A Computational Introduction to Robotics

Warm Up Project

Dexter Friis-Hecht & Dokyun Kim

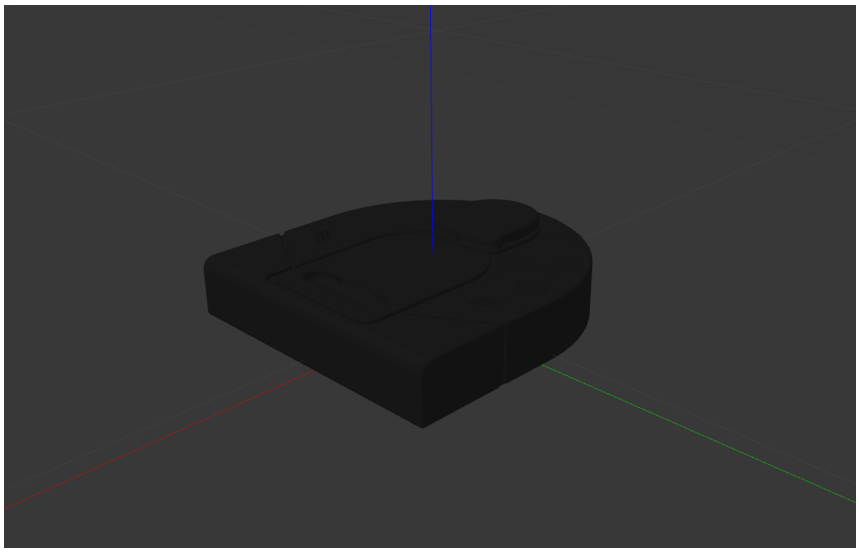

Table of Contents

1. Robot Teleop

`Teleop.py` provides the user control over a Neato through keyboard inputs.

1.1 Approach

The approach can be broken down into two main steps. First, the user input keypress needs to be logged. Then, the appropriate linear/angular velocity is communicated to the Neato depending on what key was pressed.

For accessing keyboard inputs, the `getKey()` function is used to return the key that was pressed as a Python string. Based on the pressed key, the corresponding velocity gets published to the Neato through the `cmd_vel` node. The key-to-velocity map is shown below:

w: linear velocity of 0.3
a: angular velocity of 1.0
s: linear velocity of -0.3
d: angular velocity of -1.0
x: stop
Ctrl+c: exit program

1.2 Node Structure

This program only has one publisher node that publishes to Neato's `cmd_vel` topic. A diagram showing the node relationship is shown below.

Fig 1.1 Node structure of `teleop.py`

1.3 Limitations

The `getKey()` function can only return one key at a time, which means it cannot respond to multiple keys being pressed at the same time. This results in some limitations in the Neato's movement such as it cannot move while turning. With more time, this limitation could be resolved by publishing linear and angular velocity independently from each other.

2. Drive Square

`Drive_square.py` executes a program that makes a Neato drive in a 1 meter by 1 meter square.

2.1 Approach

This problem can be broken down into three phases that repeat until the goal (completing a square) is met. First, the Neato needs to move forward for 1 meter. Then, it needs to turn 90 degrees. Once a square is complete (traveled 1 meter 4 times), the Neato stops. The function `square_draw.py` determines which phase the Neato is in.

2.1.1 Phase 1: Traveling 1 meter

The Neato's `odom` topic will be used to determine its current location in cartesian coordinates. When the program first executes, the class variable `prev_coord` is initialized at (0,0). For every subscriber callback, the `square_draw()` function will get the neato's current x,y position and store it in `crnt_coord`. Then, the function calculates the distance between `prev_coord` and `crnt_coord` to determine if the Neato has traveled 1 meter. If it has not, it sets the class variable `lin_vel` to 0.2.

2.1.2 Phase 2: Turning 90 degrees

The Neato's `odom` topic will also be used to determine its current angle in degrees. Since the `odom` topic gives the angle in quaternion (w), the following equation is used in the function `get_angle` to convert it to degrees (θ). The result of `get_angle` is stored in `crnt_ang`.

$$\theta = \arccos(w)*2$$

How much the Neato turns is determined using the class variable `goal_angle`. The idea is that the Neato needs to turn until it is 90 degrees, 180 degrees, then 270 degrees to finish drawing a square. Since the odometry readings are never integer values, the goal angles are adjusted to 88, 178, and 268 degrees for tolerance.

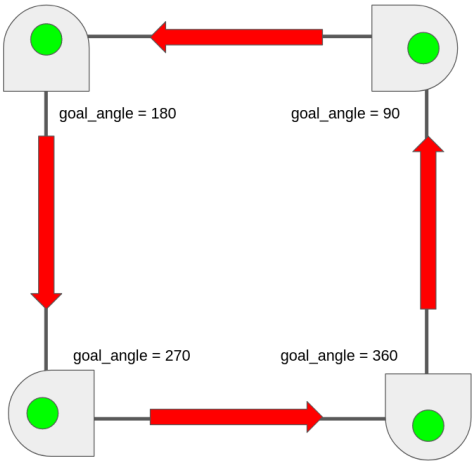

Fig 2.1 Definition of `goal_angle`

When the Neato enters the turning phase, it will set the class variable `ang_vel` to 0.5 until `crnt_ang` is less than or equal to the `goal_angle`. Once the `goal_angle` is reached, `ang_vel` is set to 0.0, and the class variable `line_count` is incremented by 1.

2.1.3 Phase 3: Stopping

Once `line_count` reaches 4, meaning the Neato completed drawing 4 lines, both `lin_vel` and `ang_vel` are set to 0.0.

2.2 Node Structure

The program has one subscriber node and one publisher node. There is a subscription to the Neato's `odom` topic, which is used to determine whether to move forward, turn, or stop in the callback function `square_draw.py`. The publisher node then publishes the `lin_vel` and `ang_vel` to the Neato's `cmd_vel` topic. Figure 2.2 shows the Node structure for `drive_square.py`.

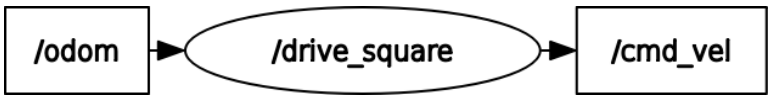

Fig 2.2 Node structure of `drive_square.py`

2.3 Limitations

Some limitations of this program are that the Neato will only rotate counterclockwise. A possible improvement could be taking the rotation direction as a parameter. Another limitation of the program is that the Neato has no self-correction functionality. If the Neato were to deviate from a

square, the program would have no way of correcting the Neato's movement. Depending on the floor's surface, there is a possibility of wheel slippage, resulting in an incorrect square.

3. Wall Follower

The purpose of `wall_follower.py` is to create a program that when a wall is present next to the Neato, it drives forward while staying parallel with the wall.

3.1 Approach

This problem can be broken down into 2 phases that repeat until the user terminates the program. First, the Neato checks if it is parallel to the wall. Then, depending on the result, it either moves forward or rotates in a way so it is parallel to the wall.

3.1.1 Phase 1: Check if parallel

Every loop iteration,
When the Neato is parallel to the wall, the lines from the 45° and 90° points of the lidar will form an isosceles triangle with ranges[45] as its base. According to the 45-45-90 triangle identity, the following equation becomes true.

$$ranges[45] = \sqrt{2} * ranges[90]$$

This characteristic will be the key point in determining whether the Neato is currently parallel to the wall.

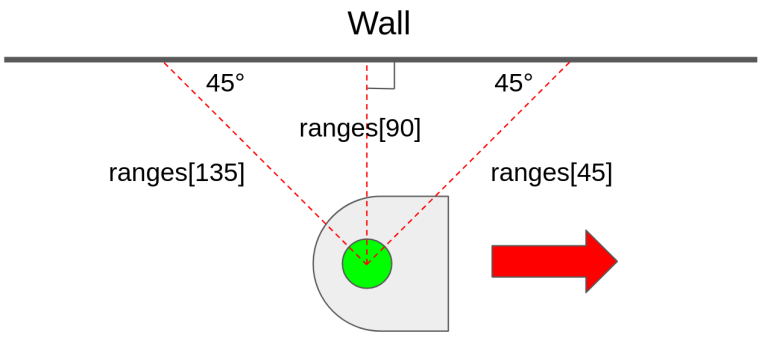

Fig 3.1 Orientation of Neato when parallel to a wall. The list `ranges` contains the distance of each point from angles 0-360°

The function `get_data()` uses this characteristic to calculate a percent error that is represented with the following equation:

$$error = (ranges[45] - \sqrt{2} * ranges[90]) / (\sqrt{2} * ranges[90])$$

This value is stored in a class dictionary variable `dists` along with ranges[45], ranges[90], and ranges[135] with the key word 'pcnt_error.'

3.1.2 Phase 2: Forward or rotate

Using the error calculated in Phase 1, the program decides whether to move forward or rotate. The function `check_angle()` returns True if $|error| \leq$ tolerance and False otherwise. If `check_angle()` returns True, the linear velocity is set to 0.2 and angular velocity to 0.0. If `check_angle()` returns False, the linear velocity is set to 0.0, and the direction of rotation is determined by the value of `pcnt_error`.

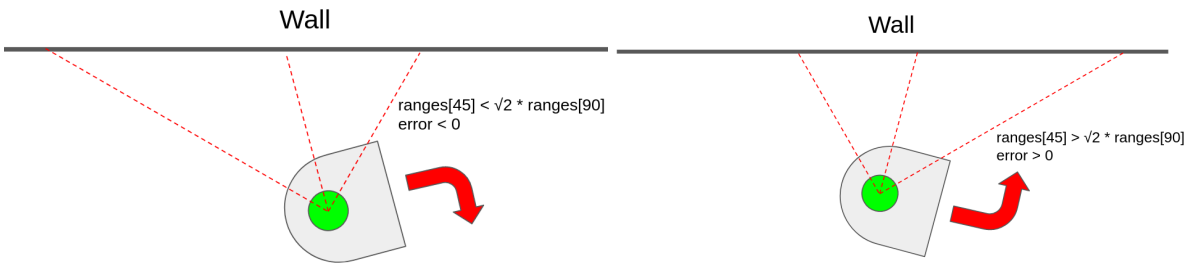

Fig 3.2 Direction rotation depending on the value of `pcnt_value`. If *ranges*[45] is less than $\sqrt{2} * ranges[90]$, the Neato will rotate clockwise. If *ranges*[45] is greater than $\sqrt{2} * ranges[90]$,the Neato will rotate counterclockwise.

3.2 Node Structure

The program has one subscriber node and one publisher node. The subscriber node is subscribed to the `/scan` topic which contains data from Neato's LiDar scans. The program accesses the `ranges` attribute from this topic, which is used to determine if the Neato is parallel to the wall. The publisher node then publishes the `lin_vel` and `ang_vel` to the Neato's `cmd_vel` topic. Figure 3.3 shows the Node structure for `wall_follower.py`.

Fig 3.3 Node structure of `wall_follower.py`

3.3 Limitations

The biggest limitation of this program is that it is built under the assumption that there is already a wall next to the Neato. A possible improvement is to implement a wall detection algorithm like RANSAC to first identify a wall, then have the Neato follow it. Another limitation is that the program only works when the wall is on the left side. This could have been improved by letting the user specify which side the wall is on as a function parameter.

4. Person Follower

The purpose of `person_follower.py` is to program the Neato in a way where it follows a moving person who is in front of the Neato in a specified area. This program is designed under the assumption that there are no bigger objects than the person present in the tracking region.

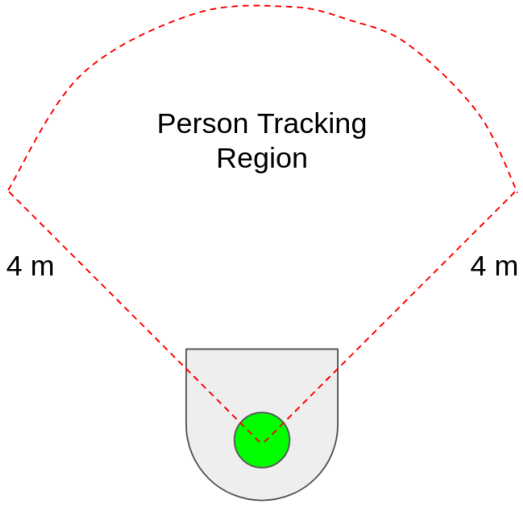

Fig 4.1 Person tracking region used in `person_follower.py`

4.1 Approach

This problem can be broken into 3 phases that repeat until terminated by the user. First, the Neato scans its person tracking region and identifies different clusters of points. Then, with a group of clusters, the largest cluster is assumed to be the person and the amount of necessary rotation to be aligned with the person is determined. Finally, the Neato moves towards the person using the angle found in the previous step.

4.1.1 Phase 1: Cluster Identification

One of the principal functionality this problem needs is a way of identifying different groups of clusters. When the Neato scans the tracking region, it will detect every object that is in that region. However, the only object of interest is a person, which is assumed to be the largest object within the tracking region.

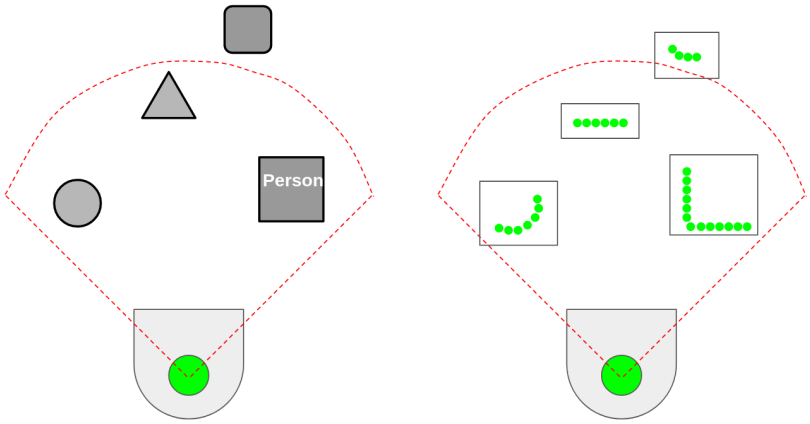

Fig 4.2 **Left**: A possible orientation of person & other objects. **Right**: Points that would get detected by the LiDar scanner. The points are grouped into their respective clusters by the rectangle.

As shown in *Figure 4.2,* there needs to be a way to group the scanned points into their respective clusters. This is achieved by iterating through all the collected points and appending them to a list if the current point is within a specified distance from the previous point. A maximum gap value of 0.5 was used in this program.

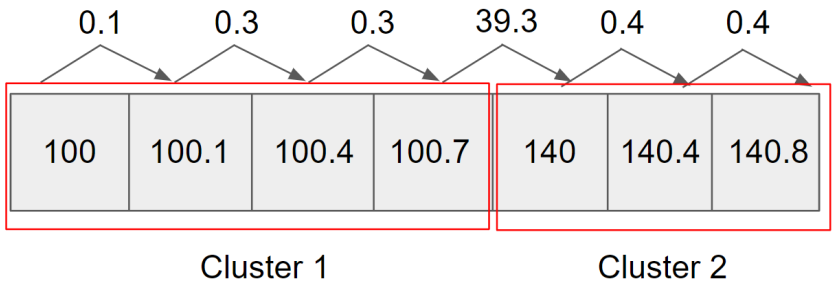

Fig 4.3 `group_cluster()` algorithm. I need a short description for this ahhhh

The output of `group_cluster()` is a list of lists where each inner list represents a cluster. An example output is shown below.

output = [[100,100.1,100.4,100.7] , [140,140,4,140,8]]

4.1.2 Phase 2: Finding the person

Once all the clusters are identified, the person needs to be identified from the group of clusters. As mentioned earlier, this program assumes that the person is the largest object within the scan area. Working under this assumption, the person is simply the remaining cluster after removing all the small clusters and clusters that are outside the scan region. The function `closest_cluster()` looks through a list of clusters and identifies the person.

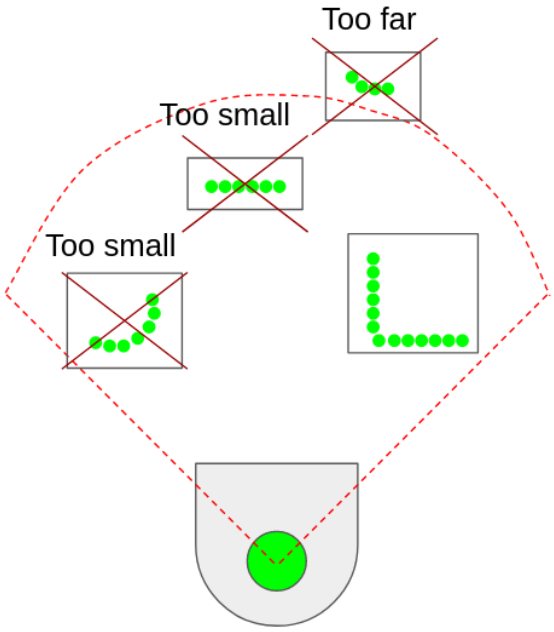

Fig 4.4 Expected output after `closest_cluster()`

4.1.3 Phase 3: Following the person

With the person identified, the Neato can now move forward while turning towards the person. The direction of rotation is determined by calculating the median angle of the person. If the angle is between 0 and 45 degrees, it turns counterclockwise, and clockwise if between 315 and 360 degrees. As the Neato rotates, it also moves forward to maintain a certain distance from the person.

4.2 Node Structure

The program consists of a subscriber node and a publisher node. The subscriber node is subscribed to the `/scan` topic to access the LiDar data. The publisher node publishes the appropriate linear /angular velocity to the `/cmd_vel` topic. The node structure is shown below.

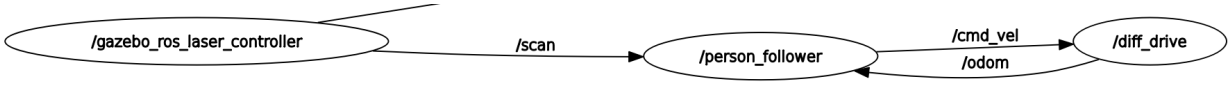

Fig 4.5 Node structure of `person_follower.py`

4.3 Limitations

The biggest fault of this program is that it is built under the assumption that the person is the largest object in the scan region. The program stops functioning if there is another object that is larger than the person. This problem could be solved by using another characteristic of a person to identify them. One possibility is to track moving clusters since other obstacles would be static.

5. Obstacle Avoidance

The purpose of `obstacle_avoider.py` is to program the Neato to drive forward until it encounters an obstacle. Upon encountering the obstacle, it will navigate around the obstacle and resume its path forward.

5.1 Approach

The algorithm can be split into two main states. There's the initial state where the Neato is just moving forward. During this phase, the Neato checks for any incoming obstacles in front of it. Upon detecting an obstacle, the Neato will check its distance from objects on both its left and right. The neato then moves into its second state where it will avoid the obstacle in front of it, turning towards the direction with obstacles the furthest away. Once the Neato has cleared the obstacle, it will reset to its initial state and resume driving forward.

5.1.1 Phase 1: Obstacle Identification

By default, the Neato's forward linear velocity will be set to 0.3 m/s. As it moves forward, we continuously access the 0 degree distance reading from the Neato's 'laser' topic. If the Neato gets too close to an obstacle, as dictated by 'self.obstacle_threshold', it will stop moving and access the 'laser' topic to check its distances to obstacles on the left and right. Since there are two directions the Neato could go to navigate around the obstacle, it chooses the direction where it would be least likely to encounter another obstacle while avoiding the first one.

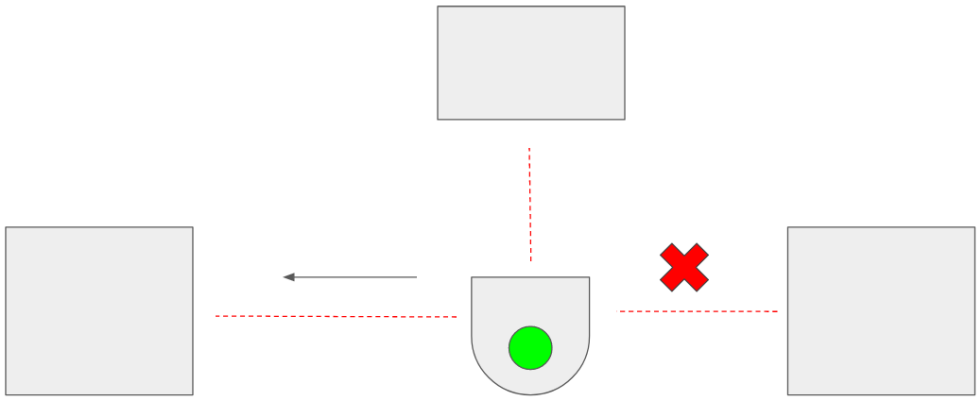

Fig 5.1.1. Neato encounters obstacle and will turn left to avoid the obstacle

5.1.2 Phase 2: Obstacle Avoidance

Once the obstacle has been identified, and the target direction has been chosen, the Neato will turn parallel to the obstacle. Once the turn is complete, the Neato will continue to sample its distance from the obstacle while moving forward. The Neato will continue to drive to the side until it no longer detects the obstacle with one of its side laser scans. If the obstacle is no longer detected, the Neato will turn towards its original direction while maintaining a small linear velocity of 0.05 m/s. This ensures that as it rotates, it will fully clear the obstacle.

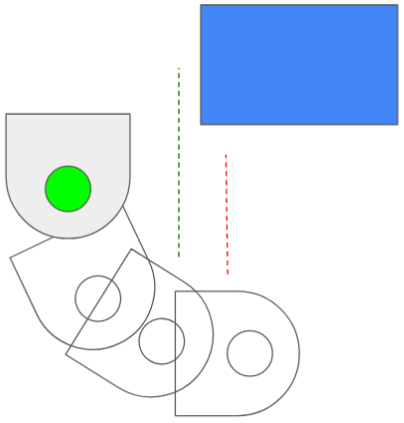

Fig 5.1.2. The Neato moves parallel to the obstacle until it's safe to drive around it

5.1.3 Neato Reset

Once the Neato has executed all of the steps required to avoid the obstacle, all of the Node's variables will be re-initialized, allowing the Neato to avoid the next obstacle it might encounter.

5.2 Node Structure

The program consists of two subscriber nodes and a publisher node. The first subscriber node is subscribed to the `/scan` topic to access the LiDar data. The second subscriber node is subscribed to the `/odom` topic to access the Neato's current rotation. The publisher node publishes the appropriate linear /angular velocity to the `/cmd_vel` topic. The node structure is shown below.

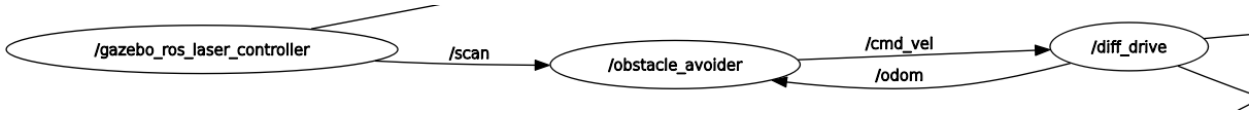

Fig 5.2. The node structure for 'obstacle_avoider.py'

5.3 Limitations

Due to the relatively simple nature of this implementation, there are a few key limitations. Once the Neato begins to avoid the obstacle in front of it, its movements become relatively hard-coded. Regardless of which direction the Neato chooses to turn to avoid an obstacle, the Neato doesn't check if the available gap is large enough for it to navigate, even if it chooses the direction with more space. This can result in the Neato colliding with a separate obstacle while avoiding the first one. Additionally, the Neato won't respond well to dynamic environments. Since any detected obstacle will immediately trigger the avoidance sequence, if an object is immediately removed after being detected, the Neato will still do an initial rotation sequence. The biggest improvement that can be made to this node in the future would be to allow

6. Finite-State Control

The purpose of `finite_state_controller.py` is to combine the behavior of `wall_follower.py` and `person_follower.py` into one program where the Neato follows the person if there is one in front of it and follows the wall if there isn't'.

6.1 Approach

The logic that switches between the two modes is quite simple. If a person is detected in the scan area, start the person following mode. If there is no longer a person in front, start the wall following mode.

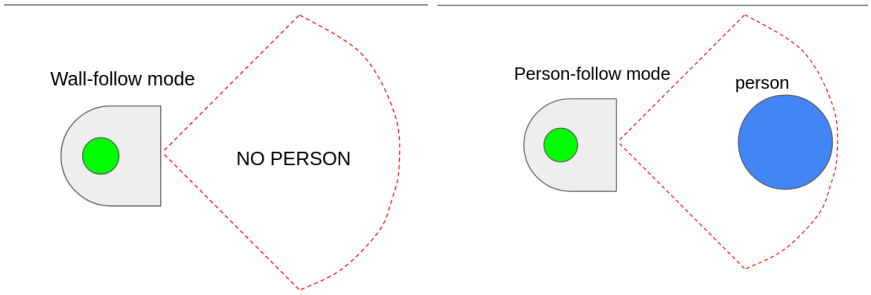

Fig 6.1 **LEFT** Neato in wall-follow mode; **RIGHT** Neato in person-follow mode

6.2 Node Structure

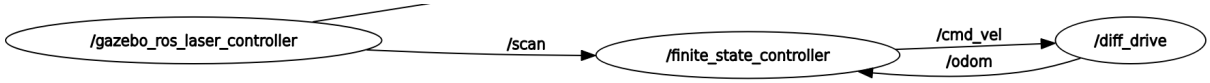

Fig 6.2 Node structure of `finite_state_controller.py`

6.3 Limitations

There are certain cases where the Neato recognizes the wall as a person if it is facing the wall, resulting in it to drive straight into it. A possible fix is to have a variable that limits the number of points the person can have, allowing the Neato to ignore the large point cluster created by the wall.

7. Lessons Learned

This project provided the fundamental knowledge for working in a ROS2 environment. Some of the key points were:

- Using subscriber and publisher nodes
- Visualization using Gazebo and Rviz2
- Familiarity with the concept of topics and nodes
- Executing ros2 functions in Linux terminal

Among the lessons learned, this project highlighted the importance of planning your structure ahead of time in an object oriented context. It also introduced us to operating within a runtime loop, which fundamentally altered the way we had to approach coding. For example, we couldn't rely easily on while loops, as they prevented us from continuously sampling odometry and laser scan data.