AA 274A: Principles of Robotic Autonomy I
Section 5: Implementing Point-to-Point Navigation

Our goals for this section:

1. Learn how to read and understand source code for more complex ROS nodes
2. Test controllers from homework on a real robot
3. Learn how to design custom launch files

1 Point to point motion around obstacles

As you saw in the last section, one of the nodes that the section launch file started was `gmapping`, which uses LIDAR readings to perform simultaneous localization and mapping (SLAM), giving us an *occupancy grid* map of the environment around the robot, as well as an estimate of the robots position within this map.[1]
A key ability of autonomous agents is the ability to navigate from point to point in the presence of obstacles. Today we'll be implementing a navigator ROS node and testing this functionality on the turtlebots!
Open `scripts/navigator.py` within the `asl_turtlebot` catkin package in a text editor, read the provided code, and think about how the node works.
Problem 1: What topics does the navigator subscribe to? What is the purpose of each of these topics? What topics does it publish to, and why?
The navigator uses a state machine to switch between different modes of operation. Carefully read the functions `run` and `publish_control`.
Problem 2: Describe what each mode of the state machine does, and intuitively when the node switches between modes.
You may have noticed that the code logic is similar to the strategy we used in HW2: planning around obstacles using A*, and using a combination of the pose controller and tracking controller to track the planned path. In fact, this code calls functions that you wrote in your previous homeworks.
Copy over the following files to `asl_turtlebot/scripts/controllers/`

```
P2_pose_stabilization.py
P3_trajectory_tracking.py
```

and the following from HW2 to `asl_turtlebot/scripts/planners/`

```
P1_astar.py
```

You will also need to make some small modification to `P1_astar.py`. Find and replace the initialization of two variables as following,

```
self.est_cost_through = {}
self.cost_to_arrive = {}
```

Also, edit `scripts/planners/path_smoother.py` and copy over the function `compute_smoothed_traj` from HW2's `P3_traj_planning.py`.
Now, we're ready to test the framework on the robot!
On genbu, run

```
roscore -p $ROS_PORT
```

Then, in a new terminal, run

```
roslaunch asl_turtlebot section5.launch
```

[1] We'll be covering SLAM in class next week and learning how gmapping is able to do this, but for now it's fine to think of it as magic.

2 Running the Navigator

From a new terminal, open `rviz`.
Add relevant topics to the display - the main ones we'll need are `/map`, the TF transform tree, and the path topics `/planned_path` and `/cmd_smoothed_path`. The `/camera` topic will also allow you you see what the robot sees as it navigates through the maze.
Create a new catkin package (like how we made one in Section 2) with the name `section5` and save the rviz configuration as `my_nav.rviz` in `section5/rviz/my_nav.rviz` on genbu.
Problem 3: What is the command to create a new package? (Hint: Take a look at Section 2's handout for a starting point). What do each of the arguments do? What modifications do you need to make for the `section5` package?
Now you can specify goal poses using the "2D Nav Goal" button in rviz and clicking and dragging on the map. The robot should move towards the goal if your controllers work correctly!
Problem 4: Test this out. Include a screenshot of rviz as your robot navigates the map.

3 Visualizing the goal position

Using what you learned in last section, write a new node that visualizes the current navigation target in rviz as a marker. Save this node in the `section5` package's `scripts` folder.
Problem 5: Describe at a high level how your goal visualizer works. Some questions to get you started are:

- **What topics should it subscribe to in order to stay up to date with the current navigation target?**
- **What message type should it publish, and to what topic?**

Include this code in your submission.

4 Custom Launch Files

It can be cumbersome to start all the nodes from scratch, and set up rviz every time we want to run the stack. To make this easier, create a launch file in your `section5` package which:

1. starts the `navigator.py` node from the `asl_turtlebot` package.
2. starts the goal visualization node you just wrote.
3. opens `rviz` with the configuration file you just saved.

Hint: run `rviz --help` to see how to pass a configuration file into rviz. Use the ROS documentation and/or Google to find out how to pass arguments into nodes through a launch file.
Take a look at the launch files in the `asl_turtlebot` package. In particular, lines `75 - 78` of the `asl_turtlebot` `project_sim.launch` file also provide an example of starting each of these nodes. In addition, `turtlebot3_nav.launch` provides a minimal example of a launch file.
Once you've written your launch file, save it as

```
~/catkin_ws/src/section5/launch/my_nav.launch
```

Test it out by running

```
roslaunch section5 my_nav.launch
```

Problem 6: Describe the components included in your launch file. Did you use any of the `asl_turtlebot` launch files as an example? If so, what changes did you make? Include the contents of this launch file in your submission