ECSE 211

Lab 3: Navigation and Obstacle Avoidance

Connor Plante, 260708461

Jérémie Marquis, 260866909

February 7, 2019

Section 1: Design Evaluation

For the hardware design, we initially decided to make our robot wider than it was from the previous lab, since the turns would be more accurate compared to the calculated angle. This would be important moving forward, as we knew that the navigation would use the odometer values to determine the next move for the robot, and if the actual theta value and that from the odometer were largely different, the task would be unsuccessful. Effectively increasing the accuracy of the odometer would increase the navigation controller's accuracy when reaching the final destination. We also decided to minimize the complexity of the software by fixing the ultrasonic sensor at the front of the robot, aiming forward at 0 degrees. However, the robot's wheel would occasionally clip the edge of the block when driving towards it at a 45 degree angle , as the sensor would not be able to reliably detect the obstacle. In map 1, this case was observed when the robot would travel from point (2,0) to point (1,1) after navigating the obstacle. For this reason, we decided to build the robot again, but this time with a thinner driving base to ensure that it would not touch the block. We knew this would decrease the accuracy of our robot, as a smaller track variable became more finicky to work with, but navigating the obstacle successfully, 100% of the time was our priority. The final hardware of our robot can be seen in Figure 1 and to our surprise it is thinner than the design of the past weeks.

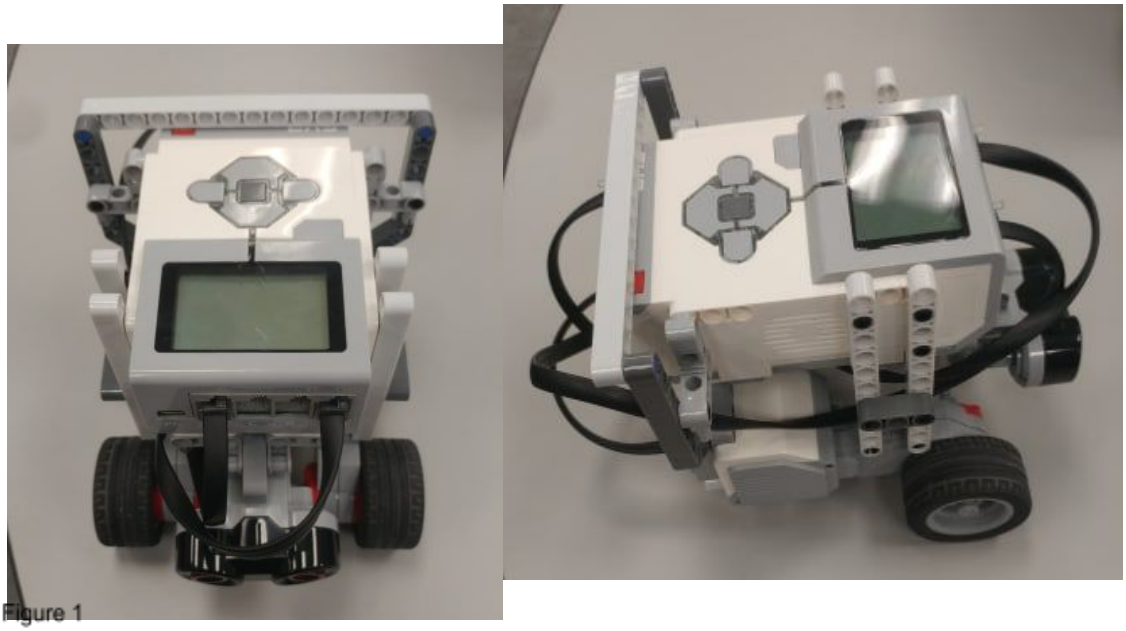


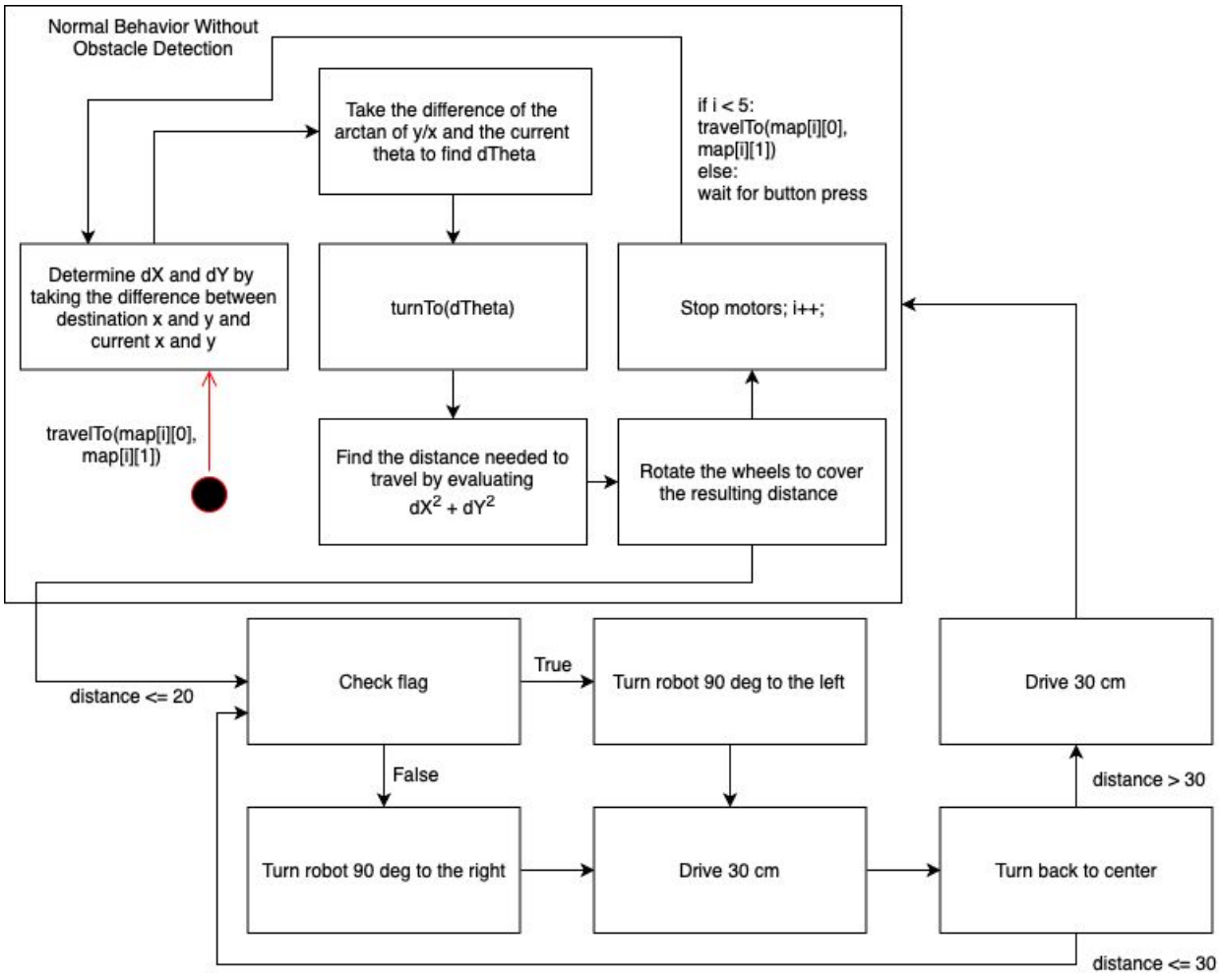
Figure 1

As for the software, the robot always runs the odometry and odometry display thread implementations used in lab 2. This is done in order to get the data needed to calculate the next waypoint, as well as to be able to display the odometer data visually for us to use during debugging. The map of the waypoints was hard-coded as an array of doubles and uploaded to the robot as specified in the lab instructions. A navigation thread is then used to physically move the robot.

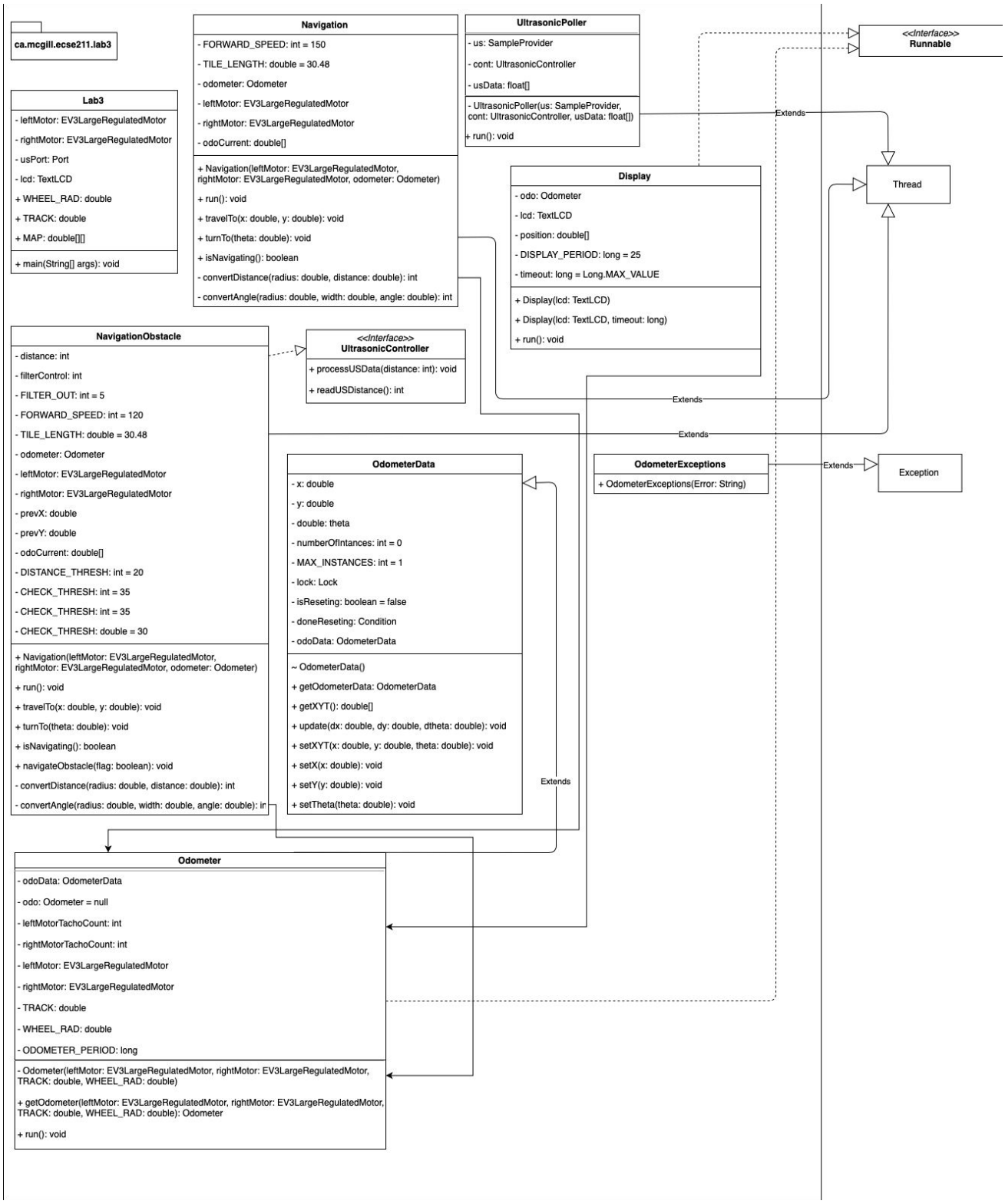
When determining how to travel to the desired waypoint, the robot uses four methods found in the navigation class: convertAngle, convertDistance, turnTo and travelTo. The two methods convertAngle and convertDistance were taken from the lab 2 SquareDriver implementation, and they convert an angle or distance (respectively) to the number of degrees needed to rotate in order to reach that distance (in the case of the angle, one of the wheels will rotate backwards). The turnTo method takes in a desired angle to rotate, and will rotate the robot in the direction that minimizes the arc length needed to reach the desired angle. In other words, if the robot is starting at 0 degrees, and needs to turn to 90 degrees, it will turn to the right rather than making a 270 degree turn to the left. The travelTo method takes in the destination x and y and then by using the current x, y and theta, computes the necessary angle to turn to and then travels the distance necessary to reach the point. When the navigation thread executes, it will use travelTo with each consecutive waypoint stored in the map array until reaching the final waypoint.

In order to account for obstacles, the program implements a different navigation class, NavigationObstacle, that extends the UltrasonicController interface and polls the ultrasonic polar every 50 ms for the distance. If the distance is below 20 cm while the robot is moving, it will execute a correction protocol by travelling 30 cm to the right and then 30 cm up before calling travelTo with the original desired x and y values to continue the course. There was an edge case where for one of the maps, turning right would cause the robot to fall off of the board, so a flag boolean was set and if it was true, the robot would turn to the left instead.

Software Execution Flow



Class Diagram



Section 2: Test Data

Table 1 : Final Position of the Robots for A Constant Path with the Destination Waypoint at (60.96, 0) cm

Trails	1	2	3	4	5	6	7	8	9	10
Xf (cm)	58.93	58.16	62.84	59.34	61.67	63.32	58.46	62.13	63.21	59.76
Yf (cm)	-1.86	-1.23	1.61	-0.96	3.16	-1.54	1.27	1.13	-2.54	-2.61

Section 3: Test Analysis

$$Euclidean\ error\ distance\ \varepsilon = \sqrt{(X - X_F)^2 + (Y - Y_F)^2}$$

Sample Calculation for the Euclidean error distance using map 1.

$$\varepsilon = \sqrt{(60.96 - 58.93)^2 + (0 - (-1.86))^2} = 2.75\ cm$$

Table 2 : Euclidean Error Distance Between the Final Position of the Robots and the Destination Waypoint

Trials	1	2	3	4	5	6	7	8	9	10
ε (cm)	2.75	3.06	2.48	1.88	3.24	2.82	2.80	1.63	3.39	2.87

$$\mu = \frac{2.75+3.06+2.48+1.88+3.24+2.82+2.80+1.63+3.39+2.87}{10} = 2.69\ cm$$

$$\sigma^2 = \frac{(2.75-\mu)^2+(3.06-\mu)^2(2.48-\mu)^2(1.88-\mu)^2(3.24-\mu)^2(2.82-\mu)^2(2.80-\mu)^2(1.63-\mu)^2(3.39-\mu)^2(2.87-\mu)^2}{10}$$

$$\sigma^2 = 2.82$$

$$\sigma = 0.53\ cm$$

Section 4: Observation and Conclusion

The Euclidean error distance average of (2.69 ± 0.53) cm that we observed during the navigation tests can be attributed mainly to the odometer. The odometer is used to obtain the position of the robot at any moment and the navigation uses these values as input to compute how and where the robot should go (in this lab, waypoints). If the navigation is taking in values that are inaccurate, the resulting destination of the robot will differ from the desired destination. There exist multiple sources of error regarding the odometer including the following: slippage of the wheels, the uneven weight distribution inside the EV3 brick as well as the imperfect board. The two wheels do not always experience the same friction due to differing residue on the tires and the percentage of weight of the brick that they support (a consequence of the asymmetrical center of mass of the robot).

The sources of error above cause inconsistencies in the tachometer counts for the wheels, and result in inaccurate data shown by the odometer. From Lab 2, we know that the odometer is not accurate without the odometry correction check with more than 2.50 cm off for a simple path. For Lab 3, we used the odometer without correction, and consequently, the navigator calculated the change in X-direction, Y-direction and angle with incorrect values which increased the error at each waypoint. Therefore we can say the navigation controller was not reliably accurate when moving the robot to its destination.

The robot would settle very quickly when reaching its destination. The method of navigating the robot consisted of turning to face the destination and then travelling in a straight line, therefore the only form of oscillation that could occur when stopping at the destination would be in the forward-backward direction. The stop() method provided by the lejos modules was also used to ensure that the motors would resist anymore motion when reaching the final waypoint. In addition to this, the motor speed was set relatively low to decrease stopping time.

The accuracy of the navigation would decrease when increasing the speed of the robot. From what was discussed previously, the robot's wheels would have more chance to slip, thereby increasing the error, whether it be when rotating, accelerating, or stopping at each waypoint. In addition to the increased chance of slippage, the robot would be more likely to experience a collision with the block, as if it is moving too fast, the ultrasonic sensor may not read the distance to the block enough times to pass the filter control for false positives before the collision.

Section 5: Further Improvements

To reduce our error, we could find a way the implement an odometry correction check like in lab 2. To do so, we would need to add the light sensor on the hardware and a thread to the software. For example, when the robot must travel in straight line over a black line on the board, the light sensor could be used to be certain that the robot is over the line. In addition, when the robot traverses the tile at a 45°, the light sensor could record the distance from the waypoint to the next waypoint. Using the Pythagorean theorem the robot could find exactly its X-position and Y-position. For instance as shown in Figure 2, the robot travels from (0,0) to (1,1) and the recorded distance is less than what it should be and the odometer computed that the X-position is less than the Y-position. The robot would set the Y-position to exactly one tile and the X-position would be calculated with the Pythagorean theorem using the distance record by the light sensor.

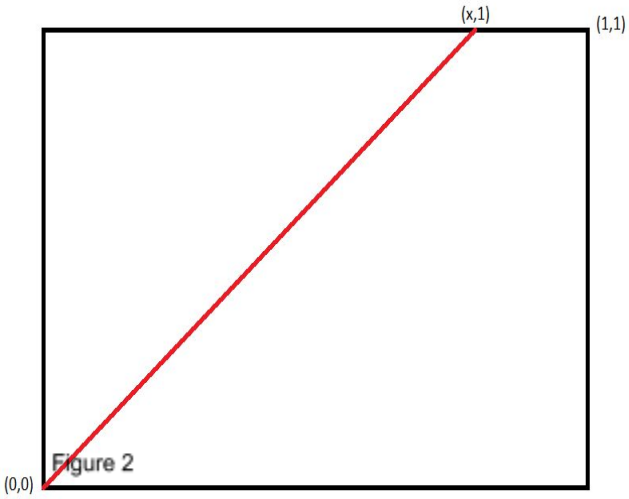


Figure 2

A hardware solution that does not involve adding sensors could be to reduce slippage of the wheels by attempting to optimize the friction experienced by them. This can be done by cleaning the wheels between runs to remove excess dust and dirt. Increasing the weight of the robot on the wheels, or lowering the brick so that it is closer to the motors and driving base could also potentially increase friction.

Regarding software, the setAcceleration() method could be used to decrease the rate at which the robot speeds up, thereby reducing the chances of any wheel slipping and increasing the accuracy and successfulness of the navigation.