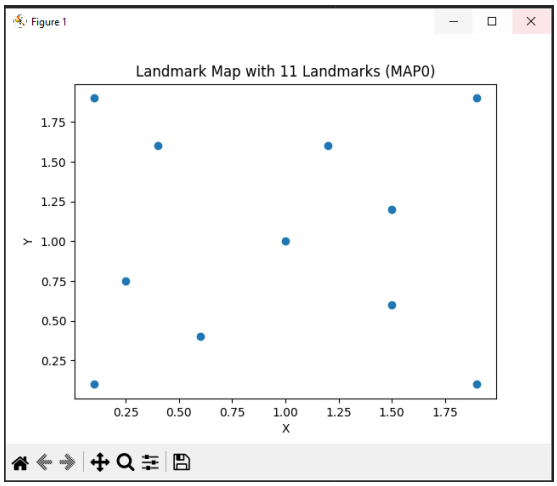
Assignment3

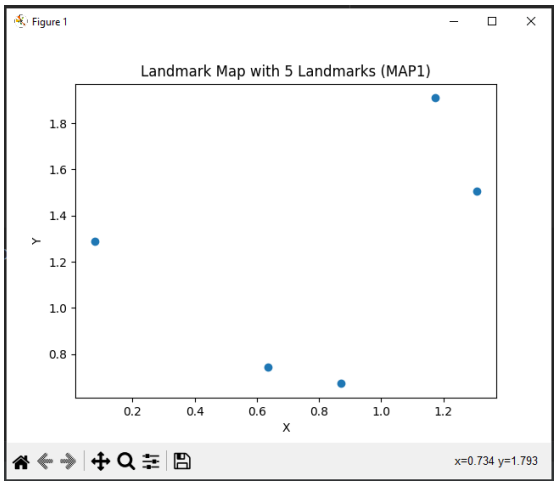
Benyamin and Sebastian

December 2023

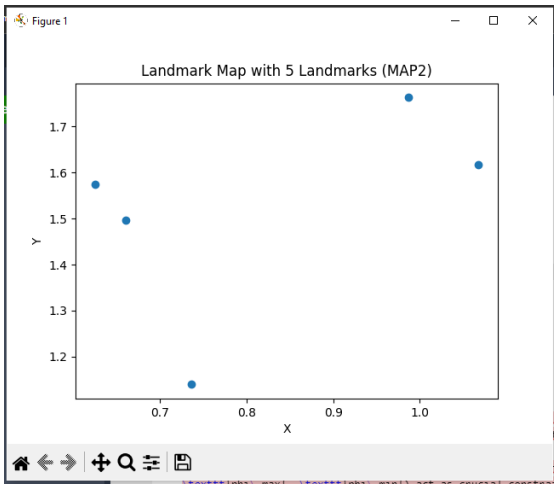
Problem 1.1



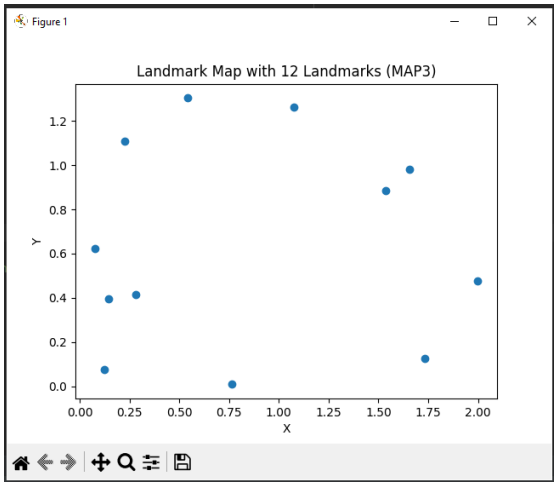
landmark_0.npy



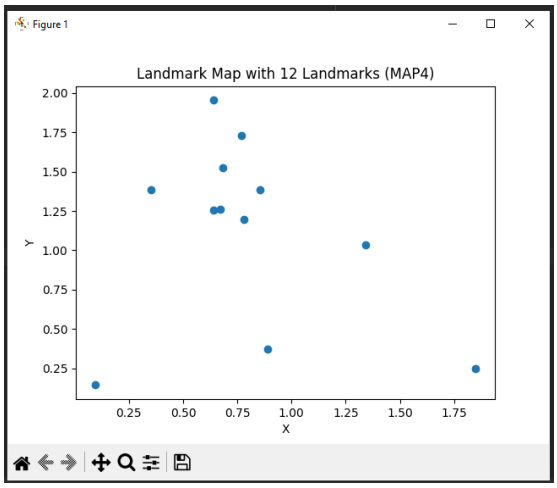
landmark_1.npy



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:

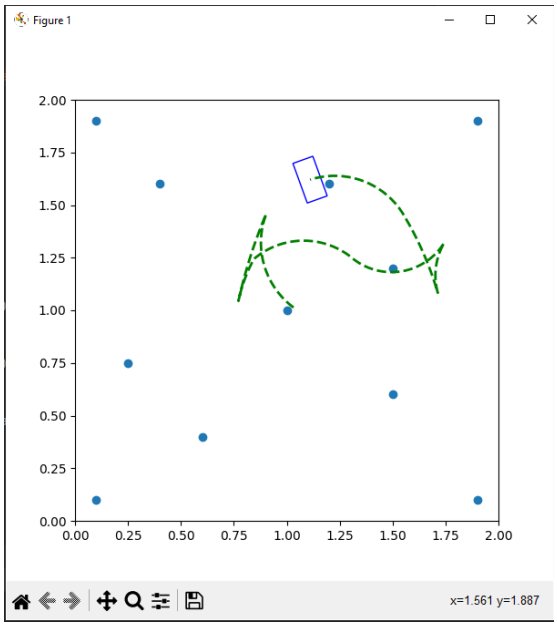
Communicating effectively, the code outputs a descriptive message to signify the occurrence of the reset, providing essential insights into the simulation's state.

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

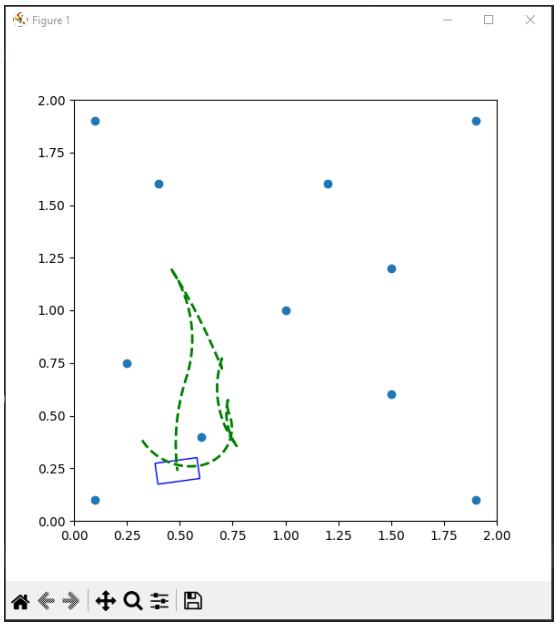
Continued Simulation:

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.

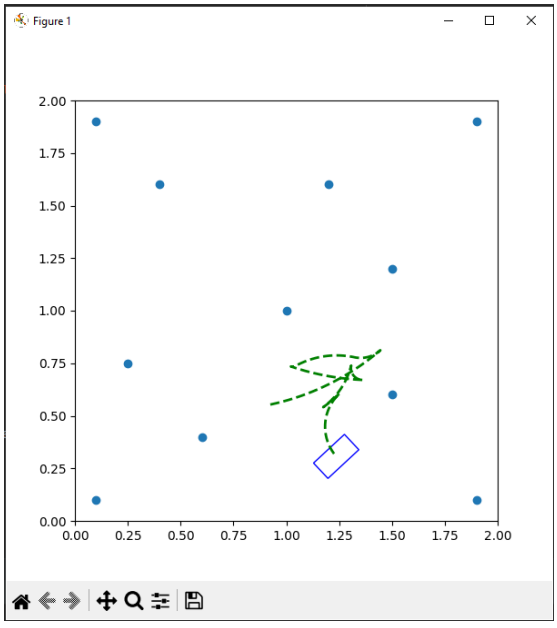
Visualizations of the 10 control sequences:



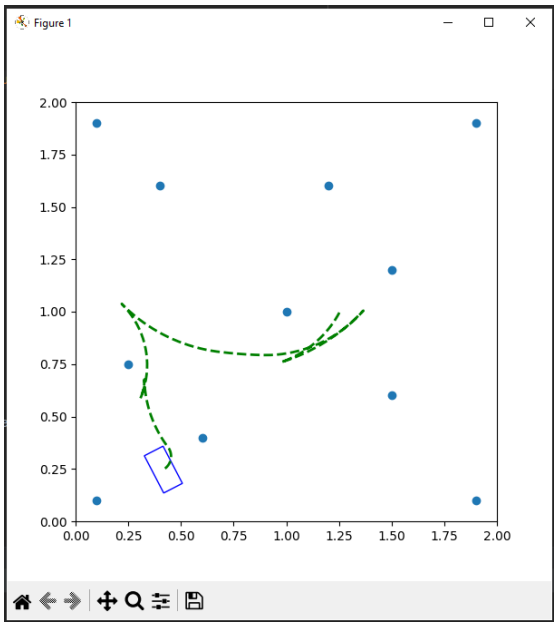
controls/$_0$/$_1$.*npy*



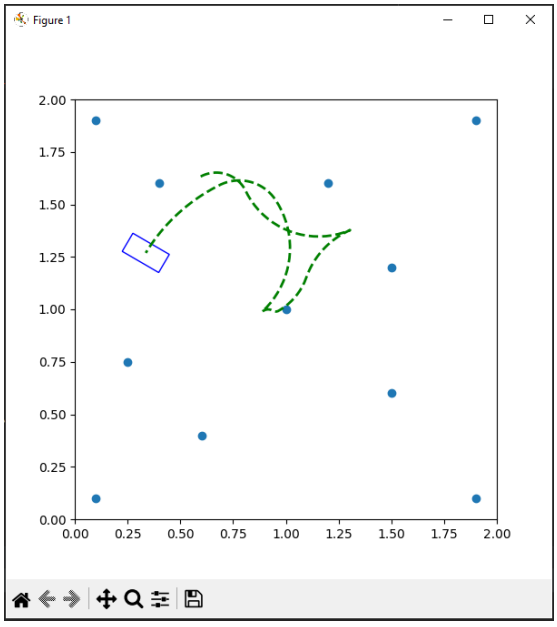
controls/0/2.npy



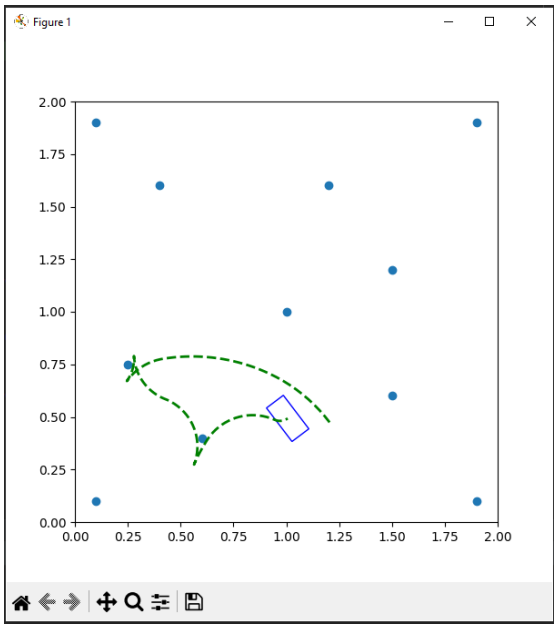
controls/1/1.npy



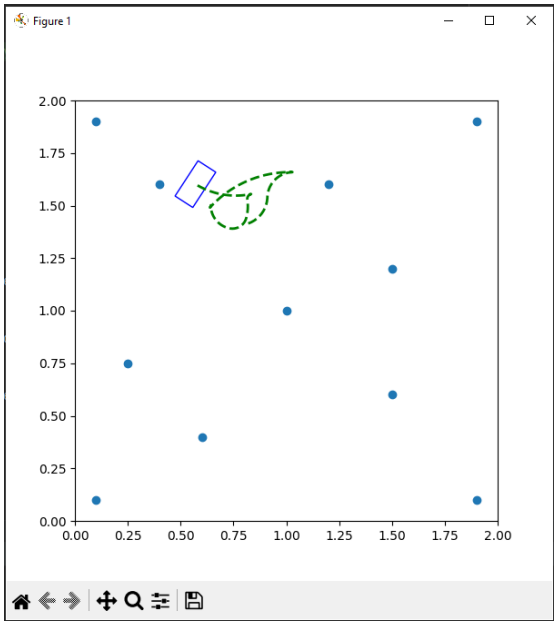
controls/$_1$/$_2$.*npy*



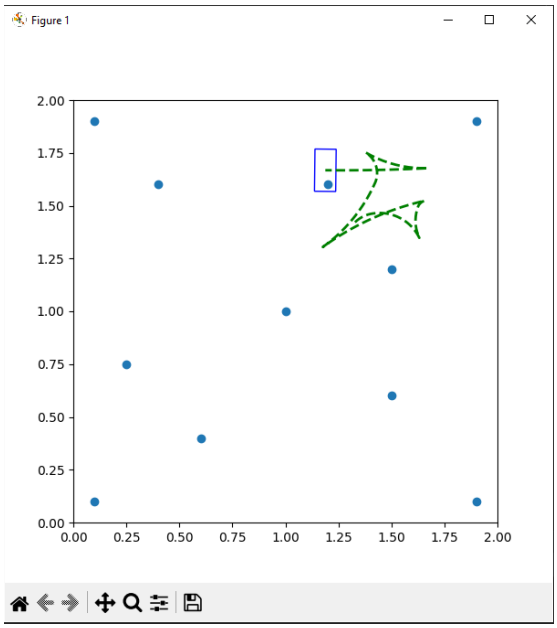
controls/$_2$/$_1$.*npy*



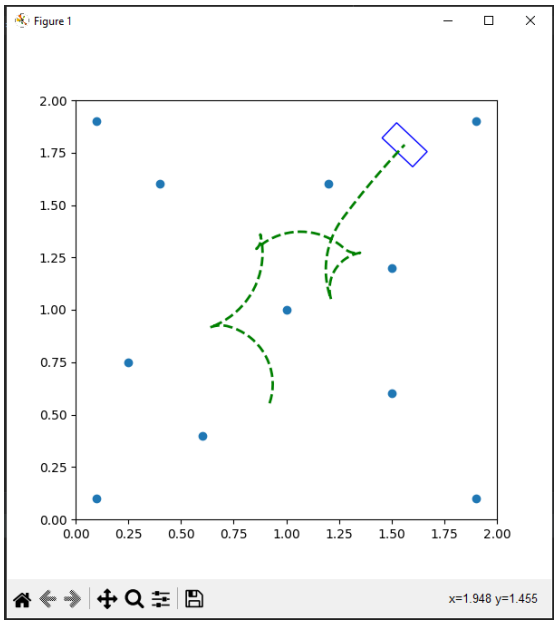
controls/2/2.npy



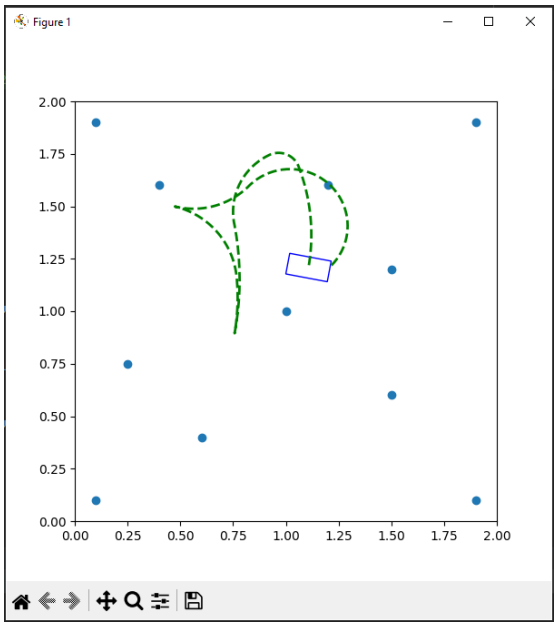
controls/3/1.npy



controls/3/2.*npy*



controls/4/1.*npy*



controls/4/2.*npy*

2.1 Detailed Function Descriptions

0.1 differential_drive_model(q, u)

- Uses numpy for element-wise operations, enhancing computational efficiency.
- Calculates the change in pose (dq) based on the provided control input (u) and the current state (q).
- The function leverages trigonometric functions (`np.cos` and `np.sin`) to efficiently handle orientation updates.

0.2 landmark_sensor(ground_truth_x, ground_truth_y, ground_truth_theta, landmarks)

- Employs vectorized operations with numpy for optimized performance.
- Iterates through each landmark and calculates the distance and angle to the robot's ground truth position.
- Uses `np.linalg.norm` for efficient Euclidean distance computation.
- Introduces noise to both distance and angle measurements using `np.random.normal`.
- Normalizes angles to ensure they fall within the range [0, 360].

0.3 generate_odometry(controls_file_path, sigma_e_omega, sigma_e_phi)

- Utilizes numpy to load a sequence of planned controls efficiently.
- Iterates through the planned controls, adding noise to both linear and angular velocities.
- Incorporates noise using `np.random.normal`.
- Appends the noisy odometry measurements, including the initial pose, to the result.

0.4 Actuation(executed_controls)

- Efficiently loads a sequence of planned controls using numpy.
- Iterates through the planned controls, introducing noise to simulate the execution of controls.
- Adds noise to both linear and angular velocities using `np.random.normal`.
- Appends the noisy executed controls, including the initial pose, to the result.

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

- Leverages numpy for loading the map of landmarks from a file efficiently.
- Calls `create_Actuation` to generate executed controls and saves them to an execution file using numpy.
- Determines the amount of noise in odometry measurements based on the value of V.
- Calls `generate_odometry` to simulate noisy odometry measurements and saves them to a sensing file using numpy.

0.6 create_Actuation(controls_file_path)

- Uses numpy for loading a sequence of planned controls.
- Iterates through the planned controls, adding noise to both linear and angular velocities to simulate control execution.
- Appends the noisy executed controls, including the initial pose, to the result.

0.7 Odometry(landmarks_map, measurements, Actuation_Model)

- Integrates odometry information and simulated sensor measurements.
- Initializes the robot's pose (q) with the ground truth from measurements.
- Iteratively applies the differential drive model and calls `landmark_sensor` to obtain simulated sensor measurements.
- The resulting trajectory contains interleaved entries of estimated poses based on odometry and simulated sensor measurements.

Problem 2.2

The animation begins by creating a `matplotlib` figure and axis (`ax`) using `plt.subplots()`. The initial plot limits are set to define the area within which the animation will take place (e.g., `plt.xlim(0, 2)` and `plt.ylim(0, 2)`).

Iterative Animation Loop:

The animation is carried out through a loop that iterates over each timestamp in the robot's trajectory. At each iteration, the following steps are performed:

Update Robot's Global Position:

The global position of the robot is updated based on the odometry information obtained from the converted control sequences using the `convert`$_{s}tructure_{c}ontrolsfunction.Theupdatedrobotpc$
$coordinate, y - coordinate, andheadingangle.$

Update Landmark Positions (if available):

`If sensor data is available (i.e., at timestamps beyond the initial state), the local landmark positions are transformed to global coordinates using the global`$_{l}andmark_{p}ositionsfunction.Thetransformedlandmarksarethenplottedontheanimation.$

Draw Robot's Body:

`The draw_rotated_rectangle function is called to draw a rotated rectangle representing the robot's body. The rectangle's position and orientation are based on the updated global position of the robot.`

Draw Trajectory Path:

`The trajectory path of the robot is visualized by plotting a line connecting the historical positions of the robot up to the current timestamp. This creates a visual trail of the robot's movement.`

Visualization of Landmark Readings:

If sensor data is available, line segments are drawn from the robot's current position to the positions of detected landmarks. This provides a visual representation of the sensor readings and the robot's perception of the environment.

Plot Update and Pause:

The plot is updated at each iteration within the loop, creating the appearance of animation. A brief pause (plt.pause(0.05)) is introduced to simulate the passage of time between frames and make the animation visually comprehensible.

Repeat Loop for Each Timestamp:

The loop continues until it has iterated through all the timestamps in the robot's trajectory, completing the animation.

Display Final Animation:

Once the animation loop is completed, the final animated plot is displayed. The animation process effectively combines the updated robot state, sensor readings, and landmark positions to provide a dynamic and informative visualization of the robot's movement and perception over time.

Impact of Noise:

The impact of noise in this simulation code is notable across various stages. The differential drive model, responsible for updating the robot's global position based on control sequences, is susceptible to noise. The simulated execution of controls in the Actuation Model introduces noise, affecting the accuracy of the robot's trajectory. Additionally, sensor measurements in the form of landmark positions are subject to noise, introducing uncertainties in the perceived environment. This noise manifests in both distance and angle measurements, influencing the accuracy of the robot's sensor readings. As a result, the animation, which visualizes the robot's trajectory and its perception of landmarks over time, reflects the cumulative impact of noise, demonstrating the inherent challenges in maintaining precision in robotic simulations, especially when faced with real-world uncertainties.

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, distance, and 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.

Observation

I honestly believe that the randomness kinda affected some of my results but in general it kept to this idea. 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.

Problem 3.2

Initialization

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

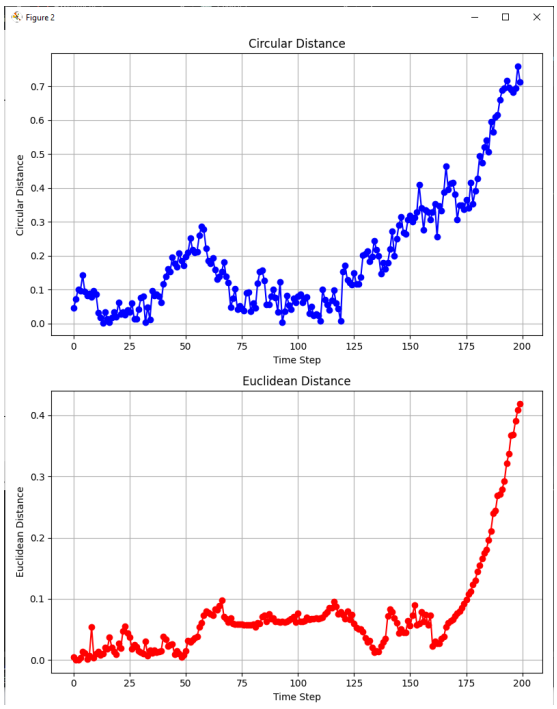
I honestly believe that the randomness kinda affected some of my results but in general it kept to this idea. In the case of low noise, the algorithm is likely to produce more accurate results, as it can distinguish the true signal from the noise with greater confidence. High noise levels can introduce 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. Due to the fact we didnt have the initial location it relied much more heavily on lower noise and higher amounts of particles. Sometimes with very low particles it didn't even seem

to get on the path in the 200 steps

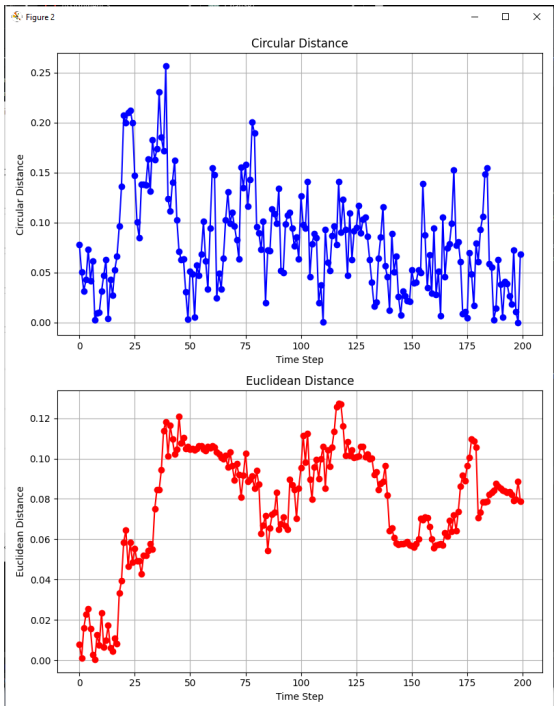
Problem 4

Experiment Data Visuals

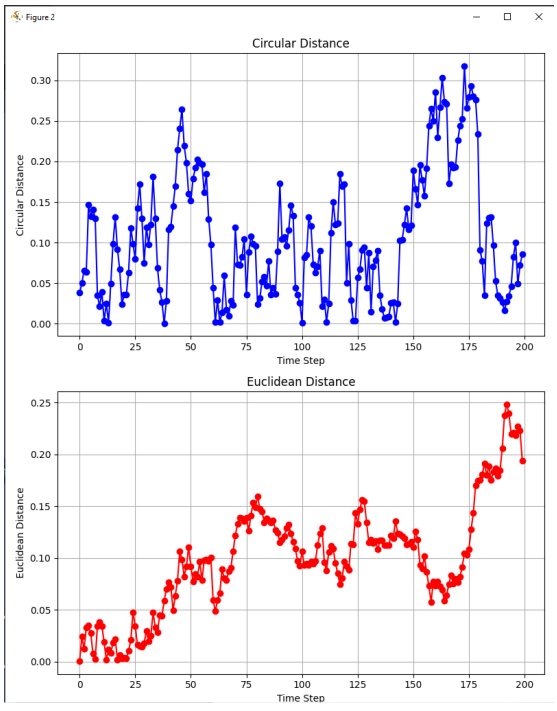
estim1_0_1_H_100



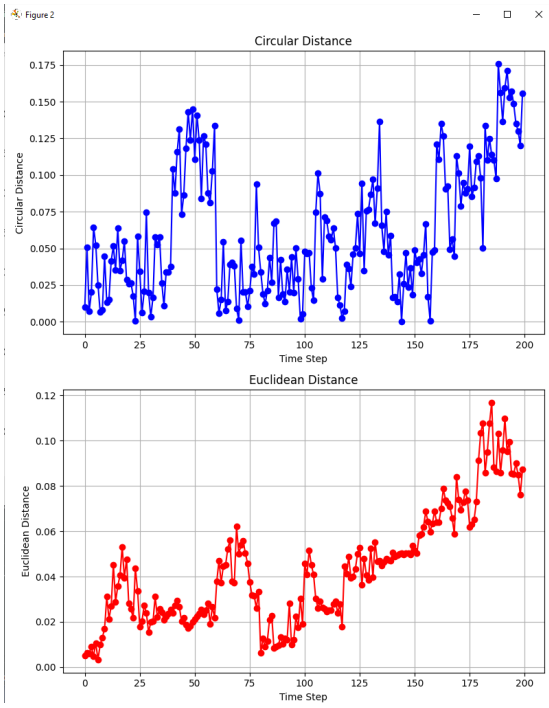
estim1_1_1_H_1000



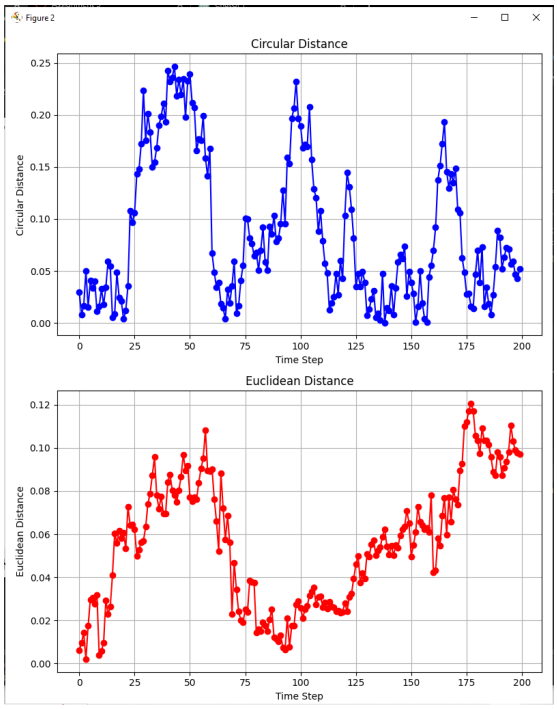
estim1_2_1_H_10



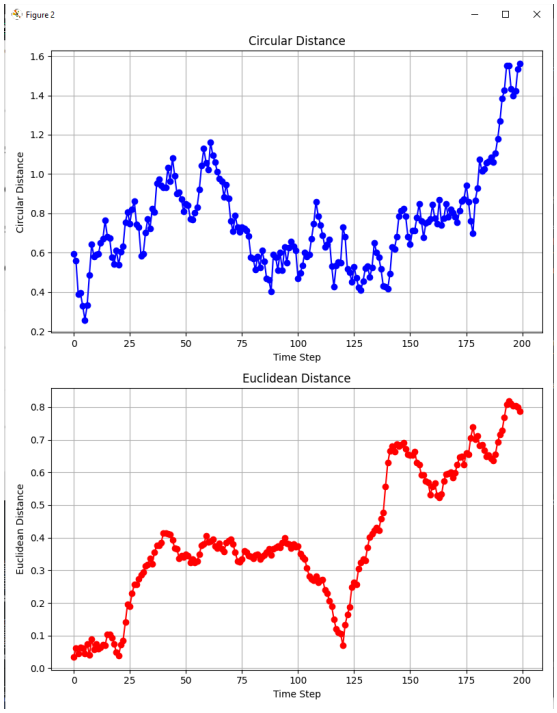
estim1_3_1_L_1



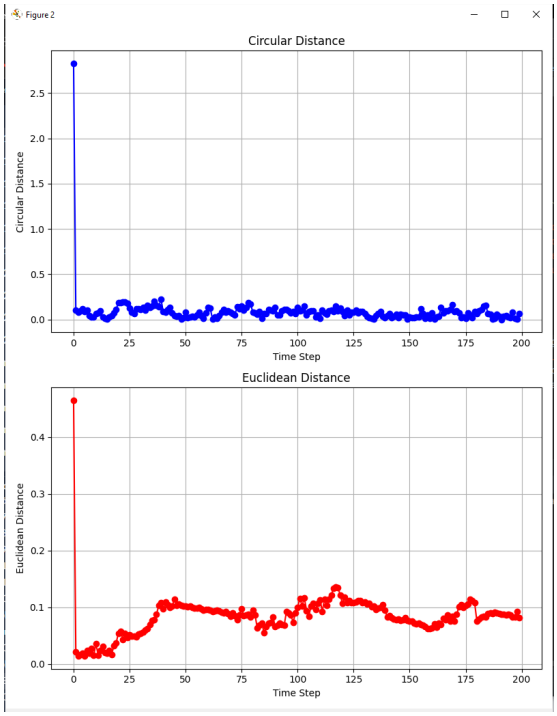
estim1_4_1_L_1000



estim2_0_1_H_500



estim2_1_1_H_5000



estim2_2_1_H_50

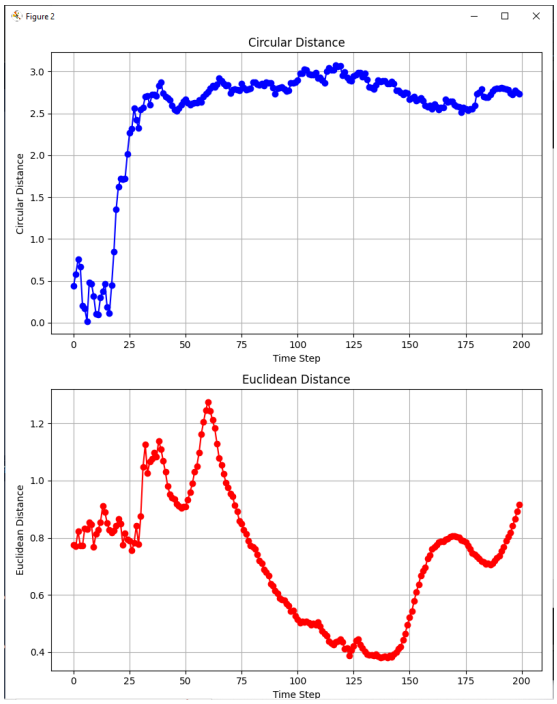


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step

estim2_3_1_L_5

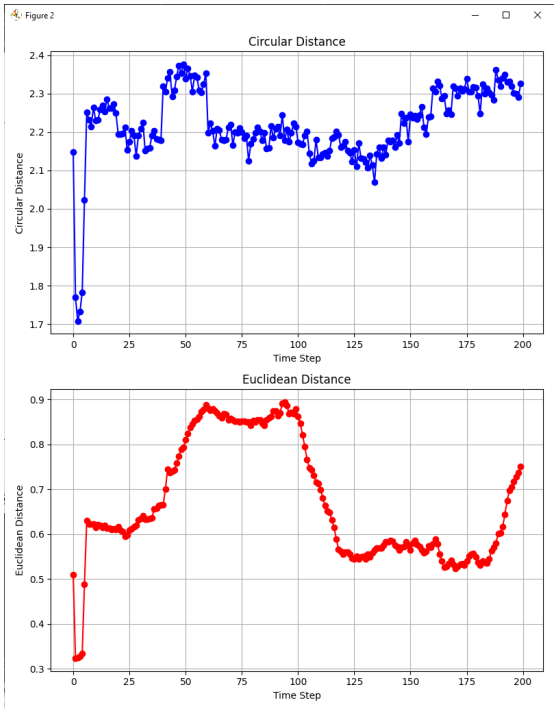
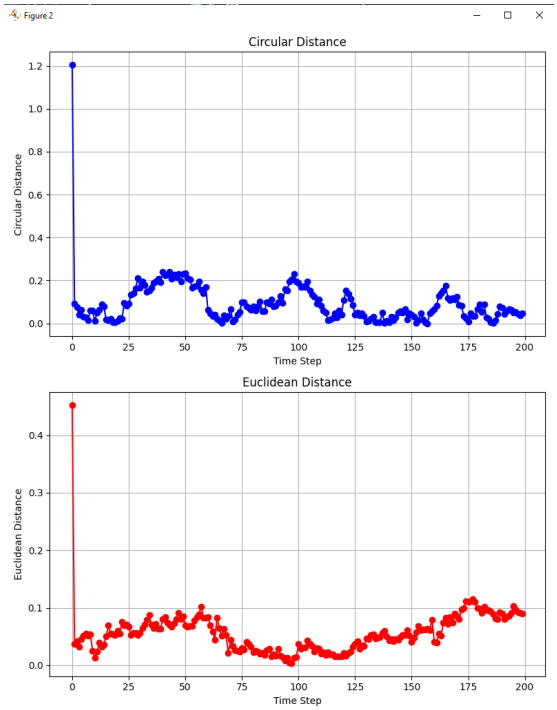


Figure 2
Circular Distance
Circular Distance
Time Step
Euclidean Distance
Euclidean Distance
Time Step

estim2_4_1_L_5000



Observation

I honestly believe that the randomness kinda affected some of my results but in general it kept to this idea. 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. Eval2 ran worse than Eval1, becuase it wasnt given a head start with being close to odometry robot.