Institute for Aerospace Studies
UNIVERSITY OF TORONTO

LABORATORY II

Planning and Navigation

ROB521 Mobile Robotics and Perception
Spring 2023

1 Introduction

This is the second laboratory exercise of ROB521-Mobile Robotics and Perception. The course will encompass a total of four labs, all of which are to be completed in the scheduled practicum periods. Each of the four labs will grow in complexity and are intended to demonstrate important robotic concepts presented during the lectures. Our robot of choice: *Turtlebot 3 Waffle Pi* running the operating system *ROS*.

1.1 Objective

The objective of this lab is to develop two global planners (**RRT** and **RRT*** from the Lecture *Using Vehicle Models in Planning - W23*) and one local planner/controller (**trajectory rollout** from the Lecture *Introduction to Planning and Graphs - W23*) that can navigate the robot from a starting to goal point. In particular, your goal is to develop both of the following:

- *An optimal planner that returns a collision free path from the starting point to the goal point.*
- *A local planner/controller that navigates the robot through the path acquired from your optimal planner.*

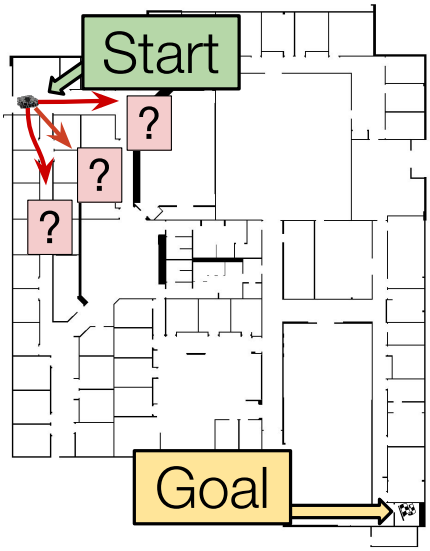

Figure 1: The willow garage map with the robot start and goal positions.

1.2 Lab Deliverables

You will be required to demonstrate functionality in both simulation and real-life environment. In this (and future) labs, look for a summary of the deliverables and their mark distribution at the end of the document. If you have to submit a lab report, include a short description of what your code is doing, and what the robot does as a result of running the code. Provide context that will allow the TA's to understand what your robot is doing in order for them to properly gauge your success. Importantly, state whether you were able to complete the deliverable. If not, explain why you weren't able to complete the task. Avoid writing more than a paragraph for each deliverable. For more details, see Section 2.3.

Finally, include a copy of your code for the TA's to look over. The code itself will not be marked, but it must be presented to demonstrate that each group has independently written their solution. If the code does not look complete, TA's will have access to all of

the computers, and will be able to run the code themselves to confirm that the deliverables have been completed.

2 Part A: Simulation

2.1 Getting Started

2.1.1 Running the RRT and RRT* Planning Skeleton Code

The `nodes/l2_planning.py` script should be run using the command `./l2_planning.py` from within the `nodes` folder. This calls the function main() in `l2_planning.py`, which produces a numpy[1] (.npy) file of the path your graph follows to the goal point. This numpy file will be used to drive your robot in the `l2_follow_path.py` script. The robot's environment is represented in 2D by the image `willowgarageworld_05res.png`. The goal of this lab is to determine the path from the top left to the bottom right room (see Fig. 1) [2]. The file `pygame_utils.py` has been included to help visualize your code running.

2.1.2 Running the Trajectory Rollout Node

The `nodes/l2_follow_path.py` script defines a class for creating a ROS node. First, launch the turtlebot in our simulated Willow Garage environment with:

```
roslaunch rob521_lab2 willowgarage_world.launch
```

Next, in a different terminal, you can launch RVIZ, which will show you a top-down view of the robot and the map:

```
roslaunch rob521_lab2 map_view.launch
```

Finally, in a third terminal, you can run your `path_follower` node:

```
rosrun rob521_lab2 l2_follow_path.py
```

Initially, it will just output a series of print statements in a loop with various tasks for you to complete. Once you have finished the lab, you will see (in RVIZ) the global plan from RRT(*), the local path from trajectory rollout, and the robot moving through the environment.

[1] Documentation for the version on the lab computers can be found at the link provided in the Additional Resources section

[2] Your planning algorithm may take 10 to 20 minutes to find the path, so start with planning a short path to verify your code.

2.2 Assignment

Two skeleton code files have been provided - `l2_planning.py` and `l2_follow_path.py`. The `l2_planning.py` file provides a framework to support the development of an RRT* algorithm. The framework inside the `l2_planning.py` file includes a Node object for constructing your graph and a planning object with empty functions that you will need to fill in. The `l2_follow_path.py` file provides the structure to implement a trajectory rollout algorithm that drives the robot along the waypoints identified in RRT*. The skeleton provided is only one way of organizing your RRT* code. Please feel free to make changes to the skeleton code if your solution requires it.

There are a large number of components to this lab, which can make completing this lab time consuming to complete and difficult to debug. We recommend that you divide up the tasks among group members and develop unit tests to verify that your functions perform as anticipated. While there are many ways to divide this assignment, we recommend that you complete Tasks 1 and 2 in parallel. Then, complete Tasks 3 and 8 in parallel. At this point, you should have a functioning RRT algorithm (Task 3) as well as trajectory rollout (Task 8). After completing Task 3, Tasks 4 and 5 can be completed in parallel. Once Tasks 4 and 5 are complete, the last two tasks to complete are Tasks 6 and 7.

2.2.1 Task 1: Collision Detection

For this task, you will be determining if a given robot position collides with a obstacle in the provided.

- The first function to complete is `point_to_cell`. This function determines the map pixel that a given metric point occupies. The necessary information to compute this transform is provided in the map's associated yaml file. Hint: The vector `origin` is the offset, (x, y, theta), of a reference frame in the lower left hand corner of the map from the origin.

- The second function to complete is `points_to_robot_circle`. Given a set of potential robot x-y positions, this function determines the cells that the robot would occupy at such positions. You may assume that the robot is fully enclosed by a 2D circle. We have provided a function `circle`[3] that determines the cells that a circle will occupy. Finally, `points_to_robot_circle` should call the `point_to_cell` function at least once.

2.2.2 Task 2: Simulate Trajectory

For this task you will be simulating the robot motion that drives the robot towards a sampled point. You may assume a constant velocity model and ignore robot dynamics (i.e the robot can instantaneously change velocities when it transitions between edges on the RRT* graph).

[3] documentation for circle can be found in the Additional Resources section.

- The first function to complete is `trajectory_rollout`. Given a linear and rotational velocity, this function computes the trajectory of the robot. Since you will need to check for collisions along this path, this function will need to output a series of points along the trajectory. As a result, the output of this function should be of shape 3-by-num_timesteps. You should use the unicycle robot kinematic equations from the Lecture *Vehicle Models - W23* slide 5 to simulate the motion of the robot.

- The second function to complete is `robot_controller`. Given a point, this function determines a linear and rotational velocity that would nominally drive the robot towards the given point. Hint: This function is very open-ended and there is more than one solution to this problem. As long as the robot ends up closer to the given point than when it started, your function is sufficient.

- Now you should be able to complete the `simulate_trajectory` function. Given a node and an x-y point, this function drives the robot from the node towards the point. You should be able to combine the previous two functions to make this function.

2.2.3 Task 3: Combine Task 1 and 2 for a functioning RRT planning algorithm

At this point, you have completed the majority of a RRT algorithm. In this section, you will create an RRT planning algorithm.

- First, you must complete `sample_map_space`, which gives a random point within the map space. Hint: You may choose a subset of the map space to sample in.

- Second, you must complete `check_if_duplicate`, which checks if a node is a duplicate of a node already in the list.

- Third, you must complete `closest_node`, which finds the closest node given an x-y point.

- Finally, you should now be able to complete the `rrt_planning` function. Follow the RRT algorithm outlined in the Lecture *Using Vehicle Models in Planning - W23* slide 15. You should take some time to verify that your RRT planning function behaves as expected. Two maps, `simple_map.png` and `willowgarageworld_05res.png`, have been provided for testing. We recommend verifying your code on `simple_map.png` before attempting `willowgarageworld_05res.png`.

2.2.4 Task 4: Rewiring Existing Nodes

In order to rewire existing nodes, we need a function that generates a trajectory to connect a node to another node. This function is required to evaluate if there is a collision free trajectory connecting the two nodes. To generate this trajectory, you will complete the function `connect_node_to_point`. Hint: Under a constant velocity assumption, connecting

two robot poses (nodes) is an over-constrained problem. A simple solution is to ignore the ending theta constraint.

2.2.5 Task 5: Trajectory Costs

RRT* requires the knowledge of a cost-to-come. In the `cost_to_come` function, you should implement a cost to get from one node to another. There are many solutions to this problem, so feel free to experiment with different cost functions.

2.2.6 Task 6: Updating Children

Finally, the costs of nodes will change as you perform rewiring. As a result, you will need to propagate the change in costs (depending on your `connect_node_to_point` function) to downstream nodes. Updating the costs of children can be performed in the `update_children` function. This function relies on the `connect_node_to_point` function from the previous task.

2.2.7 Task 7: Combine all Previous Parts for RRT Star

At this point, you have the majority of the components to implement a RRT* planning algorithm. You should follow the Lecture *Using Vehicle Models in Planning - W23* slides 45 through 60 to complete this function. For an example RRT* output, see the file `path_complete.npy`. Again, we recommend verifying your code on `simple_map.png` first.

2.2.8 Task 8: Using Trajectory Rollout for Local Planning

If we attempted to use the velocity commands used for planning in RRT(*), we would be using “open-loop" control, which, as you saw in lab 1, is fine for short trajectories, but is susceptible to ever-increasing errors. For a long trajectory, like the one in this lab, you need to implement a form of closed-loop control, in which you are always comparing desired positions with expected positions, and updating control commands accordingly. In this lab, you will program a local trajectory rollout function, from the Lecture *Introduction to Planning and Graphs - W23* sides 27 through 29, that drives the robot to each node.

1. Reuse the trajectory rollout code from Task 2 to generate the N trajectories that you want to explore.
2. Reuse the collision detection code from Task 1 to perform collision detection on your trajectory.
3. Score each generated trajectory, and choose the control command corresponding to the trajectory with the highest score (or lowest cost). There are multiple ways to generate this score, so feel free to experiment. A good place to start is using distance to the current waypoint, as well as distance to obstacles.

Here are a few more helpful hints for trajectory rollout:

- The most challenging part of this section is ensuring that your trajectory rollout loop doesn't take too long to execute. If you send commands at 5Hz (the default), your loop should take less than 200ms to evaluate all of the control commands you are considering. We've included a commented out print statement to measure your loop execution time, which you may find useful.
- We included a set of ALL-CAPS hyperparameters in the code that worked for our implementation, but feel free to experiment with others.
- To encourage you to develop this section of the lab in parallel with the RRT section, feel free to use the `TEMP_HARDCODE_PATH` at the top of the `l2_follow_path.py` file, and uncomment the `self.path_tuples = np.array(TEMP_HARDCODE_PATH)` line.

Finally, see the included video `trajectory_rollout_demo.mp4` for an example of what your trajectory rollout algorithm could look like once you've implemented it correctly.

2.3 Part A Deliverable Summary

The following is a rubric with the number of marks you can receive for completing each section:

1. (/3) A picture with a successful output from RRT, given the goal in Fig. 1. The code already provided in `l2_planning.py` is enough for visualization, and you must take a screenshot of your output.
2. (/3) A picture with a successful output from RRT*, given the goal in Fig. 1. Since RRT* is an anytime algorithm, we don't expect you to get the optimal trajectory, but it should be clear that it is reasonably close to optimal, and certainly better than your output from RRT.
3. (/3) A video with a successful run of trajectory rollout on your path from RRT or RRT*. If you were not able to successfully solve RRT or RRT*, you must show trajectory rollout working on our provided path (`path_complete.npy`).
4. (/3) A brief explanation (one paragraph each) of your algorithms.

You can include the first two deliverables and the explanations in a short report. If you were able to complete trajectory rollout, you should also include an mp4 video with your report. If you are able to complete only a partial solution to any of the above, ensure that you also submit that, because we will be attempt to give you part marks for it.

3 Part B: Real Environment Deployment

After successfully demonstrating your planning algorithm in simulation, you will be testing your algorithm on real robots in the Myhal lab space. You will be given a new map prior to testing. **More details will be provided soon.**

4 Additional Resources

1. Introduction to ROS, ROB521 Handout, 2019.
2. Robotis e-Manual, http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.
3. "SSH: Remote control your Raspberry Pi," The MagPi Magazine, https://www.raspberrypi.org/magpi/ssh-remote-control-raspberry-pi/.
4. Official ROS Website, https://www.ros.org/.
5. ROS Wiki, http://wiki.ros.org/ROS/Introduction.
6. Useful tutorials to run through from ROS Wiki, http://wiki.ros.org/ROS/Tutorials.
7. ROS Robot Programming Textbook, by the TurtleBot3 developers, http://www.pishrobot.com/wp-content/uploads/2018/02/ROS-robot-programming-book-by-turtlebo3-developers-EN.pdf.
8. Skimage circle function documentation, https://scikit-image.org/docs/0.14.x/api/skimage.draw.html#skimage.draw.circle
9. Numpy 1.11 documentation, http://194.149.136.232/numpy~1.11/