Viraj Patel 201351707

COMP 329 Mobile Autonomous Mobile Robotics Programming Assignment

Project Report

In order to have the robot navigate around the arena, I decided to implement Wall following with PID controllers. I also decided to implement the solution using Python, as I prefer the Python syntax, over Java's syntax. At the beginning of development, each run of the simulation would run extremely slow, due to the creation of a Occupancy grid in python, in order to overcome this issue, I changed the number of cells per meter from 20, to 10 (*just for the purpose of development, this was changed after prior to submission*).

The wall following algorithm I've implemented, uses a PID controller to modify the speeds of each wheel using the difference between the intended and observed distance. I created a method called (**set_velocity**) in this method both wheels are assigned a base velocity, and If the wheels exceed the maximum velocity then corrections are made to compensate for this.

I implemented a PID (*proportional-integral-derivative*) controller. This was used to make small adjustments to each motors velocity in order to achieve a smoother movement of the robot when navigating the arena. . the PID controller also ensures that the robot stays a certain distance from the walls. I also had to ensure that the **navigator class had** an instance of the **proximity sensor class**. This was done as follows:

```
my_pose = pose.Pose(0.0, 0.0, 0.0) #nav.get_real_pose()
prox_sensors = pps.PioneerProxSensors(robot, "sensor_display",
nav = pn.PioneerNavX(robot,my_pose,prox_sensors)
```

I initially had difficulty creating an instance of the proximity class in the navigator class, as the initial value for my_pose, used the variable nav before it was instantiated, therefore I had to change the **my_pose to (0.0,0.0,0.0)**. After doing so, I had to declare the instance of prox_sensors in the navigation class.

The created function **(follow_wall)** so that the robot would run parallel to a pre-determined wall, either the wall to the left of the robot, or the right. The robot would maintain a set distance to the wall based on the values fed in from the PID controller. In the **follow_wall** function the robot has a set list of rules to follow for the following scenarios*,(wall has ended, robot approaches a wall, robot is following wall).* For the case when the robot is approaching a wall/obstacle, the proximity sensor values have a set minimum distance (*distance from wall)* if this minimum distance is realized by a sensors, the robot will turn away from the wall. In my function the **direction_coefficient** variable determines weather the robot will follow the left , or right wall. If the Right wall is chosen the value is set to -1 , and 1 if the left wall is chosen.

Now that a new navigation method had been added, I had to add another Enum for my wall following states, which was done as follows :

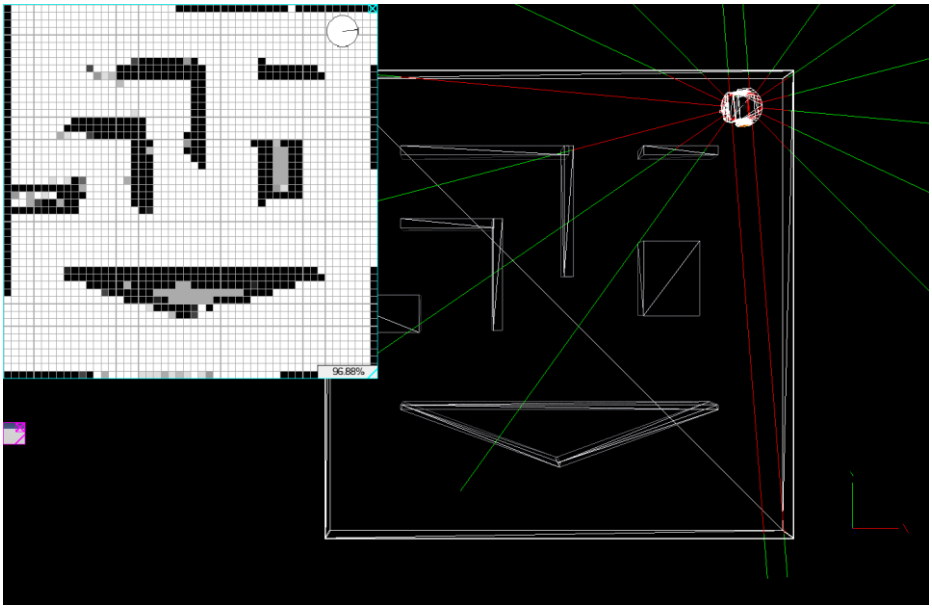
```
13 class MoveState(Enum):
14     STOP = 0
15     FORWARD = 1
16     ARC = 2
17     WANDER = 3
18     FOLLOW_WALL = 4
```

COMP 329 Mobile Autonomous Mobile Robotics Programming Assignment

In the controller class I then changed the **move.state** from Wander to **follow_wall** so that the robot was assigned the function **follow_wall**. When calling the function in the control loop, I had to play around with the values fed into the function to ensure the robot would scan the whole arena. After lots of trial and error; I found that 0.252 was the optimum value. As shown below.

```
print(timestep)
nav.follow_wall(robot_velocity, 0.252, False)
```

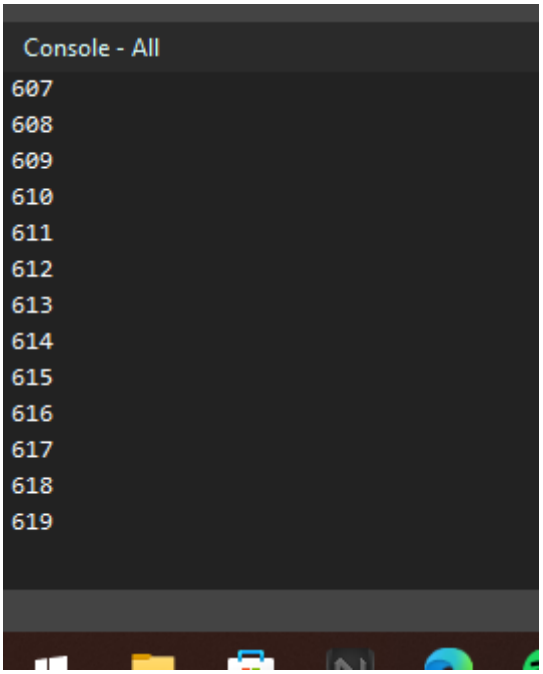
Implementation of this allowed for my robot to scan 97% of the arena. within 3 minutes. However no I was faced with the challenge of making the robot stop in the top right corner of the arena. I decided to approach this by noting down the time it took the robot to map the whole arena and when it next returned to the top right. I then aimed to stop the robot when this time had reached. As seen below is a screenshot after the robot had scanned the whole arena using the implementation outlined in this document.



I initially attempted to use the value of the timestep to see when I should stop the robot, however when I would print this value, the same number would appear with each control loop. Therfore I implemented my own method of determining the number of loops required to reach the desired location after the whole arena was mapped. I created a variable called **end_time** with an initial value of 0, and after each iteration of the control loop this value would be appended by one. This provided me with a way to measure the simulation time in terms of number of control loops / second. In order to visualise this. I printed this value into the console, as shown below :

Viraj Patel 201351707

COMP 329 Mobile Autonomous Mobile Robotics Programming Assignment



Once the robot had reached its desired location I would pause the simulation and take note of the **end_time** value. The 2207 After the 2207 has been reached by the variable **end_time** I call the **STOP** function to stop the robot, as it has scanned 97 % of the arena, and reached its desired end destination.