ECE 350/450 Intro to Robotics, Lab 7

**RRT Planning**

Nicholas LaRosa and Nolan Sornson

Department of Electrical & Computer Engineering

Lehigh University

{nil318, nls221}@lehigh.edu

Date: 5/9/2021

**Abstract**

In this lab, we introduced the usage of the rapidly-expanding random tree (RRT) algorithm for use in local planning in combination with gap follow for the selection of goal points within the region. We use the readings for the LIDAR to create a local occupancy grid in polar coordinates within the region of the car, and we then select random points within the free space area of the occupancy grid to create a tree path to the goal point. Ultimately, in the process of implementing this algorithm we ran into the computing power limitations of Python, which limited the overall performance of the implementation, but we were able to, with limited success, demonstrate the operation of the algorithm on the simulated car and track.

**Introduction**

* Occupancy Grid - A grid of 2D points in a coordinate space where the value corresponding to each point in the space represents the probability that the point is occupied by some obstacle. These values can be any point from 0 to 1 (although in our case they are binary), can be calcuated on-the-fly or in advance, or can be spatially represented by Cartesian or polar coordinates.
* RRT - Rapidly-exploring Random Tree, an algorithm that decides a path by choosing random points within the free space of an occupancy grid expanding outward from the current position and setting a path that is a continuous tree connecting the current position to the goal via the randomly selected points.
* RRT\* - Similar to the RRT algorithm except that, rather than selecting any tree that connects the position to the goal, we include a cost determination for each tree branch corresponding to the degree that it works toward reaching the goal and maintains a straight path, and then select the tree that minimizes the total cost.

**Procedures**

1. Part 1
   1. For the first part, we started with the example code provided in [1] for the RRT algorithm and implemented a variation of it for ROS compatibility.
      1. We had originally considered using an occupancy grid generated from the overall map ahead of time, with data augmentation of the grid coming from the LIDAR scan. However, we ultimately decided against doing this after some initial attempts since it would require bulk transformation from the car frame to the global frame, and back, for both the occupancy grid and the LIDAR scan points. We found from initial tests that this method would likely be too demanding for the CPU. Instead, we opted to create a partial occupancy grid in the car frame from the LIDAR scan alone, which eliminates the need for bulk transformations.
      2. To select our goal point, we could not use pure pursuit alone, since we no longer had the global occupancy grid to use as a frame to select waypoints. So, we instead opted to choose our goal points using a gap follow algorithm, then we choose the path to the goal point using the RRT algorithm. This is also processor-intensive since a new tree has to be calculated every time the goal point is changed, which may have ultimately contributed to the limited success of our implementation.
   2. We then tested our algorithm using the F1tenth simulator, but we ran into several issues, mainly related to insufficient processing power.
      1. The gap follow algorithm, when being run in conjunction with RRT, did not update the goal point quickly enough for the speed of the car of the data coming from the LIDAR. Once a new gap had opened as we progressed down the track, this would not be updated on the visualization or internally for up to 1-2 seconds, which, especially in the track areas where there were many curves, often resulted in a collision. In an attempt to rectify this issue, we made several changes:
         1. We converted, to the extent possible and practical, all of the loop-based routines in the algorithm to use more efficient Numpy methods.
         2. We tried to eliminate the disparity calculation in the gap follow algorithm, but since we needed this to correctly complete certain portions of the track we ultimately added it back.
         3. We changed our check\_collision routine on the RRT tree calculation to no longer be every scan point, since we can safely interpolate the other points in the tree branch.
         4. We only run the RRT and gap follow algorithms every n scan callbacks, where n is approximately 50. This allows us to significantly reduce the processing power necessary.
      2. Making these changes allowed us to reduce the lag associated with the goal point selection to less than 0.5 seconds, but at that point other bugs in the code started to arise, namely the fact that, intermittently after a certain period of time or a collision takes place, the first tree waypoint would begin to advance ahead of the car erroneously. We were ultimately not able to determine the source of this bug, or even whether the bug was in our code or the F1/tenth code, and this prevented us from making a full lap in the simulator.
   3. In the future, to make this algorithm perform better, it would likely be best to re-implement in C++ rather than Python to reduce the significant overhead associated with the Python interpreter. We could also potentially improve the way that the goal point is selected, possibly using a global planning algorithm if the performance could be improved.
2. Part 2
   1. Installing docker was a straightforward process when following the instructions. The only issue we encountered was that we did not realize that we could not copy and paste the second command in the “Step 1. Install Docker Engine”. When that line is copy and pasted, only “deb [arch=amd64” is sent /etc/apt/sources.list.d/docker.list. That had to be removed before continuing as it is not a valid source and prevents apt from running.
   2. We found using tmux is great for starting the multi-agent sim and running/editing the files. Being able to have one window each dedicated to one command makes it very easy to flip back and forth and kill or restart each process as desired.
      1. On the first page, we have the three commands needed to start docker:
         1. Sudo ./src/f1tenth\_gym\_ros/docker.sh
         2. roslaunch f1tenth\_gym\_ros gym\_bridge.launch
         3. roslaunch f1tenth\_gym\_ros gym\_bridge\_host.launch
      2. On the second page, we have four terminals, two for each opp/ego program
         1. rosrun <package> <opp>.py and rosed <package> <opp>.py
         2. rosrun <package> <ego>.py and rosed <package> <ego>.py
      3. Restarting the simulator only requires flipping back to the first page, ending the process in each terminal and restarting them in order. Using this progress it is easy to make changes to the opp.py and ego.py files and quickly re-simulate.
      4. Entering Ctrl-C for the sudo ./src/f1tenth\_gym\_ros/docker.sh command does appear to kill the docker container as well, so we had no problems with containers running the background.

**Analysis and Results**

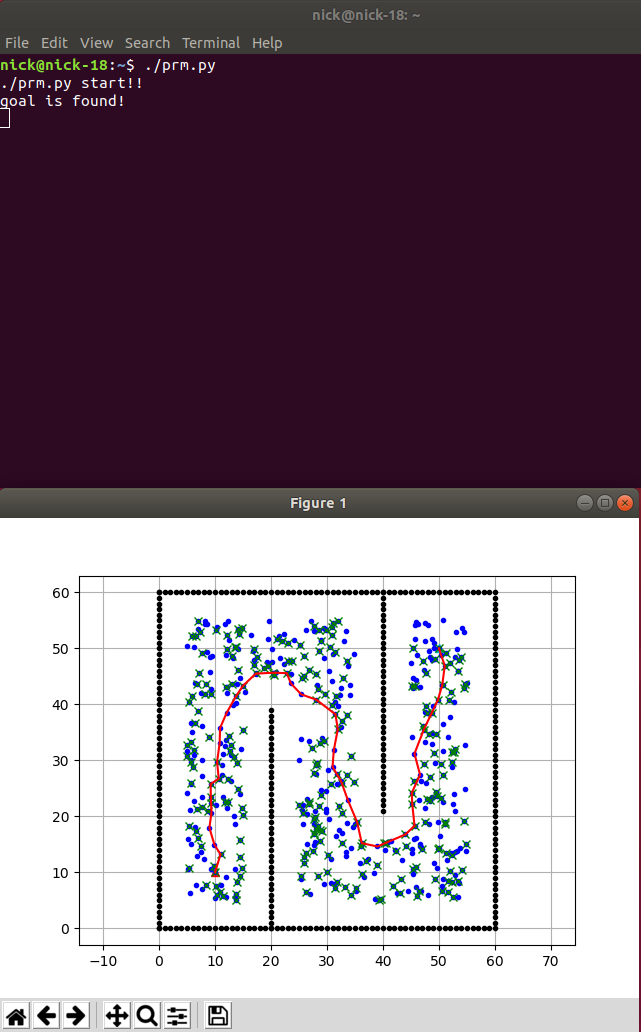
1. Part 1
   1. Q1.1. What is the basic idea of motion planning? What are the commonly used planning algorithms? For ECE450 students, pick the Probability Road Map (PRM) algorithm at the Python Robotics git repo https://pythonrobotics.readthedocs.io/ and try it out. Report your findings and explain how it works
      1. The basic idea of motion planning is to find algorithms that can find a path from a starting point to some goal point without colliding with any obstacles and often attempting to minimize a cost function associated with the planning. Some commonly used planning algorithms include the A\* algorithm, the rapidly-expanding random tree (RRT) algorithm studied in this lab, and the probability road map (PRM) algorithm explored below. The PRM algorithm works by setting the start point and goal points and then randomly selecting points in free space to assemble a graph. Once the graph is complete, a graph searching algorithm, namely the Dijkstra’s search algorithm, is used to find a good path connecting the start and end points. An example output from the PRM algorithm is shown below [1].  
           
         Figure 1: Probability road map (PRM) output
   2. Q1.2. What is the configuration space for the RRT/RRT\* algorithm in this lab? What is the workspace of the race track? How are they related to the map frame and laser scan frame?
      1. The configuration space for this lab is the car’s steering angle as well as its velocity set point (i.e. motor speed). The workspace of the racetrack is the car’s pose, which consists of its position in the global frame as well as its heading. The map frame corresponds to the workspace of the algorithm, while the laser scan frame is related to the configuration space.
   3. Q1.3. How do you constrain the vehicle speed and steering angle in the RRT/RRT\* implementation? How did you apply the RRT algorithm to the race car (grid map, occupancy detection, parameter selection, methods to speed up computation) ?
      1. In our RRT implementation, the speed is set to a constant value, and the steering angle is set to a pure pursuit curvature value (similar to race 2) corresponding to the next waypoint in the tree:  
           
         where is the straight-line distance to the waypoint and is the distance component in the forward-facing direction. We set the local grid map on a polar coordinate plane corresponding to the bounds of the LIDAR readings, with the occupancy grid set to 0 or 1 corresponding to the LIDAR point distance. Computation was sped up, with limited success, using Numpy routines and multiprocessing whenever possible. More information can be found in our procedures.
   4. Q1.4. How does the RRT/RRT\* algorithm perform in comparison to the gap follow algorithm? If you implement both the RRT and RRT\* algorithms, compare their performance too.
      1. In our implementation, due to the reduced computational complexity of gap follow when compared to our RRT algorithm, the gap follow significantly outperformed the RRT algorithm on our hardware. We ultimately concluded that, with the correct optimization and implementation, RRT/RRT\* has the potential to outperform gap follow, but this was not possible with the path we took for the implementation.
   5. Q1.5. What difficulties and problems did you encounter in Lab 7 part 1? What did you try to solve them? What did you learn from this experience?
      1. We encountered several significant challenges with our implementation of RRT, mainly related to our lack of software optimization and computing power. We tried to improve the software performance with various methods detailed in our procedures, but ultimately software bugs and continued poor performance prevented us from making a good implementation. It’s possible that this algorithm could be implemented at peak performance on a GPU or similar.
2. Part 2
   1. Q2.1. What is the Docker engine? What is a docker image and a docker container? When you install Docker and run $(sudo) docker run hello-world, what is the output you observe?
      1. Docker is an open-source, operating system level virtualization software that allows programs to be run in “Containers” that are isolated from each other. The Docker Engine is installed first and is the hypervisor that allows for multiple containers to be managed on a single machine. A Docker Image is a read only template that allows Docker to build a container. In our use case, the f1tenth\_gym Docker image allows for a container to be built that runs ROS as well as a modified version of f1tenth\_simulator.
      2. When we ran sudo docker run hello-world we observed this output:



Figure 2: Docker hello-world output

* 1. Q2.2. How is the docker engine used in the multi-agent simulator? How many docker containers were running after you run `$./src/f1tenth\_gym\_ros/docker.sh’ in Step 5?
     1. The docker engine is used to run ROS and the multi-agent simulator in a container. Running these programs in a container may normalize the performance across multiple different systems. The host computer is linked to the container via the bridge node and allows drive messages and sensor topics to be passed in and out of the multi-agent simulator.
     2. After running ./src/f1tenth\_gym\_ros/docker.sh we observed that only one container was created.

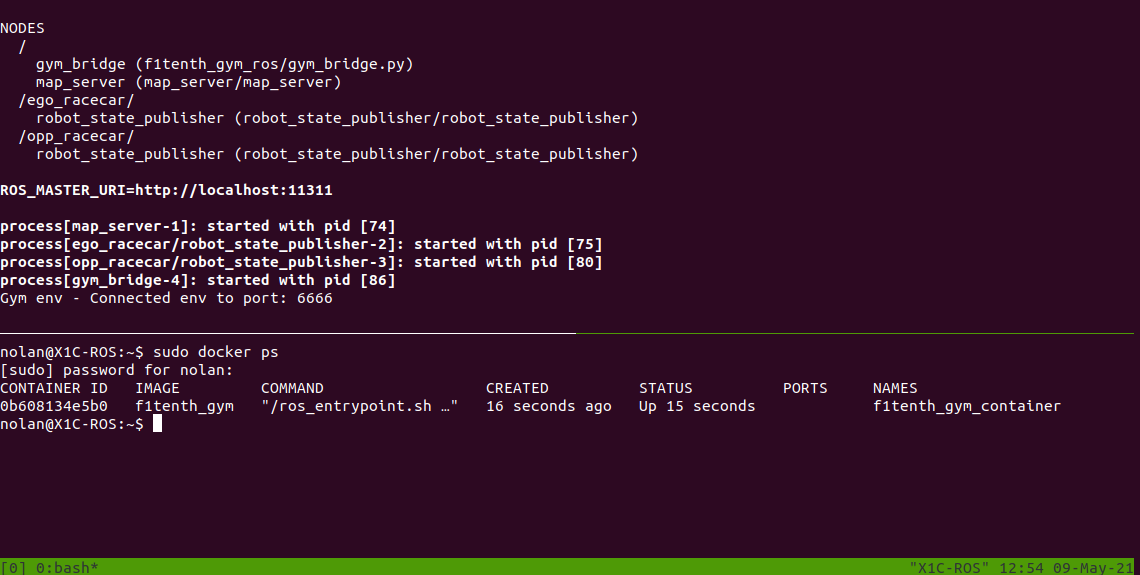
****

Figure 3: Docker container creation

* 1. Q2.3. What did you modify in your RRT/RRT\* algorithm when you run it as the ego car (blue car) in the head-to-head race? What algorithm did you use to run the opponent car (brown car)? How does the performance of your RRT algorithm differ in the two simulators? How do you detect a collision?
     1. We struggled to get RRT working well in the first place and as a result we did not perform any adjustments to the algorithms when it came time to run them in the multi-agent simulator.
  2. Q2.4. When you switch the algorithms between the ego and opp cars, what are the differences in behaviour and performance? How do you adjust the parameters of the RRT/RRT\* algorithm for the opp car? How long did it take to finish 2 laps?
     1. Our programming is seemingly taking too long to run during each callback to run effectively in the multi-agent simulator. While RRT can complete most of a lap in Berlin when it is the only car on track, the performance is vastly different in the multi-agent sim. We were unable to complete two laps and will need significant modifications if we are going to run RRT in race 3.
  3. Q2.5. What difficulties and problems did you encounter in Lab 7 part 2? What did you do to try to solve them? What did you learn from this experience?
     1. The multi-agent simulator seems to not operate at the same speed as running f1tenth\_simulator on the host computer. This is a significant performance penalty and must be taken into account as we prepare for race 3. Simpler programs that we have such as gap follow appear to run well in the multi-agent sim, so more investigation will be required to determine why our RRT performed so poorly.

**Conclusion**

In this lab, we attempted to use RRT as a local planner for our car in the simulator. Our trials in f1tenth\_simulator on the host computer demonstrate that our implementation of RRT does work; however, it requires more processing power to run well. Unfortunately, since the multi-agent simulator is running inside of a Docker container, performance is further limited and our program performs poorly. Our submission contains the best we could accomplish prior to the deadline, and we may work to improve these implementations in preparation for Race 3.

**References**

1. <https://github.com/AtsushiSakai/PythonRobotics>