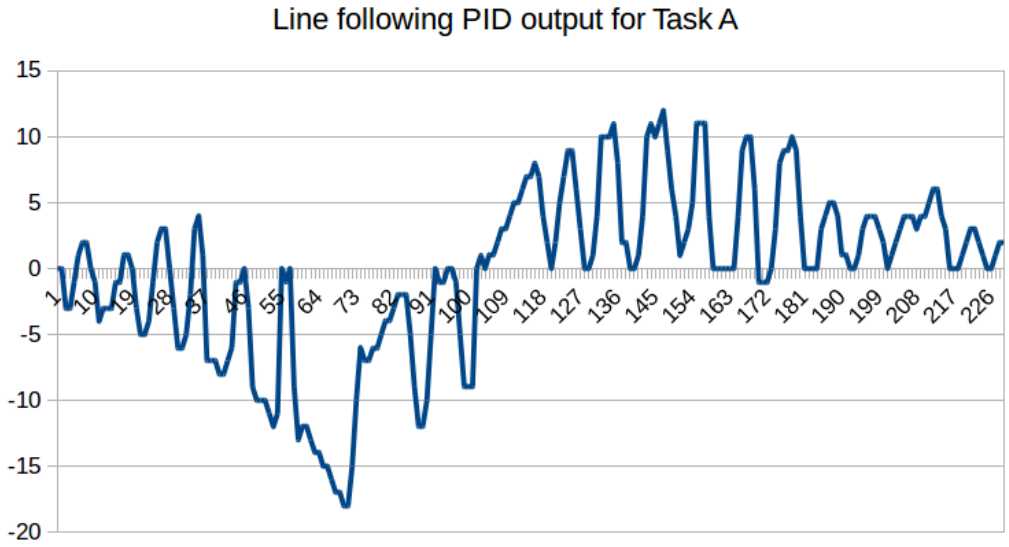


Figure 4: The results from tuning our PID controller for Task A completion.

3.2.3 Obstacle Avoidance

As discussed above, the plot of the PID controller for the obstacle avoidance is in Figure 3. We were very satisfied with its performance here as it was able to maintain the goal state almost immediately after turning each corner, which can be seen in how close the controller is to 125 after it is at 500 for so long (which are the turns).

4 Discussion

The overall performance of the robot on the tasks were satisfiable in face of the limitations and challenges that were faced, however there are of course possible improvements in certain areas.

4.1 Successes

We believe we were successful in making our algorithms fairly robust and consistent...

4.2 Challenges

The task of tuning the PID controllers turned out to be a very tedious process as it required moving from room to room adjusting the parameters and testing on the training mats. Additionally, finding the balance between balancing the coefficients for the PID controller between being able to make sharp turns and overshooting was especially challenging. With regards to hardware, the gyro sensor and servo motor tended posed interesting problems. For the gyro sensor, it records it's starting point as the point and direction in which it was facing when the sensor was plugged in and there was no way to set that value back to zero at initialization of the robot without having to recalibrate the sensor on a whole. To get around this we recorded the reading of the gyro sensor at the point of initialization and avoided making calls directly to the gyro sensor but instead to a recorded value in which is the difference of the current value and the initial value fixed in the range [-180,180] to ensure the robot would always take the shortest route to the required angle (i.e. choosing between turning left or turning right). The servo motor on the other hand would at times for example believe that its zero position was at 90 degrees. In order to get around this we made sure to fix it manually at

true zero and then indicated that this position was zero upon initialization of the robot.

4.3 Improvements

Out PID controller for line following definitely has room for improvement. It's inability to achieve a steady state is frustrating and we were unable to find the correct combinations of constants to make that happen. The robustness of the algorithm to find the next line in the staggered line follower would also be improved. If the robot over-rotates after completing a line, or travels a bit farther than expected, there is a chance that it does not find the next line. This could have been improved by making the rotation more precise, or, instead of sending the robot in a straight path to find the next line, have it sway a little and search a greater area.