DPM Lab 2 Report

Group 42
Wen Cui 260824815
Aidan Gerkis 260827581

1) Design evaluation

Our design workflow began with the evaluation of the requirements our robot must meet to implement the path-tracking algorithm. Our hardware design is similar to the robot we built for Lab 1.

Figure 1: Our first iteration robot using the Lab 1 design that had difficulty following a straight line

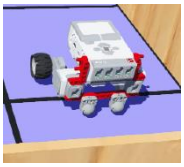

Figure 2: The second iteration robot with 2 ballcasters

The first change we made was removing the ultrasonic sensor, because the algorithm for this lab does not require the use of any external sensors. The second change was the addition of another ballcaster. Because the requirements for this lab were to make our robot move in a perfect square, which is highly dependent on the motors, wheels, and load distribution, we decided to add a new ball caster at the back to further stabilize our robot. This helped to minimize left and right deviations when moving in a straight line.

Our software design includes 4 classes: Main, Odometer, SquareDriver, and Resources. In Main, we created 2 threads corresponding to the square driver code that controlled the robot and the position tracking odometer. In the SquareDriver class we implemented several methods to make our robot drive in a square with unspecified side lengths. We also implemented the Odometer class to track the robot's position with respect to a specified starting point. The odometer value is printed on the console to show us the path our robot is following. After implementing the code one of the main challenges we faced was adjusting the wheel base parameter to ensure the robot was turning exactly 90° at every turn.

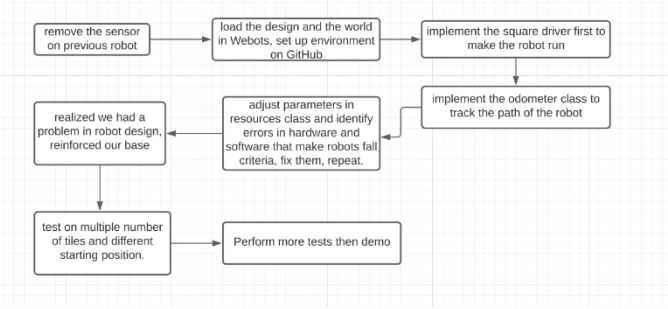

Figure 3: Lab 2 design & testing work flow

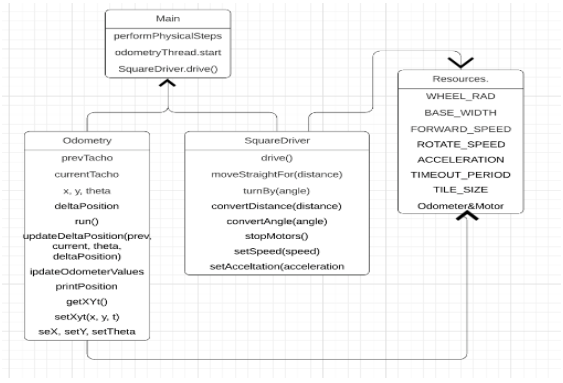

Figure 4: A class diagram of our software architecture

2) Test Data

Table 1: Test Data

Trial #	X_S	Y_S	X_F	Y_F	X_{Odo}	Y_{Odo}
1	0.15	0.15	0.016	-0.019	-0.69	-0.85
2	0.15	0.15	0.016	-0.019	-0.69	-0.85
3	0.15	0.15	0.016	-0.019	-0.69	-0.85
4	0.491	0.491	0.016	-0.019	-0.69	-0.85
5	0.491	0.491	0.016	-0.019	-0.69	-0.85

**Distances in metres*

Table 2: Variable Descriptions

Variable	Meaning
X_S/Y_S	Initial position of robot in reference to global origin.
X_F/Y_F	Final position of robot with respect to starting position. In a coordinate system with origin at (X_S, Y_S)
X_{Odo}/Y_{Odo}	Final odometer values with respect to starting position. In a coordinate system with origin at (X_S, Y_S)

3) Test Analysis

Table 3: Euclidean Error

Trial #	ε
1	1.0904114
2	1.0904114
3	1.0904114
4	1.0904114
5	1.0904114

Table 4: Mean & Standard Deviation

X		Y		Error	
Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation
-0.69	0	-0.85	0	1.0904114	0

$$\mu_X = \frac{1}{N} * (X_1 + X_2 + \ldots + X_N)$$

$$\mu_X = \frac{1}{5} * (-0.69 + -0.69 + -0.69 + -0.69 + -0.69) = -0.69$$

$$\sigma_X = \sqrt{\frac{\sum_{i=1}^{N}(X_i - \mu_X)^2}{N-1}}$$

$$\sigma_X = \sqrt{\frac{(-0.69 - -0.69)^2 + (-0.69 - -0.69)^2 + (-0.69 - -0.69)^2 + (-0.69 - -0.69)^2 + (-0.69 - -0.69)^2}{5-1}} = 0$$

The standard deviation of X_{Odo}, Y_{Odo}, and ε represents the amount of variance around the mean that we can see in these values from trial to trial. As such these values will not provide any commentary on the accuracy of our odometer. However, the standard deviation of these values (0 for all three) does tell us that in the 'ideal' world of our simulation our odometer is very precise, and will return similar values of X_{Odo} and Y_{Odo} for each trial. Changes in standard deviation will be caused by changes in the

values of X_{Odo} and Y_{Odo} recorded at the end of each trial, and these values will change depending on the physical conditions in the environment the robot traverses.

We can judge the accuracy of our odometer based on the mean value of the error (ε). A mean error of ~1.1 indicates that we can assume our odometer reading to be an accurate reflection of position to ~1.1 cm in the 3x3 grid case tested. Since odometer error will increase based on distance travelled, this representation of error will not be a valid marker of the overall performance of the odometer. To better evaluate the performance of our odometer we assume the growth in error to be proportional to distance and can then describe the error of our odometer using an error percentage.

$$\delta_{mean} = \frac{0.0109}{4.09} * 100\% = 0.27\%$$

This determined error percentage for our odometer shows that the odometer we developed is highly accurate, and the previously determined standard deviation of ε indicates that the odometer is also incredibly precise. However, it is important to note that these trials & measurements were taken from test results obtained in a simulation, and that the real-world performance of our odometer will be affected by the physical conditions our robot operates in. This will most likely result in our odometer losing some of its accuracy and precision.

Our odometer sampling frequency is 62.5 Hz (1/Webots basic time step). Increasing this sampling frequency will result in a more accurate model of the path of our robot, as we will be breaking up the path into more straight-line segments. Theoretically, an infinite sampling frequency will be able to reduce the error in our model to 0. However, increasing sampling frequency will also increase the amount of calculations we will have to complete, and will impose limitations on the computing time available to other threads as sampling frequency gets significantly high. Another limitation when setting sampling frequency is the time required to complete the distance calculations, as well as the ability of the EV3 brick to represent small values. Decreasing sampling frequency will have the adverse effect of decreasing the accuracy of our odometer but will free more computation time for other threads. When selecting a sampling rate we must evaluate the tradeoffs of increasing accuracy and decreasing available computation power.

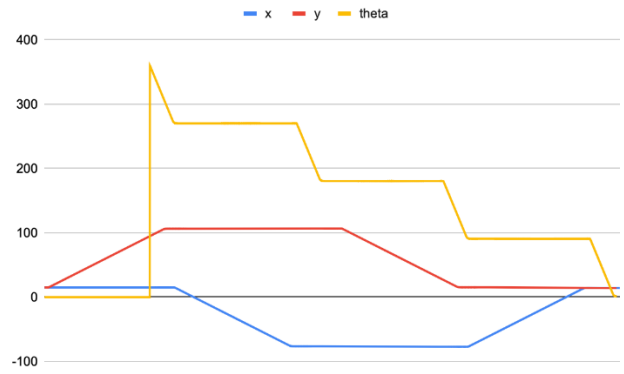

Figure 5: A plot of x, y, theta outputs from the odometer over one trial

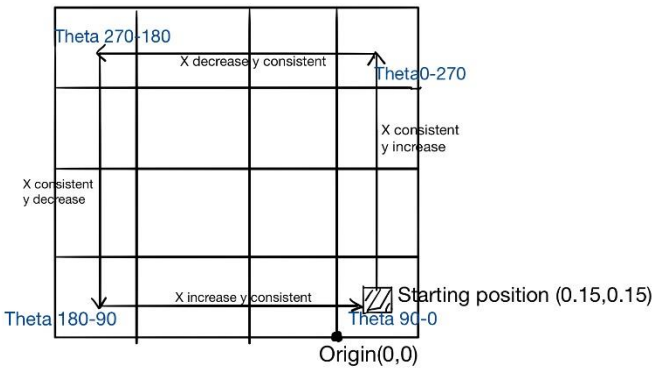

Figure 6: The path our robot follows in one trial

Figure 6 illustrates the x, y, and theta data for a trial where the starting position was (0.15, 0.15) with respect to the bottom left corner of the tile, and the robot drove in a 3x3 square. Figure 5 shows a plot of the x, y, and theta values for the same trial. The shape of the plot closely matches the behavior of

the robot in webots, however, there are still some inaccuracies, specifically when changing direction. The potential sources of error are the instability of the robot and the accuracy of the wheel base parameter, which may cause the robot to over/under rotate at the corners of its trajectory. Compared with a 'real' robot the simulation produced much higher accuracy in the measured odometer values. With a 'real' robot the friction between the tyre and the floor may differ over the course of the trajectory, which would cause error in the measured odometer value. Additionally, with a 'real' robot the two motors may not be identical (as our code assumes), which could also result in an error.

4) Observations and Conclusions

The calculated error percentage of 0.27% will yield a tolerable error for large distances. We expect that for larger distances the odometer error will grow approximately linearly. We can make this assumption since at each time step the position delta is calculated based on motor tachometer readings, then added to the total accumulated position delta. Since each individual position delta calculation is not dependent on the accumulated sum the existing accumulated error will not affect the calculation, resulting in an approximately linear growth of error. If the robot were to travel five times the distance it did in our test trials the observed error would be greater, as error accumulates over distance, but the odometer would still produce a reading accurate to within ~5cm.

Some helper methods are implemented in the SquareDriver class. For example, the moveStraightFor method moves the robot straight for the given distance; the turnBy method turns the robot by a specified angle; the convertDistance method converts input distance to the total rotation of each wheel; the convertAngle distance converts input angle to the total rotation of each wheel, etc. One benefit of helper methods is that they can reduce the lines of code we need to write. This helps to improve the readability of our code, while also reducing our workload. By putting often used formulas, conversions, or code in helper methods we can reduce the lines of code significantly. Writing helper methods to contain commonly used formulas also makes it easier to identify errors in the debugging and adjusting parameters process. With the helper methods, it was relatively easy to check if the error was a result of our code or if it could be fixed by adjusting parameters. Additionally, these methods can help clear up our software structure and make code easier to understand.

In order to test the logic for the odometer, we run multiple tests with different distances (in tiles) to be covered. By printing the odometer reading on the console it was easy to check if the measured odometer values matched the robots actual position in webots. After each cycle we also checked the difference between the measured ending position and 'real' ending position. By calculating the distance error between the 'real' value and the odometer value, we were able to show that our distance error was very small, proving that the logic for the odometer is correct. In order to test the logic for SquareDriver we first checked if the path our robot was driving was a perfect square using the 'real' position values shown in webots. Besides using the webots values, we also used the odometer program to track the path of the robot. The console print outs specified the horizontal and vertical distances and the change in heading for the duration of the test, and allowed us to verify our robot was travelling in a square. After verifying with both the console output and the webots position that our robot was driving in a square we were able to conclude that our logic for SquareDriver is correct.

We began implementing our software by implementing the SquareDriver class (without adjusting the parameters to ensure our robot drove in perfect squares) so that the robot could drive repeatedly in a similar path, helping us to test the odometer. Then, we implemented the odometer clas. After verifying the correctness of the odometer logic we adjusted the parameters in the resources class to modify the path of

the robot. Through running tests we also identified some problems in our hardware design, so we reinforced the base to help the robot drive more predictably. Finally, after adjustment and reinforcement, we ran multiple tests on different square lengths, verifying our code with error calculations between measured and ‘real’ values.

5) Further Improvements

During this lab we used the motor tachometer to calculate the distance and the heading of the robot. However, if we used a light sensor to implement or augment the odometry algorithm, the accuracy could be improved. We could add one light sensor anywhere on the robot, facing towards the floor. The sensor could be used to identify the grid lines on the floor, and could count the number of lines encountered. Assuming our robot is crossing these lines at a 90° angle we could use this information to compare measured position with ‘real’ position, and correct any errors that have occurred. We could also place two light sensors on the left and right side of the robot, facing downwards to the floor. In this way, when one sensor reached the line, the distance could be tracked by measuring wheel rotation until the second sensor reaches the line. Using this information we can calculate the true heading of the robot in reference to the line, and make error corrections to the value calculated from wheel odometry. Another benefit of using two light sensors is that it will be easier to ensure the robot is symmetrical about its centre, ensuring more ideal load distribution, and hence better handling.