Assignment One: Flocking
COSC 69, Spring 2019
Oliver Levy

Description

The robot's behavior is controlled exclusively by the program *robot.py*. This program is called in each robot's namespace by the node *start_node. start_node* has no other function. *Robot.py* has two subscribing objects. s*ub[]* listens for the positions of all the robots, while *sub_me* listens for the robot's position whose namespace *robot.py* was called in (called node robot from here on). There is one publisher to the cmd_vel topic of the node robot. *Sub[]* is responsible for calling *heading_point. heading_point* determines the point to which the robot should go. When the program receives a location message, it: updates the directory containing all robots positions, calculates the average position and orientation of all the robots within 6 meters of the node robot, and creates a heading point by projecting the average collective position 4 meters in the average collective direction. It saves this point in the heading attribute of the program.

When *sub* receives a position message, it calls *find_trajectory*. *find_trajectory* updates the attributes x, y, and theta, which store the robot's x position, y position and orientation. Then the program computes the linear and angular velocity needed to reach the heading point. If *safety* is not true, then the function sets cmd_msg equal to the linear and angular velocities computed in the previous step. If *safety* is true, *find_trajectory* sets *cmd_msg* to have no linear velocity and a set angular velocity (the robot would turn in place).

Since the cmd_msg was an attribute of the class *Robot*, I realized that one function needed to have control over changing it. Otherwise, the robot could have received conflicting messages from different programs updating *cmd_msg.* I put this control in *find_trajectory.*

Whenever the program receives a laser scan message, it passes it to *safety_call*. This program then see if there are any objects in front of it within one meter. If there are such objects, then the program sets the robot to turn to the left at a constant rate. Originally, the program only calculated if the minimum value from the laser_scan message was within the "danger" zone. However, this lead to the scenario where the minimum value was off to the side of the robot yet there was another obstacle right in front of it that was not considered.

Evaluation

All considered, the program is effective. Robots correctly show alignment, cohesion, separation and obstacle avoidance. Due to the obstacle avoidance mechanism, robots soon get separately, and eventually this prohibits them from acting as a true unified group.

Obstacle avoidance is a simple mechanism where if the robot sees an object within one meter and within the angle range $\pi/2$ and $-\pi/2$, the program will stop the robot's linear velocity and turn until it sees no object in front of it. While this prevents the robots from hitting objects, it also makes robots move in a choppy pattern. The behavior also does not account for the angle at

which the object is seen. A more efficient behavior would calculate the angle between the robot and the object and adjust linear and angular velocity appropriately.

Cohesion and alignment are bundled together in the mechanism for tracking a heading point. While efficient in theory, since the obstacle avoidance mechanism requires the robot to stop, having a single point for robots to go to means that they must eventually stop or hit each other, which interrupts the smooth movement of the robots.

I created two subscribers to be able to distinguish between the position information of this robot with the reset of the robots. While having two subscribers fulfilled my purpose, it was also inefficient because the program was double subscribing to one robot's position.

Allocation of Effort

I received help from Ben and Jennifer regarding algorithm ideas and bugs within my code. However, all of my code is strictly my own.