Assignment 2 Part B

Of

CS6650W – Smart Sensing For Internet Of Things

On

Localization of IoT Devices

Prepared by

Dipendu Ghosh

CS23M509

29th November, 2024

Contents

Project Overview:

This project focuses on developing and analyzing methods for localizing Internet of Things (IoT) devices in a two-dimensional space using randomly distributed anchor and node coordinates across an MxN grid. In IoT systems, precise localization is critical for applications such as asset tracking, environmental monitoring, and smart city management. However, achieving accurate localization is challenging due to factors such as limited anchor availability, environmental noise, and variable signal quality.

The project involves placing both anchors (fixed, known-position devices) and nodes (devices with unknown locations) randomly within the MxN space and calculating the positions of the nodes based on their distances from the anchors. These distances are typically measured by evaluating signal strength or time-of-flight information from the anchors. Key components of the project include:

1. Random Placement of Anchors and Nodes: Anchors and nodes are distributed randomly within the grid, introducing variability that reflects real-world scenarios where anchor positions may not be fixed or optimal.
2. Range-Based Distance Estimation: Using estimated distances from each node to a set of anchors, we calculate the probable locations of the nodes. Techniques such as multilateration, optimization-based methods, and noise modeling will be applied to improve localization accuracy.
3. Error and Variance Analysis: By introducing controlled noise in distance measurements, we can evaluate the impact of measurement inaccuracies on localization error. This step is essential to understand the system's robustness and to optimize the placement of anchors or algorithms for better results.
4. Performance Metrics: The project assesses localization accuracy, error distribution, and variance for various grid configurations and noise conditions, generating insights into the scalability and reliability of the localization method under different scenarios.

Overall, this project aims to develop a framework for localizing IoT devices in environments with randomly placed anchors and nodes, contributing to robust and adaptable solutions in dynamic IoT environments.

Assignment Tasks:

Task 1: Dataset Generation

Implementation details:

1. **Coordinate Generation**:

- The generate_coordinates() function generates a set of unique (x, y) coordinates within a given range, ensuring no duplicates.

- The generate_true_locations_data() function uses this to create 100 sets of locations, each containing 3 anchor coordinates and 50 node coordinates. The locations are stored in a list, true_locations_data.

2. **Writing True Locations to CSV**:

- The write_true_locations_to_csv() function writes the generated true locations data (anchors and nodes) to a CSV file, true_locations.csv, with a header for anchor and node columns.

3. **Euclidean Distance Calculation**:

- The euclidean_distance() function calculates the Euclidean distance between two points using the standard formula.

- The parse_coordinate() function converts coordinate strings from the CSV into tuples of integers.

4. **Generating Pure Range Data**:

- The generate_pure_ranges_data() function computes the range (Euclidean distance) from each node to the 3 anchors for each row in the true locations data. These ranges are stored for each node in pure_ranges_data.

5. **Writing Pure Range Data to CSV**:

- The write_pure_ranges_to_csv() function writes the calculated pure range data to a CSV file, pure_ranges.csv, with a header for anchors and nodes.

6. **Generating Noisy Range Data**:

- The gaussian_noise() function generates Gaussian noise with a given mean (mu) and standard deviation (sigma).

- The generate_noisy_ranges() function adds Gaussian noise to the pure range values for different levels of noise (mu = 0.5, mu = 1, mu = 2) to simulate noisy measurements. These noisy ranges are stored in separate lists for each noise level.

7. **Writing Noisy Range Data to CSV**:

- The write_noisy_ranges_to_csv() function writes the noisy range data to CSV files for each noise level (noisy_ranges_05.csv, noisy_ranges_1.csv, noisy_ranges_2.csv), with a header for anchors and nodes.

Key Points:

- The code generates true locations, calculates ranges, adds Gaussian noise to simulate real-world inaccuracies, and writes the resulting data to CSV files.

- It provides clean, structured data for further analysis or testing in localization systems.

Output:

a. <true_locations.csv> : Contains 100 rows of anchor and node coordinates.

Content: Each row contains one set of 3 anchor coordinates and 50 unique node coordinates for 100 different anchor-node setups.

Details: (x1, y1), (x2, y2), (x3, y3) are the randomly generated anchor coordinates. (NXi, NYi) are the unique coordinates of the 50 nodes. All coordinates are constrained within the 100x100 grid.

b. <pure_ranges.csv> : Contains 100 rows of anchor coordinates and pure range values for 50 nodes.

Content: Each row contains one set of 3 anchor coordinates, followed by the distances (ranges) of 50 nodes to these anchors for 100 different setups.

Details: (R1, R2, R3) represents the Euclidean distances of a node from the 3 anchors. Distances are calculated for each node relative to all three anchors for all 100 setups.

c. <noisy_ranges_05.csv>, <noisy_ranges_1.csv>, <noisy_ranges_2.csv> : Contains 100 rows of anchor coordinates and noisy range values with $\mu = 0.5$, 1, and 2 & $\sigma = 0.1$

Content: Each file corresponds to a specific level of Gaussian noise ($\mu = 0.5$, 1, and 2). Each row contains one set of 3 anchor coordinates, followed by the noisy distances (ranges) of 50 nodes to these anchors for 100 setups.

Details: $R_{noisy} = R_{pure} + N(\mu, \sigma)$. Noise is applied independently to each range, with Gaussian parameters:

For <noisy_ranges_05.csv>: $\mu = 0.5$, $\sigma = 0.1$
For <noisy_ranges_1.csv>: $\mu = 1$, $\sigma = 0.1$
For <noisy_ranges_2.csv>: $\mu = 2$, $\sigma = 0.1$

Task 2: Range Equations

Implementation details:

1. **Euclidean Distance & RMSE Calculation**:

- The euclidean_distance() function calculates the distance between two points in 2D space, while the calculate_rmse() function computes the RMSE between estimated and actual distances.

2. **Matrix Normalization**:

- The normalize_matrix() function scales the matrix values between 0 and 1 to facilitate comparison across different matrices.

3. **Finding Global Minima**:

- The find_global_minima() function locates the position (index) of the minimum value in a matrix, which corresponds to the most likely candidate location for the node.

4. **Plotting Heatmaps**:

- The plot_heatmap() function generates heatmaps of the RMSE values, visually representing how well candidate locations match the true node location, with annotations for minima and node positions.

5. **Data Loading and Cost Matrix Calculation**:

- The script loads true locations, pure ranges, and noisy ranges from CSV files, then computes RMSE for candidate locations in a 100x100 grid for each dataset (pure and noisy ranges). After calculating the cost matrices, it normalizes and identifies the global minima.

6. **Visualization**:

- The heatmaps of normalized RMSE values are plotted with different color schemes for each noise level. Each plot highlights the minima and node coordinates.

Output:

Selected Coordinates:

Anchor Coordinates: The coordinates of the three selected anchors.

Node Coordinates: The coordinates of the chosen node.

Heatmap Observations:

The heatmap is lightest at the actual node's location, representing the global minimum cost value (in the case of <pure_ranges.csv>, cost_value = 0 at this point). The color darkens as distance increases from the node's actual position, indicating increasing cost values. For noisy data, we observe that the node location may still have the lowest cost region, but there might be additional local minima (lighter regions) scattered around due to noise.

Explanation of Observations:

The brute-force approach scans all cells in the grid, aiming to find the position with the minimum cost value. In the non-noisy case (<pure_ranges.csv>), the global minima should align with the true node coordinates. In noisy cases, the global minima may still be near the actual node location, but noise can create local minima, which may cause slight deviations in the identified node position.

Brute-Force Approach:

This task employs a brute-force method to evaluate the cost function across all cells in a 100x100 grid. This exhaustive approach aims to locate the cell with the minimum cost, which should correspond to the true node

location, especially in the non-noisy case. While this method is effective for pinpointing the global minimum, it is computationally expensive as it requires calculating the RMSE for every cell in the grid.

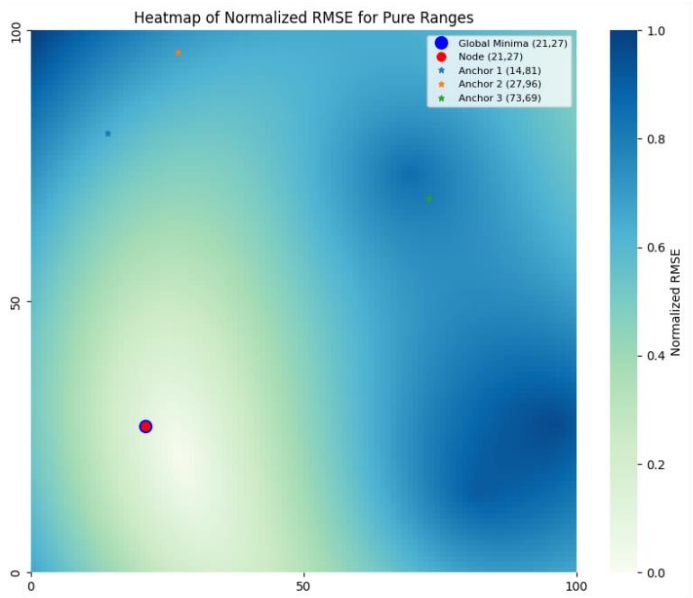
Global Minimum and Actual Node Location:

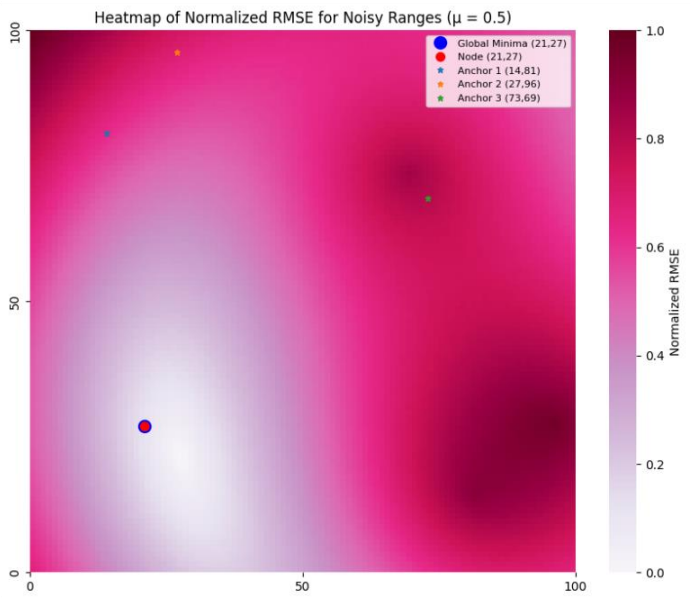
In the non-noisy scenario (<pure_ranges.csv>), the cell with the global minimum cost aligns with the actual node location, as expected. This is because, without noise, the computed ranges match precisely with the node's real position, leading to a cost value of zero at that cell. Therefore, in the non-noisy case, the brute-force approach accurately identifies the node location.

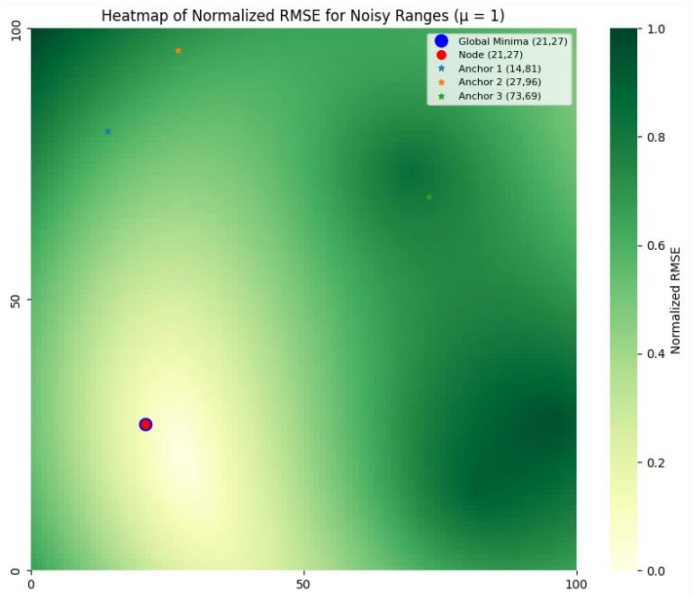
Local Minima in Noisy Cases:

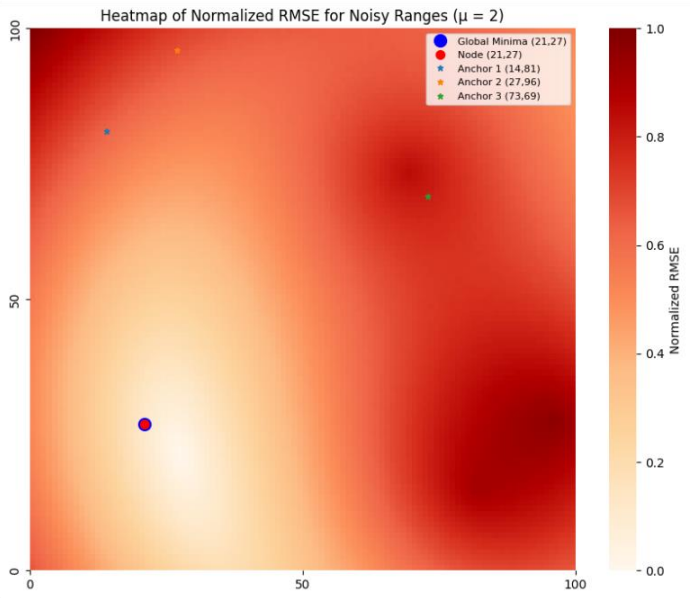
In the noisy datasets (<noisy_ranges_05.csv>, <noisy_ranges_1.csv>, and <noisy_ranges_2.csv>), noise introduces variations in range values, which slightly shifts the cost values around the true node position. Consequently, while the global minimum may still be close to the actual node location, additional local minima often appear in the heatmap. These local minima represent cells with relatively low cost values due to noise, which can mislead localization algorithms. The presence of local minima highlights the challenge of accurately localizing nodes in the presence of noise, as the brute-force search may occasionally be misled by these local variations.

```
Anchors Co-ordinates -> [(14, 81), (27, 96), (73, 69)]
Node Co-ordinates -> (21, 27)
Pure Ranges Minima Co-ordinates -> (21,27)
Noisy Ranges 0.5 Minima Co-ordinates -> (21,27)
Noisy Ranges 1 Minima Co-ordinates -> (21,27)
Noisy Ranges 2 Minima Co-ordinates -> (21,27)
```









Task 3: Trilateration.

Implementation details:

1. **Euclidean Distance Calculation:**

- **euclidean_distance(x1, y1, x2, y2)**: This function computes the Euclidean distance between two points (x1, y1) and (x2, y2) using the standard formula:

distance = {(x2 - x1)^2 + (y2 - y1)^2}^(1/2)

2. **Objective Function for Optimization:**

- **objective_function(params, anchors, ranges)**: This function calculates the residuals (errors) between predicted and actual distances for each anchor-node pair. The predicted ranges are computed based on the Euclidean distance between the node and the anchors. The goal is to minimize the residuals to estimate the node location optimally.

3. **Solving Node Location:**

- **solve_node_location(anchors, ranges)**: This function estimates the node's location by minimizing the objective function using a non-linear optimization algorithm (lmfit.minimize). It starts with an initial guess based on the average coordinates of the anchors and adjusts the node's location to minimize the distance between predicted and actual ranges.

4. **Reading Range and Anchor Data from CSV:**

- **read_ranges_from_csv(file_path)**: This function reads a CSV file containing anchors and ranges. It extracts the relevant columns (for 3 anchors and 50 nodes) and returns them as lists of tuples.

5. **Solving for All Node Locations:**

- **solve_for_all_nodes(data)**: This function iterates over all the data (sets of anchors and ranges) and solves for the location of each node using the solve_node_location function.

6. **Creating Anchor-Node Pairs:**

- **create_anchor_node_pairs(all_anchors, all_nodes)**: This function creates pairs of anchors and their corresponding solved node locations. It returns a list of dictionaries, where each dictionary contains a set of anchors and its corresponding solved node locations.

7. **Writing Solved Locations to CSV:**

- **write_solved_locations_to_csv(solved_locations_data, file_path)**: This function writes the solved node locations, along with their corresponding anchors, into a CSV file. The CSV file is structured with columns for anchor coordinates and node coordinates.

8. **Rounding Node Coordinates to Integers:**

- **round_node_coordinates_to_int(nodes)**: This function rounds the coordinates of nodes to the nearest integers. It returns the rounded node coordinates.

9. **Reading Node Locations from CSV:**

- **read_node_locations(file_path)**: This function reads the node locations from a CSV file. It assumes the node coordinates start from the 4th column and returns a list of node location tuples.

10. **Calculating Localization Error:**

- **calculate_localization_error(true_nodes, predicted_nodes)**: This function calculates the localization error (Euclidean distance) between the true node locations and the predicted node locations.

11. **Creating Anchor-Node Error Pairs:**

- **create_anchor_node_error_pairs(all_anchors, all_nodes)**: This function creates pairs of anchors and their corresponding node errors. It returns a list of dictionaries, where each dictionary contains a set of anchors and its corresponding node errors.

12. **Flattening Error Data:**

- **flatten_errors(error_data)**: This function flattens a list of error lists (for different node sets) into a single list of errors, making it easier to analyze the overall distribution of errors.

13. **Calculating Percentiles of Errors:**

- **calculate_percentiles(errors)**: This function calculates the median, 75th percentile, and 95th percentile of the error distribution. These percentiles provide insights into the typical, upper-middle, and extreme errors in the data.

14. **Reading Data:**

- The code starts by reading range and anchor data from CSV files for four datasets: pure ranges and noisy ranges with different noise levels (0.5, 1.0, 2.0).

15. **Solving Node Locations:**

- For each dataset (pure and noisy ranges), the node locations are solved using the provided range data and anchor positions.

16. **Creating Anchor-Node Pairs:**

- Anchor-node pairs are created, associating each anchor with its solved node positions for each dataset.

17. **Rounding Node Coordinates:**

- The solved node coordinates are rounded to the nearest integer to match realistic positioning scenarios.

18. **Creating Anchor-Node Pairs for Rounded Locations:**

- New anchor-node pairs are created for the rounded node locations, which are useful for analysis where discrete coordinates are required.

19. **File Paths:**

- The first block defines file paths for storing the solved and rounded node locations for different datasets: pure ranges and noisy ranges with different noise levels (0.5, 1.0, 2.0).

20. **Saving Solved Locations to CSV:**

- The next lines use the function write_solved_locations_to_csv to save the rounded solved node locations for each dataset into respective CSV files.

21. **Reading True Node Locations:**

- The true node locations are read from a file using read_node_locations.

22. **Reading Solved Node Locations and Calculating Errors:**

- The rounded solved node locations for the pure and noisy ranges datasets are read from their respective CSV files. Then, the localization error (Euclidean distance between true and predicted node locations) is calculated using the calculate_localization_error function for each dataset.

23. **Creating Anchor-Node Error Pairs:**

- Finally, for each dataset (pure and noisy ranges), anchor-node error pairs are created using the create_anchor_node_error_pairs function. This step helps associate each anchor with its respective localization error.

24. **Flattening Errors:**

- The errors for each dataset (pure and noisy ranges with different noise levels) are flattened into 1D lists for easier processing.

25. **CDF Plotting:**

- The flattened error lists are sorted, and their cumulative distribution is calculated.

- The CDF for each error distribution is plotted, and the variance of the cumulative distribution is calculated and stored.

- The plot shows the cumulative distribution of localization errors for different datasets.

26. **Percentile Calculation:**

- The median, 75th percentile, and 95th percentile are calculated for the errors in each dataset.

27. **Dataframe Creation and Display:**

- A dictionary is created to store the error type and their corresponding percentiles.

- A pandas DataFrame is created from this dictionary.

- The tabulate function is used to print the DataFrame in a grid format, showing the error statistics for each dataset.

Output:

Plot Explanation:

The plot of the Cumulative Distribution Function (CDF) for the localization errors across the four datasets (pure_ranges.csv, noisy_ranges_05.csv, noisy_ranges_1.csv, and noisy_ranges_2.csv) provides valuable insights into the effect of noise on localization accuracy:

Behavior of the CDF:

The CDF starts at 0 and monotonically increases to 1, as it represents the cumulative proportion of errors below a certain threshold.

The steepness of the curve near the origin indicates that a large fraction of errors is concentrated at lower values for the pure_ranges.csv case, reflecting high accuracy in the absence of noise.

Impact of Noise:

As noise increases (from $\mu = 0.5$ to $\mu = 2$), the curves shift progressively to the right, indicating higher localization errors. The separation between the curves represents the degradation in accuracy as the noise level increases, with the greatest separation observed for the highest noise level ($\mu = 2$).

Insights on Noise-Induced Variability:

In the pure_ranges.csv case, the plot reaches a flat plateau early, as most errors are negligible or zero. For the noisy datasets, the CDF curves rise more gradually, reflecting a broader spread of error values, particularly at higher noise levels.

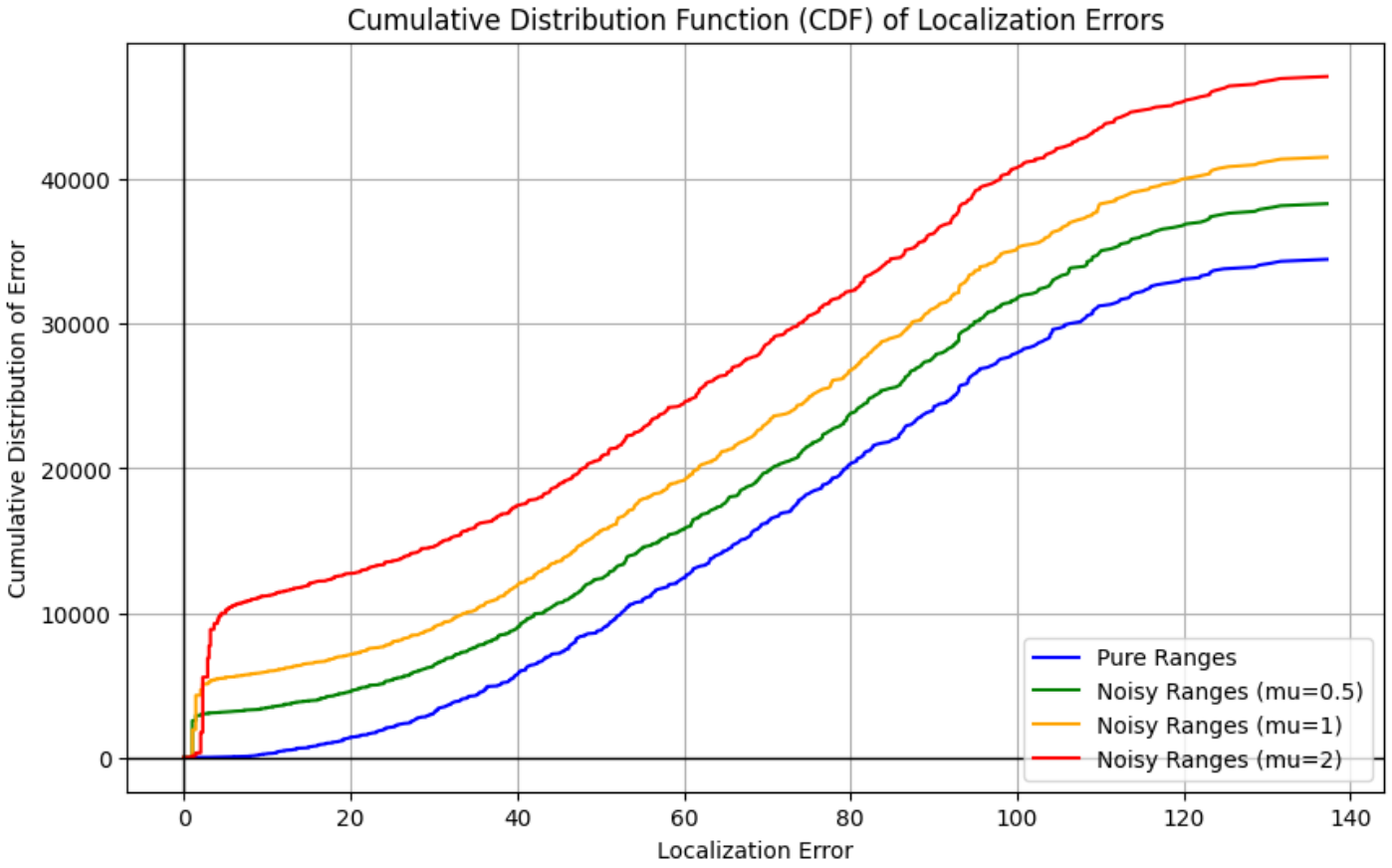
Pure Ranges:

Median and 75th percentile errors are 0, confirming that the solver perfectly estimates locations in the absence of noise. The 95th percentile error (61) suggests a few outliers due to rounding or solver limitations near grid boundaries.

Noisy Ranges:

As noise increases, the median, 75th percentile, and 95th percentile errors progressively rise, demonstrating how noise affects localization accuracy. The 75th and 95th percentile errors indicate that while most errors remain small, a few large errors occur, especially at higher noise levels.

The CDF plot highlights the increasing localization errors due to noise, with clear separation between the error distributions of different noise levels. The table reinforces this trend, with a consistent increase in median and percentile errors as noise increases. The analysis demonstrates that while the solver performs well in low-noise scenarios, its accuracy diminishes with higher noise, particularly evident in the shift of the CDF curve and the increase in percentile errors.



Error Type	Median Error	75th Percentile Error	95th Percentile Error
Pure Ranges	0	0	61
Noisy Ranges (mu=0.5)	1	1	61.7171
Noisy Ranges (mu=1)	1.41421	1.41421	61.2699
Noisy Ranges (mu=2)	2.23607	3.16228	61.8466