Introduction to Computational Robotics: Warmup Project

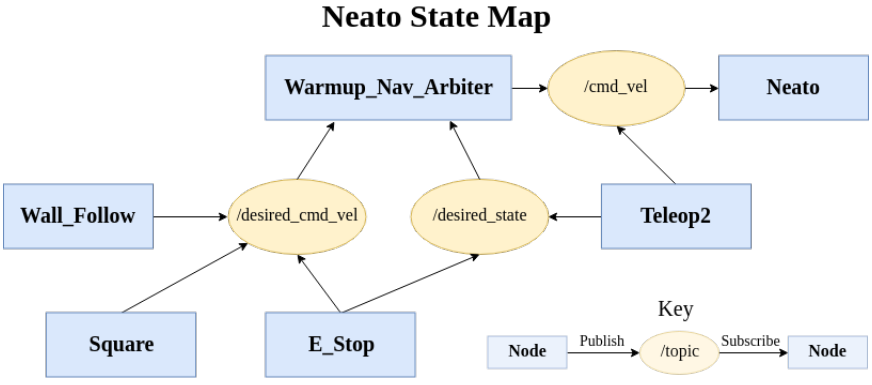
Matthew Beaudouin-Lafon, Katerina Soltan

The first project for this course was meant to be an exercise in getting to know our robot platform, a Neato vacuuming robot, and communicating with it using ROS, the robot operating system. We created a few behaviors for the robot, including wall following, driving in a square, and obstacle avoidance, and developed a few tools, such as a state arbiter and teleoperation, to enable us to interface with our robot more easily. We focused on developing an architecture for our programs that would enable future modifications and additions to the robot's behaviors.

Neato robot platform: The Neato vacuuing robot is equipped with a lidar that provides laser scan data for distance measurements and a Raspberry Pi for running ROS and connecting to laptops for remote operation.

Warmup Arbiter

Our goal for this arbiter was to simplify our workflow around simultaneously testing different behaviors while keeping both teleop control over the Neato and emergency stop behavior upon receiving a *bump* message. The arbiter would more aptly be called a state controller because it allows certain commands to be published to the robot's motors based on the Neato's state published on the *desired_state* topic. If the robot is in *WALL_FOLLOW* mode, only commands from the wall following node will be executed. Only the teleop and estop nodes publish to the *desired_state* topic. This allows the user to define the desired behavior through the teleop node, which makes sense because it serves as the user facing node. This architecture also allows the emergency stop to always take priority. It does so by changing the state to *E_STOP* when it receives a *bump* message. It should be noted that any *desired_cmd_velocity* sent by the teleop node (eg. move backwards) shifts the state to teleop, giving it priority. Below is a state diagram for clarity.



Neato State Map: All Neato behaviors publish to a *desired_cmd_velocity* topic instead of directly to *cmd_vel*. The arbiter node is in charge of publishing only the commands corresponding to the current state. THe state can only be set in the estop and teleop nodes. Teleop is a special node and bypasses the arbiter, allowing the user manual control of the robot at all times.

Instead of allowing the behaviors to publish to *cmd_vel*, directly interfacing with the motors, we created a topic for *desired_cmd_vel* that expects a custom *LabeledPolarVelocity2D* message. This message contains both the velocity command and the unique id of the behavior that commands it. The *LabeledPolarVelocity2D* message is similar to the built in *TwistStamped* message, but instead of expressing the velocity in terms of linear and angular x, y, z terms, it combines them all into a polar coordinate system. Since we are only controlling the linear x velocity and angular z velocity, we thought a polar representation would be more intuitive to work with. The arbiter subscribes to the *desired_cmd_vel* topic and chooses which command to send to the Neato based on the published state on the *desired_state* topic.

The main challenge in designing the arbiter was finding a compelling state diagram that generalized to an arbitrary number of behaviors. The key was to acknowledge that the teleop and estop behaviors are fundamentally different. Our solution ended up being simple and offered relatively low friction in adding new behaviors. A new behavior simply needs to publish its desired velocity to the *desired_cmd_vel*, add a state to the state message enum (stored in the custom *State* message), and add a command to the teleop node that will activate it.

While we assumed that the order in which messages appear is the order in which we should process them, adding a header and timestamp to the *LabeledPolarVelcoity2D* message would provide some useful information. Our callback function currently filters and publishes the correct state's command velocity to the Neato. While it is generally not a great idea to overload the callback function like this, we decided that for the purposes of this project, where the arbiter was not processing a lot of data, it would make our code more readable and simpler to write. Migrating the decision part of the code into the main loop of the node, with callbacks interrupting only to update the latest desired command, would require having good timestamps of the sent messages.

To make our configuration even more robust, we could dynamically change the *cmd_vel* topic the behavior nodes should be publishing to, depending on whether they are running with the arbiter or as standalone nodes. However, for the purposes of this exploration, this was not necessary.

Estop

The goal here was simple: stop the Neato when it hits something. The implementation was simple, too: the callback function on the subscriber to the bump topic updates the state to *E_STOP* if the bump sensors are pressed, and sends a null desired velocity command.

Teleop

The teleoperation (teleop) node can be used to control both the motion of the robot as well as its state. Specific key bindings were defined to drive the robot forwards, backwards, and turn right and left, both in place and while moving forwards. Changing the state of the robot (e.g. allowing the execution of *wall_follow*'s commands) is done by typing a ":" and a short string mapped to each behavior that could be running on the robot (such as ":fol" for *wall_follow*). While controlling the motion and state of the robot could be separated into different nodes, we decided to combine them to provide one, all-powerful interface and cut down on the number of terminal windows to type in. Instead of routing the desired velocity through the arbiter, the teleop node is special and always takes precedence, publishing directly to the Neato's *cmd_vel* topic. The arbiter is simply notified that the teleop node has taken control and it should change its current state to TELEOP. This setup allows the user to always be able to take control of the robot, even if it is in *E_STOP* mode which the robot cannot autonomously escape. Because the estop behavior is also allowed to change the state of the robot, there are slight hiccups when moving a robot after it has bumped into something. If the bump sensors are depressed fully, teleoperating the robot backwards may not release them fast enough, and the estop node will publish a desired stop and state change, overriding the teleop. In a future iteration, we could implement a grace period of a few seconds to silence the estop node messages for the user to have enough time to move the robot back from the obstacle it hit.

Square Follow

The goal of the square follow program was to make the Neato move in a 1x1 meter square. We considered two approaches. The simplest way is to sequentially command a forward velocity for a certain time to move one meter forward, followed by an angular velocity for some time to turn 90 degrees. The second approach is to use the odometry frame to plan the square movement. As our interest aligned more with the other behaviors, and in the spirit of scoping the project well, we went for the simpler first approach.

As described earlier, the square_dance node runs a while loop in which is sends a forward velocity, sleeps for some time, sends an angular velocity, sleeps for a different amount of time, and repeats. The challenge is finding the sleep times such that the square follows the specification. To do this, we used ROS's parameter GUI. This allowed us to dynamically change both sleep times, quickly finding values that worked sufficiently well.

This approach is fundamentally flawed since there is no feedback on success. The surface on which the robot moves alone adds variability and imprecision to the Neato's motion. Given more time, we could have used the odometry based implementation which would have yielded better results.

Obstacle Avoid

As the behavior's name implies, the goal is to avoid obstacles. We had previous written an iteration of this behavior that did proportional control based on the presence of obstacles in a small range in front of the Neato. We decided to improve it with a different, more general approach. We were inspired by how electrically charged particles repel each other. Each obstacle contributes to the potential field, resulting in a net force on the Neato. We do this calculation in the robot's frame of reference on every laser scan message. We treat each laser range value as a charge in polar coordinates and average those forces.

We originally wanted to keep the physical metaphor true, but upon testing we quickly found that the Neato's restricted motion did not allow for a charged particle's free motion, forcing us to tune the inverse square law distances to yield better results. For a given degree, we calculated the "force" of a charge at distance d to be: $F = \frac{3}{(\frac{d}{1.5})^2}$, where numbers 3 and 1.5 are magic values found to yield good behavior when obstacles are close and far.

We also tweaked Newton's second law to $F = mv$, because we determined that doing an open loop integration would lead to delayed and unpredictable behavior.

Since we worked in polar coordinates, in the Neato's coordinate system, we thought that it would be easier to write a class that acted like a simple polar vector class but did all of the vector math in the Cartesian frame. To do this, we used the @property keyword in Python for angle and magnitude, making them look like attributes to the class's user.

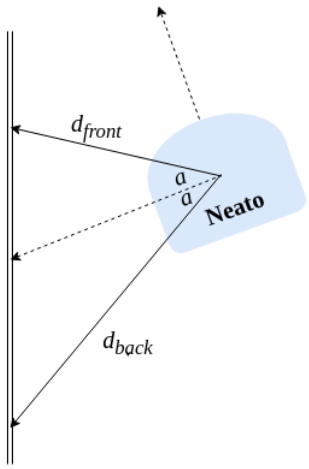
Overall, the behavior is satisfactory. When jumping in front of the Neato, it will stop, likely moving backwards before turning away. It is generally fairly conservative, particularly around walls. This is one area of improvement; the Neato seems significantly more afraid of walls than it is of smaller clusters, likely because of how walls feature more laser scan points. This could be fixed with a more aggressive fall-off distance (larger power than an inverse square).

Inebriated Wall Follow

Our first approach to wall following was meant to help us familiarize ourselves with the laser scan message and logic. We also wanted to use proportional control to get smooth behavior. At a higher level, we computed an error based on the difference between the desired position and angle relative to the wall, and the actual position and angle relative to the wall. We then used this error to turn in the direction that would minimize those values, without touching the forward speed.

We calculated the angle error by comparing pairs of range values symmetric about the side of the Neato (see diagram). The angle error is calculated as the average of the differences between d_{back} and d_{front} for each angle a. If the Neato is parallel to the wall, the angle error will average zero. If it is facing the wall, the average will be negative (since every d_{front} will be smaller than d_{back}). If it is facing away from the wall, the error will be positive. If we multiply a baseline angular velocity by the error, when the error is positive the

Neato will turn clockwise, correcting the angle. If the error is negative, it will turn counter-clockwise, once again correcting the angle. If the error is zero, then it will not turn, staying on track. We added another term to the error to maintain a given distance to the wall. With a little bit of geometry, we can determine the Neato's normal distance to the wall regardless of its orientation. The distance error term is then the difference between the desired distance to the wall, and the current distance to the wall.



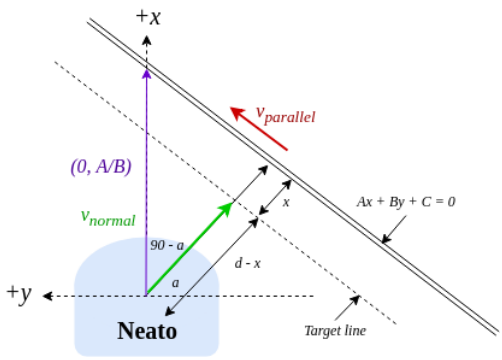
Wall Follow Diagram: To calculate the distance to the wall, the laser scan distances, symmetrical over the y-axis of the robot, are averaged over a range of angles a. The difference between d_{front} and d_{back} also provides information on the heading of the robot relative to the wall. These measurements are used to correct the Neato's trajectory parallel to the wall.

This approach was successful in giving the Neato smooth motion but quickly showed to be underdamped, resulting in a wandering motion along the equilibrium line, reminiscent of an inebriated being. This could be improved by implementing a full PID controller, particularly because the integral term would help fix the oscillation.

RANSAC Wall Follow

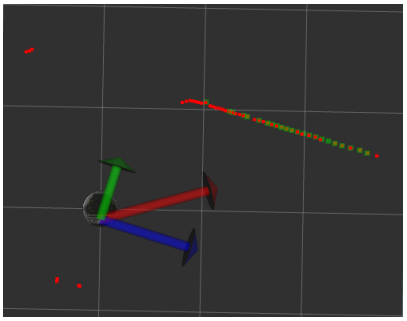
The RANSAC algorithm can be a robust way to detect walls from a set of laser scan data points. To identify a wall, the program randomly selects two points out of the set, defining a potential line which could represent the wall. Next, it uses the least-squares method to identify how far all of the points in the data set are from this potential line. If a point is within a certain distance of the line, it is called an inlier and deemed to be on the line. This process is repeated multiple times, counting the total number of inliers a potential line produces. The line with the largest number of inliers becomes the algorithm's new estimate of where the wall is. While the algorithm may not always converge and depends highly on the number of potential lines you let it check before choosing the best one, it worked reasonably well on our Neato robots. As the Neato got farther away from the wall, the estimated line tended to jump around a lot, picking up noise from other objects. In future iterations, we would implement a step in the algorithm which would update the best line only if the change from its previous estimate was below a certain threshold, attenuating these high frequency changes.

We encoded the line in terms of $Ax + By + C == 0$ instead of $y = mx + b$ form to handle vertical lines whose slopes are undefined. Once our RANSAC algorithm generated an approximation for the location of the line relative to the Neato (in the base_laser_link frame), we decided to use a carrot-and-stick approach to moving the Neato parallel to the wall and maintaining a certain distance from it by defining a target point the Neato must approach. This point, expressed as a vector, has two components, one parallel and one perpendicular to the wall. The parallel vector can be expressed as $(1, A/B)$, along the slope of the line. To calculate the normal vector, we used the same class we defined in Obstacle Avoidance to perform simple vector calculations, such as rotations by 90 degrees and finding the unit vector. The unit normal vector was then scaled by the normal distance to the wall, d, which was found by projecting the vector to the y-intercept of the line onto the normal unit vector (see diagram below).

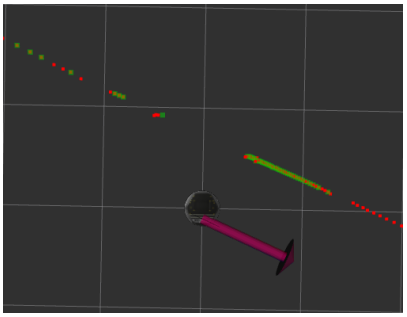


The target point was expressed as a combination of the normal and parallel vector components to the wall. We instructed the Neato to approach a point 1m along $v_{parallel}$, to keep it moving along the wall, and $d - x$ m along v_{normal}, to maintain a specified distance x from the wall.

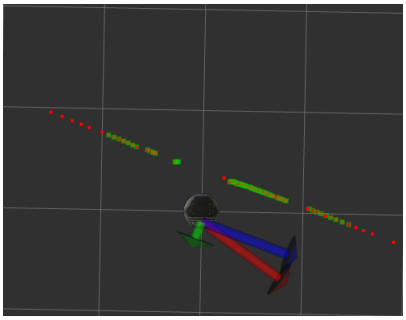
We set the target distance along the vector parallel to the wall at about 1m in front of the Neato. Thus, if it was at the correct distance from the wall, it would still be attempting to "chase" a point and move along the wall instead of standing still. For the robot to maintain a distance x from the wall, the normal unit vector was scaled by the difference between d and x. If the robot was too close, d would be too low, resulting in a negative value, pointing away from the wall. If the robot was too far, d would be greater than x and the normal vector would point towards the wall. Adding the parallel and normal components together generated an overall target point. We used Arrow Markers in Rviz to test that our algorithm was correctly identifying the wall and calculating the parallel, normal, and target vectors. Below is an Rviz visualization of the vectors in the base_laser_link frame, originating from the lidar; the robot faces the wall at a 30 degree angle. We outsourced the command velocity calculations to the Maintain_Distance node, which unfortunately does not yet work.



When the robot is too far from the wall, the green component points dominates the target vector (red), which indicates that the robot should approach the wall.



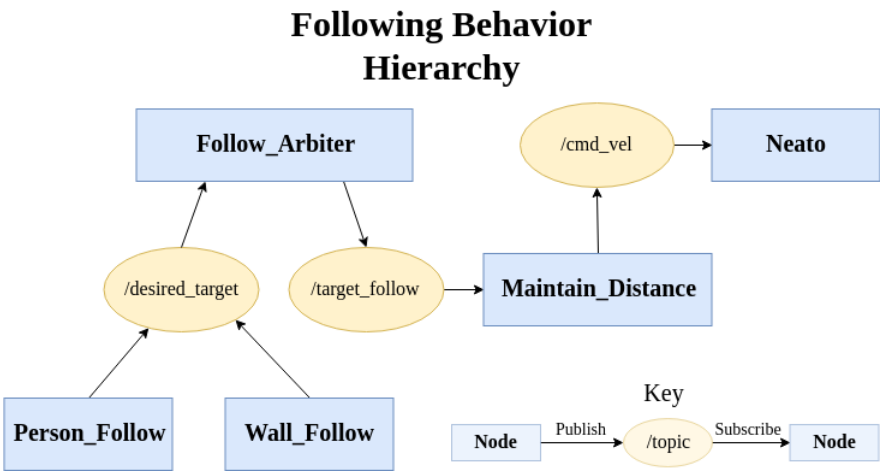
When the robot is at the correct distance from the wall, the blue component (parallel to the wall) dominates the target vector and points +1m along the wall.



When the robot is too close to the wall, the green component points away from the wall, flipping the target vector to drive the robot farther away.

Maintain Distance

The idea of the Maintain_Distance node was to provide an interface for all following behaviors, whether it be wall or person detection, by abstracting them to a target point chase, or carrot-and-stick method. Both the wall and person following behaviors are essentially the same thing: maintaining a certain distance (0 in the case of wall following) from a target point. In person following, the target is a person, while in wall following, the target is at a certain distance and angle to the wall. If implemented, this node would make possible a state diagram like the one below, where a follow arbiter would decide which target to pursue and enable more complex behaviors on the robot such as choosing to follow a wall only if there is no person to follow.



The following behavior hierarchy could be reminiscent of the Neato state map, where an the follow arbiter decides which target to pursue and enable any number of more complex behaviors on the robot.

Our goal was to use a PID control loop similar to the one we implemented in our first iteration of wall following to create a smooth motion. However, we were not able to get this behavior working in time because the robot would consistently go in the exact opposite direction of where the target vector pointed. To isolate the problem, the robot was instructed to rotate in place until its heading lined up with the target vector's, and only if it was within a few degrees to stop rotation and move forwards. It seems like the vector was always calculated to be pointing in the opposite direction of the Neato's heading, even after we transformed the point from the laser's coordinate system to the Neato's base_link frame, which is rotated by 180 degree. This problem could potentially be fixed by calculating all of the vectors in the odometry frame, taking into account the Neato's heading.

Takeaways

Our general arbiter proved to be very valuable throughout the development process. We made it simple to add new behaviors, ensuring that we would actually use it. The takeaway here is that putting thought and effort into the programming environment can improve the quality of life in programming, which is important both in being a happy coder and a productive coder. That said, there are a few kinks that we could iron out to simplify adding behaviors further. For example, we currently have to add the name of the behavior in a few places, but this could be automated via a script run through the launch file.

Another tool that proved to be extremely useful was Rviz. Choosing specific points or vectors to visualize was extremely helpful in debugging our code and hunting down where the real problems were coming from. Specifically, we visualized RANSAC's approximated line as well as the normal and parallel vectors to prove that the algorithm itself, and our math, were working.

One minor problem that we ran into was that every node's input and output used the same terminal window when used in a launch file. This is problematic for the teleop node, as it needs undisturbed input to function properly. There are settings we could specify in the file to open separate windows, but we learned too late in the project to actually implement it.

The most challenging problem in the project was prioritization. We got extremely excited about designing a robust and flexible system that could allow for infinite expansion possibilities. However, this proved to be non-trivial in the case of the following nodes and prevented us from developing the person-following and more complex behaviors. A better approach would have been to create simple implementations of all of the behaviors we wanted to include and then abstract them out or enhance them. Additionally, this workflow would have clearly showed which parts of the behaviors were redundant and would fit nicely in an object-oriented hierarchy.