Robotics and Perception *CMSC 498F, CMSC 828K, Spring 2016*

Assignment 4: Localization

DELIVERABLES:

- `assignment_4` and `localization` packages with your code
- a short report on your localization method

The goal of this assignment is to localize a robot in a known map using laser scanner data. We'll be attempting to localize in two environments, a fairly empty "easy" map and the CSE550 map from assignment 3. There are two versions of the "easy" map: low resolution `localization/EasyMap.bag` and high resolution `localization/EasyMap2.bag`. The CSE550 map is stored in `localization/CSE550Map.bag`. Low resolution map is useful for debugging your code. The laser scan and true pose of the robot are stored in `localization/easy-#.bag` and `localization/cse550-#.bag`.

1 Expected scan

Consider a laserscan from the robot and a guess of where the robot was located when the sensor reading was taken. We have to have a procedure that allows us to evaluate how likely is our guess given the laserscan data. More mathematically, given a pose x and sensor measurement z we want to calculate the conditional probability $p(x|z)$. To accomplish this implement the following functions in `assignment_4/src/assignment_4/laser.py`:

- `ray_tracing` finds the coordinates of the first occupied cell in a ray. Takes in a coordinate in map coordinate system (`x0`, `y0`), an angle, and a map. The function should return the map coordinates of the first cell in the map along the ray (starting at `x0`, `y0` at the specified angle) that is occupied i.e. has the value 100. This is meant to figure out what the laser would hit from the pose specified, given the map. If you reach the end of the map, return None. Use the `line_seg` function to find the map cells traversed by the ray.
- `expected_scan` returns a laser scan that the robot would generate from a given pose in a map. Takes in a pose (`x,y,theta`), the properties of a laser scanner (`min_angle,increment,n_readings, max_range`) and the map. Returns an array of floats that represents what ranges you would expect for the entire laser scan. You should use your `ray_tracing` function. You should return `max_range` if `ray_tracing` returns `None`, or if the ray traced is longer than the maximum range.
- `scan_similarity` computes the similarity between two laserscan readings. Takes in two arrays of floats that represent the data from the laser scanner. Assuming both were taken from the same position, come up with a metric that represents the scans' similarity to each other, with higher numbers representing more similar scans. The actual metric is up to you, but it should return a number between 0 and 1.

1.1 Testing Laser Code - Single Query

To check your laser code, you can use the `localization/src/query_pose.py` script to check the resulting scan for one pose at a time. Start `roscore`, and then run

```
rosrun rviz rviz -d localization/hill_climb.rviz
```

Then in a new terminal window run the script

```
rosrun localization query_pose.py MAPBAG DATABAG
```

Then in the `rviz` window, draw an arrow on the map using the "2D Pose Estimate Tool". This will display the expected scan, and the score will print out in the terminal.

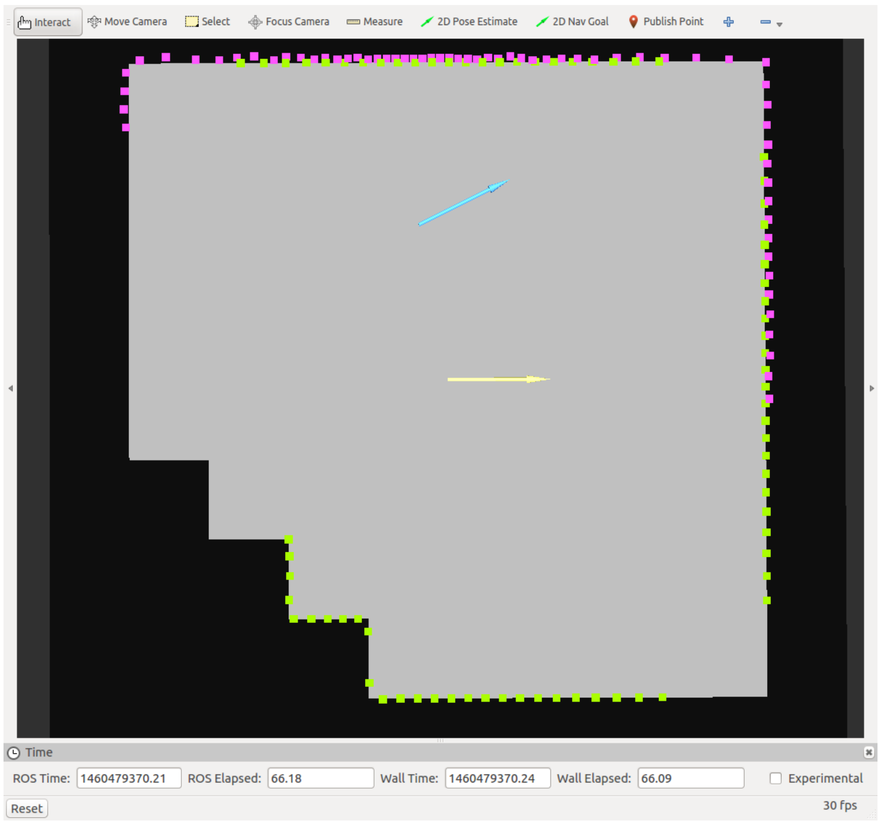

Figure 1: Testing your code with `rviz`. Yellow arrow shows the true pose, blue arrow shows the estimated pose. Green and magenta dots show the true and expected laserscans respectively.

1.2 Testing Laser Code - Brute force

A brute force approach to localizing, is to generate all possible poses in the map, calculate the likelihood that the robot is in each individual pose and choose the most likely one. That is what the script in `localization/src/find_pose.py` does. Your provide two bag files, one with the map, and one that contains a laser scan and the actual pose. The script will score each pose with your `laser.py` methods and print out the best ones. This takes quite a bit of time.

Sample Commands:

```
rosrun localization find_pose.py EasyMap.bag easy-1.bag
rosrun localization find_pose.py CSE550Map.bag cse550-1.bag -resolution 8
```

Note that due to the brute force nature of this localization approach it may take a while to run. The `-resolution 8` command only searches every 8th column and row in the map, which causes the runtime to actually be manageable. Also note that due to the resolution of the maps you are unlikely to get the exact original pose. For example, for the above commands, I get `(0.5, 0.5, 0.0)` and `(-6.460000079125166,-3.900000136345625,0.0)`.

2 Particle Filtering

Testing every possible pose takes too long. So we're going to use a particle filtering approach to find the actual pose. The pseudocode of our approach is:

```
Generate particles distributed uniformly in the map
For number of iterations:
  Evaluate the likelihood of each particle
  Resample the particles
  Add noise to the particles
```

You need to implement the following functions in `assignment_4/src/assignment_4/particle.py`:

- `random_particle` generates a random pose in the map. Returns a tuple `(x,y,theta)` for a random pose (in real world coordinates) in the map. You may find you get better results if you only return points that are in unoccupied cells.
- `new_particle` generates a new particle from an old one by adding noise to it. Given a tuple `(x,y,theta)`, returns a similar particle with slightly altered coordinates. You can add additional parameters if you desire.
- `resample` resamples the particles. Given an array of tuples `(score, (x,y,theta))`, do low variance resampling to create `n_particles` new particles (using your `new_particle` method). You should return an array of tuples `(x,y,theta)` that represent your new particles.

2.1 Testing localization

There are two ways to test this code, both using `localization/src/hill_climb.py`. To test it with no visualization, just run

```
rosrun assignment_4 hill_climb.py MAPBAG DATABAG
```

This defaults to 100 particles and 5 iterations. You can increase that number with the arguments `-particles 1000 -iterations 10`. To run it with visualization, start `rviz` as described in the Single Query Testing, and then run the same command with the `-ros` flag.

Test your code to see how well it can localize in the "easy" and CSE550 maps. Try out different values for the number of particles and iterations used, the amount of noise you add in the `new_particle` function and the error metric used in the `scan_similarity` function. To get even better results you can modify these function even further. For example you could try alternative methods for particle resampling. Investigate which parameters of your algorithm work best for different poses (3 poses for `EasyMap2.bag` and 7 poses for `CSE550Map.bag`).

Write a short report on your localization method. In the first part briefly explain your localization approach, give a list of parameters and a short description of how each parameter affects localization performance. In the second part describe your localization results for all of the test poses. For each pose provide:

- Set of parameter values that performed best for a pose.
- A screenshot from `rviz` showing your average localization result for a pose.
- If your method fails to consistently localize for a pose, an explanation of why this is happening.

Note that you are not expected to get perfect localization results for all of the test poses.

3 Grading

Grading for undergraduates:

`expected_scan` and `ray_tracing`	20 points
`scan_similarity`	10 points
`random_particle`	10 points
`new_particle`	10 points
`resample`	20 points
Report	30 points

Grading for graduates:

`expected_scan` and `ray_tracing`	25 points
`scan_similarity`	5 points
`random_particle`	5 points
`new_particle`	5 points
`resample`	20 points
Report	40 points