National Taiwan Normal University

Simultaneous Localization and Mapping

Final Project
Histogram Filter for Robot Localization

KEVIN 常凱文

11109356A

12 December 2025

Lecturer

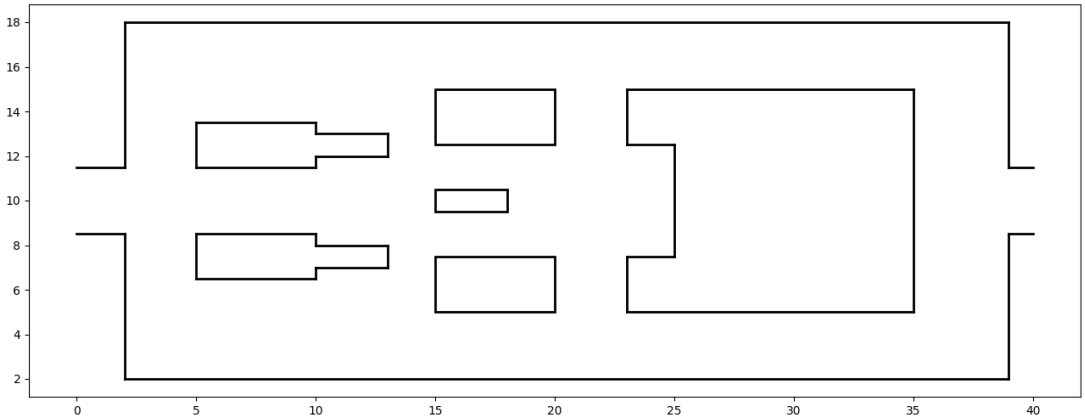
Distinguished Assistant Professor Saeed Saeedvand

I. INTRODUCTION

Histogram Filter, also known as the **Discrete Bayes Filter**, is a method of elimination and confirmation performed on a grid. It means **Histogram Filter** decomposes the approximate posteriors by a finite number of values, each roughly corresponding to a region in state space. In this Final project report, I will explain **Mobile Robot Localization** applied in a **2D Matplotlib** coordinate system (continuous spaces) by **the Discretization** using a **Histogram Filter**. The whole idea behind this project was robot estimates its position by dividing the environment into grid cells, updating these probabilities based on sensor LiDAR and motion commands.

II. IMPLEMENTATION

a) Environment & Discretization



Based on the **SLAM CH10 PDF**, I drew the exact same map as in the example given by the Professor. Inside environment.py, I wrote the environment by using line-by-line ax.plot and Rectangle.

$$\text{range}(X_t) = x_{1,t} \cup x_{2,t} \cup \ldots x_{k,t}$$

\

This formula explains how to break an infinite world into a finite number of grids. The total area where the robot can be range(Xt) is made up of the union (U) of many small individual regions. def init_grid_map() is where the implementation of this formula came, I wrote the calculation of how many regions (k) I need to cover the whole world and put it inside a 2D array self.data.

b) Probability Initialization

$$p(x_t) = \frac{p_{k,t}}{|x_{k,t}|} \qquad p(x_t) = \frac{p_{k,t}}{|x_{k,t}|} = \frac{1}{25} = 0.04$$

The idea in this formula refers to my code, which is that we put a probability value for each grid that we have defined. In the beginning, the robot has no idea where it is. Let's say we have a 25-grid world, then the probability is 1/25, which means the robot could be everywhere in the grid. This is exactly what I wrote in def init_grid_map().

c) Prediction

$$\overline{\text{bel}}(x_t) = \int p(x_t \mid x_{t-1}, u_t)\, \text{bel}(x_{t-1})\, dx$$

Prediction Step, or often **called Motion Update, Byesian Filter Cycle** means calculate the new belief based on the old belief. Inside function def histogram_motion_update(), calculate the robot's current location, divide the current location by the grid size and check if the current location exceeds the grid size (meaning it is a big move). I call the def map_shift() function to predict the new location.

$$\overline{\text{bel}}(x_t) = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{t-1}^{j}\, p(x^i \mid x^j, u_t)\, \text{bel}(x_{t-1})$$

def map_shift() did exactly what in this formula, in every grid we take the old formula, and then update it with the new predicted formula.

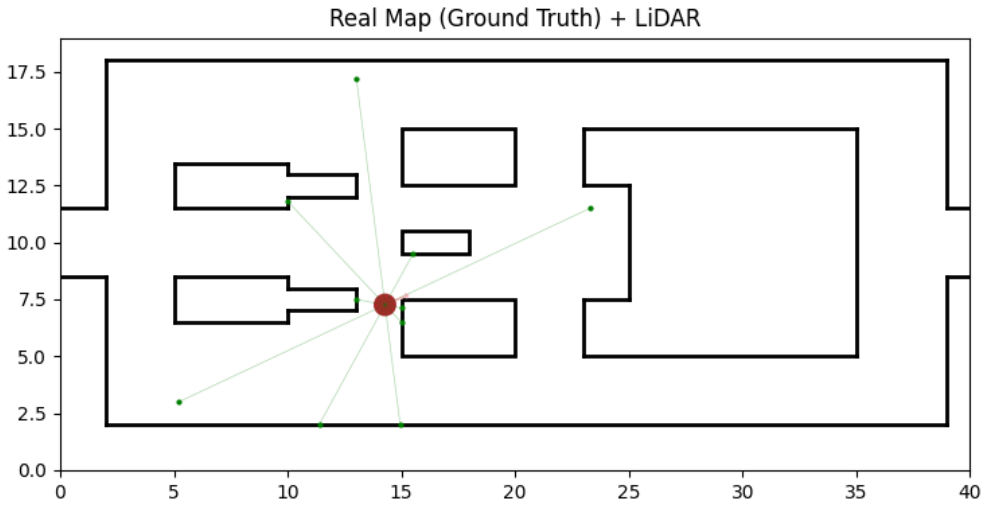
d) Correction

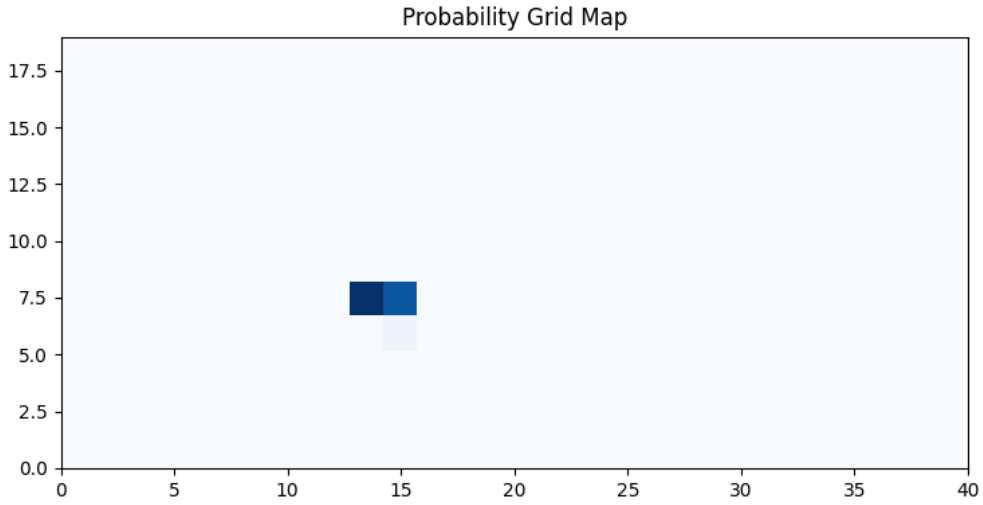
$$bel(x_t) = \eta p(o_t \mid x_t)\, \overline{\text{bel}}(x_t)$$

Correction Step, or often called **Observation Update**. This is the moment when the robot uses its sensor LiDAR, sees an obstacle, and updates its internal map. This mobile robot contains 10 LiDAR(can add more, but computationally expensive) in variable LIDAR_NUM_RAY that will be received by def lidar_scan(), and then this function loops through 360 degrees of the robot and puts the LiDAR in a certain interval. def cast_ray() function loop through every wall written in the environment.py file and check if the LiDAR hits the walls? And yes, we call the def line_intersection function, which takes two line segments (the ray and the wall), solves a linear equation to see if they cross, and returns the distance to the wall. After that, back to the def observation_update() function with those distance values (z), this is where the computational expense and it makes my visual laggy, because we take those returned values and check them against the entire map. In the def observation_update() function, it enters a triple loop to check every grid with each wall, and then recalculates the grid cell in the world, and then compares the measured_dist with expected_dist and checks the probability, and then rewards the self.data.

III. FINAL RESULT

As I mentioned, the result is really laggy, but still able to control. The Localization result is really accurate, and the robot can know its current location, although sometimes the calculation is not perfect and can be miss quite far, but mostly are correct.





IV. REFERENCE

1. **GitHub Python Robotics by Atsushi Sakai**
   https://github.com/AtsushiSakai/PythonRobotics/blob/master/Localization/histogram_filter/histogram_filter.py I got reference from this file and then modified it to fit my project case.
2. **SLAM CHAPTER 9 PDF by Assistant Professor Saeed Saeedvand**
   Each formula explanation that I mentioned in this report is based on here, and my understanding of Histogram Filter for Robot Localization is learned from here.
3. **GitHub Copilot, CLAUDE, and GEMINI**
   I use these AI tools to help with my learning and debug errors