LAB 3: PARTICLE FILTER

Due: Thursday, October 5th 11:59pm EST

The objective of Labs 3 is to implement a Particle Filter (a.k.a. Monte Carlo Localization). In this lab, you will work entirely in simulation to validate your algorithm.

Task: Your job is to implement a particle filter which takes robot *odometry measurement* as motion control input and *marker measurement* as sensor input. We will provide skeleton code in which you must complete the implementation of two functions: `motion_update()` and `measurement_update()`. Since in a particle filter the belief of robot pose is maintained by a set of particles, both `motion_update()` and `measurement_update()` have the same input and output - a set of particles, as the belief representations.

Before giving more details about the functions you need to implement, we first present some definition of the world. In this lab, we will use the grid world map shown in Fig 1., with the addition of localization markers on the wall of the arena. The grid world has the origin (0,0) at the bottom left, X axis points right and Y axis points up. The robot has a local frame in which the X axis points to the front of the robot, and Y axis points to the left of the robot.

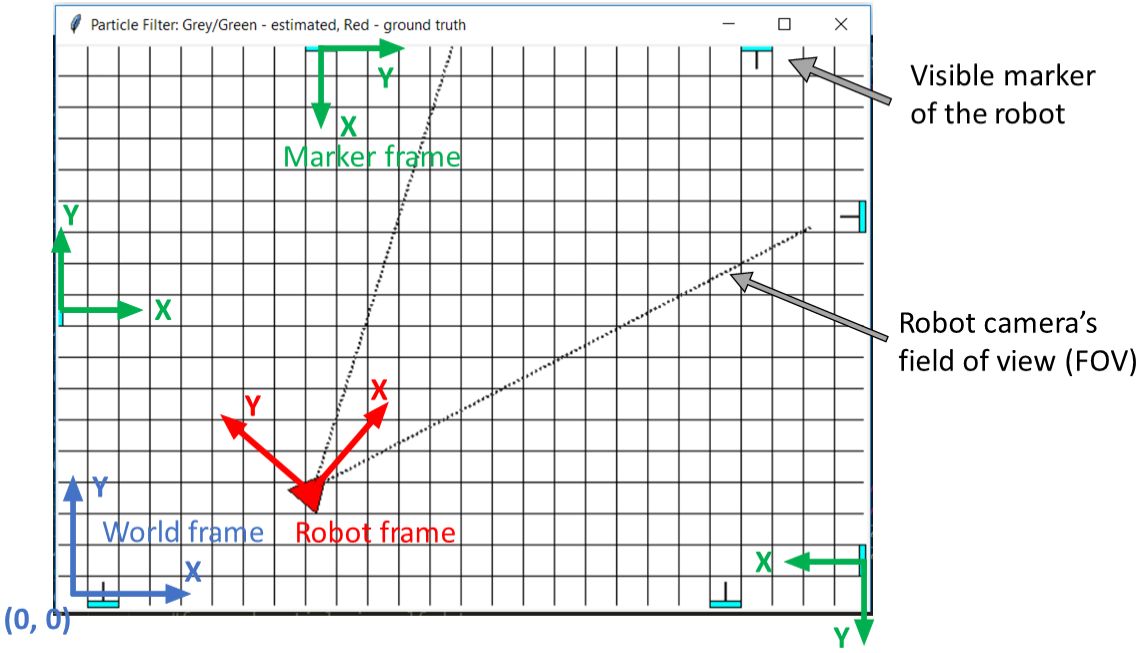

Fig 1. Coordinate frame definitions

The localization markers will appear only on the wall of the simulated arena. The direction the marker is facing is defined as the positive X direction of marker's local coordinate frame, and Y points left of the marker.

The simulated robot is equipped with a front-facing camera with a 45 degrees field of view (FOV), like Cozmo. The camera can see markers in its FOV. The simulation will output the position and orientation of each visible marker, measured relative to the robot's position. The simulated robot will also report its odometry estimates.

Motion update: `particles = motion_update(particles, odom, grid):`

The input of the motion update function includes particles representing the belief $p(x_{t-}|u_{t-1})$ before motion update, and the robot's new odometry measurement. The odometry measurement is the relative transformation of current robot pose relative to last robot pose, in last robot's local frame. It has format odom = (dX, dY, dH), as shown in Fig 2. To simulate noise in a real-world environment, the odometry measurement includes Gaussian noise, and noise level is defined in setting.py. Add noise to the updated pose of the particles using the given function *add_particle_noise* in *utils.py*. The output of the motion update function should be a set of particles representing the belief $p(x_t|x_{t-1}, u_t)$ after motion update.

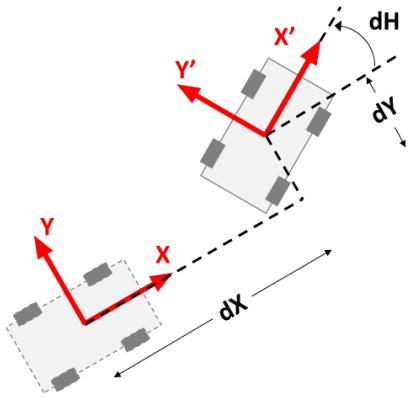

Fig 2. Odometry measurement definition

Measurement update: `particles = measurement_update(particles, measured_marker_list, grid):`

The input of the measurement update function includes particles representing the belief $p(x_t|x_{t-1}, u_t)$ after motion update, and the list of localization marker observation measurements. The marker measurement is calculated as a relative transformation from robot to the marker, in the robot's local frame, as shown in Fig 3. Same as odometry measurement, marker measurement is also mixed with Gaussian noise, noise level is defined in setting.py. The output of measurement update function should be a set of particles representing the belief after measurement update.

In addition to the noise in the odometry measurement, the marker detection may also be noisy. This is to simulate poor lighting conditions, camera issues etc. The marker detection is noisy in two ways. Either the robot could detect "spurious" markers or "drop" the correct markers. This corresponds to false positives and false negatives in your marker detector. We control the number of "spurious" or "dropped" markers using `SPURIOUS_DETECTION_RATE` and `DETECTION_FAILURE_RATE` variables defined in setting.py. These variables are the probabilities of the robot detecting "spurious" markers or "dropping" correct markers respectively.

Note: The measurement update must include resampling to work correctly. We will be setting `DETECTION_FAILURE_RATE` and `SPURIOUS_DETECTION_RATE` to 0 for most test cases and we will be cranking this up for a few test cases (See Grading section below).

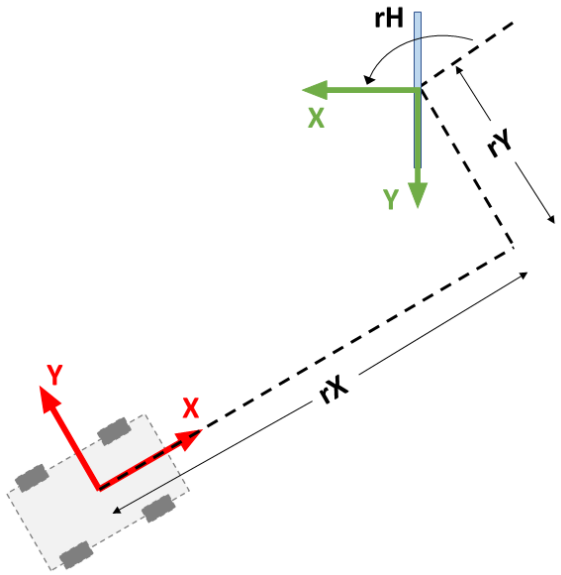

Fig 3. Marker measurement definition

In this lab you are provided the following files:

`particle_filter.py` – Particle filter you should implement

`autograder.py` – Grade your particle filter implementation

`pf_gui.py` – Script to run/debug your particle filter implementation with GUI

`particle.py` – Particle and Robot classes and some helper functions.

`grid.py` – Grid map class, which contains map information and helper functions.

`utils.py` – Some math helper functions. Feel free to use any.

`setting.py` – Some world/map/robot global settings.

`gui.py` – GUI helper functions to show map/robot/particles. You don't need to look into details of this file

You need to implement `motion_update()` and `measurement_update()` functions in `particle_filter.py`. Please refer to the particle class definition in `particle.py`.

To run your particle filter in simulation:

```
> py pf_gui.py       # in Windows
```

or

```
> python3 pf_gui.py # in Mac or Linux (or Windows in cases)
```

You will see a GUI window that shows the world/robot/particles after start. The ground truth robot pose will be shown as red (with dashed line to show FOV). Particles will be shown as red dot with a short line segment indicate the heading angle, and the estimated robot pose (average over all particles) will be shown in grey (if not all particles meet in a single cluster) or in green (all particles meet in a single cluster, means estimation converged).

Two types of simulated robot motion are implemented in `pf_gui.py`: (1). The robot drives forward, if hits an obstacle, the robot bounces to a random direction. (2). The robot drives in a circle (This is the motion the autograder uses). Feel free to change the setup in `pf_gui.py` for your debugging.

Grading: Your submission will be evaluated using the `autograder.py` listed above. Grading will separately evaluate three capabilities: (1). The filter's estimation can converge to correct value within a reasonable number of iterations. (2). The filter can accurately track the robot's motion. (3) The filter is robust enough to track robot's motion even when some of the markers are dropped or spuriously detected.

We use a total of 5000 particles, and we treat the average of all 5000 particles as the filter's estimation. The particles will be initialized randomly in the space. We define the estimation as *correct* if the translational error between ground truth and filter's estimation is smaller than 1.0 unit (grid unit), *and* the rotational error between ground truth and filter's estimation is smaller than 10 degrees. The grading is split into three stages, the total score is the sum of three stages:

1. [45 points] We let the robot run 95-time steps to let the filter find global optimal estimation. If the filter gives correct estimation in 50 steps, you get the full credit of 45 points. If you take more than 50 steps to get the correct estimation, a point is deducted for each additional step required. Thus, an implementation that takes 66 steps to converge will earn 29 points; one that does not converge within 95 steps will earn 0 points. For reference, our implementation converges within approximately 10 steps.
2. [45 points] We let the robot run another 100-time steps to test stability of the filter. The score will be calculated as 45**p*, where *p* is the percentage of "correct" pose estimations based on

the position and angle variance listed above.

3. [10 points] We let the robot run with `DETECTION_FAILURE_RATE = 0.1` and `SPURIOUS_DETECTION_RATE = 0.1` (Both these constants are defined in `settings.py`) and we will repeat the test cases in stages 1 and 2. However, we will scale the score down to 5 points for both stages 1 and 2. For instance, we will change both the variables to 0.1, run stages 1 and 2 and say you receive 45 and 40. We will then scale each of these scores to [0,5]. Therefore, your score for this stage would be 45 * (5/45) + (40 * 5/45) = 9.44. If you get more than 9 points you will receive full credit (i.e. (9,10] -> 10). Therefore, a score of 9.44 in the example mentioned above will earn full credit and you will receive 10 points. If you get less than or equal to 9 points you keep that score.

Notes and Suggestions:

1. In the first two stages of grading there will **not** be any spurious markers or dropped markers. We will enable this by setting the `DETECTION_FAILURE_RATE` and the `SPURIOUS_DETECTION_RATE` in `settings.py` to 0.
2. Your final grade will be an integer value. For example [98.5, 99.5) -> 99
3. In `autograder.py` the robot will follow a circular trajectory (see the `autograder.py` file for details). We provide several example circles in `autograder.py` for your testing, but in the final grading we will use **another 5 different circles**, and the score will be the average between these five tests. So, make sure you test several different cases to ensure reliability!
4. Particle filter is a randomized algorithm, each time you run you will get slightly different behavior. Make sure to test your code thoroughly.
5. If you need to sample a Gaussian distribution in python, use `random.gauss(mean, sigma)`.

Submission: Submit only your `particle_filter.py` file to Gradescope! Make sure you enter your name in a comment at the top of the file. Make sure the file remains compatible with the autograder. If you relied significantly on any external resources to complete the lab, please reference these in the submission comments.