PROJECT 3: PARTICLE FILTER IMPLEMENTATION IN WEBOTS SIMULATOR

Due: Wednesday, February 26 11:59pm EST

In Project 3, you will implement a particle filter in the Webots simulator. It builds upon the foundation established in prior projects, 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 robotic tasks.
2. **LiDAR Measurement Simulation**: To understand and implement LiDAR measurement simulations for particles.
3. **Differential Drive**: To understand and implement the kinematics of a differential drive robot.
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. OVERVIEW:

In this project, an E-Puck robot navigates in a simple maze environment (Figure 1) without knowing its initial pose at the start. The robot solely relies on its LiDAR sensor measurements to localize itself using a particle filter. However, the maze has symmetrical features, which leads to identical LiDAR measurements at multiple locations within the map. This prevents the particles to converge to a single cluster at the beginning (Figure 2a). Still, with a correctly implemented particle filter, the robot can successfully localize itself as it starts to explore areas with asymmetrical features (Figure 2b).

Figure 1. Top view of the maze world in Webots simulator

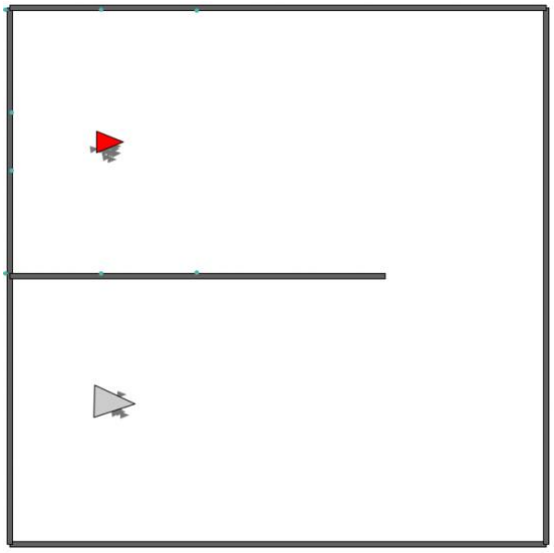

Figure 2a. Confused particle filter due to symmetry

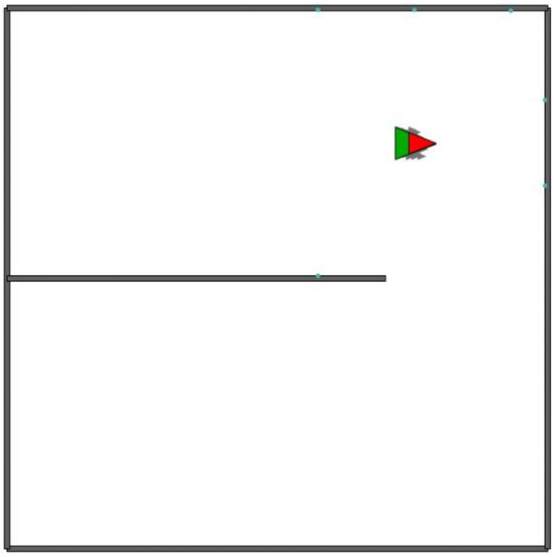

Figure 2a. Converged filter after exploring asymmetrical regions

The E-Puck robot used in this project is equipped with a LiDAR that measures distance to objects at 10 different fixed angles (Figure 3a). The measurements made by the 10 LiDAR rays change with the pose of the robot. Based on this observation, you will implement a particle filter that localizes the robot in the maze using its LiDAR measurements. The key task for achieving this is simulating the lidar measurements for particles as described in Figure 3b.

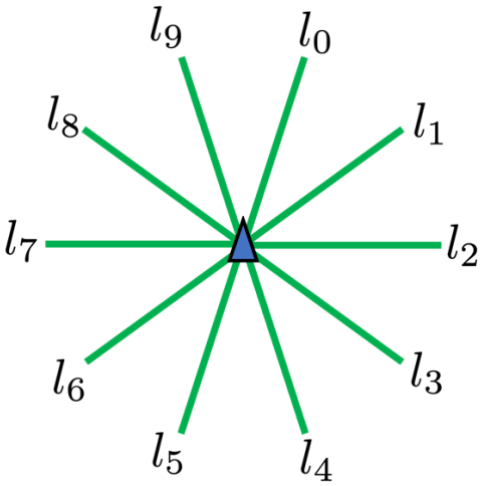

Figure 3a. 10 LiDAR rays of the robot

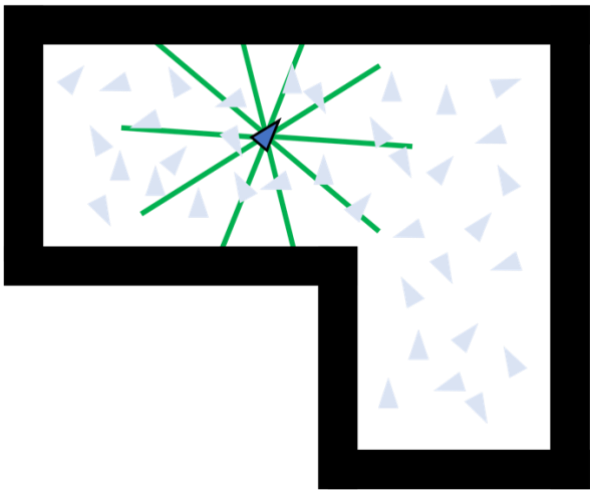

Figure 3b. LiDAR simulation for a particle

In Webots simulator, each wall is represented as a rectangle with 4 corners. Accordingly, you need to check if a LiDAR ray intersects with a rectangle and determine which edge(s) of the rectangle intersect with the ray to compute the expected distance measurement. If a ray intersects

multiple edges of wall(s), the true distance measurement is the shortest distance. An example of this is shown in Figure 4.

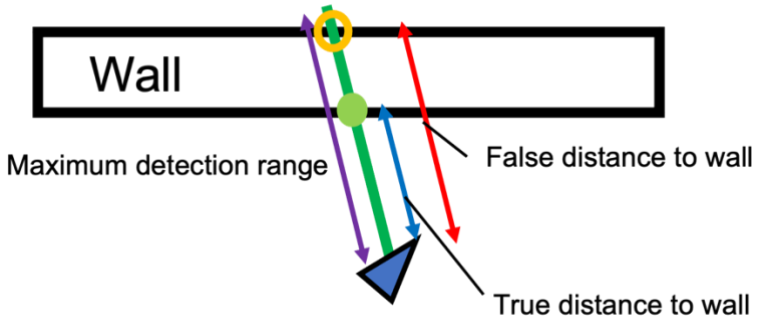

Figure 4. LiDAR distance measurement simulation

In this project, you also need to understand the kinematics of a differential drive robot (Figure 5). The E-Puck robot has two wheels, one on each side, and its linear and angular velocities, *v* and ω, are determined by the angular velocities of both wheels. Given the angular velocities of the two wheels, measured with wheel encoders in the Webots simulator, you need to compute the robot's linear and angular velocities.

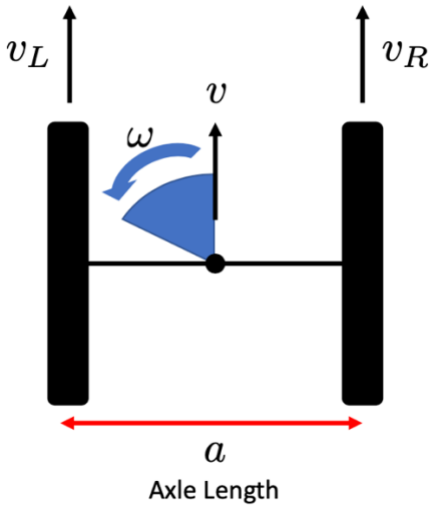

Figure 5a. Top view of a differential drive robot

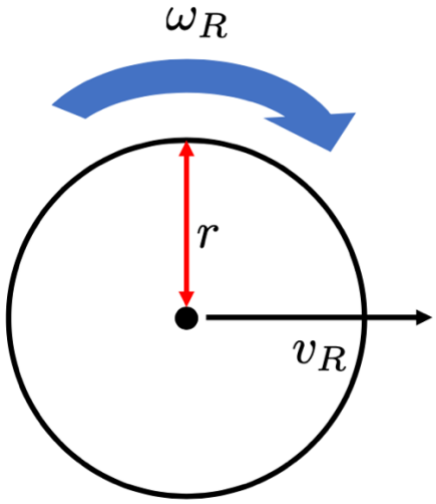

Figure 5b. Side view of a differential drive robot

B. Project Structure:

The project directory, Project_4 contains the following files:

Worlds:

- **maze_world1.wbt:** A small maze environment with symmetrical features that

confuse the robot during particle filter localization. (Ref. Figure 1a)

CONTROLLER:

- **proj4_maze_world1_controller.py:** Controls the robot's movement along a predefined trajectory in the maze world.

PARTICLE FILTER:

- **environment.py:** Handles the simulation environment including the robot 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.
- **wall.py:** Contains functions for keeping track of the 4 corner points of each wall.
- **lidar_sim.py:** Contains functions for simulating LiDAR measurements given a particle pose.
- **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, and lidar_sim.py**. **It does NOT test the implementation of particle filter.**

C. GENERAL GUIDANCE:

- You will implement functions in the following order: **geometry.py**, **environment.py**, **lidar_sim.py** and **particle_filter.py**.
- Refer to the comments in each function 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.

D. 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:** diff_drive_kinematics()
- **lidar_sim.py:** read()

Run **unit_test.py** to verify your implementations in **geometry.py** and **environment.py,** and **lidar_sim.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 file ("maze_world1.wbt") 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 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.
  - 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 map.

E. 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**
    - 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. You may find Figure 5 helpful.

- **lidar_sim.py**
    - Note that the Webots LiDAR measurement angle starts from lidar_resolution/2. For example, if the robot's lidar resolution is 90 degrees, the angles for the LiDAR rays are 45, 135, 225, 315 degrees relative to the robot and **NOT** 0, 90, 180, 270 degrees.
    - For this project, we are using 10 LiDAR rays, and the maximum distance for each ray is 1.0 m.
    - Webots LiDAR ray returns inf instead of its maximum range if no objects are detected.
    - Utilize helper functions provided in utils.py.

- **particle_filter.py:**
    - The particle_likelihood function calculates the likelihood of a particle pose being the robot's pose based on the lidar measurements. The likelihood can be computed using the following multi-variate Gaussian function:

$$P(x) = e^{-\left(\sum_{i=0}^{9} \frac{measurementDifference\_l_i{}^2}{2\sigma_{lidar}^2}\right)}$$

    - The inf LiDAR measurement values need to be replaced with the maximum detection range for the particle_likelihood function.
    - Implement the *compute_particle_weights* function to compute the important weights of particles given by the robot lidar measurements.

F. SUBMISSION:

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

G. GRADING (TOTAL OF 100 POINTS):

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

    - inverse () - 10 points
- environment.py (10 points)
    - diff_drive_odometry () - 10 points
- lidar_sim.py (30 points)
    - read () - 30 points
- Integration test (30 points)
    - maze map – 30 points

H. Additional Notes:

- Ensure your implementations adhere to the specified requirements and maximize your score potential by following the instructions carefully.
- To install the necessary libraries to complete the project and run the gui, call pip install -r requirements.txt.
- 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 geometry.py, environment.py, lidar_sim.py and particle_filter.py. Please make sure the estimation looks fine on local simulator before submitting your code to Gradescope.**
- **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!**
- **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.**