Julian Shah and Tabitha Davison

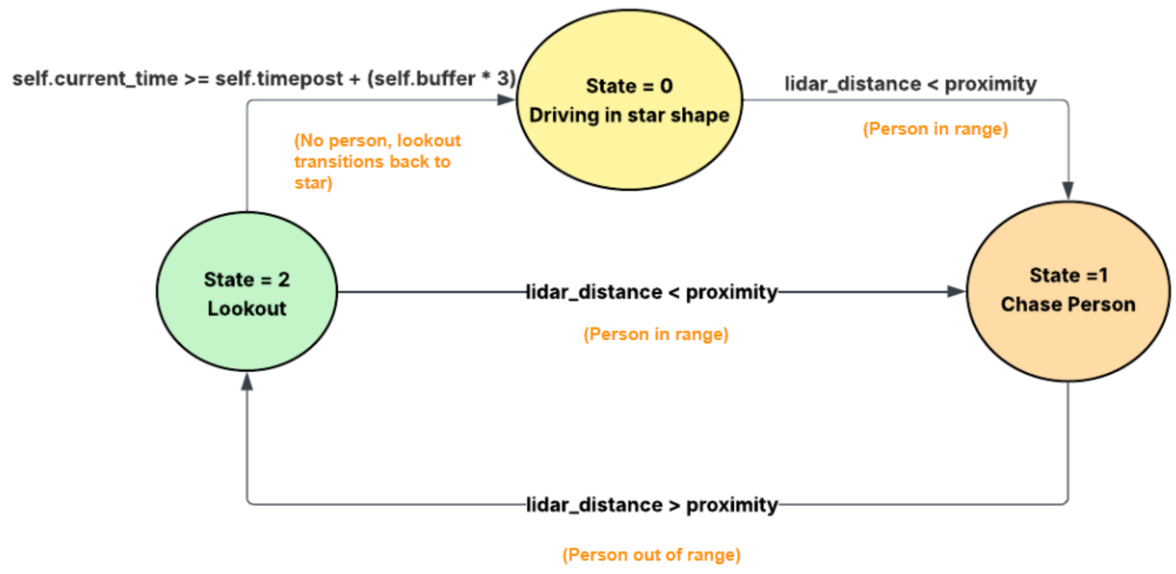
Warmup Project: Neato Sheriff

This project was an introductory warm-up exercise for CompRobo at Olin College, designed to help us gain familiarity with robotics software algorithms for perception and control. We implemented a finite state model (FSM) for a Neato robot, which we named Sheriff neato and gave it three distinct “sheriff” like behaviors:

1. Drive in star shape
2. Lookout scan (±120° pan)
3. Detect person and chase

By structuring our code as an FSM, we were able to practice switching between behaviors based on sensor input, robot state, and time. This project provided foundational experience in developing modular robot control software that includes sensing, decision-making, and movement.

Finite State Model Overview


self.current_time >= self.timepost + (self.buffer * 3)
(No person, lookout transitions back to star)
State = 0
Driving in star shape
lidar_distance < proximity
(Person in range)
State = 2
Lookout
lidar_distance < proximity
(Person in range)
State =1
Chase Person
lidar_distance > proximity
(Person out of range)

Code Layout

We implemented the FSM as a Python class, with each robot behavior written as its own function (star(), lookout(), chase()). The system transitions between states using the state_handler() function, which monitors lidar readings and timing logic.

- **State 0 – Star**: Default state. The robot drives in a five-pointed star pattern.
- **State 1 – Chase**: Triggered when lidar detects an object within 1 meter. The robot pursues the object.

- **State 2 – Lookout**: Entered when the object is out of range. The robot performs a 240° sweep (120° left, 120° right) to monitor for new targets.. If no object is found within a set time, it returns to the star-driving state.

We used threading to run both the FSM states and a state handling loop concurrently, ensuring that state transitions can occur without being blocked by a piece of executing code (for the most part). We also had other functions such as get_scan(), which processed lidar data to filter out noise and average valid ranges to estimate person position. This function ran every time a Twist() message was published over the scan/ topic.

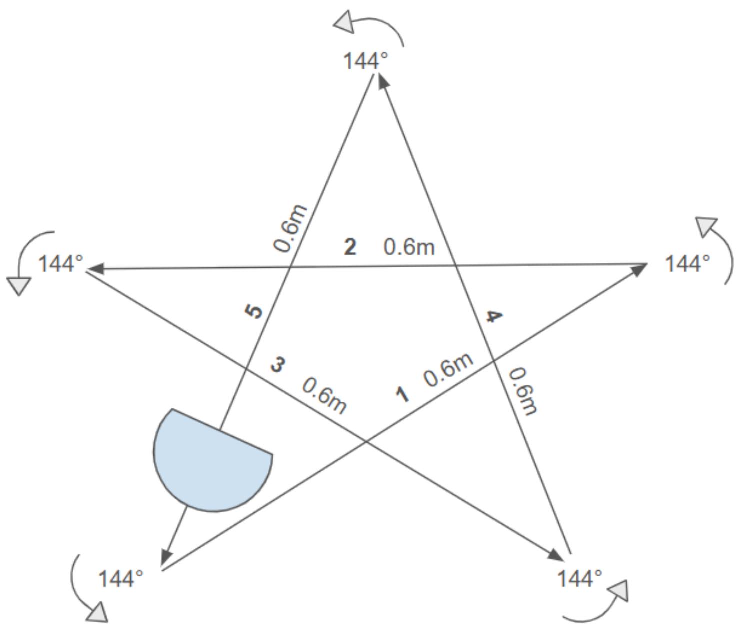
Drive in Star Shape

Description:

The star() function uses a helper function, drive(lin, ang, duration_s), which sends velocity commands for a specified duration and then stops.

Drive() sends velocity commands to the robot by publishing a Twist message with a specified linear (lin) and angular (ang) speed for a set time (duration s). It is first used to rotate the robot by 36° so the star shape is oriented with one point facing up. Then, inside a five-iteration loop, the robot drives straight for 0.6 m at 0.12 m/s, stops, and turns left 144° at 0.35 rad/s, repeating this sequence to complete the five edges of a star.

Diagram:



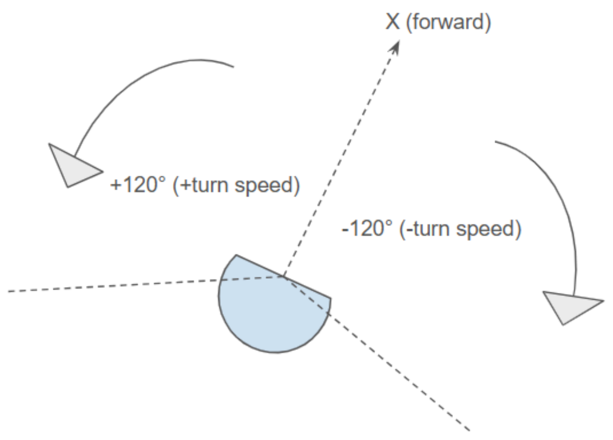
Lookout Panning and Scanning

Description:

The lookout() function uses a similar helper, turn(), which publishes angular velocity commands for a set duration.

The robot pans left 120° and then right 120°, covering a 240° window. Meanwhile, lidar data is processed to detect a person. If detected, the robot transitions into chase mode.If no person is detected after several panning cycles, the robot reverts to the star-driving state.

Diagram:



This function uses the same drive function, but named turn(), that sends velocity commands to the robot by publishing a Twist message with a specified linear (lin) and angular (ang) speed for a set time (duration s).

Turn is called outside of this function turn (turn_speed, turn_angle / turn_speed) for 120 degrees in one direction and then back in the opposite direction 120 degrees to give a 240 degrees view window.

Scanning (Lidar)

Description:
For scanning, we subscribe to the scan function and receive data from the neato lidar scan, which is filtered to remove disruptions from readings at the neato. This is done with a loop that checks which lidar distance readings are within our bots chasing range. These points are then averaged to get the direction and distance the robot should move towards before changing one of the node's instance variables for the threads to read.

Chase

This state moves towards the point specified by the scanning instance variable only if it is within our bots chase range. The 'only if' portion is handled by the scanning thread and the actual movement (publishing of velocity to the topic) is handled by the run loop thread through a match-case statement.

Object Structure

The robot functionality was executed through a one node package. The FSM node handled the logic for the three robot states, and the four different transitions. It used a load of instance variables to communicate data between threads that were running concurrently (sort of like you would use topics between nodes).

Project Learnings

This project was valuable for both technical skills and conceptual understanding shown below.

Understanding FSMs in practice: We learned about finite state models by seeing how clearly defined states simplify robot behavior design. Organizing code by state made debugging and future extensions much easier. Adding more functions or behaviours was easier in this structure.

Sensor-driven transitions: We also gained experience in using lidar data not just for mapping but also for real-time decision making. Handling noisy sensor input and building filtering logic gave us a look at perception challenges.

Threading and concurrency: Running both time-based and action-based loops concurrently proved difficult. We learned how to balance continuous behaviors with event-driven state changes, which is essential for more complex robots. There are a few issues, that we never got ironed out because of the threading, both of the movement states are blocking (despite best efforts), and therefore will only begin the chase state after completing one loop of the previous state.

In addition, we also learned the importance of checking and learning from the class and Ros2 documentation. This was crucial to this project.

Challenges

Ensuring state transitions occurred correctly was difficult, since the robot needed to both detect lidar input in real time and continue ongoing behaviors.

Thread management was another challenge, coordinating concurrent loops required careful synchronization to avoid dropped or delayed transitions.

Making sure the loops were non-blocking proved challenging.

Contribution from each team member

Julian: Designed the FSM architecture, chase behaviour, and state-handling logic.

Tabitha: Implemented the star and lookout behaviors, including the lidar scanning and panning logic

Improvements

We would improve the way that the code was structured. After realising both the power of subscriber execution functions and the limitations thereof, we believe that using individual ros nodes for the different state functionality and communication would have alleviated many of the issues where threads were blocking one another. In addition, the code would be a bit easier to look at since the nodes would be short and succinct.

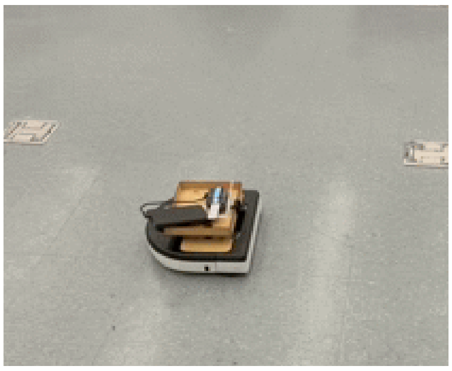
Conclusion

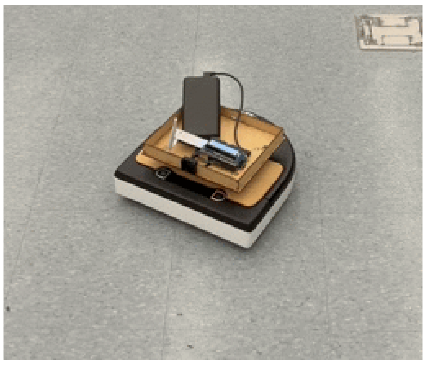
This warm-up project gave us hands-on experience with finite state machines, ROS2 communication, and lidar-based detection. By implementing star-driving, lookout scanning, and chase behaviors, we gained confidence in structuring robotic behaviors around sensor data and modular code. These skills will serve as a foundation for more advanced CompRobo projects.

Also, it might be worth looking into launch files for state machines as using threads is complicated and just using nodes and topics seems much easier.

Github Repo Link: ros_behaviors_fsm

Extra Videos


Star Pattern


Lookout