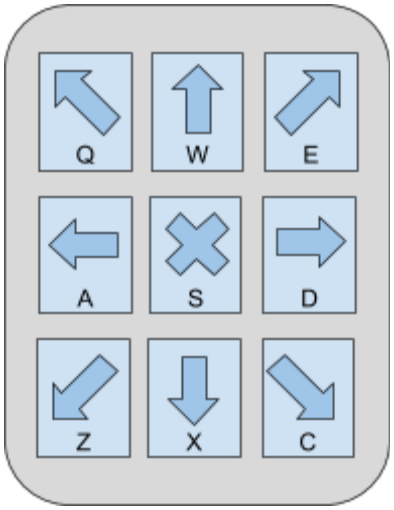
Computational Robotics: Warmup Project

Mary Keenan
9/21/18

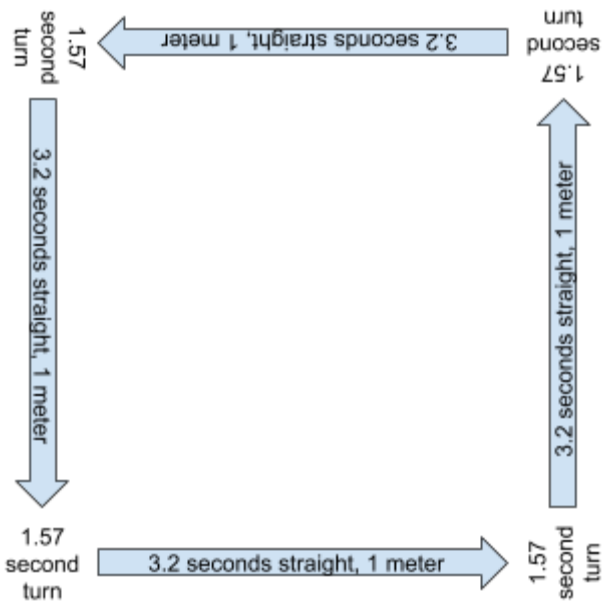
Behaviors

Robot Teleop



- *Description* -- The robot freezes as well as moves forward, forward-left, left, backward-left, backward, backward-right, right, and forward-right in response to key presses in the command window running the program.
- *Diagrams* -- To the right, there's a mapping of key presses to the direction in which the robot moved.
- *Tricky Decisions* -- I decided to have the robot keep moving in the direction of the last key press even when the key isn't being pressed since it seemed easier to pilot, but this can be annoying if you exit the program before pressing the key to freeze the robot (it will keep moving unless you break the connection or rerun the program and stop it).

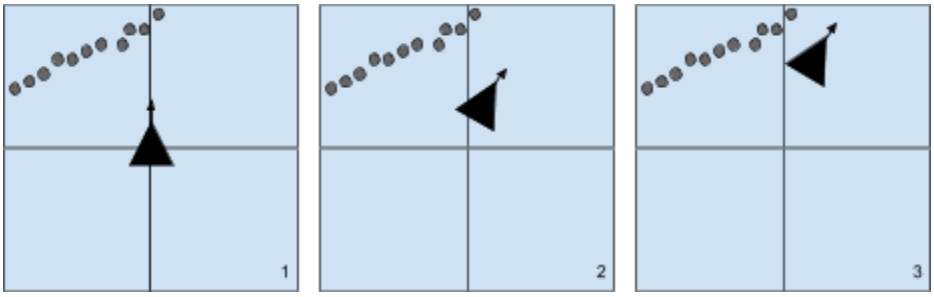
Driving in a Square

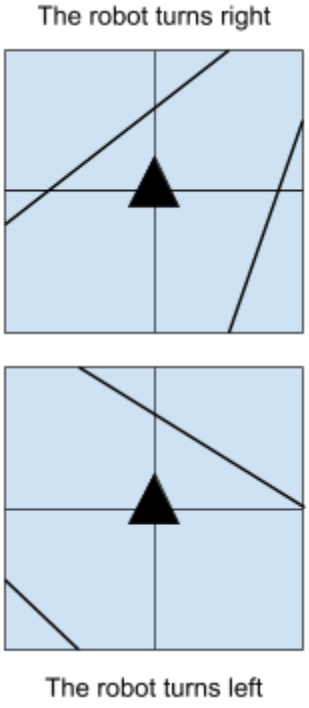


- *Description* -- The robot moves in a 1 by 1 meter square path indefinitely; it does so by turning 90 degrees every few seconds.
- *Diagrams* -- To the right, there's a diagram of the square the robot makes, labelled with the number of seconds it takes to move 1 meter forward or make a 90 degree turn.
- *Tricky Decision* -- I decided to use timing to get the robot to make the square, which meant I had to spend a little time figuring out how long it took the robot to move 1 meter at some given speed and how long I needed to give it to turn 90 degrees. I don't know if I really saved much time doing it this way or if this was the best way to do it by timing, but I stuck with it until I got a pretty good approximation of 1 x 1 meter 90 degree turns.

Wall Following

- *Description* -- The robot aligns to be parallel to the closest wall. It uses the laser sensor to first figure out if it's close to a wall. If it is within some distance (specified in __init__()), it determines whether or not it's currently parallel to the wall. If the Neato is currently parallel to the wall, then it sets the angular speed to 0 and skips the rest of the loop. If the Neato is not parallel, it determines which direction it should turn to align with the wall. It uses some rudimentary proportional control to make slower turns when it's closer to being parallel and faster turns when it's not so it doesn't under/over-turn.





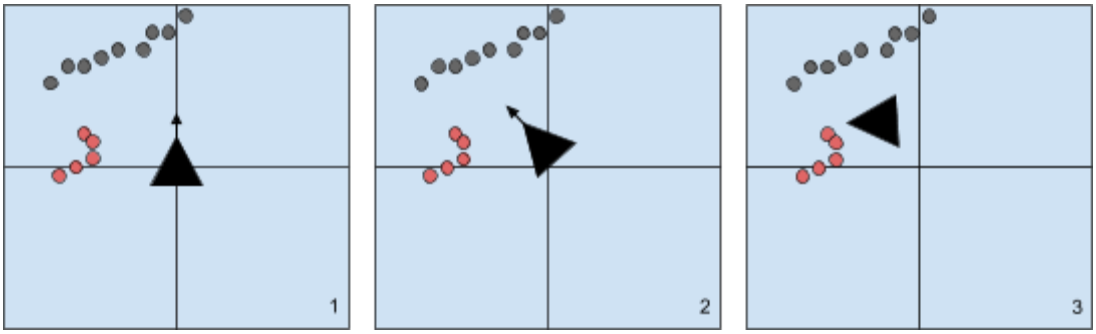
- *Diagrams* -- The three frame diagram directly above shows the behavior of the robot in response to a set of sensor readings that represent a wall to the robot. It makes a dramatic turn towards the wall in the second frame and a more gentle turn in the third frame as it aligns parallel to the wall. The two frame diagram to the top right shows four possible placements of a wall relative to the robot; in the top frame, both walls would cause the Neato to turn right, because it's either moving into or away from the walls towards the left. In the bottom frame, the opposite is true. These diagrams represent the logic of the program and how it picks the direction to turn in and speed to turn with.

Person Following

- *Description* -- The robot follows a person at a specified distance. It uses the laser sensor to detect a small bump in its 360 degree field of vision that is supposed to represent a leg. It looks at a moving average for the range, finds significant decreases in distance, then looks to see if there's a significant increase in distance within 30 degrees of the decrease. This is intended to filter out the walls as they're flat and continuous (so there are generally no dramatic decreases in distance from degree to degree), and while they may suddenly drop off (ex. for an intersecting hallway), that's typically for more than a 30 degree span in the Neato's eyes. The Neato moves towards the most dramatic bump, getting no closer than the specified distance. It adjusts the speed of its movement based on the relative angle of the leg.
- *Diagrams* -- The diagram below with the three frames, shows the behavior of the Neato in response to sensor readings that indicate a wall-like line of data points as well as a prominent bump. The bump, highlighted in red, represents a person's leg. In the second

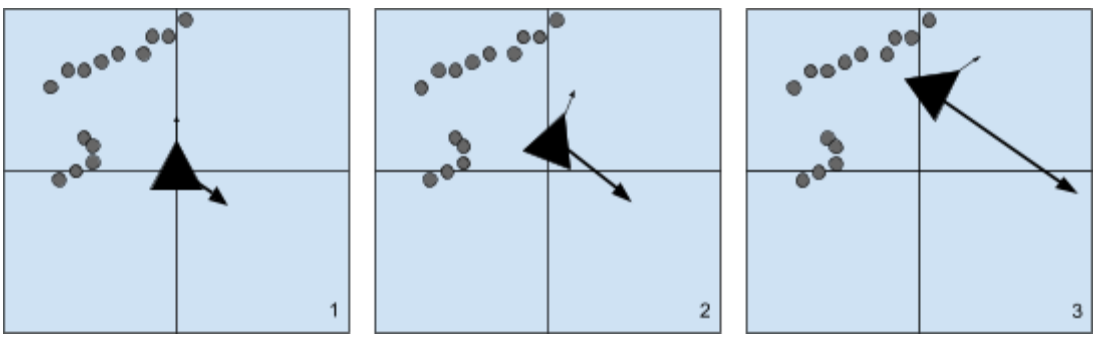
frame, the robot curves strongly to the left in the direction of the bump. In the third frame, once it's a small distance from the person, it stops moving (thus the lack of an arrow). If the person were to walk towards the Neato, it would back up. If a person were to walk away, the Neato would follow them, ignoring the wall even if it becomes closer than the leg at any given point.

- *Tricky Decision* -- I originally tried to detect the person by finding the lowest average value in a set of ranges (boxes) and turning in that direction, but the walls in the area I was testing in were consistently drawing the Neato away from me if I ever let it get too far from my legs. So, I tried to switch to a rudimentary method of detecting a leg that would effectively filter out the walls. This improved the program's performance, but created other problems. For example, whenever my feet are close together but not one in front of the other, they appear as a wall to the Neato, which means the Neato isn't interested in my feet. I was able to make the behavior more accurate, but created a different problem in doing so.



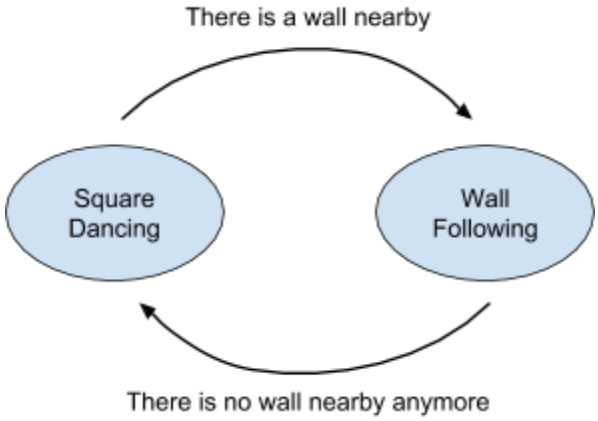
Obstacle Avoidance

- *Description* -- The Neato moves forward while avoiding obstacles. It treats the distance from objects as the magnitude of force vectors, with the angles (+ 180 degrees to reverse them) as the directions of the vectors. It converts polar coordinates to cartesian coordinates, sums the cartesian coordinates, then converts back to polar coordinates to get the overall angle and magnitude of the summed forces. It then moves in this direction with a speed proportional to the magnitude of the force vector.
- *Diagrams* -- The diagram below shows how the Neato responds to obstacles it's nearing on its journey forward. The bolded black arrows represent the summed forces in each frame. As the robot nears the obstacles, the thick black arrow gets longer, representing the increasing pull away from the obstacle. This results in a dramatic turn in frame three as the Neato swerves to avoid colliding with the object.



Finite State Controller

- *Description* -- The Neato moves in a 1x1m square until it spots a wall, at which point it transitions to following the wall. When there is no wall anymore in sight, the Neato begins moving in the square again.
- *States*
    - Square Dancing: The Neato does the “Driving in a Square” behavior.
    - Wall Following: The Neato follows the wall it spotted while Square Dancing.
- *Diagrams* -- The state transition diagram to the right shows the flow from one state to another. When the robot is Square Dancing, it is also using the laser sensor to scan its environment to see if it can spot any walls as it does in the “Wall Follower” behavior described earlier. If it detects a wall (a certain number of points are within some specified distance of the robot), it switches to the Wall Following state, where it performs the “Wall Follower” behavior. If it loses the wall (it ends or the robot gets confused), it switches back to Square Dancing, its default behavior.



Code Structure

The code for each of the behaviors described above is grouped by classes, with one class per behavior. Within each class, there is a run() method that activates the behavior and various helper functions that run() method calls, such as subscriber callbacks for whatever sensors the behavior relies on (ex. wall following uses the bumper and laser scan sensors).

Challenges

The major challenge I faced was debugging. Many of my behaviors still don’t perform perfectly, and I struggled to identify the causes of the problems. Was it my math? Was it my parameters? Was it the environment (ex. the floor-to-ceiling window in the AC hallway not appearing like a wall to the laser scan)? Was it a stupid mistake somewhere? The latter was a frequent problem; there were a number of silly oversights that cost me a lot of time. For example, I got tripped up at one point because I was adding two angles without realizing one of them was in radians and another was in degrees. Going forward, I should have a better idea of what common mistakes to look out for when it comes to coding robots, but mostly because I made a lot of them during this project!

Improvement

If I had more time, I would try to make my code cleaner and more interesting (use lambda functions, more helper functions, etc). I would like to make my person following program more precise, so I could walk down the length of the AC without losing my Neato somewhere in the

narrow sections of the hall or by the windows. It would be cool to get it to follow someone's legs like the Going Beyond section suggests -- I tried to detect just one leg, but there are a number of cases where that doesn't work so well (ex. when my legs are next to each other or my body is angled weirdly, because they appear as one massive, wall-like leg, and my code tries to filter out the walls). I'd also like to make my obstacle avoider more robust and give it more purpose (using the odometry coordinate frame to give it a specific goal instead of just moving forward as much as possible, another Going Beyond suggestion).

Key Takeaways

- Sketch out keyframes of the behavior to identify its structure and edge cases **before** you start coding
- Write helper functions galore. Every helper function should do one thing -- it makes it easier to debug
- Use multiple readings from sensors (plus ideally more than one sensor) because values are frequently missing/unreliable
- Become Nathan and Eric's new best friend
- Neatos seem to have a longer battery life than Roombas