ME 134 Homework 2 (Prep for Lab 3) : Simulate Frontier Exploration

Released: Monday 5/14/18 Due: Monday 5/21/18

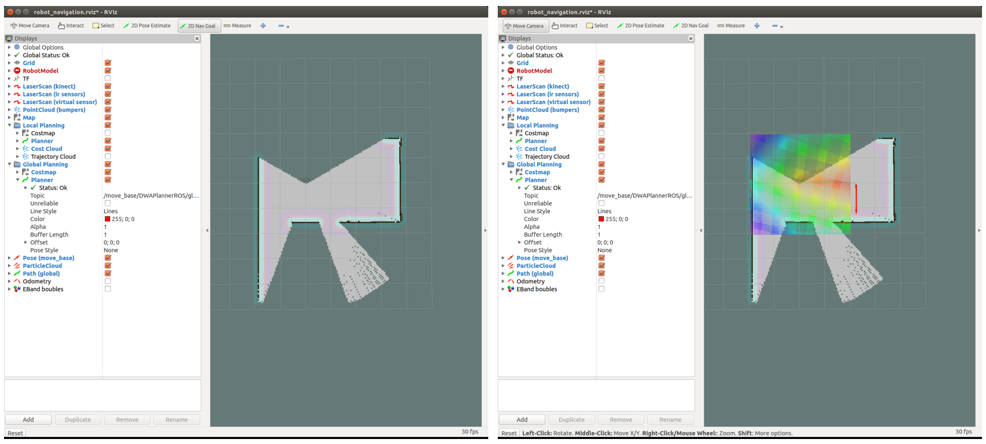
You should have already learned how to run the Turtlebot simulator and gone over how a launch file works. You can also do the TurtleBot in Stage Simulator Tutorial if you still have questions. In this prelab homework assignment, you will run a simulation of last lab's navigation code, then run a frontier exploration algorithm in simulation. You will finish this frontier exploration algorithm, and you may write your own more sophisticated algorithm for extra credit. During lab time, we will test your code in real life on the TurtleBot.

Download and install me134_explorer

1. You will need the catkin and ros_numpy packages. If you don't already have them, install them on your virtual machine by typing: `sudo apt-get install ros-indigo-catkin ros-indigo-ros-numpy`.
2. Follow this tutorial to create a catkin workspace.
3. Download the me134_explorer node for this lab, and unzip it. Place it in the ~/catkin_ws/src/ directory.
4. Build the workspace, following this tutorial. From the ~/catkin_ws directory, type `catkin_make`. Then run `source devel/setup.bash`. You will need to run this source step from every terminal in which you want to run the me134_explorer code, or put it in your .bashrc file with the line `source ~/catkin_ws/devel/setup.bash`.

Run the turtlebot_stage simulation

1. Take a look at the me134_explorer/launch/explore_in_stage.launch file. How is it different from the turtlebot_in_stage.launch file you saw before? In this particular case, in simulation, we've found that gmapping (using laser scan-matching) localizes more accurately than amcl does. Among other changes, we've commented out the amcl line to make your algorithm-writing job easier. (amcl probably requires additional tuning and/or debugging.)
2. Type `roslaunch me134_explorer explore_in_stage.launch` to bring up the simulator. Note that you will not need to run roscore, gmapping, etc.; this launch file does that for you.
3. In rviz, select the "2D Nav Goal" button at the top center of the screen, as shown in Figure 1. Click on the map to set a navigation goal at that point. The robot should plan a path to this position, and follow that path.



Topic /move_base/DWAPlannerROS/gl...
Unreliable
Line Style Lines
Color 255; 0; 0
Alpha 1
Buffer Length 1
Offset 0; 0; 0
Pose Style None

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.

Figure 1: Select "2D Nav Goal" and click on the map to navigate to that point.

4. Run your goal_send.py code from the last lab to simulate sending the robot to some nearby point. Ctrl+c to quit goal_send.py when you're done, and restart your roslaunch command from Step 2.

5. Run the provided me134_explorer.py program by typing `rosrun me134_explorer me134_explorer.py` **Note that you will have to close the 2 plot windows to continue.** This code moves the robot backwards, causing stage to run the costmap-clearing algorithm, which spins the robot in place. (Note that, due to a ros bug, you can't spin the robot in place as the first action; you need to first trigger a costmap clearing to set the robot's footprint.) The program uses the occupancy grid to find the frontiers, and selects the closest frontier point to set as a goal. (Note that this is the closest as-the-bird-flies point, not necessarily the point with the shortest travel distance.) As written, however, the code fails because the robot sets a goal point that is too close to an obstacle. You will need to fix this in the next step.

   Once it finds a goal, the program adds that goal to a queue and publishes it using actionlib. Note that, although gmapping constantly updates the map with laser scan data, you will be blocked from using that data to set a new goal until the current goal is reached. Since we are not using amcl (which generates PoseStamped messages), we are updating position using the map-to-base_link transform that gmapping publishes to the /tf topic. Once the robot has reached that goal, the program again uses the lastest map and pose data to set a new goal.

6. Rename me134_explorer.py to me134_explorer_[initials].py, with the initials of your group members (or any other unique identifier that you choose). Edit this code so that the robot successfully explores all the frontiers. Use the simulation to test and debug your code.

Deliverables and extra credit

A frontier exploration algorithm continually provides the next goal location on a frontier, until all the frontiers have been explored. Of course, there are several different ways to choose that next frontier. The me134_explorer.py code implements one of these strategies. For full credit on this assignment (140 points), you must run me134_explorer.py in simulation and get it working correctly. Turn in a few pages that answer the questions above and describe how this program works, as in Approach 1 below. Bring your completed me134_explorer_[initials].py to lab next week on a USB key.

In addition, you may write your own exploration program that implements one of the algorithms described in Approaches 2-4 for extra credit. Feel free to copy as much of me134_exporer.py as you want; you may find it useful that this program subscribes to the map and transform topics for you, to provide a occupancy grid and current pose. If you write your own implementation, be sure to include in your pre-lab writeup a description of how you implemented the algorithm, what issues occurred, and how it worked in simulation. Name your program me134_explorer_[initials]_EC.py and bring it to lab as well. This assignment will be worth 140 points plus any extra credit points, and the lab report will be worth 60 points.

- Approach 1 (full credit): To get full credit on this assignment, you must run the me134_explorer.py exploration algorithm, fix its errors, and turn in a short pre-lab report. How does the existing code work? How did you fix it?
- Approach 2 (+20 points): Analyze the current map to find the frontiers. Choose a goal on the frontier that will require the shortest path from the robot's current position.
- Approach 3 (+40 points): Choose your next goal location to maximize expected information gain (EIG).
- Approach 4 (+60 points): Choose your next goal location using value iteration.