Introduction to Vision and Robotics
Robotics Practical: Line Follower

Dylan Angus, Matthew Martin

November 22, 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 Methods

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 developed 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 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:

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

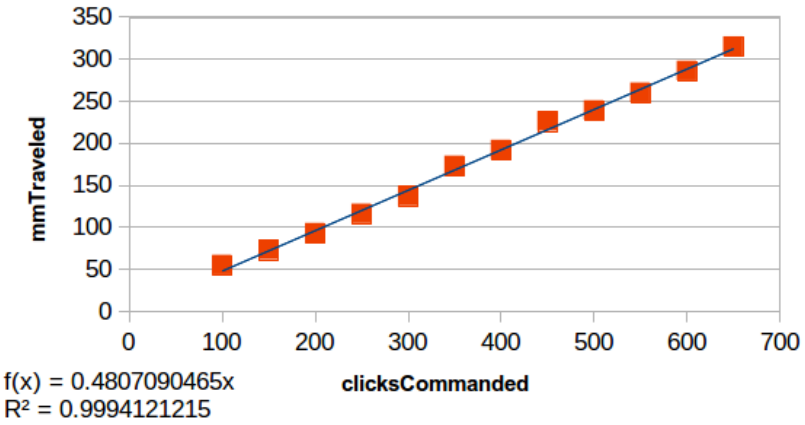
.

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).

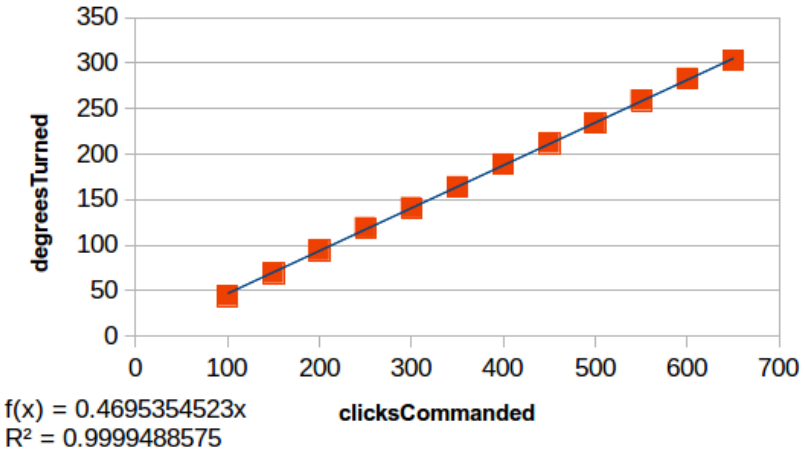
2.2 Odometry and Dead Reckoning

2.3 Tasks

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



(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.

2.3.1 Line following

Our first solution to this problem was functional, but naive. We set up a loop that operated by the following logic: if the robot was on the line (which could be determined by the value of the downward facing color sensor), turn right, otherwise, turn left. This resulted in the robot wiggling along the line with fairly wide turns and slow progress.

Then we revised our approach to use a PID controller instead. The overall idea was to set the goal of the PID controller to be the color/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. We created a function that returned a 0 if the robot was on the edge of the line, a -1 if it was not on the line, and a 1 if it was on the line. 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.

Tuning the controller was an extensive process. We had to first scale the output of the PID controller to values that made sense to the motors. We set the base `duty_cycle_sp = 25` for each of the motors, so that when it had 0 error, it would move forward at 25% speed. So the output values had to be scaled to make sense for a change in motor power percentage. One motor's speed was increased and the other's decreased, or vice versa, based on the sign of the output.

Tuning process, and data from it.

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 varaible 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.

2.3.2 Staggered line navigation

2.3.3 Obstacle avoidance

3 Results

4 Discussion

Appendix

`code`