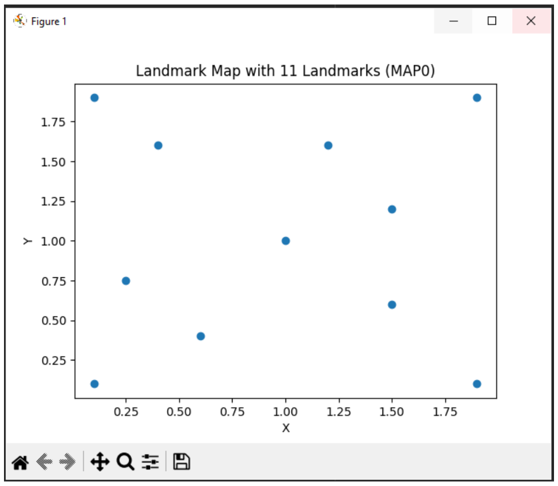
Assignment3

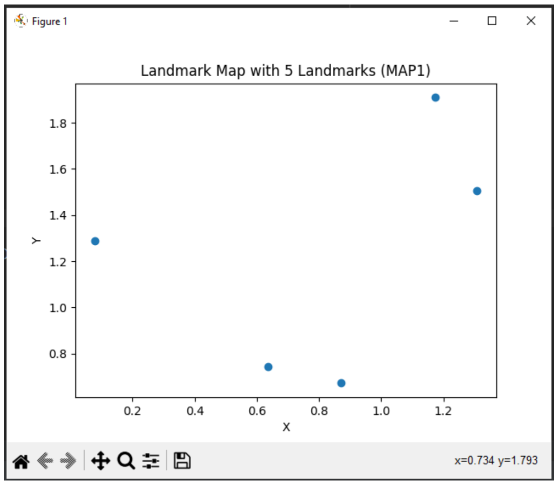
Benyamin and Sebastian

December 2023

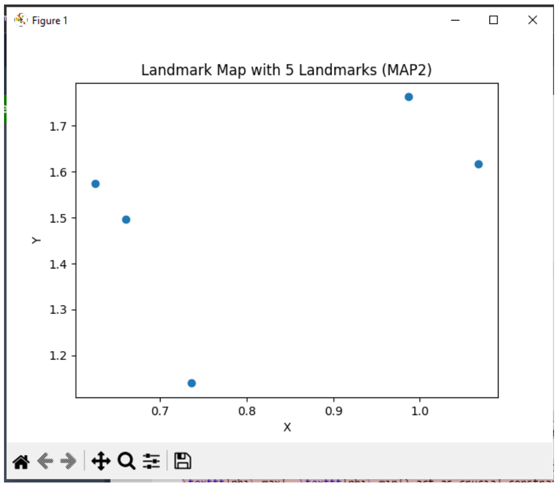
Problem 1.1



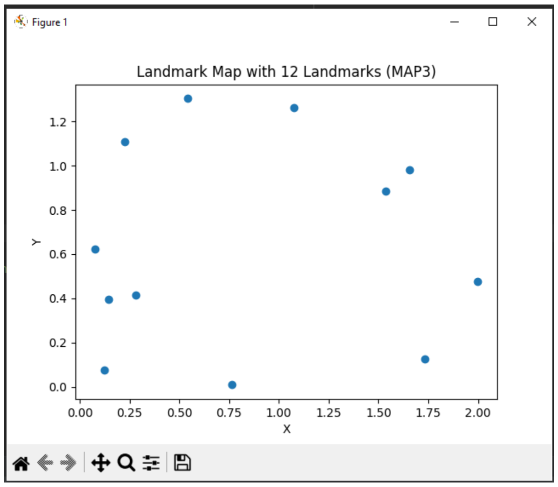
landmark_0.npy



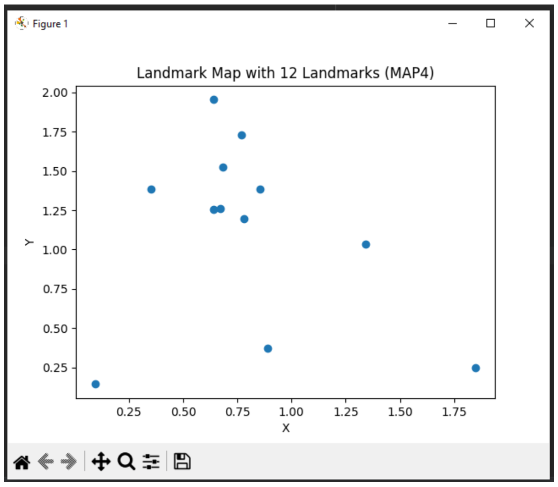
landmark_1.npy



landmark_2.npy



landmark_3.npy



landmark_4.npy

Problem 1.2

Control Sequence Generation

Control Initialization:

The initial control input (`u`) is meticulously set to zero velocity and zero angular velocity, establishing a neutral starting point for the robot's movement.

Explicit control limits (`v_max`, `v_min`, `phi_max`, `phi_min`) act as crucial constraints, defining the allowable range for linear and angular velocities.

Differential Drive Model:

The `differential_drive_model` function intricately employs a fundamental kinematic model, capturing the nuanced relationship between the robot's state and the applied control input.

Control Sequence Loop:

The core simulation loop, governed by a `while` loop structure, orchestrates the robot's movement for a predetermined number of steps (`total_steps`).

A condition within the loop ensures the simulation's resilience: if the robot nears the predefined boundaries, an intelligent reset mechanism is triggered.

Updating State and Storing Trajectory:

Precision is maintained as the robot's state (`q`) undergoes continual updates, gracefully following the kinematic model within each iteration.

The trajectory is systematically crafted, with the current position elegantly appended to a dedicated list, allowing for a detailed record of the robot's path.

Visualization:

The visual representation is carefully crafted using Matplotlib, offering a clear and illustrative display of the simulation.

The green dashed line signifies the evolving trajectory, providing a visual narrative of the robot's journey.

The robot's body is artistically portrayed as a rotated rectangle, contributing to an intuitive understanding of its orientation.

Landmarks from the map are thoughtfully scattered across the plot, enhancing the visual context.

Control Change:

Introducing an element of dynamism, the control input undergoes a deliberate randomization process every `steps_per_sequence` steps.

This deliberate variability injects diversity into the robot's motion, simulating scenarios where the robot adapts its behavior over time.

Control Sequence Storage:

A systematic approach to data management is maintained as the generated control sequences are structured into a NumPy array and judiciously saved to a designated file in the "controls" folder.

Experimentation

Random Initialization:

The introduction of randomness is methodically incorporated into the simulation, where the robot's initial state, encompassing both position (`x`, `y`) and orientation (`theta`), is randomly set within predetermined ranges.

Multiple Maps:

Demonstrating versatility, the code seamlessly navigates through different maps, each characterized by distinct landmark configurations. Control sequences are meticulously generated for each, facilitating a comprehensive exploration of various scenarios.

Trajectory and Landmarks:

Real-time visualization emerges as a powerful tool, offering an immersive insight into the robot's trajectory and its interaction with landmarks present in the environment.

This dynamic visualization contributes significantly to a nuanced understanding of the simulated robot's motion and spatial awareness.

Control Change Strategy:

The deliberate randomization of control inputs introduces a strategic layer to the simulation, simulating scenarios where the robot exhibits adaptive behavior over time.

This strategy enhances the exploration aspect of the simulation, allowing for a diverse range of robot behaviors to be observed and analyzed.

Resetting Simulation on Border Hit

Boundary Check:

A meticulous boundary check condition diligently evaluates whether the robot's current position, represented by `q[0]` and `q[1]`, transgresses the predefined spatial boundaries (0.2 to 1.8 for both `x` and `y`).

Resetting State and Controls:

When the boundary condition is met, the simulation gracefully resets, ensuring a seamless transition to a new scenario:

The robot's state (`q`) is recalibrated to a freshly randomized position (`[x, y, theta]`).

The current control input (`current_control`) is judiciously

Control Sequence and Trajectory Reset:

Prudent data management is exemplified as the control sequence (`control_sequence`) and trajectory (`trajectory`) are systematically cleared. This resets the data collection process, preparing the simulation for a new phase.

Print and Continue:

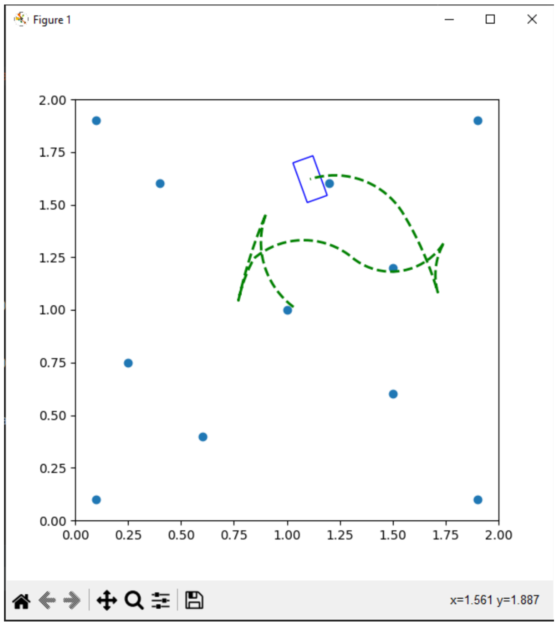
Communicating effectively, the code outputs a descriptive message to signify the occurrence of the reset, providing essential insights into the simulation's state.

The loop counter (`step`) is adeptly reset to zero, ensuring a seamless continuation of the simulation from the new initial state.

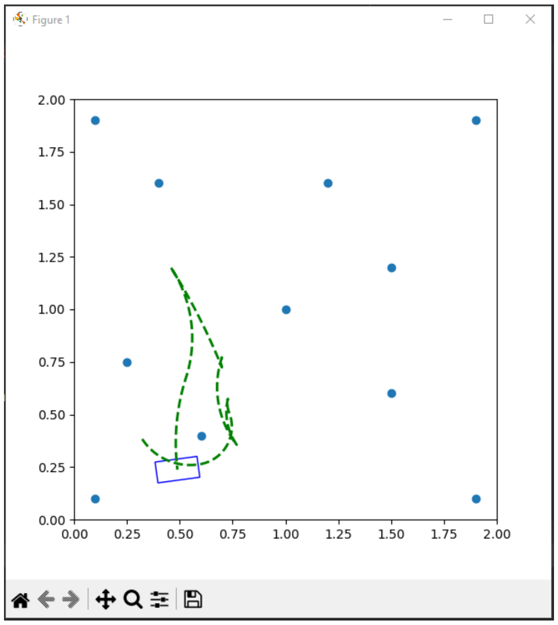
Continued Simulation:

The reset mechanism ensures a consistent and responsive simulation experience as the code seamlessly resumes from the updated initial state. This ensures that the robot's movement remains dynamic and adaptable, catering to different environmental conditions and scenarios.

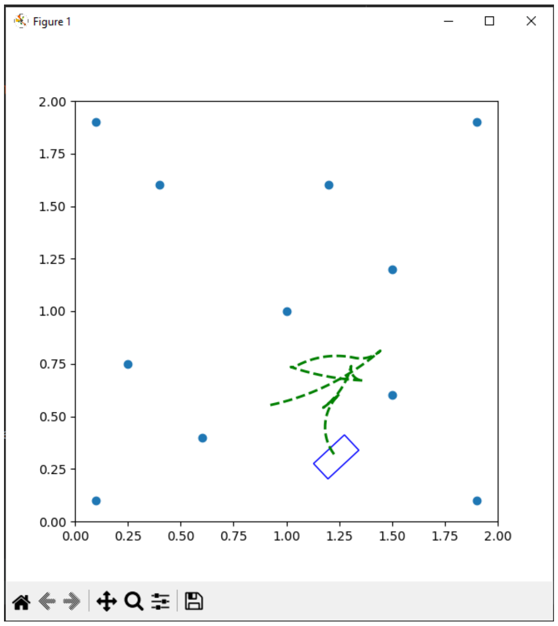
Visualizations of the 10 control sequences:



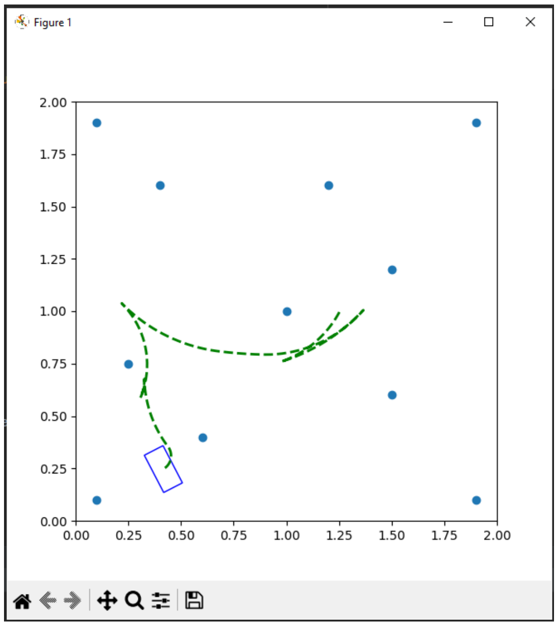
controls/$_0/_1.npy$



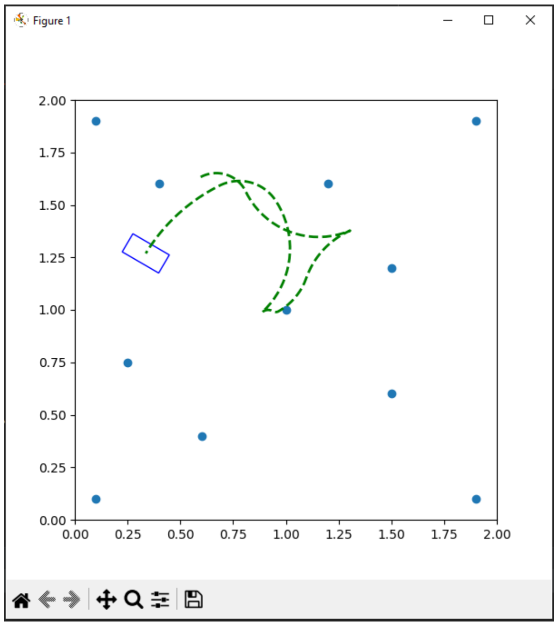
controls/$_0/_2.npy$



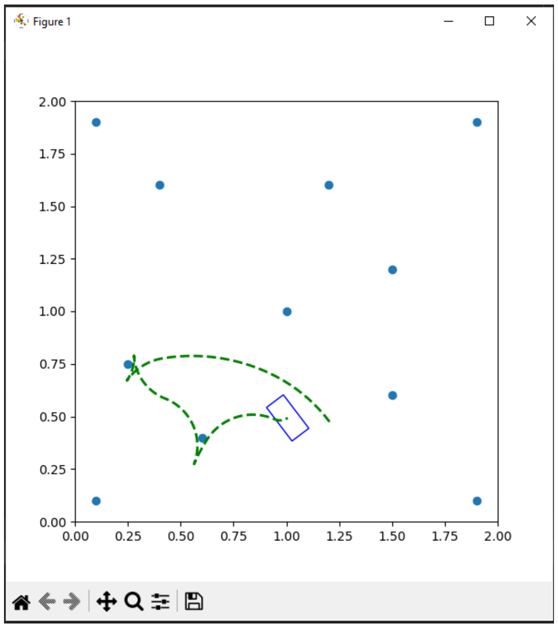
controls/$_1/_1.npy$



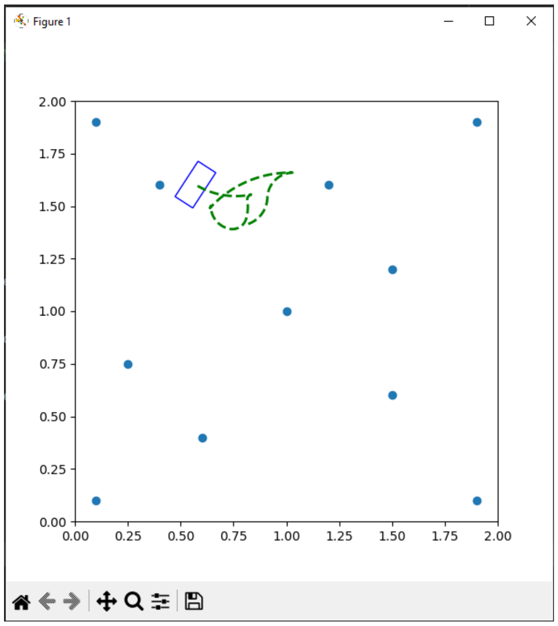
controls/$_1/_2.npy$



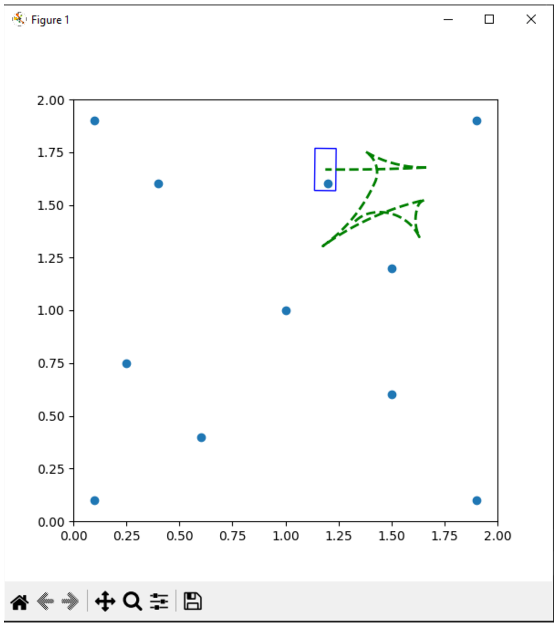
controls/$_2/_1.npy$



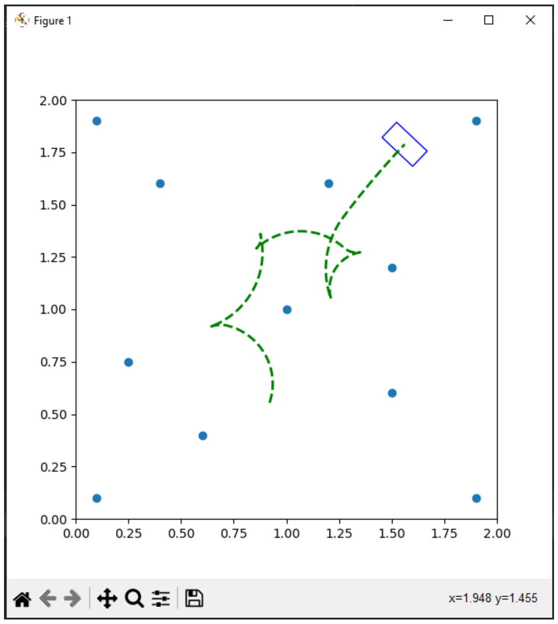
controls/$_2$/$_2.npy$



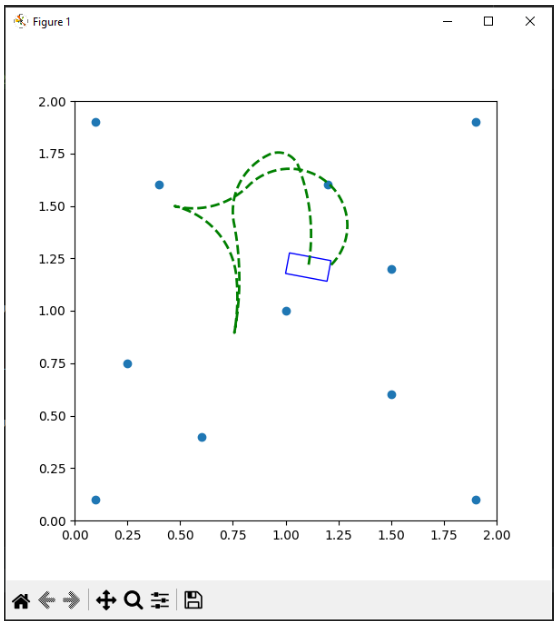
controls/$_3$/$_1.npy$



controls/$_3/_2.npy$



controls/$_4/_1.npy$



controls/$_4/_2.npy$

Problem 2.1

How My Code Works

`differential_drive_model(q, u)`

This function implements the basic kinematics of a differential drive robot. Given the current pose q (position and orientation) and control inputs u (linear velocity and angular velocity), it calculates the change in pose dq over a small time step (dt). The resulting dq represents the incremental change in position and orientation.

`landmark_sensor(ground_truth_x, ground_truth_y, ground_truth_theta, landmarks)`

This function simulates a simple landmark-based sensor model. Given the ground truth position ($ground\_truth\_x$, $ground\_truth\_y$) and orientation $ground\_truth\_theta$ of the robot, as well as a list of landmark positions (landmarks), it calculates simulated sensor measurements for each landmark. Gaussian noise is added to both the distance and angle measurements to mimic real-world sensor noise.

`Odometry(landmarks_map, measurements)`

The `Odometry` function simulates the movement of the robot based on a sequence of control inputs (`measurements`). It initializes the robot's pose with the initial location from the measurements and then iteratively applies the differential drive model to update the pose. For each step, it also calculates simulated sensor measurements using the `landmark_sensor` function and appends both the measurements and control inputs to the trajectory. The final trajectory, consisting of robot poses, measurements, and control inputs, is returned.

`Actuation(executed_controls)`

This function simulates the executed controls, similar to the `Odometry` function. It initializes the robot's pose with the initial location from the executed controls and iteratively updates the pose using the differential drive model. The trajectory of the robot's positions during the simulation is printed to the console.

`generate_odometry(controls_file_path, sigma_e_omega, sigma_e_phi)`

This function generates noisy odometry measurements based on a planned sequence of control inputs loaded from a file (`controls_file_path`). Gaussian noise is added to both linear and angular velocities to simulate sensor noise. The resulting odometry measurements, including the initial pose, are returned.

`create_Actuation(controls_file_path)`

This function generates executed control inputs with noise based on a planned sequence of control inputs loaded from a file (`controls_file_path`). Gaussian noise is added to both linear and angular velocities to simulate the actual executed controls. The resulting executed controls, including the initial pose, are returned.

`simulate(plan_file, map_file, execution_file, sensing_file, V)`

The `simulate` function serves as the main driver for the simulation. It loads the landmark map from a file (`map_file`), generates executed controls using the `create_Actuation` function, simulates actuation using the `Actuation` function, and saves the results to an execution file. It then generates odometry measurements using the `generate_odometry` function, simulates odometry using the `Odometry` function, and saves the results to a sensing file. The choice between "H" and "L" for the parameter `V` affects the variance used in generating odometry measurements.

`Command Line Interface (__main__ block)`

The script includes a command-line interface (CLI) using the `argparse` module. Users can specify the plan file, map file, execution file, and sensing file as command-line arguments. It also creates directories ”readings” and ”gts” if they don’t exist.

`File Saving`

The script uses NumPy’s `np.save` function to save the simulation results, including executed controls and odometry measurements, to specified files.

`Directory Handling`

The script checks for the existence of directories ”readings” and ”gts” and creates them if they don’t exist. These directories are used to organize and store the simulation results.

Problem 2.2

Robot Motion Simulation and Visualization Process:

`convert_structure_controls` Function:

Purpose: Initializes the robot’s position and simulates its movement based on control inputs.

Implementation: Begins with an initial position and iteratively applies control inputs using the `differential_drive_model` function to update the robot’s position. The resulting trajectory is stored and returned.

`global_landmark_positions` Function:

Purpose: Transforms local landmark positions into global coordinates based on the robot’s current position and orientation.

Implementation: Iterates over local landmarks, calculating their global positions using trigonometric functions. Returns a list of global landmark positions.

`differential_drive_model` Function:

Purpose: Models the kinematics of a differential drive robot, determining how its position changes over time given control inputs.

Implementation: Utilizes control inputs (linear velocity and angular velocity) to compute incremental changes in the robot’s position and orientation over a small time step (`dt`).

`draw_rotated_rectangle` **Function:**

Purpose: Facilitates the visualization of a rotated rectangle on a 2D plot, representing the robot.

Implementation: Uses matplotlib to draw a rectangle at a specified position and then applies a rotation transformation to represent the robot's orientation.

`separate_data` **Function:**

Purpose: Reads data from a file containing both trajectory and sensor readings, separating them into two distinct lists.

Implementation: Iterates through loaded data, appending trajectory and sensor readings to separate lists based on their indices (even or odd).

`animate_robot_trajectory` **Function:**

Purpose: Creates an animated visualization of the robot's trajectory and landmarks.

Implementation:

- **Initialization of Plot:** Creates a figure and axis using `plt.subplots()`. Sets up the initial plot with markers for landmarks, plotting the initial trajectory points, and defining plot limits.
- **Iterative Animation:** Iterates through each time step in the robot's trajectory. Updates the plot by clearing the previous plot and recreating the axis. Represents the current robot position with a rotated rectangle using `draw_rotated_rectangle`. Draws the robot's path with a red dashed line. Updates and plots trajectory points in blue, possibly representing an alternative trajectory. If the current time step is beyond the initial point, calculates and visualizes global landmark positions as red 'x' markers. Line segments connect the robot's position to each landmark.
- **Real-time Plot Updates:** Introduces a slight delay (`plt.pause(0.05)`) to create a visible pause between frames, enhancing the animation effect.
- **Final Plot Display:** After iterating through all time steps, displays the final plot using `plt.show()`.

Effect of Noise

In summary, the noise introduced in both the odometry and sensor measurements simulates the inherent uncertainties in real-world robotic systems. This noise can affect the accuracy of the robot's estimated trajectory and sensor measurements, making the simulation more realistic and applicable to practical scenarios. It does cause the robot to go a bit off path

Problem 3.1

Function Summaries

1. `global_landmark_positions(robot_x, robot_y, robot_theta, landmarks_local):`

- Computes the global positions of landmarks given the robot's global pose and local landmark positions.
- Utilizes trigonometric transformations to consider the effect of robot orientation on landmark positions.
- Returns a list of global landmark positions, incorporating the robot's pose.

2. `convert_structure_controls(controls):`

- Extracts the initial location and control sequence from a given set of controls.
- Utilizes a differential drive model, a fundamental robotics model, to simulate the robot's movement over time.
- Returns an array representing the simulated trajectory of the robot based on the provided controls.

3. `next_position(control, q):`

- Predicts the next position of the robot based on the current state (position and orientation) and a given control command.
- Implements a simple differential drive model to estimate the change in position.
- Returns the new position of the robot after applying the control command.

4. `differential_drive_model(q, u):`

- Implements the basic differential drive model for the robot's movement.
- Takes the current state q (position and orientation) and a control command u (linear and angular velocities).
- Computes the change in state (position and orientation) based on the control command.
- Returns the change in state as a result of the applied control.

5. `draw_rotated_rectangle(ax, center, width, height, angle_degrees, color='r')`**:**

- Visualizes a rotated rectangle on a given Matplotlib axes.
- Enables customization with parameters such as center coordinates, width, height, rotation angle, and color.
- Uses Matplotlib patches to draw the rotated rectangle on the specified axes.

6. `separate_data(sensing)`**:**

- Processes a sensing file containing both control commands and sensor measurements.
- Separates controls and sensor measurements into distinct lists for further analysis.
- Achieves separation by indexing even and odd elements, corresponding to controls and measurements, respectively.
- Returns two lists: one for control commands and one for sensor measurements.

7. `sort_particles_by_weight(particles)`**:**

- Sorts a list of particles based on their weights in descending order.
- Utilizes Python's built-in `sorted` function with a custom sorting key based on particle weights.
- Returns a new list of particles sorted by weight, facilitating efficient particle selection.

8. `landmark_sensor(ground_truth_x, ground_truth_y, ground_truth_theta, landmarks)`**:**

- Simulates the robot's sensor observations of landmarks given its ground truth pose.
- Computes distances and angles from the robot to each landmark using trigonometric relations.
- Returns a list of simulated landmark observations, including distance and angle.

9. `particle_filter(prior_particle, control, measurement, landmarks):`

The `particle_filter` function is a core component of the particle filter algorithm for robot localization, employing a Bayesian approach to estimate the robot's pose over time. The detailed steps are as follows:

1. **Initialization:**
    - Initialize an empty list `particles` to store the updated particles.
    - Set the variables n, $\sigma\_distance$, and $\sigma\_direction$ for later use.
2. **Prior Particle Sorting:**
    - Sort the prior particles based on their weights in descending order. This ensures that particles with higher weights are more likely to be selected during resampling.
3. **Particle Resampling:**
    - Iterate through each particle in the sorted prior set.
    - Sample a new particle from the prior set based on its weight using numpy's random choice function.
4. **Motion Model:**
    - Predict the new position of each particle by applying the control command.
    - Introduce Gaussian noise to the control command to simulate uncertainty in the robot's motion.
5. **Sensor Model:**
    - Simulate sensor observations for landmarks based on the predicted particle positions.
    - Compute the observed distances and angles to landmarks.
6. **Likelihood Calculation:**
    - For each observed landmark, calculate the likelihood of the particle's sensor observations using Gaussian probability for both distance and direction.
    - Update the joint likelihood for all observed landmarks.
7. **Weight Update:**
    - Update the weight of each particle based on the joint likelihood.
    - Add a small value (1e-100) to avoid numerical instability.

8. **Normalization:**
    - Normalize the weights of all particles to ensure that they sum up to 1.
9. **Sorting of Updated Particles:**
    - Sort the updated particles based on their weights in descending order. This prepares them for the next iteration.
10. **Return:**
    - Return the list of updated particles, each now associated with an updated weight.

10. `animate(controls, measurements, initial, landmarks_map, length, width, particles, estim_file):`

- Creates an animation to visualize the particle filter process over time.
- Plots the robot's trajectory, particles, and landmarks at each time step.
- Saves the estimated trajectory to a file for later analysis.
- Utilizes Matplotlib for plotting and animation, providing a visual understanding of the algorithm's performance.

11. `create_initial_particle(initial, n):`

- Generates an initial set of particles with uniform weights.
- Takes the initial pose and the number of particles as input.
- Initializes particles at the same pose with equal weights.
- Returns a list of particles with equal weights for the initial step.

Observations

I noticed that with more particles the closer the robot runs compared to the reading model, and this is likely due to the fact the filter has more to sample giving a much more accurate result.The noise spreads the particles.The more noise being added might be helpful if the robot is far from the reading robot.Though most of the time because we are starting from initial location the robot would almost completely accurately trace the reading robot

Problem 3.2

Implementation

Its almost the same a 3.1 except i start all the particles at random locations around the map pretty much same how i setup robot randomly in 1.2. The weight is still 1/Num particles to keep it normalized.

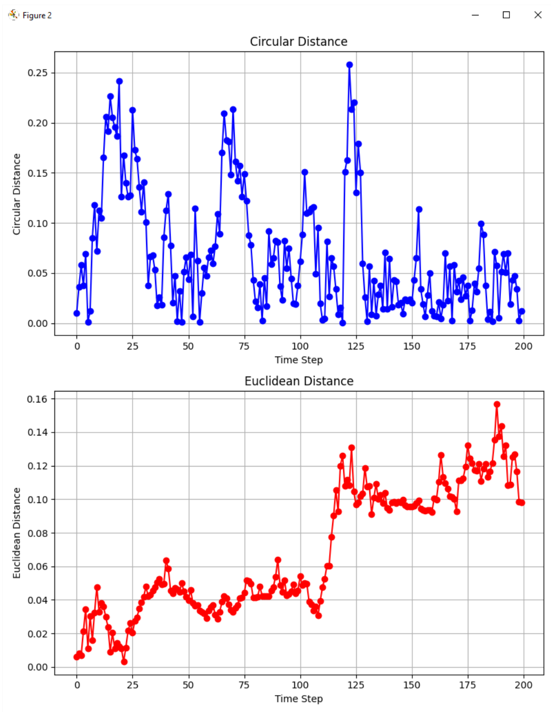
Observation

I found that that having higher amount of particles spread out caused the robot longer to find the location because on average it would stay in the middle and then it would start going toward the readings robot.I did a bunch of tests with 3001 particles, and they had trouble finding the path. Though the one with 500 had an easier time.As you can see by the videos i provided under video2.

Problem 4

Eval1 results

0_1_h_1000



1_1_h_1

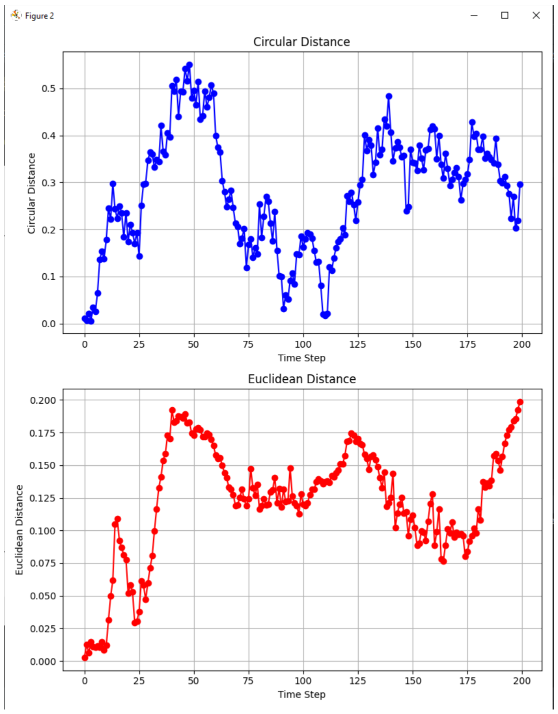


Figure 2
Circular Distance
Circular Distance
0.0
0.1
0.2
0.3
0.4
0.5
0
25
50
75
100
125
150
175
200
Time Step
Euclidean Distance
Euclidean Distance
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0
25
50
75
100
125
150
175
200
Time Step

2_1_L_10

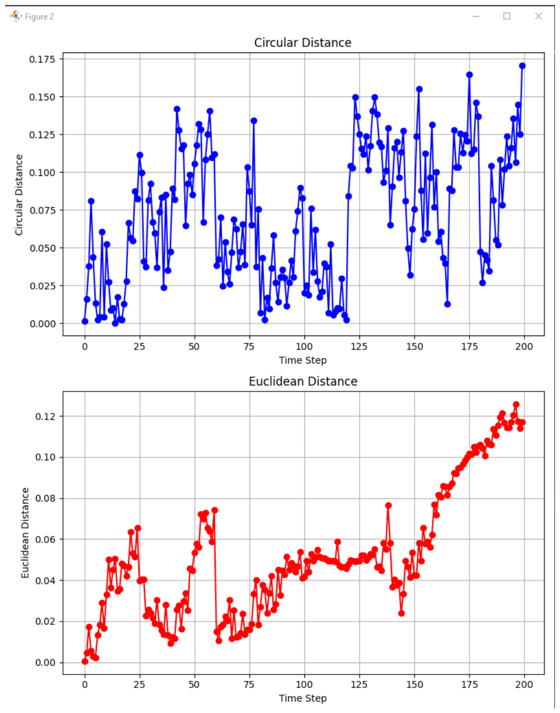


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step

3_2_L_100

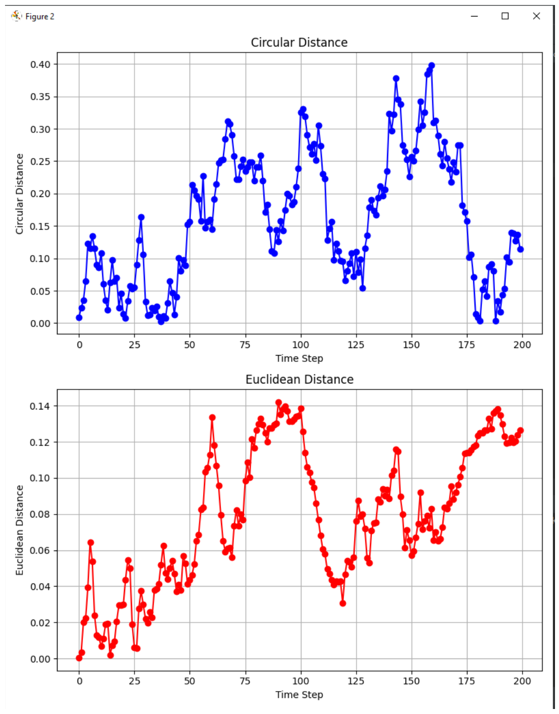
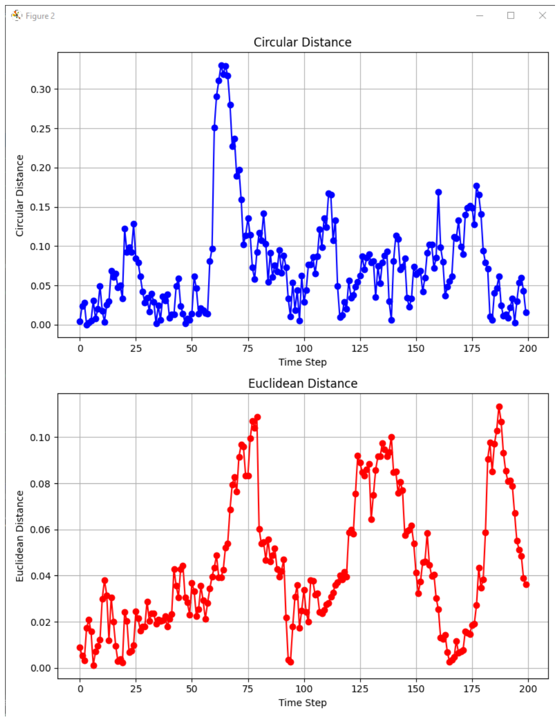


Figure 2
Circular Distance
Circular Distance
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
25
50
75
100
125
150
175
200
Time Step
Euclidean Distance
Euclidean Distance
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Time Step

4_2_L_3



Eval2 results

0_1_h_3001

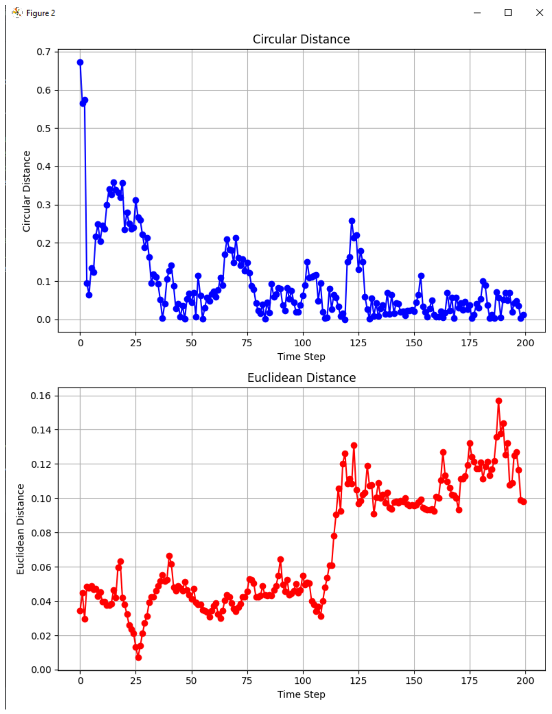


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0
25
50
75
100
125
150
175
200

1_1_h_3001

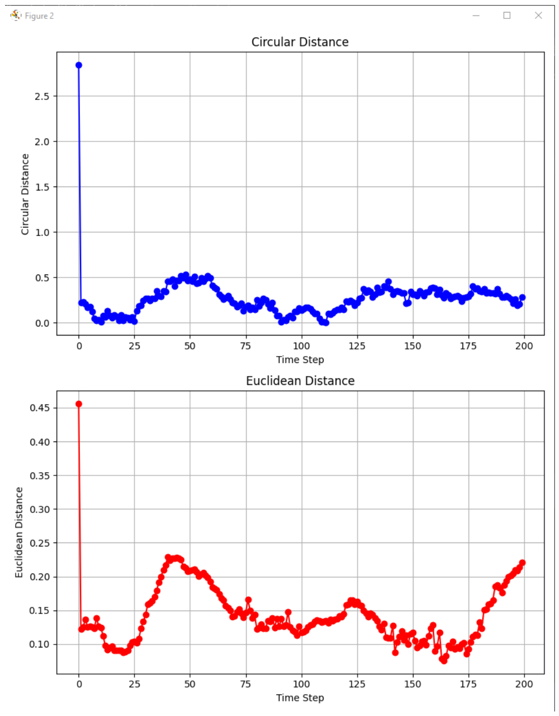


Figure 2
Circular Distance
Circular Distance
Time Step
0.0
0.5
1.0
1.5
2.0
2.5
0
25
50
75
100
125
150
175
200
Euclidean Distance
Euclidean Distance
Time Step
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45

2_1_L_3001

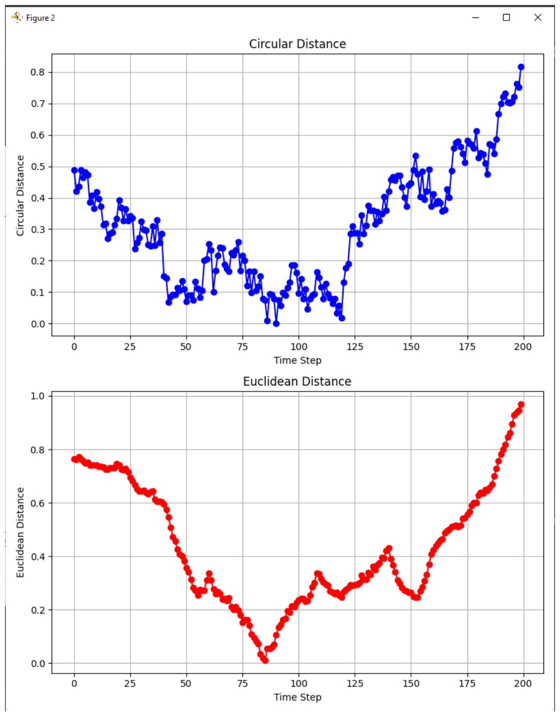


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step

3_2_L_500

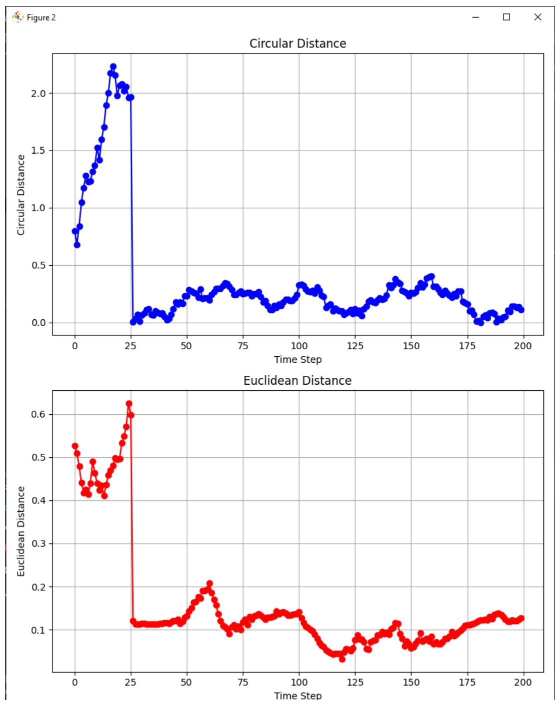
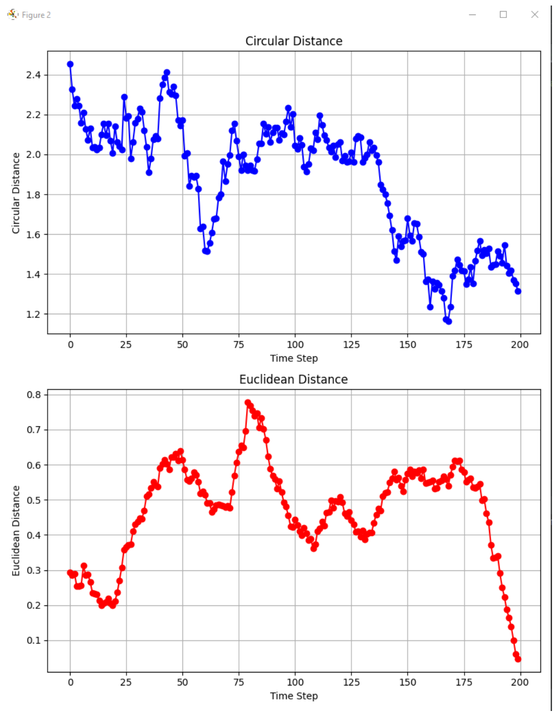


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step

4_2_L_500



Discussion

I honestly believe that the randomness kinda affected some of my results but in general In the case of low noise, the algorithm is likely to produce more accurate results, as it can distinguish the true signal from the noise with greater confidence. High noise levels can introduce uncertainty and errors in the readings, leading to a decrease in accuracy. The algorithm might struggle to differentiate between the actual measurements and noisy data. Low noise levels generally result in faster computation, as the algorithm can efficiently process cleaner data. High noise levels may require additional computational resources and time to filter out noise and generate accurate estimates. This can lead to increased runtime. Increasing the number of particles often leads to improved accuracy, as it allows the algorithm to explore a larger state space and generate a more robust estimate of the true system state. A lower number of particles may result in a less accurate representation of the system, especially in complex environments or scenarios. A higher number of particles generally requires more computational resources and time for processing. The algorithm needs to propagate and update a larger set of particles, which can lead to increased runtime. Conversely, a lower number of particles can reduce computational requirements but may sacrifice accuracy.