CSCI-473/573 Human-Centered Robotics
Project 2: Reinforcement Learning for Robot Wall Following

Project Assigned: Feb 15, 2021
Multiple due dates (all deliverables must be submitted to Canvas)
Deliverable 1 (Gazebo Simulation) due: Feb 22, 23:59:59
Deliverable 2 (Problem Formulation) due: March 8, 23:59:59
Spring Break: March 18–26
Deliverable 3 (Reinforcement Learning) due: April 3, 23:59:59

In this project, students will design and implement reinforcement learning algorithms to teach an autonomous mobile robot to follow a wall and avoid running into obstacles. Students will be using the Gazebo simulation in ROS Melodic to simulate an omni-directional mobile robot named Triton, and using an environment map that is provided to you. Students will be using a laser range scanner on the robot to perform sensing and learning, where the robot is controlled using steering and velocity commands. Students are required to program this project using C++ or Python in ROS Melodic running on Ubuntu 18.04 LTS (i.e., the same development environment used in Project 1). Also, students are required to write a report following the format of standard IEEE robotics conferences using LaTeX. Please **START EARLY!**

A note on collaboration: Students are highly encouraged to discuss this project amongst yourselves for the purposes of helping each other understand the simulation, how to control the robot, how to read sensor values, etc. This collaboration is solely for the purpose of helping each other get the simulation up and running. You may also discuss high-level concepts in helping each other understand the reinforcement learning algorithms. However, each person must individually make their own decisions specific to the learning aspects of this project. In particular, this means that individuals must make their own decisions over the representations of states, actions, and rewards to be designed in this project, and must implement their own learning algorithms and prepare all the written materials to be turned in on their own.

I. The Wall Following Problem

Wall following is a common strategy used for navigating an environment. This capability allows an autonomous robot to navigate through open areas until it encounters a wall, and then to navigate along the wall with a constant distance. The successful wall follower should traverse all circumferences

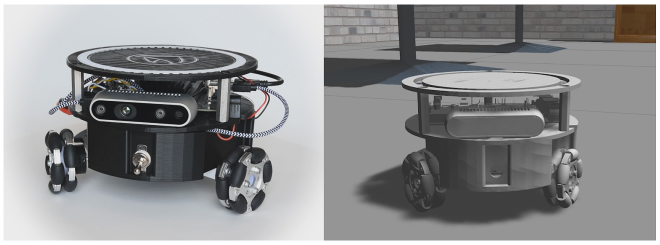


Fig. 1: The Triton robot and its simulation in Gazebo.

of its environment at least one time without getting either too close, or too far from walls, or running into obstacles. Many methods were applied to this problem, such as control-rule based methods and genetic programming. In this project, students are required to solve the problem using reinforcement learning.

II. Deliverable 1: Gazebo and SLAM

Before working directly on the wall following robot, students will first familiarize themselves with using Gazebo in ROS. Gazebo is a powerful tool used in robotics for robot simulation. It allows you to simulate a robot and its environment for testing or data-driven methods. In order to simulate the wall-following robot for reinforcement learning we will use a gazebo. As a tutorial for utilizing gazebo you are expected to launch a robot in the gazebo environment, map the environment using SLAM, and understand what ROS topics are being used. To do this you will follow the steps below:

1) Install ROS packages for simulating Turtlebot3 in Gazebo, mapping, and planner:
```
sudo apt-get install ros-melodic-turtlebot3*
sudo apt-get install ros-melodic-slam-gmapping
sudo apt-get install ros-melodic-dwa-local-planner
```
2) Go through Gazebo beginner tutorial 1-3 "Overview and Installation, Understanding the GUI, and Model Editor" to get familiar with Gazebo: http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b2

If you have any questions or comments, please contact the instructors Mark Higger at mhigger@mines.edu, Terran Mott at terranmott@mines.edu or TA Shane Copenhagen at scopenhagen@mines.edu.
This write-up is prepared using LaTeX.

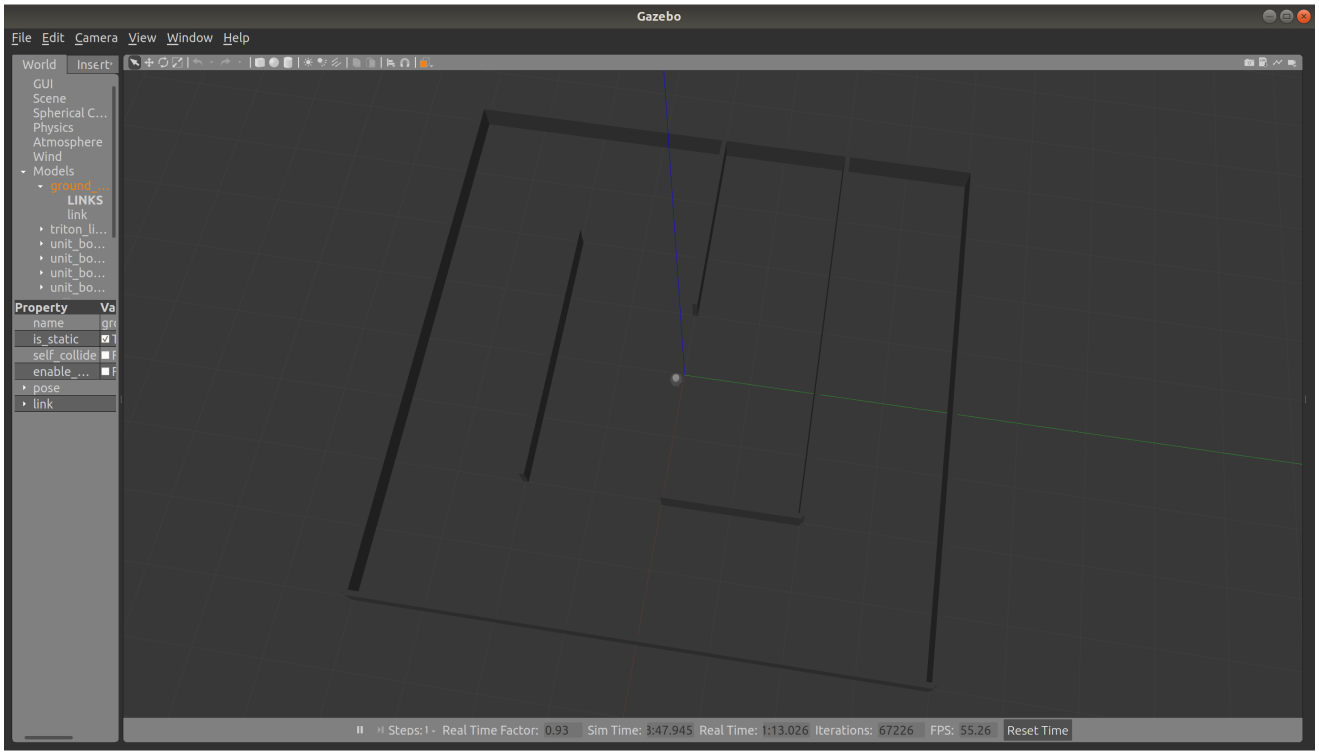


Fig. 2: Gazebo simulation of the Triton robot and the experiment environment for wall following.

3) Go through TurtleBot3 Simulation tutorial from 6.1.1 "Install Simulation Package" to 6.3.4 "Set Navigation Goal." `https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/`
   Notes: Choose waffle_pi as your TURTLEBOT3_MODEL since it is also equipped with a camera.
   Make sure you select Melodic at the top of the page.
   You will need to (1) load a pre-defined simulated world, (2) execute RViz to visualize the robot sensing, (3) build a map of your simulated world, and (4) use RViz to navigate through the world.
4) Find what ROS topic gazebo is subscribing to control the robot and what topics you would need to subscribe to for getting lidar data. Students should use rqt_graph to find the available ros topics: `http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics`

III. Gazebo Simulation of Triton in ROS Melodic

This project will use a simulated Triton robot as illustrated in Fig. 1. The Gazebo simulator of Triton can be set up by following instructions in the GitLab repository: `https://gitlab.com/HCRLab/stingray-robotics/Stingray-Simulation`.

The Gazebo simulation allows for visualizing the Triton robot and the world, controlling the robot via ROS topics and services, and visualizing robot sensing and navigation path in RViz. Students are required to follow the instructions in the repo's README and become familiar with basic functions before implementing your package.

The default simulation environment can be launched via:

`wall_following_v1.launch`

which is available at:
Stingray-Simulation/catkin_ws/src/stingray_sim/launch/.

After launching, you will see a simulation environment that is similar to Fig 2. To experiment with other maps, you need to place a new world file under the *worlds/* directory in the *stingray_sim* package and call `roslaunch` with a world_file argument.

The robot movement is controlled using the ROS topic `/triton_lidar/vel_cmd`. This topic accepts a message of type `geometry_msgs/Twist`, the structure of this message can be found at:
`http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html` The laser readings of the robot is obtained by listening on the topic of `/scan`, which uses the message of type `LaserScan`. The structure of this message can be found at:
`http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html`

Communication services between ROS and Gazebo are necessary to let the robot repeatedly explore the environment. You can see all available services while the simulation environment is launched by typing the command `rosservice list` in a new terminal. You are likely to use the following services in this project:

- `/gazebo/reset_simulation`
- `/gazebo/get_model_state`

- `/gazebo/set_model_state`
- `/gazebo/unpause_physics`
- `/gazebo/pause_physics`

Explanation of each service can be found at:
`http://gazebosim.org/tutorials/?tut=ros_comm#Tutorial:ROSCommunication`.
Examples of calling services in C++ and Python are presented in ROS Tutorial 1.1.14 and 1.1.15, respectively.

Any questions you have on Gazebo and Triton simulation should be directed to the TA, and cc to the instructor.

IV. Deliverable 2: Problem Formulation

As part of the project, students are responsible for formulating the wall following problem. Specifically, in this part of the project, students will need to determine how to represent the input state (based on the input Lidar data) and the motor actions, and the best level of discretization of these values. Please note that you are NOT allowed to use the grid-world representation of states. States should be a function of sensor values (and other measurements, as appropriate), for example, as implemented in [1]. As always, there is not just one single way to define states and actions. Explore various design possibilities and find the one that you believe works best.

Based upon the discretized states and actions, students are required to implement a Q-table and manually set the values of the Q-table to determine the policy. The manually selected Q-values will be applied for two purposes:

- Allowing the robot to follow a straight wall using these manually defined Q-table as the policy.
- Using these manually defined Q-values to encode expert knowledge to facilitate Q-learning in Deliverable 2.

In Deliverable 2, students are required to demonstrate that the manually defined Q-values as the policy enable the robot to follow a straight wall. Accordingly, the ROS package you submit in Deliverable 2 should allow a user (e.g., the TA or instructor) to test the policy manually defined in the Q-table for a robot to follow a straight wall in Gazebo. This means that (1) your submitted ROS package in Deliverable 2 should include your manually defined Q-table, and (2) a launch file designed to start the whole simulation to show the capability of robot following a straight wall in Gazebo.

Additionally, you are required to use LaTex to generate a 1-2 page report in IEEE 2-column conference format for deliverable 2. You can find the IEEE LaTex template at `https://www.overleaf.com/gallery/tagged/ieee-official`. To write this report in LaTex it is recommended to use Overleaf, which provides a web platform for writing and compiling documents in LaTex. A tutorial for how to use LaTex and overleaf can be found here: `https://www.overleaf.com/learn/latex/Tutorials`.

A. ***CSCI-473***: *What to Submit for Deliverable 2*

CSCI-473 students are required to submit a single tarball, named *D1_firstname_lastname.tar* (or .tar.gz) to the Canvas portal named P2-D1, which must contain the following items:

- A 1-2 page **Report** written in LaTex which contains:
  - An introduction to the problem
  - Definitions of your states and actions as well as why those states and actions were chosen
  - A table including your manually defined Q-values for each state-action combination (HINT: this table should n_{states} x $n_{actions}$ in size and populated by mostly zeros)
  - A screenshot of your robot in the simulation environment
  - Some notes on the performance of the robot (i.e. Does your solution wobble? Is it generalizable to corners and turns?)
  - Bibliography containing any references that you used to complete this assignment (including [1]).
- Your **ROS package** (that must include the source files, launch files, package.xml, CMakeLists.txt, world files, README, etc.) to demonstrate the robot capability of following a straight wall. The launch file must automatically start your demonstration in Gazebo (as shown in Fig. 2). The README file must provide sufficient information of your package, and clearly describe how to use the launch file to run the demonstration.
- A short **demo video** showing that the robot successfully follows a straight wall (no need to implement or show the capability of turning around wall corners; following a *straight* wall is sufficient for this deliverable). You can either submit a video or provide a link to the video on YouTube. If you are submitting a video, make sure the video size is less than 5M. You can use *ffmpeg* to speed up the video in Ubuntu. If you choose to provide a link, you are responsible to ensure that the video link allows public access.

B. ***CSCI-573***: *What to Submit for Deliverable 2*

CSCI-573 students are required to submit a single tarball, named *D1_firstname_lastname.tar* (or .tar.gz) to the Canvas portal named P2-D1, which must contain the following items:

- A 1-2 page **Report** written in LaTex which contains:
  - An introduction to the problem
  - Definitions of your states and actions as well as why those states and actions were chosen
  - A table including your manually defined Q-values for each state-action combination (HINT: this table should n_{states} x $n_{actions}$ in size and populated by mostly zeros)
  - A screenshot of your robot in the simulation environment
  - Some notes on the performance of the robot (i.e. Does your solution wobble? Is it generalizable to corners and turns?)
  - Bibliography containing any references that you used to complete this assignment (including [1]).
- Your **ROS package** (that must include the source files, launch files, package.xml, CMakeLists.txt, world files, README, etc.) to demonstrate the robot capability of

following a straight wall. The launch file must automatically start your demonstration in Gazebo (as shown in Fig. 2). The README file must provide sufficient information of your package, and clearly describe how to use the launch file to run the demonstration.

- A short **demo video** showing that the robot successfully follows a straight wall (no need to implement or show the capability of turning around wall corners; following a *straight* wall is sufficient for this deliverable). You can either submit a video or provide a link to the video on YouTube. If you are submitting a video, make sure the video size is less than 5M. You can use *ffmpeg* to speed up the video in Ubuntu. If you choose to provide a link, you are responsible to ensure that the video link allows public access.

V. Deliverable 3: Reinforcement Learning

In this deliverable, students will implement reinforcement learning algorithms. Before starting to code your algorithms, you need to read the lectures to understand them:

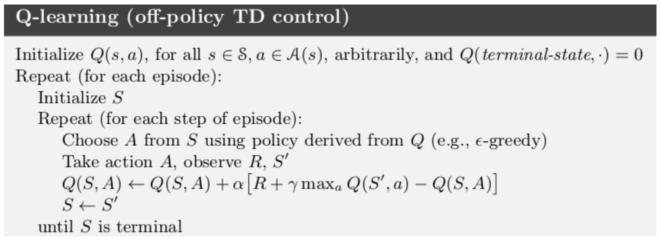
After understanding the lecture, please **START EARLY!** It may take a while for the reinforcement learning algorithms to converge.

In class, we discussed two classic reinforcement learning algorithms, including Q-learning (Fig. 3) and SARSA (Fig. 5). This Deliverable 2 mainly focuses on expanding the ROS package you already developed in Deliverable 1 and implementing the reinforcement learning algorithms to update the Q-values through learning (instead of using manually defined values). In particular, your reinforcement learning algorithms should satisfy the following requirements:

- Temporal Difference (TD) with a pre-defined learning rate (i.e., StepSize) must be implemented to update the Q-values in an incremental and iterative fashion.
- ϵ-greedy policy must be implemented with a pre-defined ϵ value to balance exploration and exploitation.

These requirements are already included in the provided Q-learning and SARSA algorithms. You may use your manually defined Q-values in Deliverable 1 as the initialization, or set all Q-values to 0 in order to let the algorithms to learn from scratch without prior knowledge.

As part of the deliverable, you are responsible for fine-tuning how you will represent the input state and the motor actions, and the best level of discretization of these values. In addition, you will need to decide the reward function you'll use for learning; presumably the robot will have a positive reward for following a wall and a negative reward for running into a wall. The robot may also have a component of the reward function that rewards moving longer distances along a wall, rather than have the robot creep along the wall. You may also define your reward function any way you like, including the use of scaled or graduated rewards as a robot moves toward or away from a wall, if you find such rewards helpful. Moreover, you can decide yourself where the starting position of the robot will be for each learning episode. You'll need to determine how many episodes are needed for your program to learn.



Q-learning (off-policy TD control)

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

Fig. 3: The Q-learning algorithm.

A. *CSCI 473: What to Submit for Deliverable 3*

CSCI 473 students are required to implement Q-Learning ONLY (NO requirement to implement SARSA), and submit a single tarball, named *D3_firstname_lastname.tar* (or .tar.gz) to the Canvas portal named P2-D3, which must contain the following **three items**:

- Your **ROS package** (that must include the source files, launch files, package.xml, CMakeLists.txt, world files, README, etc.) that implements Q-learning. Your ROS package should include TWO options – one option is the reinforcement learning algorithm in learning mode, and the second option is to test the best learned policy for an arbitrary robot starting position. This means that your package submission should include the best policy you found during your own training, so that you can use those during testing mode. Your package should allow the user to specify whether the code should be run in the training mode or the testing mode. It is up to you to design the interface to allow the user to specify which mode the code should be in, e.g., by implementing two launch files or by implementing one launch file with arguments. The README file must provide sufficient information of your package, and clearly describe how to use the launch file(s) to run your package in either training or testing mode.
- A **demo video** showing that your policy learned by your Q-learning algorithm enables the robot to successfully follow the wall in ALL FIVE scenarios defined in Fig. 4. You can either submit a video or provide a link to the video on YouTube. If you submit a video, make sure the video size is less than 20M. You can use *ffmpeg* to speed up the video in Ubuntu. If you choose to provide a link, you are responsible to ensure that the video link allows public access.
- A 2-3 page **Report** written in LaTex which contains:
  - An introduction to the problem
  - Definitions of your states and actions as well as why those states and actions were chosen
  - What your rewards were and why they were chosen
  - How you chose actions (i.e. Epsilon Greedy)
  - The algorithm you used to update your Q-table, including an equation with the bellman update equation
  - Some notes on the performance of the robot.
  - Bibliography containing any references that you used to complete this assignment (including [1]).

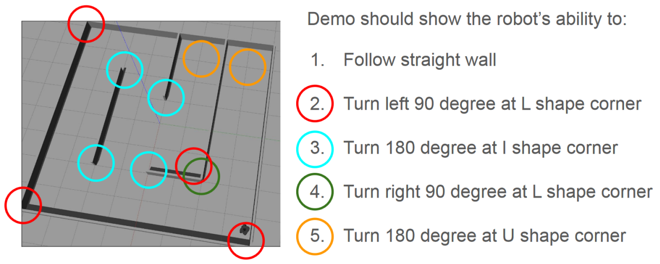

Demo should show the robot's ability to:
1. Follow straight wall
2. Turn left 90 degree at L shape corner
3. Turn 180 degree at I shape corner
4. Turn right 90 degree at L shape corner
5. Turn 180 degree at U shape corner

Fig. 4: The five scenarios in which the robot should be able to follow the wall using the learned policy.

Sarsa (on-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 $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Fig. 5: The SARSA algorithm.

B. *CSCI 573: What to Submit for Deliverable 3*

CSCI-573 students are required to implement and submit both Q-Learning (Fig. 3) and SARSA (Fig. 5), and compare them. For this deliverable, you need to submit a single tarball, named *D3_firstname_lastname.tar* (or .tar.gz) to the Canvas portal named P2-D3, which must contain the following 5 items:

- Your **ROS package** (that must include the source files, launch files, package.xml, CMakeLists.txt, world files, README, etc.) that implements Q-learning. Your ROS package should include TWO options – one option is the reinforcement learning algorithm in learning mode, and the second option is to test the best learned policy for an arbitrary robot starting position. This means that your package submission should include the best policy you found during your own training, so that you can use those during testing mode. Your package should allow the user to specify whether the code should be run in the training mode or the testing mode. It is up to you to design the interface to allow the user to specify which mode the code should be in, e.g., by implementing two launch files or by implementing one launch file with arguments. The README file must provide sufficient information of your package, and clearly describe how to use the launch file(s) to run your package in either training or testing mode.
- Your **ROS package** (that must include the source files, launch files, package.xml, CMakeLists.txt, world files, README, etc.) that implements SARSA. Your ROS package should include TWO options – one option is the reinforcement learning algorithm in learning mode, and the second option is to test the best learned policy for an arbitrary robot starting position. This means that your package submission should include the best policy you found during your own training, so that you can use those during testing mode. Your package should allow the user to specify whether the code should be run in the training mode or the testing mode. It is up to you to design the interface to allow the user to specify which mode the code should be in, e.g., by implementing two launch files or by implementing one launch file with arguments. The README file must provide sufficient information of your package, and clearly describe how to use the launch file(s) to run your package in either training or testing mode.
- A **demo video** showing that your policy learned by your Q-learning algorithm enables the robot to successfully follow the wall in ALL FIVE scenarios defined in Fig. 4. You can either submit a video or provide a link to the video on YouTube. If you submit a video, make sure the video size is less than 20M. You can use *ffmpeg* to speed up the video in Ubuntu. If you choose to provide a link, you are responsible to ensure that the video link allows public access.
- A **demo video** showing that your policy learned using your SARSA algorithm enables the robot to successfully follow the wall in ALL FIVE scenarios defined in Fig. 4. You can either submit a video or provide a link to the video on YouTube. If you submit a video, make sure the video size is less than 20M. You can use *ffmpeg* to speed up the video in Ubuntu. If you choose to provide a link, you are responsible to ensure that the video link allows public access.
- A 2-3 page **Report** written in LaTex which contains:
  - An introduction to the problem
  - Definitions of your states and actions as well as why those states and actions were chosen
  - What your rewards were and why they were chosen
  - How you chose actions (i.e. Epsilon Greedy)
  - The algorithm you used to update your Q-table for Q-learning
  - The algorithm you used to update your Q-table for SARSA
  - A description of On-Policy vs Off-policy learning
  - Detailed information on the final performance of the robot as well as the learning algorithm. This should contain quantitative metrics such as total cumulative reward per episode during training.
  - Bibliography containing any references that you used to complete this assignment (including [1]).

This means your submitted single tarball must include two ROS packages, for Q-Learning and SARSA, respectively.

In addition, during your implementation and testing, you are suggested to collect the following information to compare both algorithms

- Rate of convergence of learning (e.g., in terms of the number of episodes).
- Effect of different reward assignments on the quality and speed of learning.

	Excellent (100%)	Need Work (70%)	Poor(30%)
Code (10 points)	Code runs without issues and is clearly documented	Code contains significant bugs	Code does not run
Demonstration (10)	Robot follows wall without significant swerving	Robot follows the wall but frequently stops or swerves	Robot does not follow the wall
Write-up (10)	Write-up contains all the necessary sections, is well written and well formatted	Write-up does not contain sufficient information or is not well written	Writup is not formatted or is missing significant sections of work

TABLE I: Grading rubric for Deadline 3

- Change of the accumulated reward as the number of episodes increases.

You may also analyze the effect of the layout of the learning environment (if you want to try different environments), or the effect of different state and motor output representations (e.g., using more or less resolution) on the quality and speed of learning.

References

[1] D. L. Moreno, C. V. Regueiro, R. Iglesias, and S. Barro, "Using prior knowledge to improve reinforcement learning in mobile robotics," *Proc. Towards Autonomous Robotics Systems*, 2004.