Wall Follower

Files Involved: `wall_follower.py`

Bag Files: `wall_follower.bag, wall_follower_viz.bag`

- For each behavior, describe the problem at a high-level. Include any relevant diagrams that help explain your approach. Discuss your strategy at a high-level and include any tricky decisions that had to be made to realize a successful implementation.

The problem we're trying to solve here is to have the robot move forward while aligning its direction of motion to be parallel to the nearest wall.

Our approach is to let the robot find the nearest object first and try to move alongside it. Here we made a design decision to assume the nearest object is the wall. Meanwhile, the robot is keeping track of the distance from the object. When the distance is closer than the distance set by the user, the robot will turn away from the wall until it reaches to the target distance; similarly, when the robot is far away from the wall, it will drive towards the wall. We also added code to stop the robot when a bump occurs.

The script is constantly checking the smallest non-zero value in the range reading of the stable_scan topic. The code will then record the measurement angle, distance, and timestamp as the current minRange. Based on the current and previous minRange, the velocity of the robot can be calculated. If the velocity is positive, it means the robot is driving away from the wall and a negative velocity means it's driving towards the wall. Along with the angle of the current minRange, the script will be able to find out which direction the wall is located and thus makes the robot drive in the correct direction.

- How was your code structured? Make sure to include a sufficient detail about the object-oriented structure you used for your project.

We made the script into a class and have instance variables that track the current minRange, previous minRange and if the robot is stopped(during bump). As for sending instructions to the robot, we have two statuses to determine how the robot should move. The first one is the relative position to the wall: inRange, tooClose and tooFar. The second status is the the direction of the wall: leftFront, rightFront, leftBack and rightBack. Based on those two statuses, the script will find the correct movement direction from a state machine encoded as a dictionary. For example, if the robot is at the correct distance from a wall (inRange) and the wall is to the robot's front right (rightFront), then robot will move forward while curving right.

- What if any challenges did you face along the way?

Early on, we reasoned that we could make decisions based on only LIDAR ranges in each cardinal direction from the robot (front, left, right, back). However, we quickly realized that creating logic on these directions was challenging to perform logic on, because they change both with linear and angular motion. We realized that a much easier way to perform logic was to use the smallest distance measurement (hopefully, the distance to the wall) and it's angle to the robot.

As our code progressed, we were impeded by our original, byzantine if statements that made it hard to understand and debug the robot's behavior. We restructured the code to use a straightforward state machine, which makes decisions based on the direction and distance of the nearest obstacle.

- What would you do to improve your project if you had more time?

Our robot became a fairly safe driver, but overcompensates for walls - over rotating when too close. If we extended the project, a better reaction would be to use proportional control to start moving cautiously forward the more confident the robot becomes that it will not run into a wall. Similar proportional control would help mediate an overreaction from the robot when too far from the wall.

- What are the key takeaways from this assignment for future robotic programming projects?

From our experiences with the wall-follower, simple state machines appear to be a good way to mediate robot behavior without sacrificing the code-quality to piles of if-statments. Additionally, using the LIDAR's minimum distance to obstacle proved a productive approach to obstacle avoidance behaviors.

Obstacle Avoidance

Files Involved: `obstacle_avoidance.py, utils.py`

Bag Files: `obst_avoid.bag, obst_avoid_viz.bag`

Demo: *https://youtu.be/3hs4rSE86Uo*

- For each behavior, describe the problem at a high-level. Include any relevant diagrams that help explain your approach. Discuss your strategy at a high-level and include any tricky decisions that had to be made to realize a successful implementation.

In a nutshell, the problem presented by the Obstacle Avoidance behavior is trying to get the robot to avoid colliding with obstacles while navigating about its environment. Our strategy was to treat all obstacles as objects that exerted a certain amount of repulsive "force" on the robot, described by a quartic distance function. The closer the robot came to an object, or the more objects/points of an object there were in a certain direction, the less willing the robot became to travel that way.

Some tricky considerations were the fact that, given our approach, the robot had no incentive to move once it found a happy equilibrium where all the repulsive "forces" relatively equaled out; at what point to have the robot stop moving forward and purely rotate if faced with an obstacle that was too close; and how exactly to incorporate proportional control into our algorithm (i.e just for the turning with a steady forward velocity, for turning and forward motion independently, etc.).

For the first consideration, we initialized the robot with a constant forward velocity, which we called our "adventurousness factor." Now when the robot had zero incentive to move, it would instead default to the initial forward velocity until the "forces" of other objects overcame its basic behavior caused it to change course once again. For the second consideration we toyed around with different cutoff angles for which the robot would cease moving forward and opt to turn away from the obstacle. We found that a cutoff angle of 180 degrees was optimal, anything less, and the robot felt "uncomfortable" traveling parallel (or roughly parallel) to the obstacle which made it very "cautious" and "timid" in its movements. Anything greater, and the robot was already facing away from the object, so there wasn't much of an effect. We opted to implement proportional control by making the forward velocity a function of the cutoff angle and the difference angle between the robot's heading and the location of the obstacle, and added a gain to the difference angle for turning.

- How was your code structured? Make sure to include a sufficient detail about the object-oriented structure you used for your project.

We created an ObstacleAvoider class with an instruction instance variable to store the Twist, a stop boolean for emergencyStop which we incorporated into our code, a cutoffAngle, and a gain. If the stop boolean was true, we published instructions for the robot to remain stationary, otherwise we published the instructions that were set in the instruction variable by the processScan function. The processScan function loops through the scan ranges and ultimately sets the instruction instance variable based on the data. We calculated the x and y components of the ranges and aggregated them into a sum of all the ranges, resulting in a vector pointing away from obstacles to be avoided. Initially we gave the x variable a starting value so that our robot would always be trying to move forward unless deterred by an obstacle. We determined the ultimate velocity as a function of the cutoff angle and the difference angle between the robot's heading and repulsion vector from all the obstacles, and applied our gain in determining how much the robot should turn. We assigned these values to our instruction variable which gets called in the main loop.

- What if any challenges did you face along the way?

Calculating the right behaviors based on the trigonometric functions was a bit tricky. We also had to adjust the distance from a square to a quartic so that distant, large objects did not overpower small local objects. We also subtracted a small amount from the value in our ranges to account for the size of the robot so it wouldn't accidentally crash into objects it thought it could clear.

- What would you do to improve your project if you had more time?

We're pretty happy with our obstacle avoider. Given the limitations, we think it behaves remarkably well.

- What are the key takeaways from this assignment for future robotic programming projects?

A key takeaway is the importance of a strong trigonometric background in calculating robot motions and heading. Also using rViz and print statements are very useful in debugging, and keeping in mind the different coordinate frames of the robot and how they are physically related is invaluable.

Person Follower

Files Involved: `person_follower.py, utils.py`

Bag Files: `person_follower.bag`

Demo: **https://youtu.be/BFjYHY5-BlY**

- For each behavior, describe the problem at a high-level. Include any relevant diagrams that help explain your approach. Discuss your strategy at a high-level and include any tricky decisions that had to be made to realize a successful implementation.

For the person follower, our script tries to detect the nearest object in front of the robot and uses proportional speed to follow the target. The script first clusters all the points in the /prjoected_stable_scan topic into blogs based on their locations. If two points' distance is less than 0.05m, they're clustered together. Then the script calculates the center of mass for each blob and filters out all the blobs that don't have a center of mass in the front of the robot. In this way, the robot won't be "distracted" by obstacles that are not in the front (or partially in the view). Now that the target is narrowed down to a few centroids, the script then picks the nearest centroid and follows it. Here we made a design choice that the robot will assume the nearest object as person. Even if it isn't, the person can always put his/her feet closer so that the robot will then follow the person.

- How was your code structured? Make sure to include a sufficient detail about the object-oriented structure you used for your project.

We structured our code in object-oriented style. We tried to keep each function relatively short to increase the readability.

- What if any challenges did you face along the way?

Filtering out the noise from scan and tuning parameter values to improve robot performance.

- What would you do to improve your project if you had more time?
  - Have a more developed method for detecting moving objects.
  - Enabling robot to make right decision in some corner cases (narrow space, moving obstacle and etc.)
- What are the key takeaways from this assignment for future robotic programming projects?

Always start simple, don't try to accomplish everything in the first iteration.

Finite State Machine

Files Involved: `automaton.py, robot_tag.py, utils.py`

Bag File: `robot_tag.bag`

- For each behavior, describe the problem at a high-level. Include any relevant diagrams that help explain your approach. Discuss your strategy at a high-level and include any tricky decisions that had to be made to realize a successful implementation.

For the finite state machine, we decided to focus highly on creating a reusable, cleanly written structure that could be reused in future projects. The main difficulty here was establishing a useful framework that carried as much of the FSM implementation as possible without overly constraining what could be done within that framework. With that reusable structure as the focus, we decided to set an implementation objective for the finite state machine of "robot tag".

In our robot tag implementation, the robot 'chases' a person using the person-following behavior and, when it 'tags' them with its bump sensor, it starts to instead 'run away' using its obstacle avoidance behavior. We chose this implementation in a large part because, with the behaviors already built and functioning, we were free to focus on the FSM structure.

- How was your code structured? Make sure to include a sufficient detail about the object-oriented structure you used for your project.

To implement this and future FSM or FSM-like robots, we built two classes, 'Automaton' and 'Behavior'. An automaton is a standalone ROS node which has multiple Behaviors. To build a Behavior, one subclasses Behavior and overloads any callbacks of interest (eg. onBump, onScan). Convenience methods, such as setSpeed and changeState are provided by Behavior for subclasses to make use of, and are hooked into an update method called by the Automaton that has this behavior. When changeState is called and the next update runs, Automaton will substitute the newly specified behavior for the current one, calling its sensor callback and update messages instead of the original behavior's.

To create an automaton, one must simply instantiate Automaton with the appropriate arguments, including the behaviors it should have.

- What if any challenges did you face along the way?

Because the FSM structure described was designed based on our anticipation that sensor callbacks would be the critical interface between Automaton, we handle them well. However, our system anticipated a static number of subscribers that are hard coded into Automaton

(sensors are not created on the fly). Besides “/cmd_vel”, publications are not as static; each behavior or automaton instantiation may need its own set of publications. To handle this we built in a system of marker publications that could be created at the automaton’s instantiation. It works admirably, but enforces an expectation that all behaviors will only ever need to publish `/cmd_vel` and markers.

- What would you do to improve your project if you had more time?

This implementation so far has been remarkably clean and robust even to clean up the behavior logic of robot behaviors that don’t need to be part of an FSM. The only addition we may make with additional time is documentation so that this FSM structure can be used by other students or in future projects when we have forgotten the implementation details.

- What are the key takeaways from this assignment for future robotic programming projects?

This FSM structure was successful enough that it encourages future investment in code reuse and architecture. It also simplifies the creation and publication of markers to the point that visualization has become less painful and that encourages us to visualize our behaviors earlier (eg. while they’re still useful).