Design Principles and Methods

ECSE211

Tony Ou

260867785

Hongtao Xu

260773785

# Lab 2 – Odometry

Group 27

# Design Evaluation

## Hardware Design

For the hardware part of our robot, we reused almost the whole design except the sensor. The base was stable and reliable enough to be kept. We first detached the ultrasonic sensor and removed its mounting parts. We then used a simple mechanism to hold the color sensor exactly at the middle of our wheels and at an appropriate height to get the best results. Finally, We added 2 supporting sticks on each side to ensure stability of the sensor during movement.

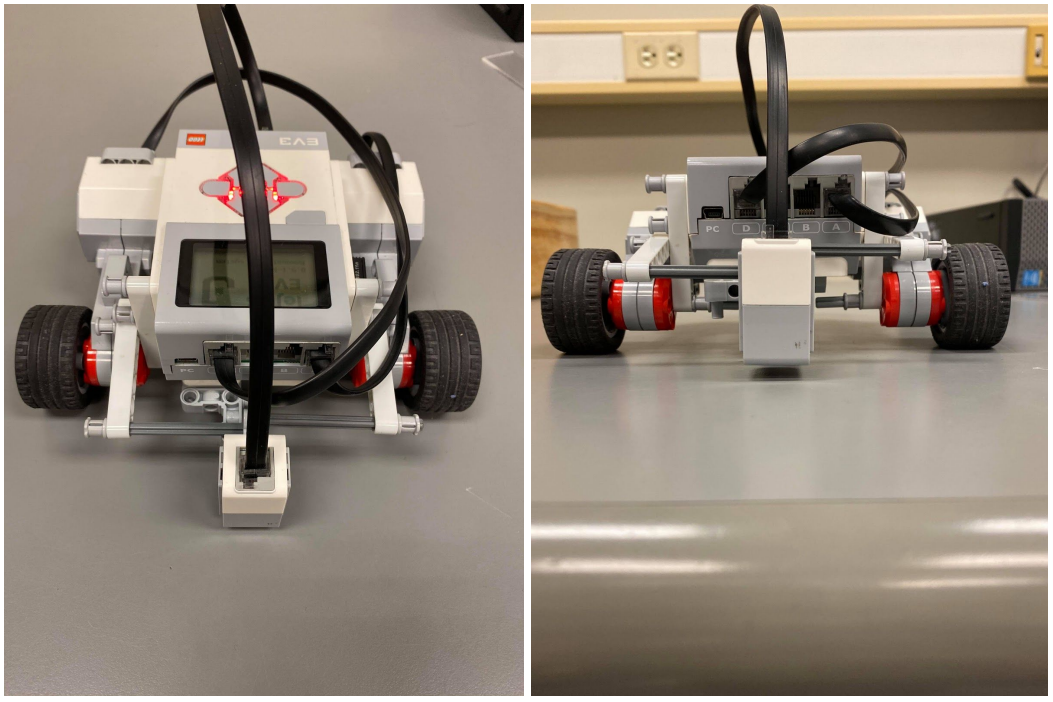

Figure 1.1 Final design of the Odometry robot

## Software Design

For the software design of Odometer.java, we implemented the code for the run function, which is executed at a frequency of ODOMETER_PERIOD. The code first calculates the displacement of each wheel using the current and last tacho counts. Afterwards, the last tacho count is updated with the new tacho count to compensate for the next time the code is executed. It then calculates the vehicle displacement (deltaD) and the change in heading (deltaT). The x/y components of the vehicle displacement are also calculated by getting the sin/cos of the current theta value in degrees multiplied by the vehicle displacement (deltaD). The vehicle displacement needs to be separated into components so that the LCD can update its X and Y values separately. Finally, the odometer is updated with new coordinates using the "update" helper function.

The software design of OdometerCorrection.java uses a completely different logic by using detected lines to correct the robot's coordinates. To detect lines, it compares the last and current values of the color sensor. If there is a large difference between the 2 values, it beeps to alert us that a line has been detected. Once a line has been detected, the current direction in degrees is calculated and a different statement is executed for each cardinal direction (North, South, East, West). If the robot is going north/south, the y line counter is incremented/decremented respectively. If the robot is going south/west, the x line counter is incremented/decremented respectively as well. Note that the y/x counter is decremented first before setting its new coordinates when going south/west (compared to setting the new coordinates before incrementing the counters when going north/east). This needs to be done because when the robot reaches the northmost position, the y/x line counter is at a value of 4, when it only traveled 3 lines (caused by the increment first logic). This explanation applies to both the x and y line counters.

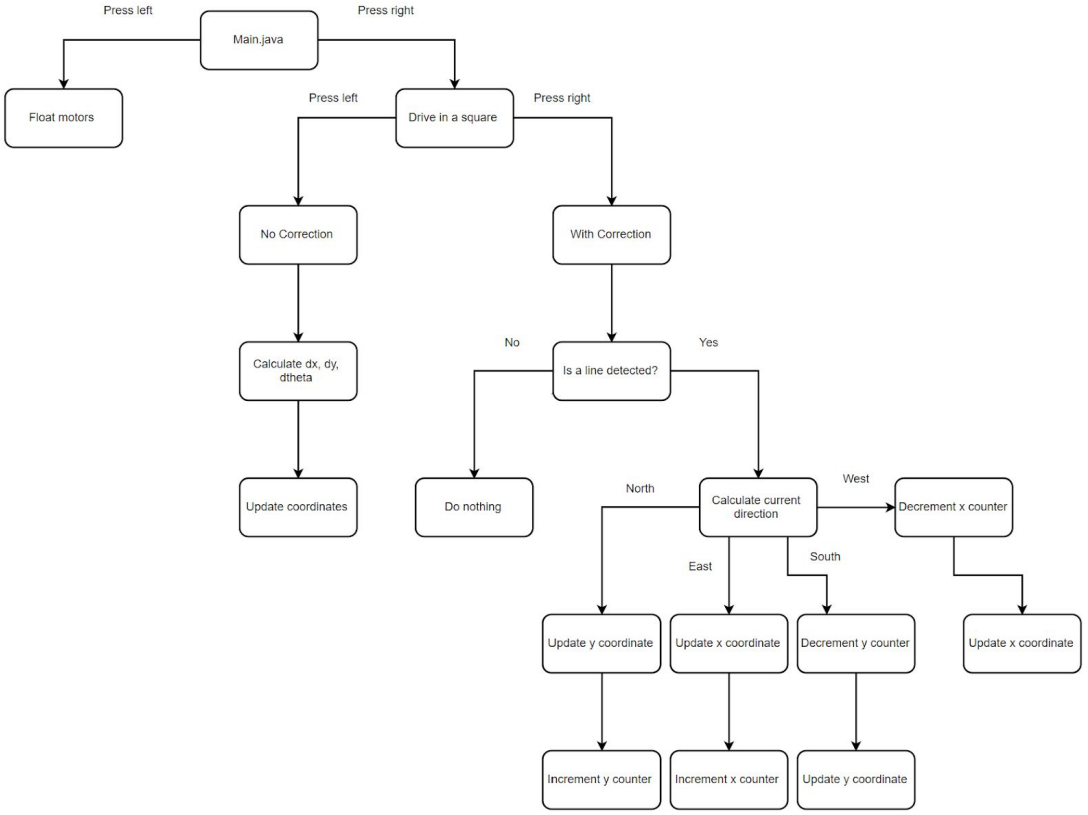


Figure 1.2. Software flowchart

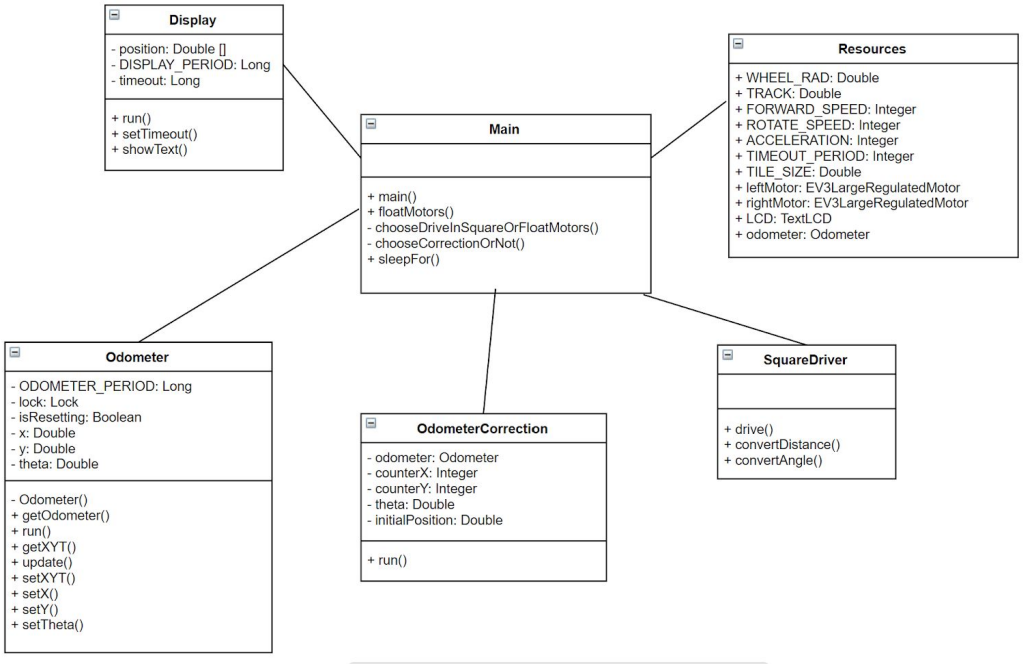


Figure 1.3. Software class diagram

## Workflow

To complete the lab, we first started by making modifications to the robot to replace the ultrasonic sensor with the color sensor, as it was quick and easy to do. We then implemented the code for the odometer by using the same logic in the slides and tested it in "float motors" mode to make sure the values were updating correctly. Afterwards, we ran the robot multiple times in square driving mode and updated the TRACK variable until it would turn at the right angle. After making sure the odometer worked correctly, we implemented the code for "odometer correction". Finally, we debugged the code after each test run until the x position, y position and theta error was at its minimum.

## Test data

Figure 2.1. Odometer test data

| Starting X (cm) | Starting Y (cm) | Final X (cm) | Final Y (cm) | Odometer X (cm) | Odometer Y (cm) |
|---|---|---|---|---|---|
| 15.5 | 15 | 16.5 | 13 | 0.122 | 0.97 |
| 15 | 16 | 14 | 18 | 1.428 | 0.29 |
| 14.5 | 17 | 13 | 18.5 | 0.770 | 0.09 |
| 14.5 | 17 | 12.5 | 17.5 | 0.18 | 0.32 |
| 14.5 | 17.5 | 12.5 | 19 | 0.095 | 0.01 |
| 15 | 16 | 13 | 17.5 | 0.17 | 0.08 |
| 15 | 16.5 | 13 | 16.5 | 0.18 | 0.47 |
| 15.5 | 17 | 13.5 | 19 | 0.511 | 0.160 |
| 15 | 17.5 | 13.5 | 19 | 0.111 | 0.08 |
| 14.5 | 17.5 | 13.5 | 19.5 | 0.772 | 0.30 |

Figure 2.2. Odometer correction test data

| Final X (cm) | Final Y (cm) | Odometer X (cm) | Odometer Y (cm) |
|---|---|---|---|
| 16 | 19 | 15.103 | 20.691 |
| 17 | 14 | 15.963 | 11.34 |
| 17 | 14 | 15.524 | 15.439 |
| 17 | 15 | 15.081 | 15.773 |
| 17 | 17 | 15.314 | 15.430 |
| 17 | 16 | 16.677 | 15.170 |
| 16.5 | 17.5 | 15.092 | 18.328 |
| 16 | 14.5 | 15.052 | 18.879 |
| 14 | 18 | 15.44 | 19.713 |
| 13.5 | 17.5 | 15.18 | 19.637 |

# Test analysis

Figure 3.1 Eucledean error distance e of the position (No correction)

| Run Number | Eucledean error distance e |
|---|---|
| 1 | 3.097061 |
| 2 | 2.969728 |
| 3 | 2.672265 |
| 4 | 2.187419 |
| 5 | 2.570822 |
| 6 | 2.593318 |
| 7 | 2.23009 |
| 8 | 3.112992 |
| 9 | 2.147492 |
| 10 | 2.455603 |

Figure 3.2 Eucledean error distance e of the position (Correction)

| Run Number | Eucledean error distance e |
|---|---|
| 1 | 1.914181 |
| 2 | 2.85499 |
| 3 | 2.061382 |
| 4 | 2.068838 |
| 5 | 2.3038 |
| 6 | 0.890634 |
| 7 | 1.633416 |
| 8 | 4.48044 |
| 9 | 2.237849 |
| 10 | 2.718303 |

Figure 3.3 Mean and standard deviation of X, Y and e (No correction)

| Variable | Mean | Standard dev |
|---|---|---|
| X | 0.4339 | 0.439245 |
| Y | 0.277 | 0.282373 |
| e | 2.603679 | 0.362948 |

Figure 3.4 Mean and standard deviation of X, Y and e (Correction)

| Variable | Mean | Standard dev |
|---|---|---|
| X | 15.4426 | 0.517286 |
| Y | 17.04 | 2.886896 |
| e | 2.316383 | 0.938579 |

How do the mean and standard deviation change between the design with and without correction? What causes this variation and what does it mean for the designs?

The means of the X and Y values of both designs are very different because when it has correction, the correct coordinates are displayed (relative to the origin at the corner of the platform). On the other hand, when it does not have correction, it displays the coordinates relative to its starting position, which it thinks is the origin. The mean of the error is also slightly lower for the design with correction, which means that the design is slightly more reliable. The standard deviations for the design without correction are also lower than the ones with correction. This means that the odometer fluctuates a lot less when it has no correction.

Given the design which uses correction, do you expect the error in the X direction or in the Y direction to be smaller?

We expect that the error in the Y-direction (North/South) is smaller than the error in the X-direction (East/West). From our observations, we found that the wheel slippage of the robot usually occurs when it turns, which leads to errors on the angle of theta.

Change of x displacement = deltaD * sin(theta)
Change of y displacement = deltaD * cos(theta)

From the formula above (calculations of the changes in x and y displacements), we know that the angle theta affects the x and y components of the displacement. When the robot is travelling on the platform for one round, the robot turns twice on the x axis and only once on the y-axis (the last turn does not count because it is not travelling). Therefore, there are less turns on the y-axis, which leads to a smaller error.

## Observations and conclusions

Is the error you observed in the odometer, when there is no correction, tolerable for larger distances? What happens if the robot travels 5 times the 3-by-3 grid's distance?

When there was no correction, the error we got from test runs was still tolerable (around 1-2 cm error) for smaller distances. However, if the robot travels large distances, the error would inevitably increase to intolerable levels. For instance, if the robot travels 5 times the 3-by-3 grid's distance, the error would increase to about 5-10 cm, which would be problematic, as it could easily miss an entrance or crash into a positioned obstacle.

Do you expect the odometer error to grow linearly with respect to travel distance? Why?

We get the position of the robot by adding the changes of distance in x-direction, y-direction and theta to their to the previous value.

Vehicle displacement (deltaD) = pi * WR * (L+R) / 90 deg.
Change of x displacement = deltaD * sin(theta)
Change of y displacement = deltaD * cos(theta)
Change of theta = (pi * WR * (R-L)/90) / WB

From the upper formula, we can see that both x and y values are related to the angle theta. We observe that the slip usually happens when the robot turns, which causes an initial small error on the angle theta. If the robot keeps travelling in the same direction, the deviation in angle will increase linearly. As a result, the odometers' x and y coordinates will also increase linearly with respect to travel distance because the angle is not corrected. Therefore, we expect the odometer error to grow linearly with respect to travel distance.

## Further improvements

Propose a means of reducing the slip of the robot's wheels using software.

We could reduce the FORWARD_SPEED and ROTATE_SPEED values so that the robot travels at slower speeds, which would increase the friction with the wheels. More friction will reduce the chances the the robot slips and result in a more precise trajectory.

Propose a means of correcting the angle reported by the odometer using software when:

The robot has two light sensors.

We could place both sensors on the same axis in front of the robot so that each time a line is detected, both sensors will react to the event. If both sensors do not read the line at the same time, it means that it is not going perfectly straight. We could therefore adjust its trajectory by changing the speeds of each motor. To check the timing of both sensors, we can set a small

error threshold and if the difference of both detection times exceed that threshold, it would mean that the angle is not correct.

The robot has only one light sensor.

When the robot is going in a specific direction, we could set an error threshold and if the difference between the timings of successful detections exceeds that threshold, it would mean that the angle is not correct, as the robot is not going straight. We could then correct its pathing. We can also make the software start correcting the direction only after the first line is detected to make sure that it will travel full tiles.