LAB 5: PARTICLE FILTER PART I

Due: Tuesday, November 14th 3:00pm

The objective of Labs 5 and 6 are to implement a Particle Filter (a.k.a. Monte Carlo Localization). In this lab, you will work entirely in simulation to validate your algorithm. In Lab 6, we will adapt the algorithm to work on Cozmo.

Task: You 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, with the addition of localization markers on the wall of the arena, as shown in Fig 1. 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 left of the robot.

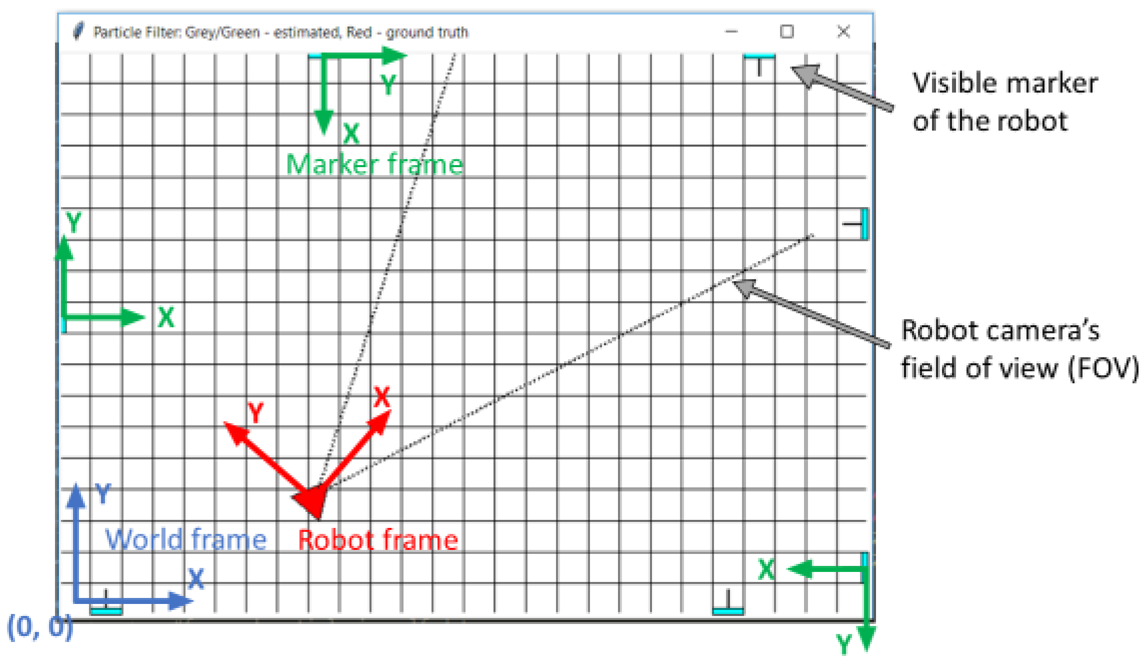

Fig 1. Coordinate frame definitions

The localization markers will appear only on the wall of the arena, and will be replaced by real QR-style markers in next lab. 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-degree field of view (FOV), similar to 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)`

The input of the motion update function includes particles representing the belief $p(x_{t-1}|u_{t-1})$ before motion update, and the robot's new odometry measurement. The odometry measurement is a pair of robot pose, $u_t = (\bar{x}_{t-1}, \bar{x}_t)^T$, with $\bar{x}_{t-1} = (\bar{x}, \bar{y}, \bar{\theta})^T$ and $\bar{x}_t = (\bar{x}', \bar{y}', \bar{\theta}')^T$. To simulate noise in real-world environment, the odometry measurement includes Gaussian noise, and noise level is defined in `setting.py`. The output of the motion update function should be a set of particles representing the belief $\tilde{p}(x_t|u_t)$ after motion update.

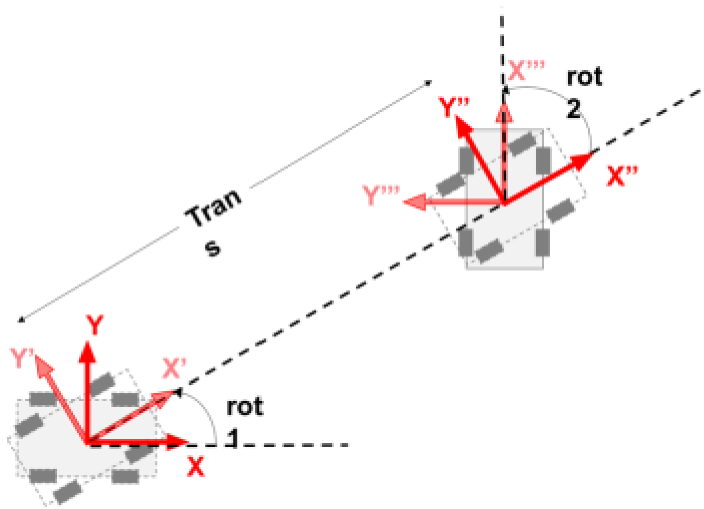

Fig 2. Odometry measurement definition

Measurement update: `particles = measurement_update(particles, mlist)`

The input of the measurement update function includes particles representing the belief $\tilde{p}(x_t|u_t)$ 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 list may contain several measurements (if the robot sees multiple markers in its FOV), or the list may be empty (if no markers are visible). The output of measurement update function should be a set of particles representing the belief $p(x_t|u_t)$ after measurement update. Note that the measurement update must include resampling to work correctly.

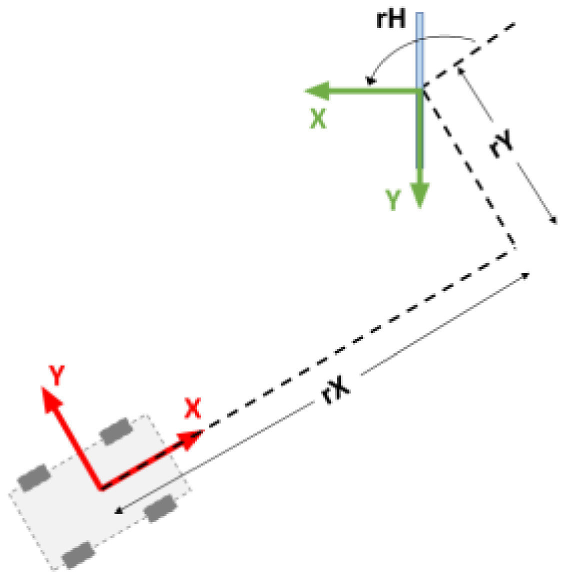

Fig 3. Marker measurement definition

In this lab you are provided the following files:

`particle_filter.py` – Particle filter you should implement

`autograder.py` – Auto-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 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
```

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, robot bounces to a random direction. (2). The robot drives in a circle (This is the motion auto-grader 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 two 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.

We use 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 15 degrees. The grading is split into two stages, the total score is the sum of two stages:

1. [50 points] Let the robot run 100 time steps to make the filter find global optimal estimation. If the filter gives correct estimation in 50 steps, you get full credit of 50 points. If you spend more than 50 steps to get correct estimation, a point is deducted for each additional step required. Thus, an implementation that takes 66 steps to converge will earn 34 points; one that does not converge within 100 steps will earn 0 points. For reference, our implementation converges within approximately 10 steps.
2. [50 points] Let the robot run another 100 time steps to test stability of the filter. Due to stochasticity of Monte Carlo method, we require the robot is close to ground truth state for 90 time steps. If your implementation can track the correct pose more than 90 time steps, you will earn 50 points. For every missing time step, you will lose one point. Therefore, if you track 70 time steps out of 100, you will get 30 points. For less than 40 time steps, you will get 0 point.

In `autograder.py` the robot will follow a circular trajectory (see the `autograder.py` file in 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. <u>So make sure you test several different cases to ensure the reliability!</u>

To use auto-grader, you have to run the following. Note auto-grader will not launch GUI window. Make sure your implementation well behaved before trying auto-grader.

```
> py autograder.py gradecase1.json       # in Windows
```

or

```
> python3 autograder.py gradecase1.json # in Mac or Linux
```

Since we have an auto-grader, there will **not** be an in-class demo for this lab.

Notes:

- In this lab you do not have to worry about the robot being 'kidnapped,' the robot will remain on a continuous trajectory.
- Particle filter is a randomized algorithm, each time you run you will get slightly different behavior. Make sure to test your code thoroughly.
- If you need to sample a Gaussian distribution in python, use `random.gauss(mean, sigma)`.

Submission: Submit only your `particle_filter.py` file, make sure you enter the names of both partners in a comment at the top of the file. Make sure the file remains compatible with the auto-grader. Only one partner should upload the file to T-Square. If you relied significantly on any external resources to complete the lab, please reference these in the submission comments.