PROJECT 2: PARTICLE FILTER

Due: Wednesday, February 12 11:59pm EST

The objective of Project 2 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.

Recall that a particle filter repeats these main steps:

- A motion update step, in which the particles representing the robot's potential location are updated according to the motion control command sent to the robot
- A measurement update step, in which each particle is weighted according to the likelihood of its position being correct given the sensor inputs
- An importance resampling step, where before the next motion update, the particles are resampled and their weights re-normalized according to their weights from the previous step.

Please see the lecture slides for more details.

In this lab, we provide skeleton code in `utils.py` and `particle_filter.py`, and your job is to fill in the missing pieces. To install the necessary libraries to complete the project and run the gui, call `pip install -r requirements.txt.`

Start by implementing the three missing functions in `utils.py`. Next, implement all the missing functions in `particle_filter.py` to build a full particle filter! (Note: in our implementation we combine the measurement update and importance resampling into one function called `measurement_update()`). You may look at any of the other provided files, but please do not modify them, as the autograder expects those files to remain unchanged.

IMPORTANT: the autograder for this lab is *slow* (will take around 5 minutes to finish grading) and not helpful for debugging purposes (we hide the test cases). **To debug your implementation, we strongly recommend using the local unit tests and local simulator instead of the autograder.** This will allow you to get feedback more quickly and see how the particle filter is actually behaving. **The autograder may not be able to execute your code successfully unless you finish both `utils.py` and `particle_filter.py`. Please make sure the estimation looks fine on local simulator before submitting your code to Gradescope.**

To run the local sanity checks for `utils.py` and `particle_filter.py`, call `python3 local_tests.py` in your terminal. This will provide sanity checks for all functions except `measurement_update`, which is expected to be debugged through the local simulator. **Passing the sanity checks does not guarantee a successful implementation as these tests are minimal, but they should indicate that you are on the right track!**

To run the local simulator, call `python3 pf_gui.py` in your terminal. (Note this will not work until you implement the required functions or temporary workarounds, the method docstrings will call out any such functions/workarounds).

The local tests and local simulator are just designed for you to debug. We only consider your score on Gradescope when calculating your final score for this project.

Below, we provide more information on the local simulator, some helpful tips for implementing a particle filter, and additional details on grading and submission.

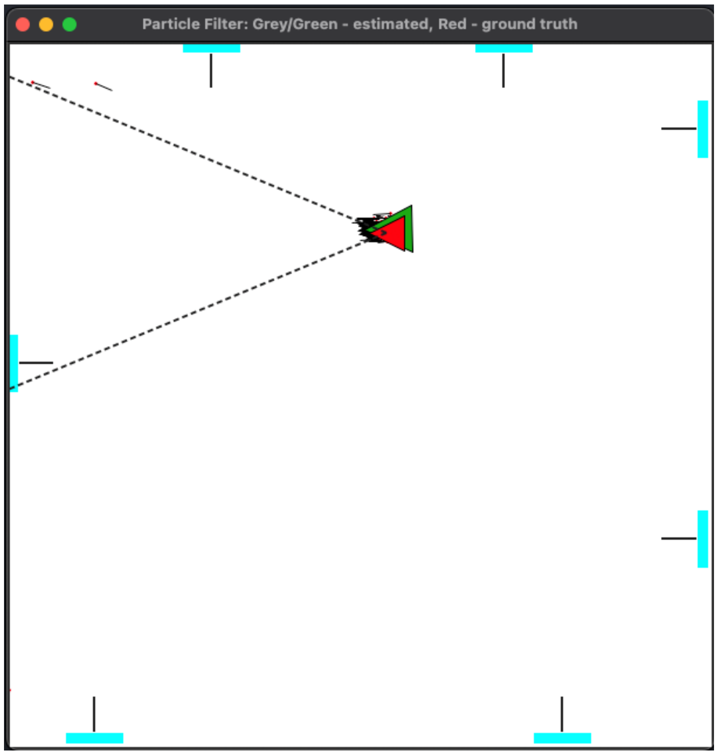

Figure 1: A correctly implemented particle filter. Notice (nearly) all the particles have converged around the same estimate.

Simulator: After launching the local simulator, you will see a GUI window that shows the world/robot/particles. 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, meaning your estimation has converged).

The simulated robot is equipped with a front-facing camera with a 45 degrees field of view (FOV), like a real robot called a 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, once they come into view. The simulated robot will also report its odometry estimates.

Two types of simulated robot motion are implemented in `pf_gui.py`: (1) The robot drives forward until it hits an obstacle, then it bounces in a random direction and repeats. (2) The robot drives in a circle (this is the motion the autograder uses). Feel free to change the setup at the top of `pf_gui.py` for debugging.

At each step of the simulator, the following sequence occurs:

1. Odometry Measurements are received
2. Your `motion_update` function is called with the odometry measurements
3. Markers read by the robot are received
4. Your `measurement_update` function is called.

And this sequence repeats for as long as the simulator is active.

Coordinate frames: One of the key challenges in implementing a particle filter (or mobile robotics in general) is coordinating coordinate frames. We provide a diagram to help you with this in Figure 2. The 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. You will have to handle the difference between local robot frame and global grid-world frame in the motion update.

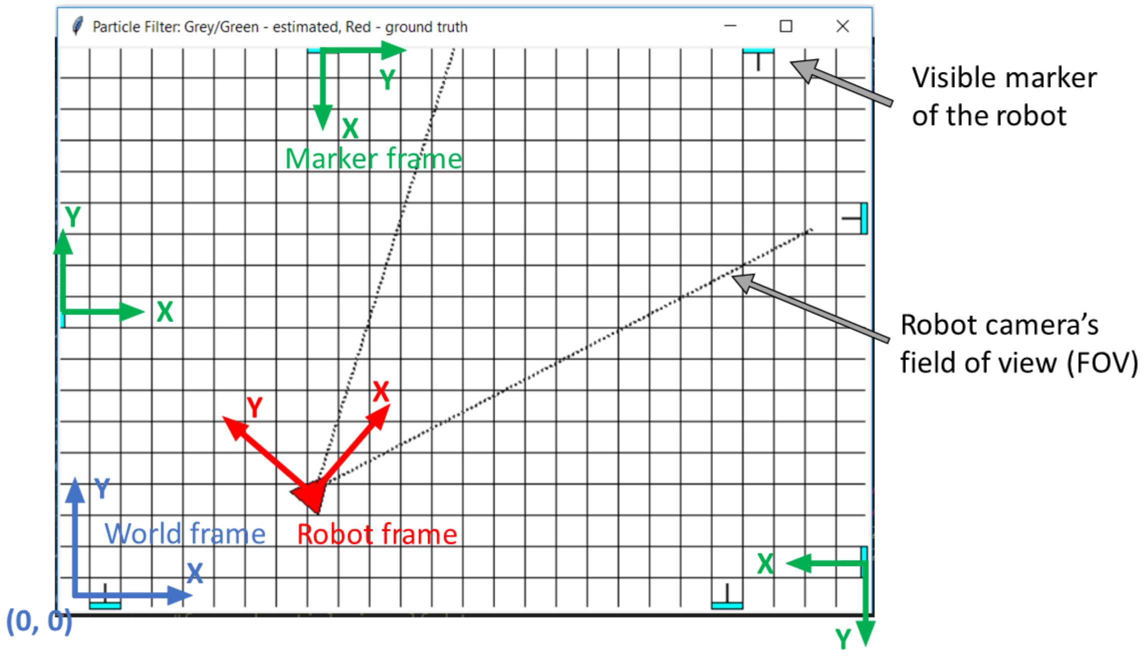
Particle Filter: Grey/Green - estimated, Red - ground truth
Y
X
Marker frame
Visible marker of the robot
Y
X
Robot camera's field of view (FOV)
Y
X
Y
World frame
Robot frame
X
(0, 0)
X
Y

Figure 2: 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. Again, you will have to handle the discrepancy in coordinate frames, this time in the measurement update.

Motion update details: 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, as shown in Figure 3. To simulate noise in a real-world environment, the odometry measurement given to you includes Gaussian noise. 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. **Your motion update must account for the noise in the odometry measurement in order for your filter to converge properly.**

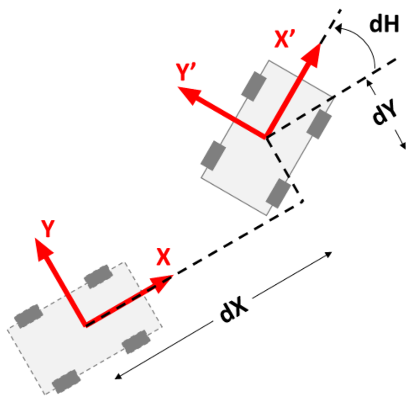

Figure 3: Odometry measurement transform

Measurement update details: 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 Figure 4. Same as odometry measurement, marker measurements are also given to you with Gaussian noise. The output of your measurement update function should be a set of particles representing the belief after the measurement update and resampling.

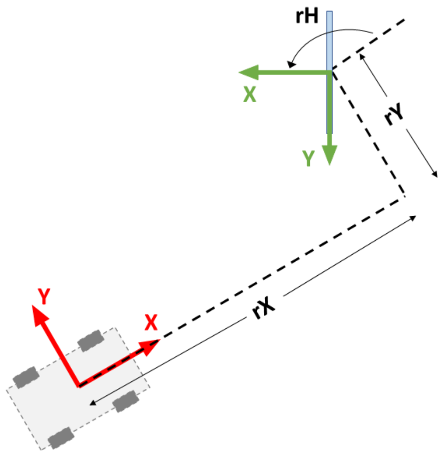

Figure 4: Marker measurement transform

Note: During measurement update, we want to find the likelihood of the particle pose being the robot's pose based on how well the markers seen by the robot align with the makers "seen" by the particle. We model the probability of a marker read by the particle matching a marker read by the robot as two Gaussian distributions centered about 0, one capturing the probability of two markers matching as a function of the distance between them, and the other capturing the probability of two markers aligning as a function of the heading difference between them. Therefore, for a given marker seen by the robot and a marker "seen" by the particle, we can find the probability of the two matching as:

$$P(x) = e^{-\left(\frac{distBetweenMarkers^2}{2\sigma_{trans}^2}+\frac{angleBetweenMarkers^2}{2\sigma_{head}^2}\right)}$$

The relevant standard deviations can be found in `settings.py`.

Submission: Submit both your `utils.py` and `particle_filter.py` file to Gradescope. Enter your name in a comment at the top of the file. Do not modify any other files, or your code may not work with the autograder! If you relied significantly on any external resources to complete the lab, please reference these in the submission comments.

Grading: Your grade on Gradescope is your final grade. The autograder evaluates whether:

1. [50 points] Your filter's estimation can converge to correct value within a reasonable number of iterations.
2. [50 points] Your filter can accurately track the robot's motion.

More details on grading: 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 two stages, the total score is the sum of two stages:

1. 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 50 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 34 points; one that does not converge within 95 steps will earn 0 points. For reference, our implementation converges within approximately 10 steps.
2. We let the robot run another 100-time steps to test stability of the filter. The score will be calculated as 50**p*, where *p* is the percentage of "correct" pose estimations based on the position and angle variance listed above.
3. Note: we round all scores to the nearest integer (up or down as needed).