Design Challenge 5: The IR Olympics

Due: November 6, 2014

1 Goal

The goal of this design challenge is to program your team's robots to compete in two of the following IR Olympics challenges, then present your design to the class. Both robot events and presentations will be scored by our panel of judges. The top team wins fabulous prizes!

2 The Rules:

1. You will form three teams of at least 4 and at most 5 students. Elect one person to be "team communicator". This person is not in charge, but will coordinate communication between teams and staff. The staff reserves the right to move students between teams for balance.
2. Each team selects two challenges. As a class, there can only be two teams doing any one challenge, so some coordination is required.
3. You will write your programs as a team, but each person needs to have their "part" of the code. It might be useful to use Google docs (or Git) to make it easy to share software within your team. Each team's robot will run the same code during the challenge.
4. You must start your robots in a random position and orientation within the 1-meter start circle.
5. Each event is only run once, so you only get one scoring run.
6. Your presentation of your two designs must be less than 2 minutes, just tell us the core of the solution. No powerpoint, but you can plan to use the white board.

3 Event Descriptions

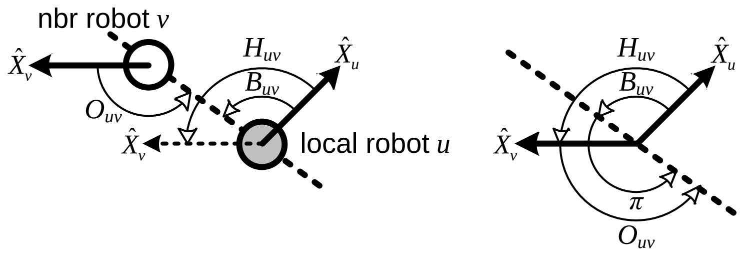
Each team needs to participate in two events. But each event can only have two teams in it. The team communicator will need to coordinate with the other teams to arrange this. Use the LEDs to indicate the status of your code.

3.1 Sorting:

Goal: Get all the robots on your team to follow each other, in ID order of robot ID. But, you cannot program specific IDs for each robot to follow, it must be dynamic. The lowest ID robot should drive straight and avoid walls. The staff will remove robots at random to see if your code self-stabilizes to the correct order. Staff will provide a basic wall-avoid behavior, or you can make your own based on the PS04 code.

3.2 Flocking:

Goal: Get all the robots to face the same direction and move as a group. When flocking, you want to match the average *heading* of all of your neighboring robots. Geometry for computing the heading is shown in the following diagram:



Once you compute the heading, you can use the follower RV controller to rotate towards the average heading, while moving at constant velocity. Use a small velocity, around $50\frac{m}{s}$, larger velocities will work, but will be harder. If you want to be especially clever, you can ad wall-avoid to this behavior too, but be sure to test flocking by itself first.

Each neighbor needs to compute the *heading* of each neighboring robot. Use `neighborsX.get_nbr_orientation()` and `neighborsX.get_nbr_bearing()` to figure out the heading of a neighbor in that robot's coordinate frame— the way the leader is facing relative to the direction the follower is facing. Compute the error in heading, the put this error into your equation to compute RV. Ta-da: Matching heading!

3.3 Orbiting:

Goal: Get all the robots to orbit one of the robots. We assume that we start in an position so that all the robots can see all the other robots. The robot that is being orbited is the pivot, and it must be the robot with the lowest ID. All other robots must orbit this robot. This must be self-stabilizing, meaning that if we re-arrange the robots, they should still settle into an orbit around the pivot.

Want to impress us? Figure out what to do if a robot can't directly communicate with the pivot robot. Hint: solving this might look a lot like the sorted follow event from above.

4 Hints:

1. We've given you a function to sort a list of robot's neighbors by ID. This may come in handy when, well, writing something that needs to sort neighbors by ID.
2. The math for computing the heading of a robot in your local coordinate frame will be useful for flocking. We will do this math on the board. I would write a function of the form:
   `def compute_heading(bearing, orientation)`
3. Understanding how to average angles will be useful for flocking. Note that this is exactly what you did to compute bearing and orientation in PS07, but abstracted to take a list of angles as input. I would write a function of the form:
   `def average_angles(angle_list)`
4. If you use a RV controller for orbiting, you might need a bit of offset to make the robots turn a little more than they would otherwise. This is expected because of the lag in the neighbor system. (huh? If the lag comment isn't clear, the need for offset will be clear when you write the code.)
5. Start testing early. Figuring out some of these events will take a while. Do early runs to figure out how well your algorithms are working.