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 `turtlebot3_bringup_jetson_pi` launch file starts is `gmapping`, which uses the LIDAR readings to perform SLAM (simultaneous localization and mapping), 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!

First, let's git pull the latest changes to the `asl_turtlebot` package which contains the starter code for the navigator node in `scripts/navigator.py`. As before, from the `asl_turtlebot` directory, run

```
git pull
```

If you're using docker, you will need to also update your docker config. From the `aa274-docker` directory, run

```
git pull
./build_docker.sh
```

Next open `scripts/navigator.py` in your favorite text editor and 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.

[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.

Copy over the following files to `scripts/controllers/`

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

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

```
P1_astar.py
```

Also, edit `scripts/planners/path_smoother.py` and copy over the function `compute_smoothed_traj` from HW2's `P3_traj_planning.py`.

For SCPD students: use the `scp` command to copy these files to genbu.

```
scp <file> groupXX@genbu.stanford.edu:~/catkin_ws/src/asl_turtlebot/scripts/...
```

Now, we're ready to test the framework on the real robot!

1.1 On Campus Students

Power on the robot, and have one person in your group ssh into it. Ensure it's running the latest version of our codebase by running the following commands

```
roscd asl_turtlebot
git pull
```

Finally, run `turtlebot3_bringup_jetson_pi.launch` as we did in last section.

From your computer, (after running `rostb3`), **one person per robot** should run the following to start the navigator

```
rosrun asl_turtlebot navigator.py
```

Again, only one navigator should be running per robot, so take turns to test your code!

1.2 SCPD Students

Log into genbu and run

```
roscore -p $ROS_PORT
```

Then, in a new terminal, run

```
roslaunch asl_turtlebot turtlebot3_nav_sim.launch
```

2 Running the Navigator

From a new terminal, open rviz. As before, if using docker, run the following command from the `aa274-docker` directory.

```
./run.sh --display 1 --rosmaster <lowercase_robot_name>.local rviz
```

Add relevant topics to the display - the main ones we'll need are `/map`, the TF transform tree, and the path topic `/cmd_path`. Save the rviz configuration as `my_nav.rviz` into the package you made in section 2

```
~/catkin_ws/src/aa274_s2/rviz/my_nav.rviz
```

Before starting, have someone else working with the robot ready to run the teleop node to take over control and stop the robot if necessary

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

(Don't actually run this until you need to! This overrides any commands to the turtlebot so your navigator won't work if teleop is running.)

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 3: Test this out on the robot (or simulation). 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 `aa274_s2` package's scripts folder.

Problem 4: 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 `aa274_s2` 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.

Once you've written your launch file, save it as

```
~/catkin_ws/src/aa274_s2/launch/my_nav.launch
```

Test it out by running

```
roslaunch aa274_s2 my_nav.launch
```

Problem 5: Include the contents of this launch file in your submission