LAB 5: Particle Filter – Part I
Due: Thursday, April 21nd 11:59pm

The objective of lab 5 is to implement the **Monte-Carlo Localization** (using Particle Filters) for Cozmo. This lab consists of two parts. In the first part, you will work entirely in simulation to validate your algorithm. In the next part, you will adapt the algorithm to work on the Cozmo robot.

Part I

In this part, you will implement a particle filter which takes robot ***odometry measurements*** as motion control input and ***marker measurements*** as sensor input. We have provided you with a starter code in which you must complete the implementation of two functions: `motion_update()` and `measurement_update()`. Since in the 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 that is a set of particles representing the belief.

Before giving more detail about the two functions, we first need to define a few coordinate frames. In this part as shown in Fig. 1, we will use a grid map with the addition of localization markers on the walls of the arena. The grid world has a coordinate frame on the origin (0,0) with X and Y axes pointing to the right and up respectively. The robot has a local coordinate frame in with X and Y axes pointing to the front and to the left of the robot respectively.

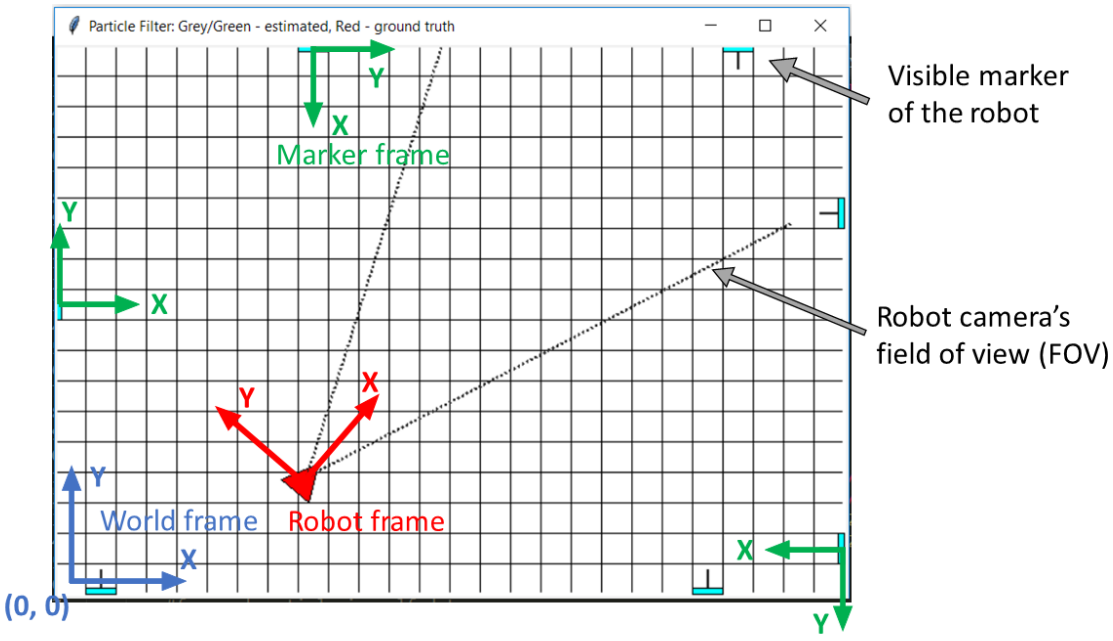

Figure 1. Coordinate frame definitions

The localization markers will appear only on the wall of the arena, and will be replaced by real AR markers in part 2 of the lab. Each marker has a coordinate framework with the X axis

perpendicular to the marker and pointing outwards, and the Y axis pointing to the left of the marker.

Similar to Cozmo, the simulated robot is equipped with a front-facing camera with a 45-degree field of view (FOV). 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.

Now, let's take a look at the *motion update* and *measurement update* functions.

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

The input of the `motion_update` function includes a set of particles representing the belief $p(x_{t-1}|u_{t-1})$ before motion update, and the robot's new odometry measurement. The odometry measurement includes 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 the real-world environment noise, the odometry measurement adds Gaussian noise to the values. The noise level is defined in the file `setting.py`. The output of the motion update function should be the set of particles representing the belief $\tilde{p}(x_t|u_t)$ after motion update.

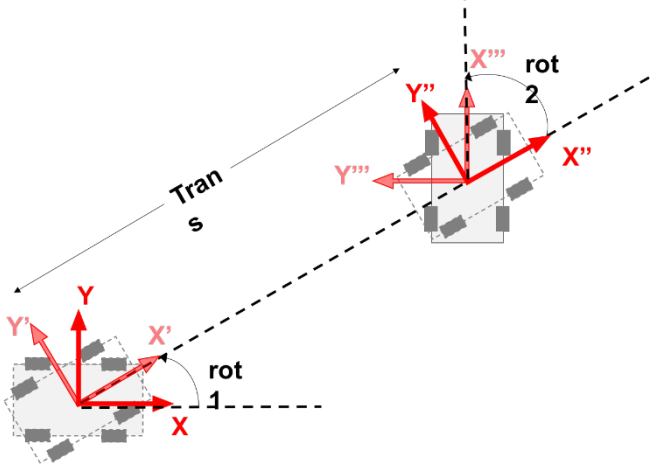

Figure 2 Odometry measurement definitions

The next step is to implement the measurement_update function.

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. As illustrated in Fig. 3, the marker measurement is calculated as a relative transformation from robot to the marker, in the robot's local frame. Same as the odometry measurement, marker measurement is also mixed with Gaussian noise and the noise level is defined in the file `setting.py`. Note that, the list may contain several measurements (if the robot sees multiple markers in its FOV), or no measurement (if no marker is 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 perform resampling to work correctly.

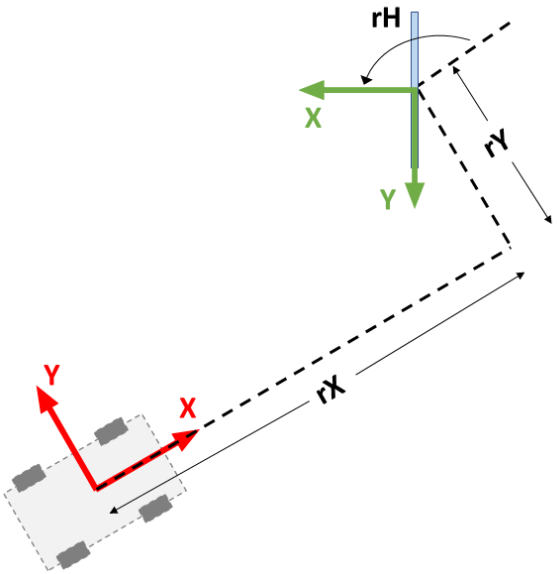

Figure 3 Marker measurement definitions

Files: In this part of the lab, you are provided with the following files:

`particle_filter.py` – the particle filter algorithm that you will implement
`autograder.py` – auto-grades your particle filter implementation
`pf_gui.py` – a script with GUI to run/debug your particle filter implementation
`particle.py` – particle and robot classes and some helper functions
`grid.py` – grid map class which contains map information and some helper functions
`utils.py` – includes some math helper functions
`setting.py` – includes some world/map/robot global parameters
`gui.py` – a GUI helper function to illustrate map/robot/particles

You will implement the `motion_update()` and the `measurement_update()` functions in the `particle_filter.py` . please refer to the particle class definition in `particle.py` for more information.

Run: To run your particle filter in simulation:

- `python3 pf_gui.py`

You will see a GUI window that shows the world/robot/particles. The ground truth robot pose will be shown in red (with a dashed line to show FOV). Particles will be shown as red dots with a short line segment indicating their headings. The estimated robot pose averaged over all particles will be shown in grey if particles do not meet in a single cluster, and in green if all particles meet in a single cluster (i.e., when estimation has converged).
Note that two types of simulated robot motion have been implemented in `pf_gui.py` : (1) the robot drives forward and bounces to a random direction when hits an obstacle; (2) the robot drives in a circle (the motion used by the auto-grader). Feel free to change the setup in `pf_gui.py`.

Grading: Your submission will be evaluated using the `autograder.py`. We will evaluate two capabilities: (1) the filter's estimation can converge to the correct value within a reasonable number of iterations; and (2) the filter can accurately track the robot's motion.
We use a total of 5000 particles and consider 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. A total of 100 points for this part can be gained in to 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 your 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 the Monte Carlo method, we require the robot to remain 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. Therefore, we have provided you with several instances of circular trajectories in `autograder.py` for your testing. However, in the final grading, we will use another 5 different circles and the score will be the average of these five tests. So make sure to test your implementation with several different cases to ensure the reliability!

To use the `auto-grader`, you have to run the following:

- `python3 autograder.py gradecase1.json`

Note that the auto-grader will not launch the GUI window, so make sure your implementation works before trying the auto-grader.

Notes:

- There will be no in-class demo for this part of the lab.
- In this part of the lab, you do not have to be worried about the robot being 'kidnapped'. The robot will remain on a continuous trajectory.
- Particle filter is a stochastic algorithm and each time you run it you will get a slightly different behavior, so, make sure to test your code thoroughly.
- To sample a Gaussian distribution in python, use `random.gauss(mean, sigma)`.

Submission: Submit only your `particle_filter.py` file and make sure to 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 Blackboard. If you relied significantly on any external resources to complete the lab, please reference these in the submission comments.