Design Principles and Methods Michael Li and Cecilia Jiang

ECSE 211 (Fall 2019) 260869379, 260795889, Group #36

Lab 3: Navigation & Obstacle Avoidance

Prof. Frank Ferrie

McGill Faculty of Engineering

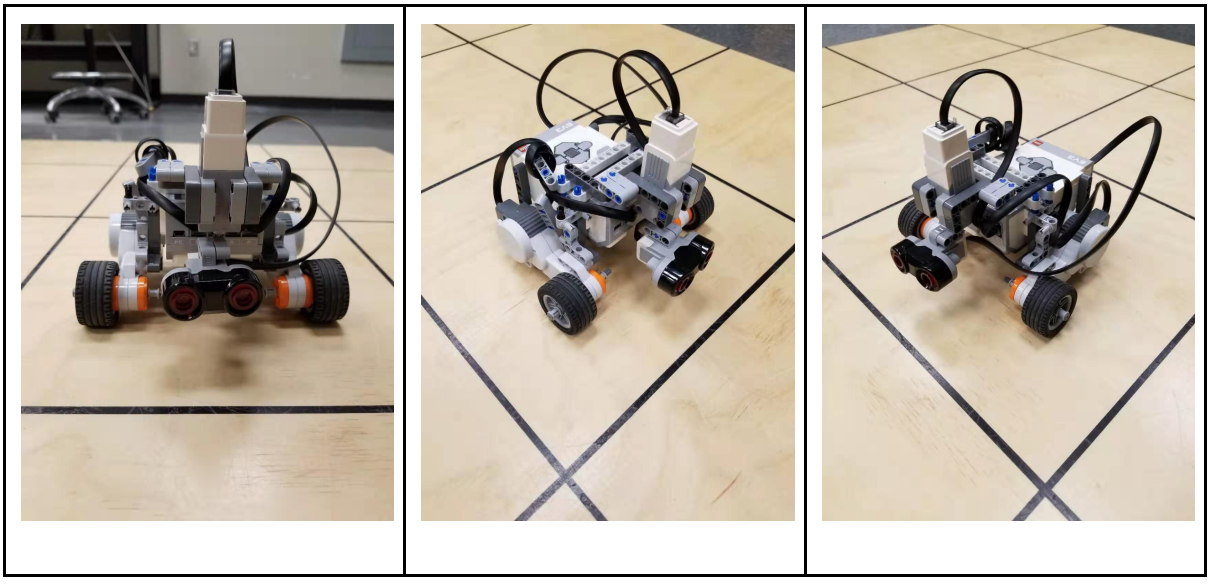
Due October 3rd, 2019

Section 1: Design Evaluation

1.1 Hardware Design

The robot was designed to have a basic, wide but strong structure shown in Fig. 1 below. The main design objective is to ensure a robust robot structure. Cross shape joints were used to connect the main brick to the wheels to prevent them from falling apart or experience any minor displacement. The two front wheels are assembled to two sides of the brick and we used a single rollable steel ball as the back wheel. In fact, the reason why the steel ball was chosen instead of the plastic wheel or a third wheel is that the steel ball can rotate in all directions Thus, this design decision can significantly reduce sliding and friction. Besides, the height of the steel ball is perfect. After attaching it to the rear of the brick, the brick remained parallel to ground level.

An EV3 Medium Servo Motor is installed in front of the “Brick” in order to allow the ultrasonic sensor to rotate while it is moving. The motor is oriented towards the ground because we wanted to install the ultrasonic sensor lower and closer to the ground. If the EV3 Medium Servo Motor was installed with its head (i.e. the part that rotates) towards the sky and its port towards the ground, the ultrasonic sensor would need to be installed on top of it which would be higher than the woodblocks. Therefore, we concluded that it was better to invert the orientation of the EV3 Medium Servo Motor and have its port pointing towards the sky to be able the install the US sensor closer to the ground (see Fig. 1).



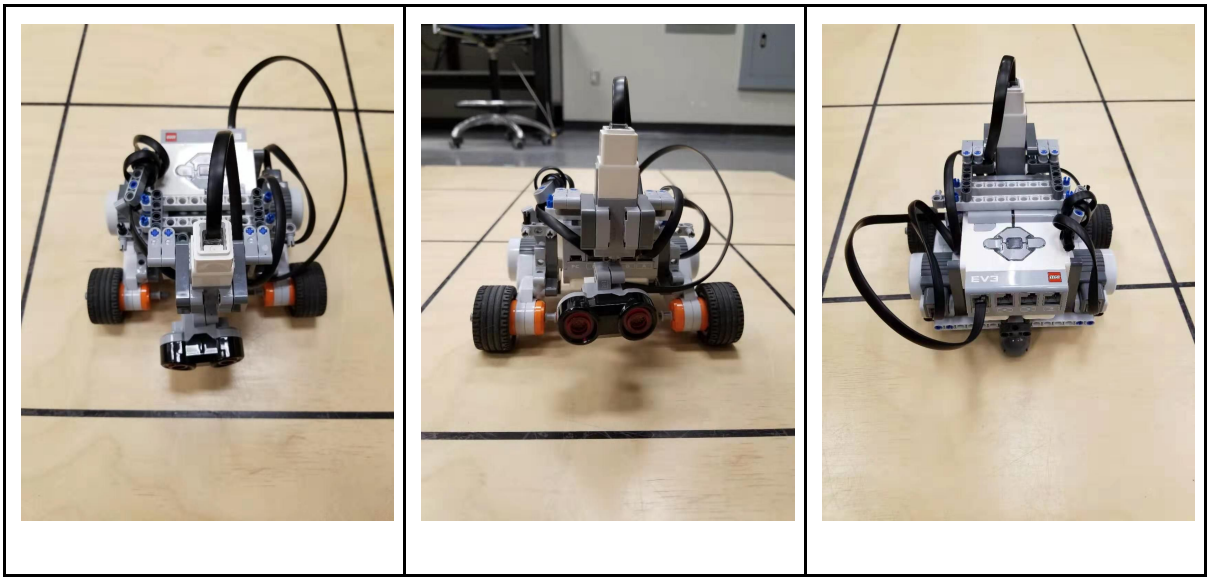


Figure 1: Robot's hardware design viewed from 6 different angles

1.2 Software Design

The software system was designed to complete two objectives:

1. Navigate to a specified set of coordinates, known as waypoints, where the robot's final position must be equal to the final waypoint's coordinates within an error margin of 2 cm.
2. Navigate to a specified set of coordinates, known as waypoints, while avoiding any obstacles in its path. The robot's final position must be equal to the final waypoint's coordinates within an error margin of 2 cm.

For the first objective, the odometer implemented in lab 2 was used along with a simple Navigation class. This objective was completed using two threads, one from the odometer class and another one from the Navigation class. The Navigation and NavigationWithObstacles classes both implement the singleton pattern. Therefore, there can only be one instance of the Navigation class or the NavigationWithObstacles class in this program. In addition, instances of these classes can only be created in their respective classes. A brief flowchart for Navigation class is shown below in figure 2.

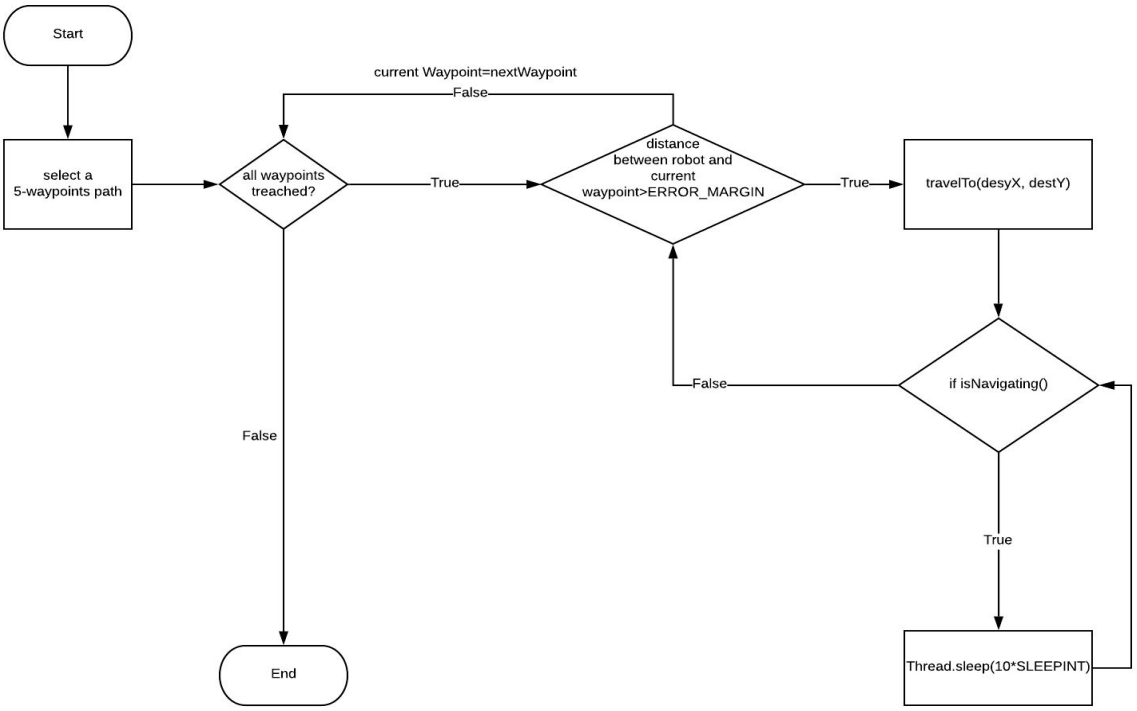


Figure 2: Flowchart for Navigation Class

The Navigation class contains three methods:

1. travelTo()

This method causes the robot to travel to the absolute field location (x, y), specified in tile points. First and foremost, it sets the navigator's traveling attribute to true. Then, it computes the difference between the destination's (x,y) coordinates and the odometer's current (x,y) coordinates and stores the differences as dx and dy. These differences are used to compute the distance required to reach the destination using the Pythagorean Theorem:

$$distance = \sqrt{((dx)^2 + (dy)^2)}$$

Afterward, travelTo computes the angle at which the robot needs to travel this distance. Using the atan2 and toDegrees methods from the built-in Java Utilities Library.

double theta = Math.toDegrees(Math.atan2(dx, dy)) - odometer.getXYT()[2];

The function atan2(y,x) takes as argument an x and y distance and computes:

$$\theta = tan^{-1}(x/y)$$

However, in our design, dx was passed in the "y" position and dy at the "x" position, because we want to compute the angle with respect to the Y-axis. Then, the

difference between this angle and the robot's current orientation is the angle needed to turn to get to this waypoint.

double theta = Math.toDegrees(Math.atan2(dx, dy)) - odometer.getXYT()[2];

Nonetheless, the theta from this formula is often bigger than 180 degrees or smaller than -180 degrees. Therefore, an "if" and "else if" statements were added in order to convert theses cases into the minimal angle. In fact, if theta is bigger than 180, travelTo will execute:

$$theta = 360 - theta$$

And if theta is smaller than -180:

$$theta = 360 + theta$$

Therefore, theta will always be the smallest angle to its target. Theta will be passed into turnTo method which will handle which way the robot must turn. After turnTo returns, the convertDistance helper method is used to convert the distance to the next waypoint into the number of revolutions for each motor. Finally, navigator's traveling boolean variable is set to false.

2. is Navigating()

This method returns true if another thread has called travelTo() or turnTo() and the method has yet to return; false otherwise.

3. turnTo()

This method causes the robot to turn to the absolute heading theta. It turns according to the angle passed as arguments. Helper method convertAngle is used to convert an angle to a distance for the BasicMotor.rotate() function. If the angle passed as argument is negative, it will make the robot turn left. On the other side, if the angle is positive, the robot will turn right.

For simplicity, the instance of the Navigation class will be called "navigator". When a thread from the Navigation class runs, it iterates over the selected path using a "for loop". At every iteration, it stores the destination waypoint's x and y coordinates as navigator's "destX" and "destY" attributes. Then, while the absolute value of the difference between the odometer's X or Y values and their respective "destX" and "destY" values is bigger than an error margin of 1 cm, the thread will call the travelTo() method. The navigator's destX and destY attributes are passed as arguments into the travelTo method. Finally, the Navigation thread is slept until the navigator is done navigating.

The implementation of the odometer is exactly the same as lab 2. At each iteration, the odometer stores the current tacho count of both motors. Hence, it is

able to track how many degrees each wheel has spun from its previous position. Then, this information can be used to calculate the distance traveled by both wheels using the following formula:

$$\pi * wheel\ radian\ * (currentMotorTachoCount - lastMotorTachoCount)/180$$

Moreover, the vehicle's displacement is obtained by computing the average of the distances traveled by both wheels (i.e. (distance traveled by the left wheel + distance traveled by the right wheel)/2). We will call the latter *d*. The change in orientation is also computed by:

$$\theta_{heading} = \theta_{old\ heading} + \theta$$

where θ is *d* divided by the robot's wheelbase. Afterward, the displacement in x and y is determined by using the following formulas:

$$x = d * sin(\theta)$$

$$y = d * cos(\theta)$$

Finally, the odometer will update the current position and the current orientation by doing:

$$x_{current} = x_{old} + x$$
$$y_{current} = y_{old} + y$$
$$\theta_{heading} = \theta_{old\ heading} + \theta$$

For objective number 2, the robot's runs one thread from each of the following class:

- UltrasonicPoller
- Odometer
- SensorRotation
- NavigationWithObstacles

The ultrasonic sensor pulls at an interval of 20 Hz. At each interval, the UltrasonicPoller thread fetches data from the ultrasonic sensor and stores the distance into an array that is passed into the PController's processUSData method. The PController has a proportion gain of 4 and is implemented exactly as in the first lab. The latter contains a state machine with 4 states, shown below in figure 2:

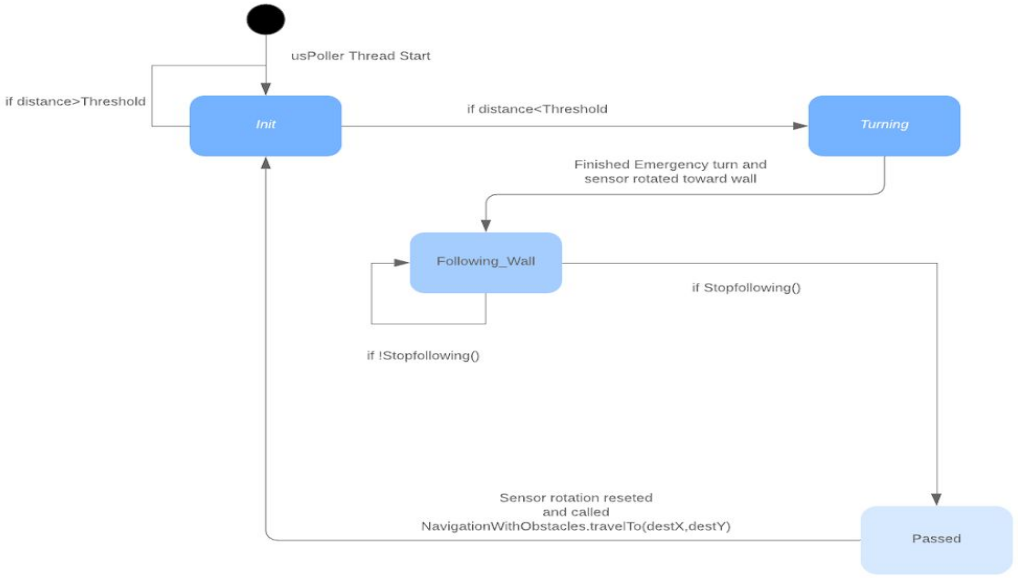


Figure 3: State diagram of PController's processUSData method

1. Init

The robot starts in the "Init" state. As soon as it detects an object whose distance is smaller than 20 cm, it changes its current state to "Turning".

2. Turning

The robot starts by getting its current (x,y) coordinates as well as its orientation from the odometer. Then, it makes a decision based on current orientation and which is the quadrant it is currently in.

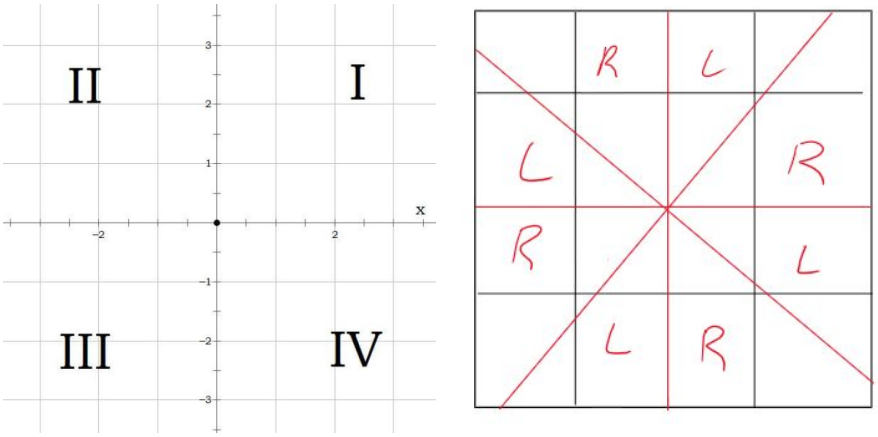


Figure 4: Scheme describing which way the robot will turn based on the quadrant

For quadrant 1, if the current orientation is within [210,360] OR [0,30], it will turn left. Otherwise, it will turn right.

For quadrant 2, if the current orientation is within [300,360] OR [0,120], it will turn right. Otherwise, it will turn left.

For quadrant 3, if the current orientation is within [210,360] OR [0,30], it will turn right. Otherwise, it will turn left.

For quadrant 4, if the current orientation is within [300,360] OR [0,120], it will turn left. Otherwise, it will turn right.

When the robot turns right, it will turn the sensor to the left in order to keep track of its distance from the wall. When to robot turns left, it will turn its sensor to the right, perpendicular to the wall. After turning, the robot will be in the state “FollowingWall”.

Note that during the whole obstacle avoidance process, the “traveling” variable in the NavigationWithObstacles class is always true, since the robot’s position is not on a waypoint. Therefore, the “while loop” in the run function of the NavigationWithObstacles class will call travelTo another time and set “traveling” to true. Then, the NavigationWithObstacles thread will sleep until the “traveling” variable is set to false.

3. FollowingWall

Based on the side on which it turned during the “Turning” state, the robot will activate the corresponding version of PController. If it turned right, it will follow the wall while staying on its right. On the other side, if it turned left, it will follow the wall on the left using an adapted version of PController. The PController computes the error given by the difference between the band center (18 cm) and its current distance from the wall. Then, the robot will make the decision on either turning right or left. If the absolute value of the difference between the angle of the next waypoint w.r.t. the Y-axis and the odometer’s current orientation is smaller than an error threshold (i.e. 5 degrees), the robot will be in the state “Passed”. Whenever the angle on the odometer will match the angle of the next waypoint w.r.t. to the Y-axis, it means that the robot is facing the next waypoint.

4. Passed

In this state, the robot resets its sensor back to looking forward and reset the EV3 medium motor tacho count. In addition, the robot turns the NavigationWithObstacles class’ traveling variable to false, stops both motor and switches the current state back to “Init”.

The Odometer tracks the robot's current (x,y) coordinates and orientation while it is moving.

The SensorRotation rotates the sensor of 40 degrees in the opposite side at each iteration (i.e. if the sensor is pointing left, if will rotate to the right and vice versa).

The NavigationWithObstacles uses the exact same logic as the Navigation class explained above for the most part. However, it contains an additional public setter for the static variable "traveling" in order to allow other classes such as the PController class to change this variable. In fact, the PController class needs to set it to true when it is following the wall and false once it is done.

Section 2: Test Data

Final Waypoint X-Coordinate X_d	Final Waypoint Y-coordinate Y_d	Final Position X-Coordinate X_f	Final Position Y-Coordinate Y_f
91.14	30.48	91.74	28.78
91.14	30.48	92.44	29.18
91.14	30.48	89.24	29.28
91.14	30.48	91.34	28.38
91.14	30.48	91.44	28.68
91.14	30.48	91.34	28.68
91.14	30.48	89.64	28.98
91.14	30.48	89.94	27.38
91.14	30.48	91.54	28.88
91.14	30.48	91.44	28.68

Section 3: Test Analysis

Error between final waypoint and real final position:

$$\varepsilon = \sqrt{(Xf - Xd)^2 + (Yf - Yd)^2}$$

Mean formula:

$$\mu = \frac{\sum_{i=1}^{n} \varepsilon^i}{n}$$

Standard Deviation Formula:

$$\sigma = \sqrt{\frac{\Sigma|\varepsilon - \mu|^2}{n}}$$

Error (ε, cm)	Error Mean(μ_ε , cm)	Error Standard Deviation (σ_ε , cm)
1.802776	2.06	0.48
1.838478		
2.247221		
2.109502		
1.824829		
1.811077		
2.12132		
3.324154		
1.649242		
1.824829		

Section 4: Observations and Conclusions

Q: Are the errors you observed due to the odometer or navigator? What are the main sources?

The errors are mainly due to the odometer, since the navigator relies on the values returned by the odometer to compute the angle and distance to reach the next waypoint. In addition, the robot travels further than its destination coordinates. In fact, it starts to decelerate once it thinks it arrived at the waypoint. Consequently, the robot travels further than necessary, because it doesn't break instantly. The track and wheel radius values vary depending on the wooden board on which the tests are executed. Therefore, there is no way to find a perfect track and wheel radius value that performs well on every wooden board. In our software implementation, there is a while loop that verifies the robot's current position against the destination waypoint's coordinates. However, if the values on the odometer contain error, then this verification is useless and imprecise. On the other hand, the threshold values set for this lab are not small enough (i.e. we allowed +/- 1 cm compared to destination waypoint when implementing the while loop above). Finally, the error on the odometer accumulates on each turn at each waypoint. In addition, when the robot is going around the obstacle, it adds a lot of errors on the odometer's orientation and current position.

Q: How accurately does the navigation controller move the robot to its destination?

Both Navigation with and without obstacle are relatively accurate. The navigation without obstacle gets a smaller error than 2 cm once every two trials. In fact we can see that the average of the error from the Navigation without obstacle is 2.06 cm. However, the Navigation with obstacles tend to be less performant as it can only get an error less than 3 cm once every three trials. In fact, it doesn't travel far enough on the last waypoint and tends to stop before arriving there.

Q: How quickly does it settle (i.e. stop oscillating) on its destination?

Except during obstacle avoidance, the robot doesn't oscillate at any point of the program. In fact, while the absolute value of the difference between the odometer's X or Y values and their respective "destX" and "destY" values is bigger than an error margin of 1 cm, the thread will call the travelTo() method again. Therefore, The robot will iteratively adjust itself on the waypoint if it detects that its current odometer's position is not equal close enough to the waypoint. However, the travelTo is never called a second time in the simple Navigation class, but tend to be called one to three times in the NavigationWithObstacle class. In addition, the robot doesn't oscillate since the Motor.stop methods' "immediateReturn" parameter is set to true and false respectively. Hence, the program will wait till the last wheel stops completly, before executing anything else. In this way, there is no oscillations.

Q: How would increasing the speed of the robot affect the accuracy of your navigation?

Increasing the speed of the robot could result in the robot slipping and therefore inducing errors in the odometer's values. Consequently, it will decrease the accuracy of the robot's navigation. If it is moving too fast and the threads are not fast enough to retrieve various information (i.e. the ultrasonic sensor perform at a 20 Hz rate), it could miss any threshold. For instance, if the robot is going too fast and the US sensor is not fast enough to retrieve the obstacle's distance, the robot could crash in the wall. Another example would be when turning around an obstacle, if the robot is turning too fast, it could miss the point at which its current orientation is equal to the orientation of the destination waypoint w.r.t. the Y-axis. Therefore, it would turn around the obstacle forever.

Section 5: Further Improvements

What steps can be taken to reduce the errors you discussed above? Identify at least one hardware and one software solution. Provide explanations as to why they would work.

For the hardware solution, it would be useful to install a light sensor at the rear of the robot where the distance between the sensor and the wheelbase (i.e. the rotation sensor) is the largest. The light sensor can also be installed close to the wheelbase in order to implement the odometer correction. Depending on how the software for the light sensor is implemented, the position of the sensor on the robot will change. Nonetheless, installing a light sensor will allow the robot to localize itself and correct its position and orientation according to the grid lines. In fact, we are able to have more information about the current location of the robot. For instance, if the light sensor is installed in front of the wheelbase, between the wheels, we are able to adjust the X and Y positions of the robot every time it detects a black line depending on its current orientation. Alternatively, if the sensor is installed on the rear of the robot, we obtain the greatest circumference possible each time the robot will be making a 360 degrees turn on itself (see software solution). Hence, the bigger the circumference is, the greater are the chances that the light sensor detects the black lines while turning.

The software solution consists of implementing the light sensor. There are two possible implementations; however, the first one will be preferable and selected. The first implementation thrives to use the light sensor as a localization tool. The light sensor will be used to detect the grid lines' positions relative to the robot. The idea is that at every waypoint, the robot makes a rotation with respect to its center of rotation (i.e. its wheelbase) and the light sensor has to cross every 4 grid lines as shown in Fig. 6 below. For this implementation, the light sensor should be located away from the center of rotation of the robot (i.e. wheelbase), as shown in the image below:

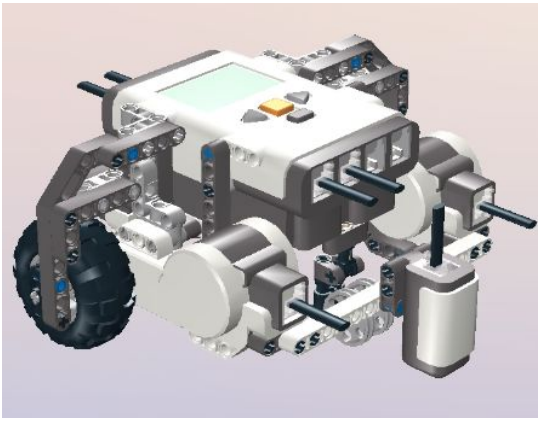


Figure 5: Position of the light sensor for software improvement

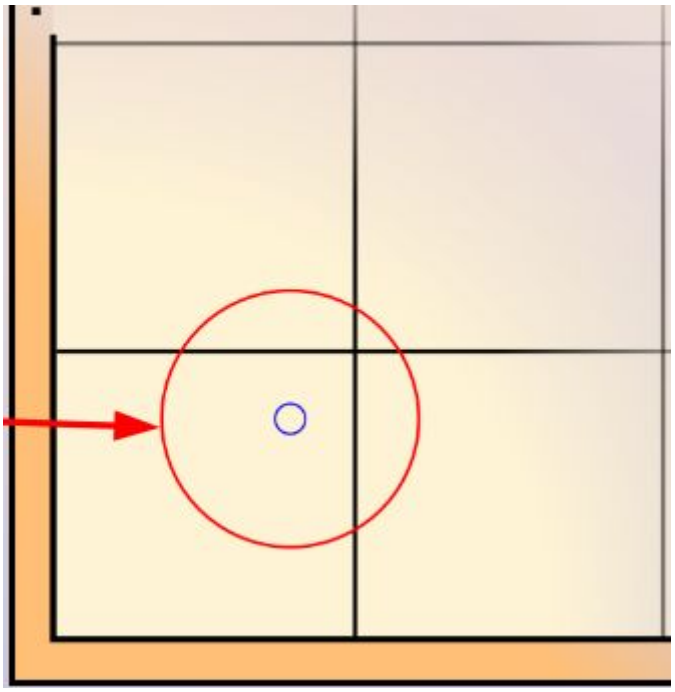


Figure 6: Path of light sensor when robot rotates w.r.t. its center of rotation.

Hence, it is possible to find the position of its center of rotation (see Fig 7).

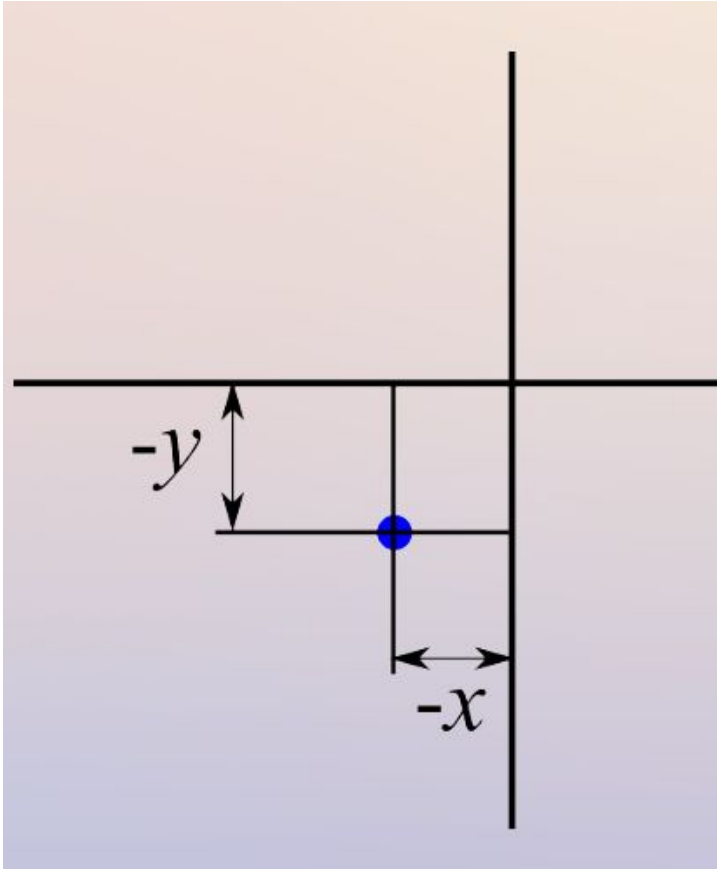


Figure 7: Location of the robot's center of rotation w.r.t. a waypoint.

The angle created by the arc connecting the intersections of the light sensor's path with the y-axis is called θy.

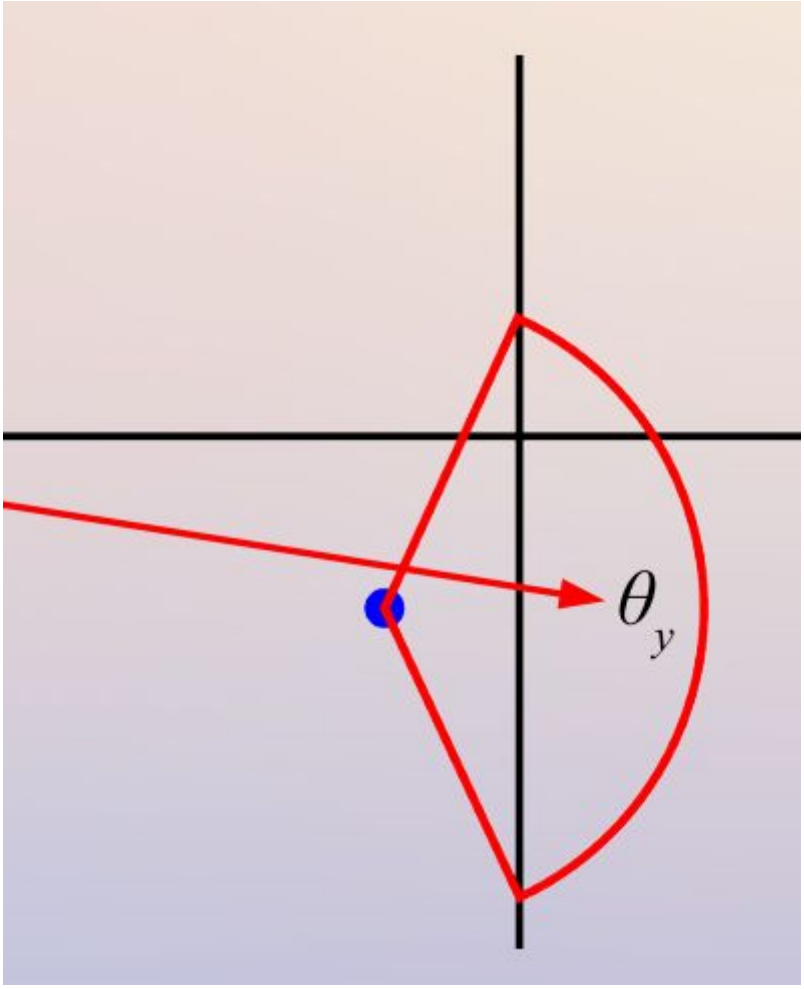


Figure 8: Angle θy create by the arc representing the intersection of the light sensor's path along the Y-axis.

It is possible to find the distance "x" between the robot's center of rotation and the Y-axis using the formula below:

$$x = -d\cos(\theta y/2)$$

where *d* is the distance from the light sensor to the center of rotation (see Fig. 9).

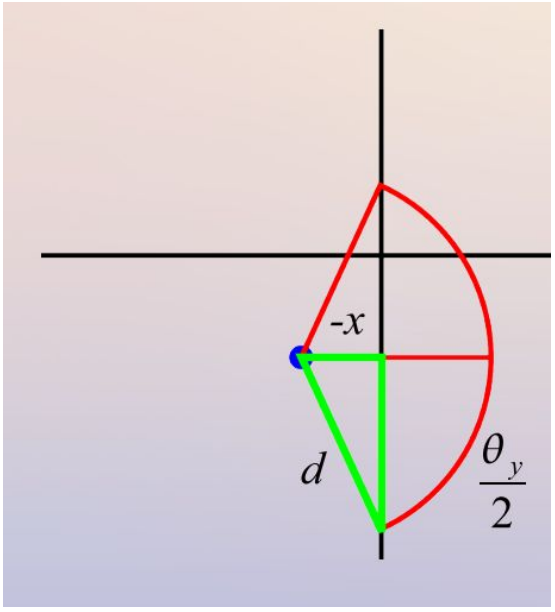


Figure 9: Finding the robot's x coordinate.

Similarly, for the X-axis, we define θx representing the angle formed by the intersection between the light sensor's path and the X-axis. Then, it is possible to find the distance between the center of rotation and the X-axis (see Fig. 10) using the formula below:

$$y = -d\cos(\theta x/2)$$

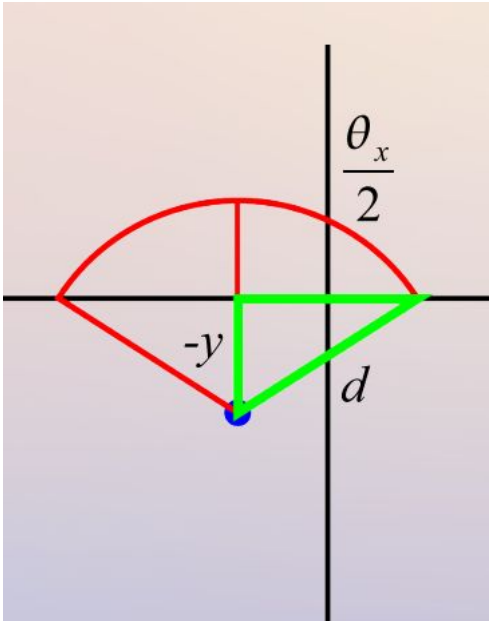


Figure 10: Finding the robot's y coordinate.

Finally, if θy is the heading of the robot reported by the odometer when the light sensor's path intersects the negative y-axis, then solving:

$$90 = \Delta\theta + (\theta y - - 180) - \theta y/2$$

for Δθ will yield the angle by which to correct the robot's heading (see Fig.11).

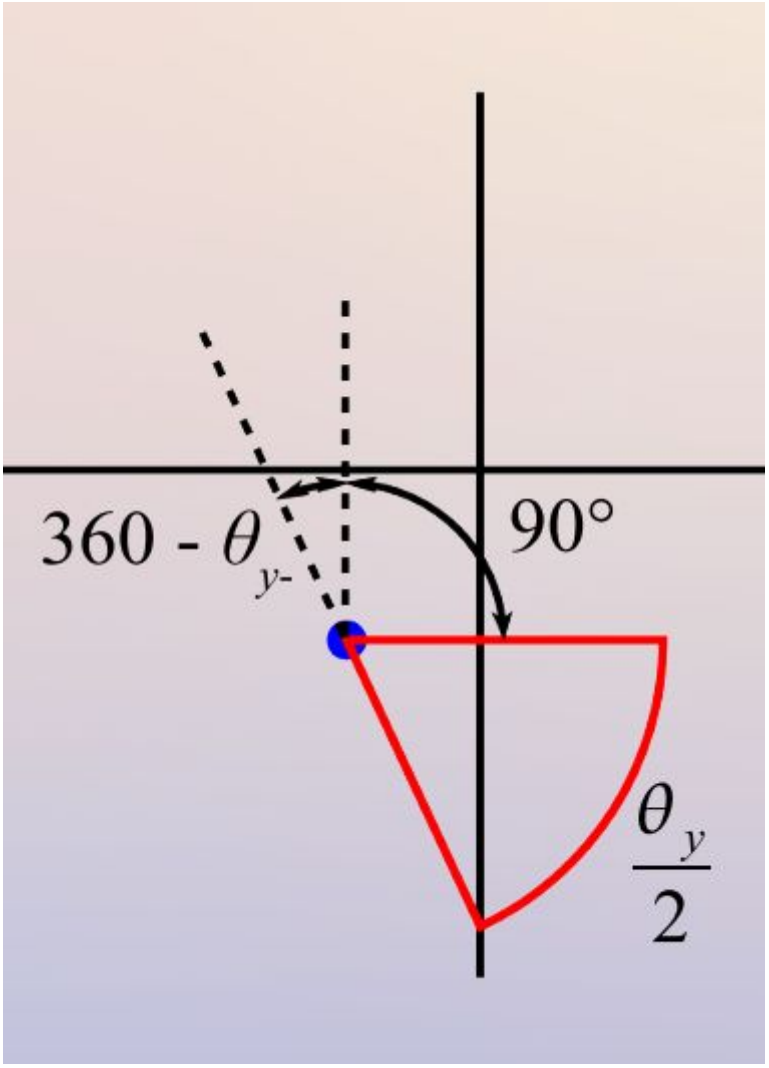


Figure 11: Correction of the robot's heading.

The second implementation is the odometer correction class used during lab 2. For this implementation, the light sensor would be placed in front of the wheelbase, between its wheels. However, this software improvement can only be used when the robot is not traveling diagonally. If the odometer's orientation is around zero (i.e. [350, 360] AND [0, 10]), then each time it detects a black line, we set the robot's Y to the value given by (z * TILE_SIZE where TILE_SIZE is equal to 30,48cm and "z" is the counter of the number of black lines detected). This counter

“z” will increase every time the robot detects a black line. Moreover, when the robot is traveling at an orientation of 180 degrees (i.e. in the other way on the Y-axis), the odometer will correct its position in the same way. However, this time the counter decreases each time it detects a black line when the orientation is around 180 degrees. The same logic is implemented for the X-axis. When the robot is traveling around 90 degrees, its X position is corrected to (x * TILE_SIZE) where “x” increases every time it encounters a black line. When the robot is navigating with an orientation of around 270 degrees, its X value will be set to (x * TILE_SIZE) with “x” decreasing every time it encounters a black line. Note that in this lab, the variable “x” and “z” would start at 1 every time since this point is the first waypoint of all 4 paths. In addition, it is also possible to set the robot’s X and Y values according to the closest value given by (r * TILE_SIZE where r is a discrete value between {1,2,3}) to its respective X and Y values.