PROJECT 4: PARTICLE FILTER IMPLEMENTATION IN WEBOTS SIMULATOR

Due: Wednesday, March 13th 11:59pm EST

In Project 4, you will implement a particle filter in the Webot simulator. It builds upon the foundation established in prior labs, emphasizing on:

1. **Coordinate Transformation**: To understand and implement coordinate transformations between the world frame, sensor frame, and robot frame to enable accurate localization and perception in robotics tasks.
2. **Differntial Drive:** To understand and implement the kinematics of a differential drive robot.
3. **Detection Failure and Spurious Detection Handling**: To develop strategies to address detection failures and spurious detections, ensuring reliable performance of robotic perception systems in challenging scenarios.
4. **Operating in Ambiguous Environments**: To explore techniques for navigating larger environments with strong ambiguity, including solving the kidnapped robot problem, to enhance adaptability and robustness in real-world robotics applications.

A. Project Structure:

The project directory, Project_4 contains the following files:

Worlds:

- **simple_world.wbt:1** A simple square world for testing the particle filter. (Ref. Figure 1b)
- **maze_world1.wbt:** A square world with more markers and a wall for evaluating the filter's performance in a more complex scenario. (Ref. Figure 1a)

Figure 1a: Top view of the maze world

Figure 1b: Top view of the simple world

Controllers:

- **proj4_simple_world1_controller.py:** Controls the robot to turn in-place in the center in the simple world.
- **proj4_maze_world1_controller.py:** Controls the robot's movement along a predefined trajectory in the maze world.

Particle_filter:

- **contour.py:** Deals with extracting contours from camera images.
- **environment.py:** Handles the simulation environment, including robot, markers, and their interactions.
- **geometry.py:** Contains functions related to geometric calculations and transformations.
- **gui.py:** Creates a Graphical User Interface (GUI) for visualizing the particle filter simulation.
- **particle_filter.py:** Has the particle filter algorithm.
- **run_pf.py:** Reads captured data in "data" folder and run particle filter without Webots simulator.
- **sensors.py:** Contains functions related to converting sensor data for the particle filter algorithm.
- **setting.py:** Holds various configuration settings used throughout the project.
- **utils.py:** Contains utility functions commonly used in the project.
- **unit_tests.py**: Contains unit tests for functions in **geometry.py, environment.py**. **It doesn't test the implementation of particle filter.**

B. General Guidance:

- You will implement functions in order in **geometry.py**, **environment.py**, and **particle_filter.py**.
- Refer to the comments in each **TODO** for specific implementation guidance.
- Use reference lectures and additional resources for understanding the particle filter algorithm.
- Utilize the provided unit tests to validate your code by running **unit_tests.py**.
- Submit the modified files in the controller's folder as instructed.

C. Instructions

Please follow these instructions carefully to ensure successful project completion.

First, complete code modifications in geometry.py and environment.py:

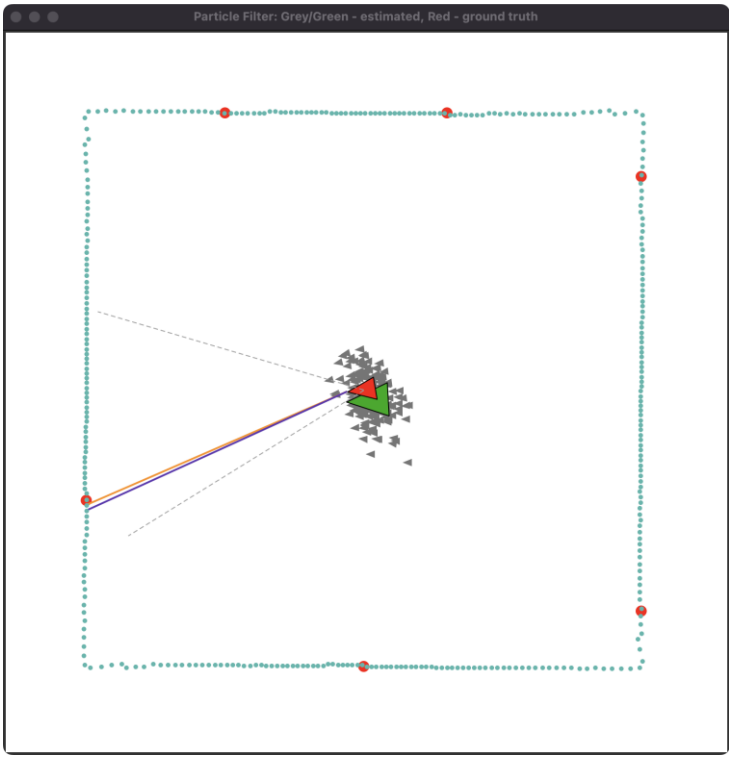
- **geometry.py:** transform_point(), compose(), inverse()
- **environment.py:** read_marker_measures() and diff_drive_kinematics()

Run **unit_test.py** to verify your implementations in **geometry.py** and **environment.py**. Ensure all test cases pass before proceeding to the next steps. This step is crucial for validating the correctness of your code modifications. When the provided test cases in **unit_tests.py** pass, implement functions in **particle_filter.py**

- particle_likelihood()
- compute_particle_weights()

Your goal is to make the estimated robot pose as close as possible to the ground-truth robot pose. You can open the world files in "worlds" folder and run the robot controller in Webots. The robot will move along a predefined trajectory. A Python GUI will display to visualize the particles. The larger

triangles represent the robot and its estimation, and the smaller triangles indicate the particle poses. The orange and purple lines in front of the robot illustrate marker measurements by cameras and lidar, respectively, and the grey dashed lines are field-of-view of the robot. The small cyan dots represent the lidar array.



You can verify the particle filter convergence in one of the following ways:

- Test in Webots
  - Set *DATA_CAPTURE_MODE = False* in **setting.py**.
  - Open a Webots world file and run the attached controller.
  - A python GUI will show up to visualize the particles.
- Capture data in Webots then run particle filters separately
  - Set *DATA_CAPTURE_MODE = True* in **setting.py**.
  - Open a Webots world file and run the attached controller. The captured data will be stored in the "data" folder.
  - Change *SCENARIO_NAME* variable to the desired world name in **run_pf.py**.
  - Run **run_pf.py** in particle filter.
  - A python GUI will show up to visualize the particles.

It usually takes several minutes to finish running particle filter on the maps.

D. Hints for code completion:

- **geometry.py:**

  - The SE2 class represents a 2D pose/transformation, incorporating both position and orientation components.
  - Use this class to represent the pose of a coordinate frame, perform coordinate frame transformations, and apply transformation operations to rotate and translate coordinate frames or points.
  - Implement methods for point transformation (*transform_point*), composition of transformations (compose), and inversion of transformations (inverse).
- **environment.py:**
  - The *read_marker_measures* function generates expected ground-truth marker measurements given the pose of the robot.
  - Utilize coordinate transformations to compute the marker positions relative to the robot's pose.
  - Remember to handle visibility checks and consider hints provided in the function comments.
  - Utilize the provided robot and wheel radius to convert wheel rotational speeds into linear and angular velocities for the *diff_drive_kinematics* function. And then, apply the kinematic equations specific to differential drive robots to calculate the forward speed and counterclockwise rotational speed based on the rotational speeds of the left and right wheels.
- **particle_filter.py:**
  - The particle_likelihood function calculates the likelihood of a particle pose being the robot's pose based on observed marker measurements.
  - Treat unmatched particle marker measures as detection failures and unmatched robot marker measures as spurious detections.
  - Utilize helper functions like *generate_marker_pairs* and *marker_likelihood* implemented in Project 3.
  - Incorporate the likelihood of unmatched markers, considering parameters like spurious detection rate and detection failure rate.
  - Implement the *compute_particle_weights* function to compute the important weights of particles given by the robot marker measures.
  - Optionally, handle scenarios where no markers are observed as no information is available and adjust the weights accordingly.

E. Submission:

- Submit the modified files (**geometry.py**, **environment.py**, and **particle_filter.py**) to Gradescope by the deadline.
- Include your name in a comment at the top of each file.

F. Grading: (Total 100 points)

- Geometry.py (30 points)
  - transform_point () - 10 points
  - compose () - 10 points

  - inverse () - 10 points
- environment.py (30 points)
  - read_marker_measures () - 20 points
  - diff_drive_odometry () - 10 points
- Integration test (40 points)
  - simple map – 15 points
  - maze map – 25 points

G. Additional Notes:

- Run the provided local unit tests on the simple world before proceeding to the maze world.
- Ensure your implementations adhere to the specified requirements and maximize your score potential by following the instructions carefully.