IVR Robotics Assignment Report

Onji Bae (s1683818)
Jennifer Perry (s1683710
Karen Xu (s1672390)

University of Edinburgh
Fall 2016

1. Introduction

Our ev3 robot utilizes a color sensor and ultrasonic sensor with a PID controller in order to autonomously navigate and complete line following and circumventing tasks.We explored several different ways to achieve the tasks given by testing the abilities of the sensors and robots, testing the accuracy of the PID algorithms we created. Our goal was to have the robot move smoothly, have the robot move as fast as possible while still completing the task, and create repeatable success.

2. Methods

2.1. PID Controller Overview

The PID controller corrects for deviation from the line.

P (Proportionality Constant)

P determines how much we turn by adjusting the output of our wheel motors proportionally to the distance we have deviated from the line. As p is a proportionality constant we can imagine our movement following a straight line on a graph using the equation y(turn) = mx(sensor reading) + c(0 for going straight). By subtracting the average of our sensor readings for white and black ((a+b)/2), we can produce a value 'error' which will tell us how far off the line we are. An error value of 0 will mean we are straight with the line. Our equation becomes y=mx or turn = m*error.

I (Integral Constant)

By just using P over time large errors pile up, the 'I' term corrects for this. The integral, holds a running sum of the errors. We add every error we calculate to I, multiply by proportionality constant, Ki, then add to our turn variable/previous formula. Turn P = Kp*error + Ki*integral. As time progresses the integral plays a large part in correcting previous errors and excels in correcting small errors.

D (Derivative Constant)

D tries to predict any upcoming errors and adjusts for that. We based D on assuming the next change in error is the same as the last change in error. It assumes our next error will be error = error + derivative (the difference between the two preceding samples). D is useful because if the current error is worse than the previous error, D will try to correct this error. If it is better, than D will try to stop the controller from correcting this error because it shows we are actually improving and veering closer to our line, towards 0. This stops us from overshooting/overcorrecting by predicting when we reach the desired position before we actually reach it. Giving us our final equation of

$$Turn\ P = Kp*error + Ki*integral + Kd*derivative$$

The final Kp, Ki, and Kd values were fine tuned through trial-and-error and the "Ziegler–Nichols Method".

2.2. Sensor and Motor Testing (See: Appendix, *sensor_testing.py*)

Color sensor testing: *color_calibrate()*
We chose the mode COL-REFLECT because it returns a range of values, allowing us to determine a target value for when the robot is right on the edge of the black line, sensing half black and half white. We then recorded sensor values by placing the robot on black and white areas of the testing map.

Ultrasonic sensor testing: *sonarTesting()*
This test prints the sonar values the Ultrasonic Sensor picks up. We manually moved the robot around obstacles to record the values. This allowed us to identify when values are equivalent to important changes in the robot's physical surroundings such as the detection of an obstacle or an empty space.

Motor testing: *motorTesting()*
Motor testing was used to determine the optimal power level (duty_cycle_sp) and time (time_sp) for each motor when achieving a certain task. Finding the right values for specific turns proved to be especially difficult because the motors were affected by battery level, internal friction, and external friction.

2.3. Task A (See: Appendix, *taskA.py*)

Brute force approach: *taskA.followLine()*
First, we used a brute force method that follows the left or right edge of the black line. It used a series of if-else statements to control the motors based on the color sensor values. If the color sensor returns a color value of over 50, the robot turns left towards the line. If it is under 15, the robot turns right. Otherwise, the robot would move straight.

PID controller approach: *taskA.followLine_PID()*
Next, we created a method that uses a PID controller. The robot constantly aimed to be as close as possible to the desired position of color value 45 which is the edge of line. To do this, it calculates the error between its current position and desired position, and uses the error value to determine how much power to apply to each motor. As a result, the robot follows the line in very short, small turns, constantly readjusting itself.

The robot's movement was much more smoother and accurate for *followLine_PID()* than it was for *followLine()*. This is because instead of using constant power and time values to move the motors, it was able to use corrected values from the PID controller. Thus, the robot avoids a choppy pattern of movement. In conclusion, we used the PID controller approach over the brute force approach for Task A due to smoothness.

In order to determine when the line ended, we tracked when the color sensor was consistently returning white by checking for a high sensor value, above 82, and a low difference between each iteration, below 5. When both values were read a certain amount of times, the robot would stop following the line.

2.4. Task B (See: Appendix, *taskB.py*)

For Task B, the robot moves forward along a line under the guidance of a PID controller until the end of the line is reached, which is detected by the colour sensor as white. The robot pauses to state its next action before rotating 90° to advance straight until the next line is detected, by the colour sensor detecting black. The robot rotates 90° once more to face down the line and the procedure repeats until all the lines have been navigated.

The above method was the simplest and easiest to implement. We tried a recursive function using distance values from the given track. However, because of factors such as the depleting charge of the battery and uneven surface of the ground which were prone to angle variation, the robot turned differently each time. This made it highly prone to errors as it had no way to adapt. Incorrect trajectories were hard to correct without the use of a PID controller.

2.5. Task C (See: Appendix, *detector.py, taskC.py*)

Brute force approach: *detector.circumvent()*
The *circumvent()* method makes the robot turn around a cube-like obstacle with 90° angle turns. First, the robot turns left once and its medium motor turns to face the obstacle. Then, it moves forward. The robot turns right at 90° only when the difference between the previous sonar value and current value is greater than 290 because it means that the sonar currently does not detect anything, so the robot can safely turn right without bumping into anything.

PID controller approach: *taskC.aroundObstacle()*
aroundObstacle() both follows the line and circumvents around an obstacle. It uses a PID controller for both tasks. For avoiding the obstacle, it uses the desired position of sonar value 90. As a result, the robot moves so that it is neither too far or too close to the obstacle until the color sensor detects a black line.

The *circumvent()* method only works for a cube obstacle with specific dimension. It does not cover variation in obstacle placement or types of obstacles. Therefore, we chose the *aroundObstacle()* method for Task C because of its usefulness in navigating any type of obstacle. The robot can constantly readjust its turn angles depending on the PID error value, whereas for *circumvent()* it is strictly limited to moving forward in a straight line and 90° turns.

3. Results

3.1. Color Sensor Trials

We ran the function *color_calibrate()* three times. The function returned two values for white, two values for black and then averaged all four values to return the offset value that could be used in the PID controller.

Trial	**White**	**Black**	**Average (PID offset value)**
1	87, 85	6, 8	47
2	86, 84	7, 7	46
3	79, 85	7, 9	45

3.2. Ultrasonic Sensor Testing

The *sonarTesting*() method allowed us to test how far the ultrasonic sensor can detect obstacles and what values equate to certain obstacles. The following are the results from our testing:

Value	**Meaning**
X < 150	Obstacle is in front of the robot
150 < X < 300	Obstacle is near robot but far enough for it to move
300 < X < 900	Robot senses something; Recommended it continue moving until X is < 150
900 < X < 1800	No obstacle but most likely within obstacle course
1800 < X (~2000+)	In an empty room

From this testing, we came up with the following sensor value ranges for Task C:

Task C, *aroundObstacle()*

Obstacle Detected Turn Left	Optimal Distance from Obstacle Move Forward	Too Far from Obstacle Turn Right
X < 90	X = 90	X > 90

3.3. Motor Sensor Trials

For the large motor, we tested different combinations to turn the robot 90°. To determine how successful the power levels were performing, we ran each combination 10 times over the course of a day on different surfaces and at different battery levels. Two mostly successful combinations are listed below:

Left Motor duty_cyle	**Right Motor duty_cycle**	**Percentage** 90° **Achieved**
-60	60	60%
-90	10	90%

The medium motor was especially inconsistent. We ran the medium motor at several different power levels over the course of a week and noted the angle at which it turned. Our target angle was 90° so that we could turn the ultrasonic sensor to face the obstacle in Task C.

duty_cycle	**Best Result**	**Worst Result**	**Percentage Best Result Achieved**
20	45°	0°	10%
25	45°	0°	70%
30	90°	0°	80%
60	90°	180°	50%

3.4. Task A PID Trials

followLine_PID()

Kp	**Offset**	**Tp (Power)**	**Time**	**Result**
.45	35	30	150	Followed the line smoothly but misses steep curves
.73	35	30	150	Followed the line and successfully passed steep curves but movement was extremely choppy
.45	30	25	150	Followed the line smoothly but offset was so low, the integral and derivative values were no longer accurate
.26	45	24	150	Good balance between speed, smoothness and accuracy.

We used the numbers from the last row. Lowering the Tp allows the robot to move slower but more steadily, thus it does better at navigating steeper curves. We also made the offset 45 according to our color sensor testing. To address the changes, we fine tuned the Kp value by hard coding the value rather than using the original formula.

3.5. Task B PID Trials

Kp	Power left	Power right	Time	Result
26	-10	90	500	Increased power and momentum(time) leads to overshooting and missing the line .
26	-5	85	475	Most optimum time thus far.
26	0	70	450	Low power and momentum leads to falling short and not being in the position to advance in a direct line.

The component that had the most significant effect on the path of the robot in Task B was the 90 degrees turn taken to position the robot before its next move forward. Much experimentation was required to find the optimum power output for the wheels. The table reflects the range of values that were most successful (for a left hand turn). Kp had an affect on the angle the turn began from and so affected to a degree the position the turn finished in. The task was unable to be continued once the robot overshot of fall short of the next black line.

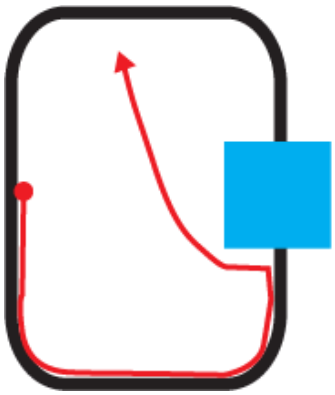
3.6. Task C PID Trials

aroundObstacle.circumvent()

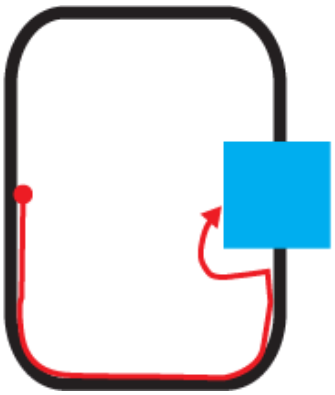
Kp	Offset	Power	Time	Result
.10	90	30	150	Robot did not turn enough and would stray further and further away from obstacle
.15	90	30	150	Robot overturned at obstacle corners and ran into obstacle
.12	90	30	150	Good balance between speed, smoothness and accuracy.

The results in the chart are visually graphed out below. The red line represents the robot trajectory.

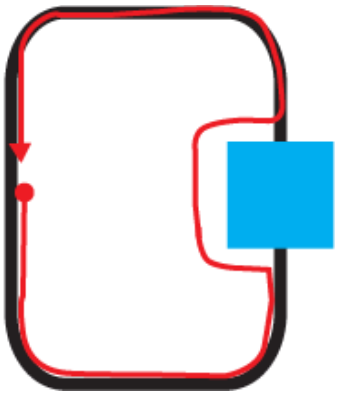
Kp = 0.10



Kp = 0.15



Kp = 0.12



4. Discussion

While our ev3 robot successfully completed all tasks, it was prone to certain errors. For task A depending on which end of the curved line we began with we could have different rates of success. One reason for why it did not perform consistently starting from the end we had least successes with could be because of our PID values. We relied on the P constant more than the I and D and eventually did not even use the I value. If we had more time to test out the I and D values, we could possibly have the robot be more sensitive the changes in the color readings. Another reason would be the inaccuracy of the color readings. Although theoretically the edge of the black line should return the median value between the black and white readings, which would be 45, when we printed out the color values as the robot followed the line, the average value returned was 65 and the median was 78. This meant the color sensor did not return values around 45, but rather mostly white values and occasionally a low black value. Adjusting for this fact by testing the offset and power further may have made the following line functionality more consistently successful.

Task B was less heavily reliant on PID but in exchange was dependent on its ability to arrive successfully at the next lines edge.This required a strong ability to detect a head and stop at the appropriate time to be able to position ourselves to advance along the next line. The algorithm we went with in the end relied on experimentation with motor output and time to get the robot to advance at the right angles, while this means it successful on the given task track, it gives the robot limited ability to perform on similar tracks of different distances without prior testing. To further improve the performance of our robot it would be good to consider the use of a gyroscope for plotting our previous locations in order to work out where our next destination should be. The ultrasonic sensor on the other hand has limited usage in that it is unable to detect the next black line when it is in close proximity but may have the potential to detect black lines further off and so expand the potential range of what the robot can do. Our current design when placed in the exactly the same conditions it was programmed performs largely the same every time so has a high rate of successful performances.

In Task C, the robot would also fail to circumvent the obstacle once in awhile. It would be successful 7 out of 10 runs. These could be explained by the inconsistent behavior of the motors. As shown in the *Results* section, the power levels would often cause very different degrees of movement in the large and medium motors. With more time, we would try using a different motor attribute other than *duty_cycle_sp* in order to improve turns. We would have instead tried to have the motor move to a predetermined angle rather than get it to a perfect 90 degrees using trial and error.

We learned as we worked with the robot that the battery level affected the motor behavior as time went by, possibly explaining why sometimes certain power levels did not make the motors move at all or very slightly. Motors are not perfectly linear, and due to a variety of uncontrollable environmental factors and natural differences between robots e.g motor output, friction, our

proportionality constants have to be tailored to our robot, and may not be true have the same results when applied to another robot, forming natural restrictions on what our current algorithms can and can't do when extended to tasks outside the ones designed for.

Contributions/Percentage of Time Spent:

Onji (15%)
- Built robot
- Task A (theory)
- Report

Jennifer (35%)
- PID theory
- Task B
- Report

Karen (50%)
- Task A (code)
- Task C
- Report