

AA 274: Principles of Robotic Autonomy

Problem Set 4: Planning and Decision Making

Due Wednesday, March 13

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/StanfordASL/AA274_HW4.git` in a terminal window.

For your final submission, **you will submit the python files (zip of Problems 1 and 2) that contains your work for the code items and a pdf that contains work for written questions** to Gradescope. We will be primarily grading the plots you submit for the first two problems. In the case that your results are incorrect or incomplete, we will refer to your code to assign partial credit.


Problem 1

In this problem we will implement the A^* motion planning algorithm, outlined in pseudocode below:

Algorithm 1 A^*

Require: \mathbf{x}_{init} , \mathbf{x}_{goal}

```
1:  $\mathcal{O}.\text{INIT}(\mathbf{x}_{init})$ 
2:  $\mathcal{C}.\text{INIT}(\emptyset)$ 
3:  $\text{SET\_G\_SCORE}(\mathbf{x}_{init}, 0)$ 
4:  $\text{SET\_F\_SCORE}(\mathbf{x}_{init}, \text{HEURISTIC}(\mathbf{x}_{init}, \mathbf{x}_{goal}))$ 
5: while  $\mathcal{O}.\text{SIZE} > 0$  do
6:    $\mathbf{x}_{current} \leftarrow \text{LOWEST\_F\_SCORE}(\mathcal{O})$ 
7:   if  $\mathbf{x}_{current} = \mathbf{x}_{goal}$  then
8:     return  $\text{RECONSTRUCT\_PATH}$ 
9:   end if
10:   $\mathcal{O}.\text{REMOVE}(\mathbf{x}_{current})$ 
11:   $\mathcal{C}.\text{ADD}(\mathbf{x}_{current})$ 
12:  for  $\mathbf{x}_{neigh}$  in  $\text{NEIGHBORS}(\mathbf{x}_{current})$  do
13:    if  $\mathbf{x}_{neigh}$  in  $\mathcal{C}$  then
14:      continue
15:    end if
16:     $\text{tentative\_g\_score} = \text{GET\_G\_SCORE}(\mathbf{x}_{current}) + \text{DISTANCE}(\mathbf{x}_{current}, \mathbf{x}_{neigh})$ 
17:    if  $\mathbf{x}_{neigh}$  not in  $\mathcal{O}$  then
18:       $\mathcal{O}.\text{ADD}(\mathbf{x}_{neigh})$ 
19:    else if  $\text{tentative\_g\_score} > \text{GET\_G\_SCORE}(\mathbf{x}_{neigh})$  then
20:      continue
21:    end if
22:     $\text{SET\_CAME\_FROM}(\mathbf{x}_{neigh}, \mathbf{x}_{current})$ 
23:     $\text{SET\_G\_SCORE}(\mathbf{x}_{neigh}, \text{tentative\_g\_score})$ 
24:     $\text{SET\_F\_SCORE}(\mathbf{x}_{neigh}, \text{tentative\_g\_score} + \text{HEURISTIC}(\mathbf{x}_{neigh}, \mathbf{x}_{goal}))$ 
25:  end for
26: end while
27: return Failure
```

- (i)  Implement A^* for 2D geometric planning problems (state $\mathbf{x} = (x, y)$) by filling in `AStar.solve`

and `Astar.get_neighbors` in `astar.py` (make sure you read the documentation for every function!).

- (ii)  Include a plot of your A* implementation successfully planning to the goal state.

Problem 2

In this problem we will implement the Rapidly-exploring Random Tree (RRT) [1] sampling-based motion planning algorithm, outlined in pseudocode below:



Algorithm 2 RRT [1] with goal biasing.

Require: \mathbf{x}_{init} , \mathbf{x}_{goal} , maximum steering distance $\varepsilon > 0$, iteration limit K , goal bias probability $p \in [0, 1]$

```

1:  $\mathcal{T}.\text{INIT}(\mathbf{x}_{init})$ 
2: for  $k = 1$  to  $K$  do
3:   Sample  $z \sim \text{Uniform}([0, 1])$ 
4:   if  $z < p$  then
5:      $\mathbf{x}_{rand} \leftarrow \mathbf{x}_{goal}$ 
6:   else
7:      $\mathbf{x}_{rand} \leftarrow \text{RANDOM\_STATE}()$ 
8:   end if
9:    $\mathbf{x}_{near} \leftarrow \text{NEAREST\_NEIGHBOR}(\mathbf{x}_{rand}, \mathcal{T})$ 
10:   $\mathbf{x}_{new} \leftarrow \text{STEER\_TOWARDS}(\mathbf{x}_{near}, \mathbf{x}_{rand}, \varepsilon)$ 
11:  if  $\text{COLLISION\_FREE}(\mathbf{x}_{near}, \mathbf{x}_{new})$  then
12:     $\mathcal{T}.\text{ADD\_VERTEX}(\mathbf{x}_{new})$ 
13:     $\mathcal{T}.\text{ADD\_EDGE}(\mathbf{x}_{near}, \mathbf{x}_{new})$ 
14:    if  $\mathbf{x}_{new} = \mathbf{x}_{goal}$  then
15:      return  $\mathcal{T}.\text{PATH}(\mathbf{x}_{init}, \mathbf{x}_{goal})$ 
16:    end if
17:  end if
18: end for
19: return Failure

```

- (i)  Implement RRT for 2D geometric planning problems (state $\mathbf{x} = (x, y)$) by filling in `RRT.solve`, `GeometricRRT.find_nearest`, and `GeometricRRT.steer_towards` in `rrt.py`.
- (ii)  Implement RRT for Dubins car planning problems (state $\mathbf{x} = (x, y, \theta)$, see [2] for steering connection details) by filling in `DubinsRRT.find_nearest` and `DubinsRRT.steer_towards` in `rrt.py`. Computing steering solutions in this code relies on installing the following python package

```
$ sudo pip install dubins==0.9.2
```

(see <https://github.com/AndrewWalker/pydubins/blob/master/dubins/dubins.pyx> for usage details).

- (iii)  Include plots of your RRT and Dubins RRT implementations successfully planning to the goal state.

Problem 3

In this problem, we will test your A^* implementation from Problem 1 in Gazebo/Rviz.




We have updated `asl_turtlebot` once more for this problem. On the `/master` branch of `asl_turtlebot`, `git pull` the new files. Some of the files that were added for this demo are:

- `scripts/navigator.py*`
- `scripts/grids.py*`
- `scripts/pose_controller_nav.py*`¹
- `scripts/supervisor_nav.py*`²
- `world/project_city.world*`
- `launch/turtlebot3_nav_sim.launch`
- `rviz/nav.rviz`

Files denoted by `*` are identical to the corresponding files in the project branch.

This should not overwrite any of your code as these are new files (except for `launch/turtlebot3_nav_sim.launch` but you should not have changed that file in previous homeworks anyways). You can run `git status` to make sure.

Inside `navigator.py`, you are provided a path planner that will: 1) use your A^* implementation (from Problem 1) to generate a trajectory through an occupancy grid, 2) smooth it using cubic spline interpolation and 3) apply the differentially flat control scheme from (from HW 1) to send controls. This script is accompanied by `grids.py` which is used to perform collision checking on the occupancy grid. The simulation world, `project_city.world`, is a Gazebo model file of an environment very similar to that of the final project. The launch file, `turtlebot3_nav_sim.launch`, will launch a Gazebo simulation using this world file and start the appropriate nodes to perform motion planning. Please take some time to look inside this file and understand the details. Finally, `nav.rviz` will be used to visualize the simulation in Rviz.

- (i)  Take a look at `turtlebot3_nav_sim.launch`. Which scripts are we using for motion planning? You do not need to provide an exhaustive list of ALL nodes and Gazebo files being launched by this file. Rather, just look at which python scripts are being used. Keep these in mind as they should be called by your final project launch file (or added to the `turtlebot3_bringup_jetson_pi.launch` file if you are using it as a starting template).
- (ii)  To which topics does `navigator.py` subscribe? What do each of the corresponding callback functions do?
- (iii)  To which topics does `navigator.py` publish? What is the purpose of each of the associated published messages? **Note:** It would help to *thoroughly* read through the `run_navigator()` function. The comments in the code should help.

To get started with the simulation, take your `astar.py` file and place it in the scripts directory `asl_turtlebot`. Next install `scipy` if you haven't already (`sudo pip install scipy`).

Then, launch `turtlebot3_nav_sim.launch`. RViz should open up and look something like Figure 1.

¹This is essentially the same `pose_controller.py` that you have been working with before but with some minor modifications to make it compatible with `navigator.py`. This `scripts/pose_controller_nav.py` is the same as the provided `pose_controller.py` in the project branch.

²This is essentially the same `supervisor.py` that you have been working with before but with some minor modifications to make it compatible with `navigator.py`. This `scripts/pose_controller_nav.py` is the same as the provided `pose_controller.py` in the project branch.

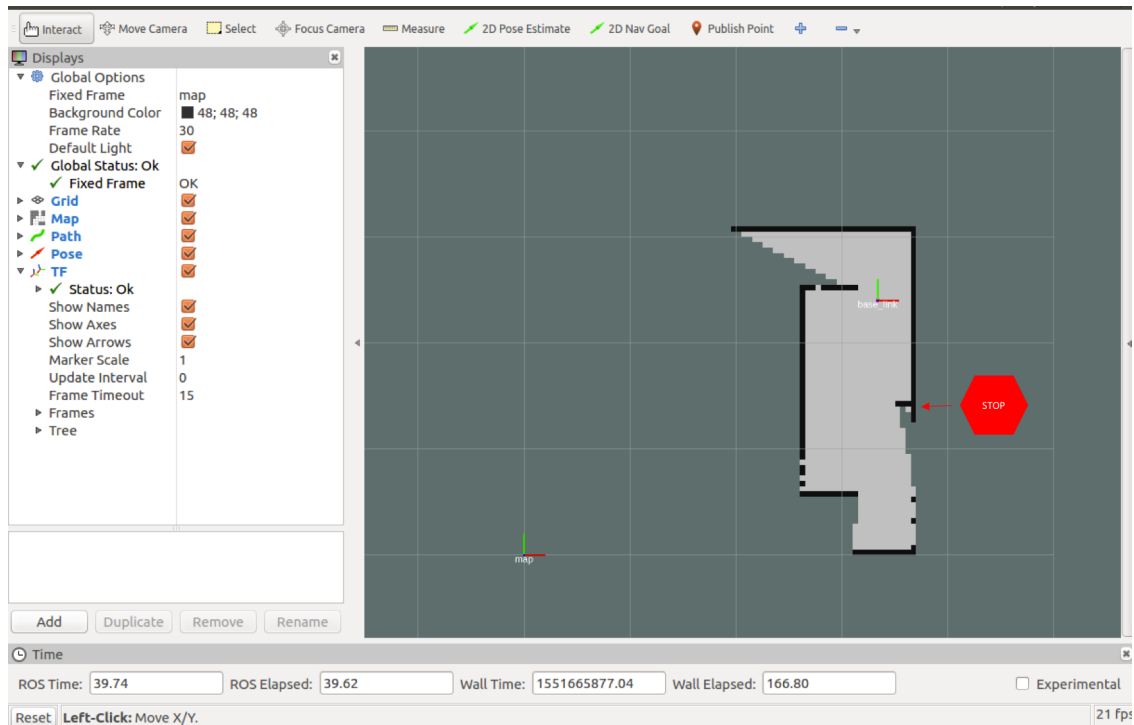


Figure 1: Screenshot of nav.rviz when running `turtlebot3_nav_sim.launch`

You should now be able to send a desired nav pose to your turtlebot in Rviz by clicking **2D Nav Goal** at the top and then clicking on some location on the map. Make the robot move to several locations in the environment so that its map is completed. Get a feel for how the robot follow the paths, and in what cases it cannot compute a feasible path. What parameters may affect how easily the robot can find a feasible path. Make sure you are able to successfully do this part as this is a core component to your final project.

- (iv)  Take a screenshot of your RViz as the Turtlebot is following some interesting path (plotted in green).

For the purposes of your project, you can extend this motion planning in a number of ways. Possible extensions include:

- Frontier exploration
- Safe motion planning to avoid cutting corners too tightly
- Contingency plans in case of collisions
- Stopping at stop signs in the correct direction
- Lane following and motion planning that obeys the rules of the road
- Identifying and avoiding puddles on the ground

Note that manipulating the occupancy grid might be important for a lot of these tasks.

At this point, you have all of the foundational pieces for your final project. With very little effort, you should be able to create (or adapt) a launch file to start up all of the necessary components for your autonomy stack

(lidar/camera perception, gmapping for SLAM, motion planning). This will give you a starting point that should work for the most part. However, you are free and expected to modify your stack for the project as you please!

References

- [1] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” Illinois State University, Tech. Rep., 1998.
- [2] L. E. Dubins, “On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents,” *American Journal of Mathematics*, vol. 79, pp. 497–516, 1957.