Project 2 Deliverable 3

John Ripple
CSCI
Colorado School of Mines
Golden, United States
jripple@mines.edu

***Abstract*—This document discusses one method to create a wall following robot.**

I. Introduction

A common application for robots is object detection and avoidance. This paper discusses how a robot simulated in Gazebo and ROS is able to follow walls and turns using a Q-table with Q-learning for state-based decisions.

II. Q-table

Q-table implementation for robotic control allows for extensibility into more states and the possibility of using machine learning to better optimize choosing states. The Q-table designed for this application was set up with nine states. Q-Learning with a greedy epsilon algorithm was used to choose states and actions and fill out the Q-table.

A. States

TABLE I
Definition of States

State	Front	Left	Right
0	>	>	>
1	<	>	>
2	>	>	<
3	>	<	>
4	<	>	<
5	<	<	>
6	<	<	<
7	>	<	<

A 360-degree liDar sensor is placed on the robot which can move in the x and y direction linearly and the z direction rotationally [1]. LiDar data was then read using the /scan topic [2]. The laser with the smallest distance value at +/- 10 degrees at 0 degrees and +/-45 degrees at 90 degrees and 270 degrees are used to measure the distance in front, to the left, and to the right of the robot respectively. Each state is defined as each sensor reading distance values greater than or less than the tolerance [3]. The states in Table I were chosen because it covers each side of the robot with sensors while still having minimal states to reduce the curse of dimensionality.

The Q table in Table II shows that for every state there are three possible actions. These actions were chosen because they were the minimal actions required to find and follow a wall. The three actions are turning to follow the wall, following the wall, and finding a wall. These are the minimal actions required to follow the wall keeping the Q-table dimensions to 8x3.

TABLE II
Q Table of States and Actions

States	Find wall	Turn Left	Follow Wall
0	2.9808565844326087	0.7150296469232191	0.6852970670094203
1	0	1.0960474544165195	0
2	-2.868780580472797	1.5763846853311454	4.0283334397623145
3	5.2005340807832585	1.236819794267857	1.1849904571027294
4	-14.659073051572154	0.2164383596336809	-14.622830146897051
5	0	1	0
6	-26.682280438795782	-13.338257303375574	-22.846608172632394
7	2.832833138913845	1.045735425615309	2.167094291252819

B. How Actions Were Chosen

Actions were chosen using the Epsilon Greedy algorithm. This means, initially the robot would take random actions to fill out the Q-table, and as time went on the random actions would become less likely until the robot was only choosing the action with the highest value in a state. The degradation of random actions was done with an exponential decay function of $10^{-\gamma * t}$ where gamma was chosen to make the exponential function be 0.1 after a desired number of episodes had occurred.

C. Rewards

Rewards are used in reinforcement learning to tell the robot if it is doing well or not. For this application, the reward was an exponential function that increased the reward (up to 1) the closer the robot got to the desired distance from the wall. This allowed the robot to always get feedback as it moved around the environment and encouraged it to get to the desired distance from the wall. A graph of the reward function is shown in figure 1. An exponential was chosen over a linear reward because it encouraged the robot to get closer far more and discouraged being farther away faster than a linear reward would. The only time the exponential function is not used as a reward is when the robot is touching the wall. When the robot touches the wall a set negative reward is applied to tell the robot that touching the wall is the worst action it can take. The overall reward function is shown in equation 1 where x is the desired distance minus the sensor distance reading.

$$Reward = \begin{cases} 10^{-\gamma * x} & x > 0 \\ 10^{\gamma * x} & -0.1 < x < 0 \\ -1 & x < -0.1 \end{cases} \quad (1)$$

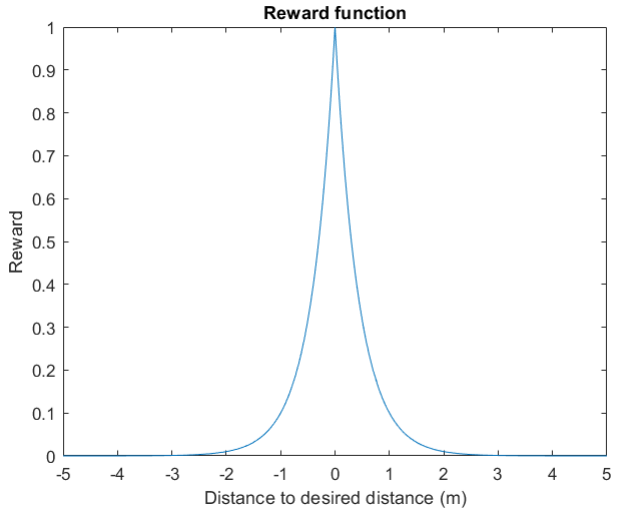

Fig. 1. Reward function was the robot gets closer to a desired distance.

D. Updating Q Table

The Q Table was updated at the end of every action. When an action was completed, depending on the state a reward was chosen by inputting a sensor value and desired distance into the reward function. This reward was then used in the Bellman Q-Learning function to update the Q table. The general pseudo code for this operation is shown in figure 2 with the Bellman equation. Alpha and gamma values were experimentally chosen in the Q update function based on what worked resulting in $\alpha = 0.1$ and $\gamma = 0.8$.

Q-learning (off-policy TD control)

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Fig. 2. The Q-learning algorithm.

III. Test Environment

The test environment was set up to be a maze the robot could follow a wall around. The robot starts in a random position and finds a wall then follows that wall around the course as shown in figure 3.

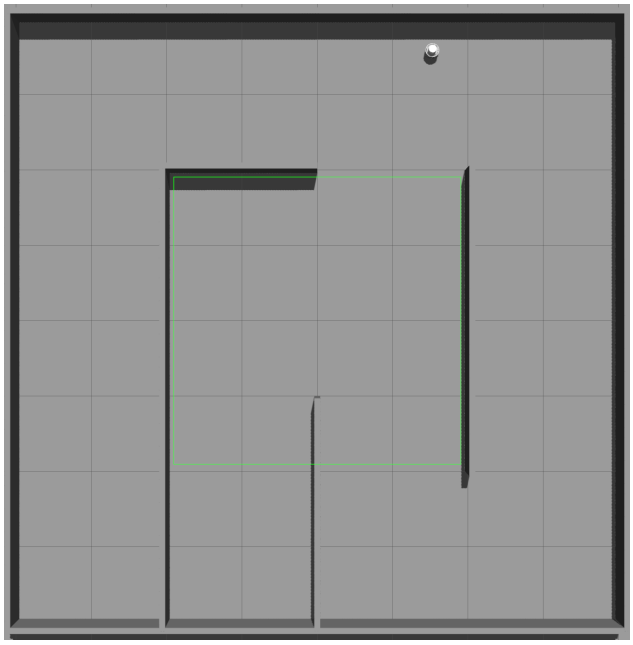

Fig. 3. Robot in test environment.

IV. Performance of the Robot

Running the robot through the test environment allowed for a full circuit of the course and when tested on the standalone wall the robot was able to make 180-degree turns. A class-based implementation with two P controllers was used to make the robot follow a wall [4]. The first P controllers would rotate the robot while it was not aligned with the wall it was following. The second P controller would control the distance the robot was from the wall. These controllers were only used when the follow wall action was chosen from the Q-table. Besides the movement as the robot gets into the correct distance from the wall, there is very little to no wobble while the robot follows the wall. A PID controller was tested as well but did not yield better results than the two P controller method [5].

Since a Q table implementation was used with sensors on the left, front, and right of the robot, the code is generalizable to increase the number of sensors used (more states) and create more actions allowing for a more robust wall follower to be implemented if necessary.

To view a video of the robot demonstrating wall following capabilities click the link below.
Robot wall following link

References

[1] N. Gyory, "HCRLab / Stingray Robotics / Stingray-Simulation · GitLab," GitLab. https://gitlab.com/HCRLab/stingray-robotics/Stingray-Simulation (accessed Mar. 03, 2023).

[2] "Sensor_msgs/laserscan message," sensor_msgs/LaserScan Documentation, Mar. 02, 2022. [Online]. Available: http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html. [Accessed: Mar. 01, 2023].

[3] M. Arruda, "Exploring ROS with a 2 wheeled robot 7 - Wall Follower Algorithm," The Construct, May 23, 2019. https://www.theconstructsim.com/wall-follower-algorithm/ (accessed May 23, 2019).

[4] A. Parundekar, “turtlesim/Tutorials/Go to Goal - ROS Wiki,” Ros.org, Oct. 07, 2021. http://wiki.ros.org/turtlesim/Tutorials/Go%20to%20Goal (accessed Mar. 01, 2023).
[5] M. Lundberg, “simple-pid: A simple, easy to use PID controller,” PyPI, Apr. 11, 2021. https://pypi.org/project/simple-pid/ (accessed Mar. 02, 2023).