Particle Filter Simultaneous Localization and Mapping with Texture Mapping

Chhabra Rachit (rachhabra@ucsd.edu)
Department of Mechanical and Aerospace Engineering, University of California San Diego

I. INTRODUCTION

The goal of this project is to create a particle filter simultaneous localization and mapping model using data from an autonomous vehicle. SLAM, as commonly known in the industry, is a mathematical problem of building a map or an unknown environment, and at the same time track the position of an agent (a vehicle in this problem) in this space. SLAM is used in a variety of applications such as autonomous vehicles, virtual and augmented reality applications, etc. In this project, using the data provided first dead reckoning is done using one particle, and no added noise to a differential drive model. Next, using Light Detection and Ranging (LiDAR) data provided, and scan grid correlation observation model, SLAM is carried out, and an Occupancy Grid Map is constructed for multiple particles and added gaussian noise to the model. Finally, using the stereo camera model, and given data from the two cameras on the vehicle, a textured map is created of the environment of the vehicle.

II. PROBLEM FORMULATION

We are provided with the following data from the vehicle:

(i) Left and Right wheel encoder data.
(ii) LiDAR data from a 2D LiDAR scanner mounted on the vehicle
(iii) Roll, Pitch and Yaw data from the FOG sensor installed in the vehicle.
(iv) Stereo images from the two cameras (mounted on the front left and front right side of the vehicle)
(v) The parameters for all of the above sensors.
(vi) Transformation matrices to compute a pose from one frame to another (LiDAR, Stereo, Vehicle, World).

Using the Initial Particle Set $\mu_{0|0}^{(k)} = (0,0,0)^T$ with k particles and particle weights $\alpha_{0|0}^{(k)} = \frac{1}{N}$ we need to estimate the:

(i) Position of vehicle at each time step τ using the differential drive model -

$$x_{t+1} = f(x_t, u_t) + Noise$$

where x_t is the position of a particle in $\mu^{(k)}$ at time t, u_t is the control being applied on the particle (velocity from the encoder and the angular velocity from the gyro sensor).

(ii) Map of the surrounding environment using the LiDAR scan from the sensor and the position of the vehicle at every timestamp. Given sensor observations z, we need to generate the map m, and estimate the probability over time:

$$p(m|z_{0:t}, x_{0:t}) = \prod_i p(m_i|z_{0:t}, x_{0:t})$$

The above two points need to be done simultaneously. The joint distribution can be written as following and using Bayesian Inference the problem of SLAM can be solved.

$$p(x_{0:t}, m, z_{0:t}, u_{0:t-1}) = p_0(x_0, m) \prod_{t=0}^{t} p_h(z_i|x_t, m) \prod_{t=1}^{t} p_f(x_t|x_{t-1}, u_{t-1})$$

In the second part, given two perspective camera's stereo images, z_t, the camera parameters f, s_u, s_v v_L, u_L; the occupancy grid map m, the particle trajectory x_t we need to create a texture map on the grid map corresponding to the road color of the environment.

III. TECHNICAL APPROACH

In this section, we discuss the methods used in constructing the different parts of the project, LiDAR Scan Visualization, Dead-reckoning, Particle Filter SLAM and Texture Mapping.

LiDAR Scan Visualization:

The first step if to visualize the given data. We start with the LiDAR data which is provides us a distance reading with a 190˚ (-5˚ to 185˚, step of 0.666˚) field of view at every timestamp. This data is in the LiDAR frame of reference. We first plot this to visualize it in the LiDAR frame and then the data is transformed to the world frame using the Transformation Matrices, Vehicle to LiDAR (T_{VL}) and Vehicle to World (T_{VW}, calculated in the next step) given to us/calculated by us.

Dead Reckoning:

In this step, we use the Differential Drive model (stated in Section II) as the motion model to estimate the position of the vehicle. Euler discretization over time interval of length τ is considered and only the encoder clicks data from the encoder and the yaw angle from the FOG sensor is used. We first calculate the velocity of the vehicle with wheel diameter d, encoder count z resolution of the encoder res, at each time stamp using the formula:

$$v_t = \frac{\pi dz}{res\ \tau}$$

While performing dead-reckoning, only one particle is used, and no noise has been added to the model or the sensor data. Then the position can be estimated using the equation,

$$x_{t+1} = x_t + \tau \begin{bmatrix} v_t \cos(\theta_t) \\ v_t \sin(\theta_t) \\ \omega_t \end{bmatrix}$$

where θ_t is the yaw angle ω_t is the angular velocity of the vehicle at a time t.

We can plot this data to find out and look at the overall trajectory of the vehicle.

Particle Filter SLAM:

Here, we consider a motion model and an observation model simultaneously, use encoder data to the predict the motion of the vehicle, observe our environment using the LiDAR data, map correlations between the current predicted position of the vehicle using the latest LiDAR values from the scanner, and then update the position of the vehicle. We also resample if required. Now, we will go into depth for each of the above step and look into the assumptions, steps and computations performed and their reasoning.

1. Predict

First step for our SLAM model is to predict the position of our vehicle at time t = 1(let this be the second time step, after t = 0). We consider N = 10 particles to start with. Using our motion model described in Dead-reckoning step, we can estimate the position of each particle at t = 1. Currently each particle trajectory is the same, as all have same values. Thus, a key step in SLAM to add noise to each particle in the model. Gaussian Noise was added to the sensor reading (Encoder and FOG) of each particle rather than to the final predicted particle as usually, sensors and electronic signals are the most susceptible to external noise. The noise was added relative to the value of the data. The mean μ for both the sensors noise was 0 and standard deviation σ for was 10% relative to the encoder and yaw values.

There was a need in the project to synchronize the data to get the most accurate model. So, for this step, the LiDAR and FOG data were synchronized according to their timestamps.

As we are adding the x and y position of the vehicle at every time step, the final trajectory we obtain is in the World Frame.

2. Update

After completing the predict step for one timestep, we have the predicted positions of the N particles, we need to update the particle's position. The motive of this step is to determine the Most Likely particle out of N particles based on the LiDAR reading at the next time interval. We first initialize the weights of each particle as $\frac{1}{N}$. The weights define the likelihood of the particle to be the most likely particle. So, greater the weight, greater its likelihood.

(a) As the data is asynchronous, the first step is to find the lidar reading nearest to the current predict step timestamp.

(b) We then need to filter the LiDAR data. As there were a lot a zero (LiDAR gives 0 if the object is more than 80m away), the points closer than 0.5m and father than 60m were filtered out (We do not require far points).

(c) Next, as the data is in LiDAR frame, we convert that into world frame, using consecutive transformations from LiDAR to vehicle and then vehicle to world frame.
To calculate the later, the predicted value of each particle and its corresponding yaw angle is used to create the transformation matrix as:

$$T_{VW} = \begin{bmatrix} R & \boldsymbol{p} \\ 0^T & 1 \end{bmatrix}$$

where R is the rotation of θ° about z axis, and $\boldsymbol{p} = [x\_pos, y\_pos, z\_pos]^T$, is the position of LiDAR points in vehicle frame.

(d) Now we have N particles and need to figure out which one is the most likely particle. This is done by the MapCorrelation function. This function takes in 5 arguments such as the map, two 2D arrays defining the steps in height and width of the map, and two 9 x 9 grids which represents the LiDAR point and its surrounding we are looking at. The function returns a grid of correlation values, and we consider the maximum values out of all. Similarly, we find this value for all the particles. This gives us a max correlation value of each of the particle, let $c^{(k)}$, and the updated $\alpha^{(k)}$ can be computed as

$$\alpha^{(k)}_{updated} = norm\,(\alpha^{(k)} \text{ x } c^{(k)})$$

Now we can find the best particle using the updated weights. The particle with the maximum weight is the most likely particle and then is used to do further computations.

(e) The LiDAR values of the best particle estimated above is fed into the bresenham2D function. This function essentially gives us all the cells in the map between the given starting and end point. These points are then used to update the Occupancy Grid Map. The map is updated using the following algorithm.
- if a cell is occupied, the log odds of the cell is decreased by log (64). This is capped to a value of -100.
- if a cell is unoccupied, the log odds of the cell is increased by log (9). This is capped to a value of 100.

The change in log odds is more in occupied cell depicts more confidence in LiDAR readings.

(f) In theory the update step should be done after every predict step. But due to the small timestep (0.01s), the update is done every 5th iteration.

3. Resampling

This is an important step to correct the weights at times when the number of effective particles, N_{eff} is less a threshold limit $N_{threshold}$, i.e., number of hypotheses we have gets really small. Here stratified resampling is done to populate all the particle near the most likely particle, as it is optimal in terms of variance. It guarantees that particles with large weights appear at least once and those with small weights - at most once.

$$N_{eff} = \frac{1}{\sum_{k=1}^{N}(\alpha^{(k)}_{t|t})^2},\ N_{threshold} = 20\%\ of\ N$$

If at any time, the $N_{eff} < N_{threshold}$, resampling is done, and the weights are reset to $\frac{1}{N}$.

After the loop has been run for all the time steps, the final map is passed through a function which assists in increasing the distinction between the occupied and unoccupied cells.
- The cells with value 0.5 will keep their values(neither occupied nor unoccupied, or outside our scope).
- cells with value > -10 are set to 0 and cells with value <=10 are set to 1.

Using the above algorithm, we are able to see a clear difference between the occupied and unoccupied cells.

Texture Mapping:

We can also use the images from the stereo to create a textured map of the environment with RGB values at each cell in the occupancy grid map. This is done using the following steps.

(a) Not all the left and right stereo images given were in sync and thus had to be pre-processed before using. The timestamps were matched for all the images, and a 2D array was maintained for all corresponding images.

(b) Disparity image is calculated using the function compute_stereo(). It takes in the left and right image at a particular time step and give out the disparity image. This image has value of disparity at every pixel of the left image.

(c) Then using disparity and the parameters provided, the coordinates of each point in vehicle frame were calculated using the equations:

$$z = \frac{fs_u}{d}$$
$$x = \frac{(u_L - c_u)}{fs_u} z$$
$$y = \frac{(v_L - c_v)}{fs_v} z$$

where x, y and z are the coordinates of each stereo image point in vehicle frame, f, s_u, s_v are the given intrinsic parameters of the camera, v_L, u_L are the pixel number image of the .

Using these we calculated the coordinates in vehicle frame using:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = {}_oR_r R^\top (\mathbf{m} - \mathbf{p})$$

where the ${}_oR_rR^T$ rotation matrix from optical frame to vehicle can be calculated using taking the inverse of the given matrix, and **m** is the coordinates in vehicle frame and **p** is the position transformation vector.

Now, similar to the transformation in update step, we can transform these coordinates from vehicle to world frame.

(d) Finally, to create a texture, we project the colored points from the left camera pixel value onto the grid. Only those points are considered who lie on/near the plane we require, i.e., the road. So, the height constraint is put as:

(e) $-0.5 < height < 0.5$

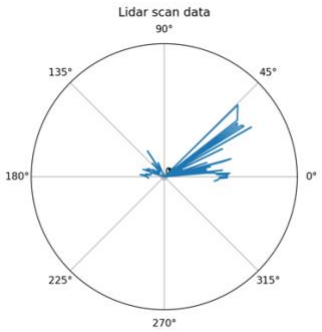
Finally, the textured image is converted to type int so that all the float values (0 to 1) get converted to RGB values.

IV. RESULTS

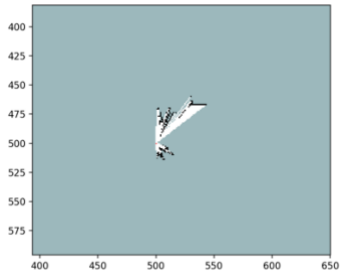
After we have used the SLAM algorithm and done texture mapping, here we discuss the results of the all the four parts mentioned in the Introduction, and how the algorithm performed on the real-life vehicle data.

LiDAR Scan Visualization

We can see using the LiDAR points in polar coordinates (in LiDAR frame) and in world coordinates as well after we have passed it through the bresenham function and lidar plotted the rays onto the occupancy grid map for first update step.



Lidar in polar coordinates



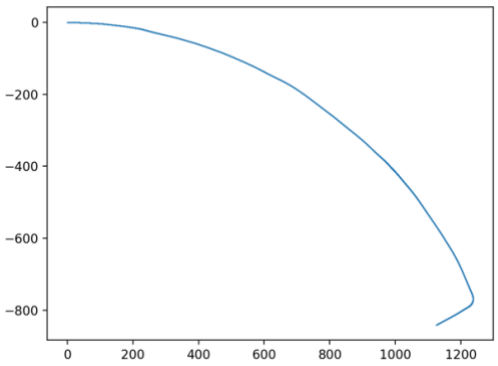
Lidar in world coordinates

For analysis of the following parts, the output was taken at four timestamps, so as to understand the development of the trajectory and mapping with time.

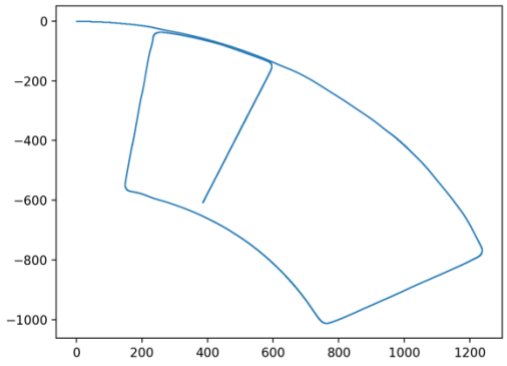
Dead Reckoning

Dead reckoning is a crucial part of the project, it helps us to get the grid size for the environment for our problem.

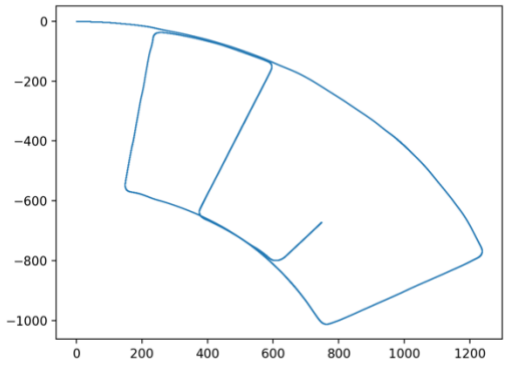
The path of the particle at different time stamps can be visualized by the following images:



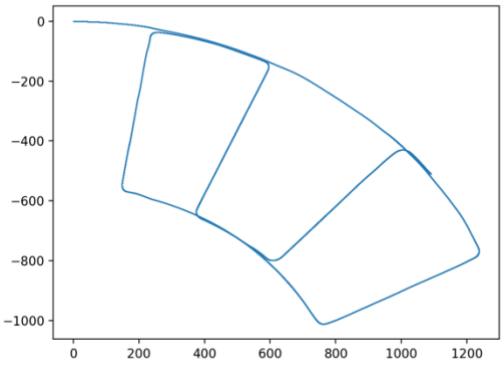
Timestamp: Iteration 40000 (400 seconds)



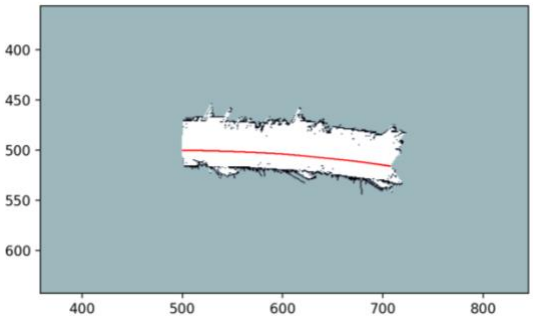
Timestamp: Iteration 80000 (800 seconds)



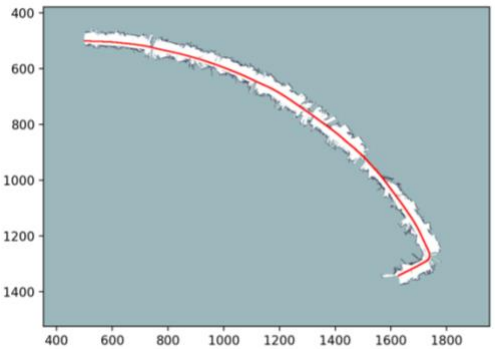
Timestamp: Iteration 101000 (1010 seconds)



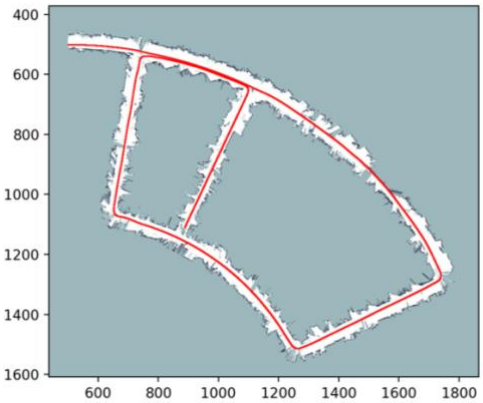
Timestamp: Iteration 116046 (1160 seconds)



Timestamp: Iteration 10000 (100 seconds)



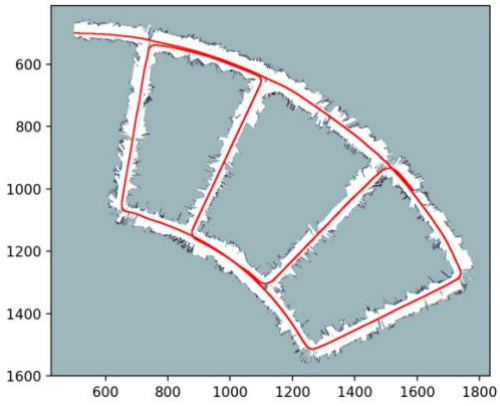
Timestamp: Iteration 40000 (400 seconds)



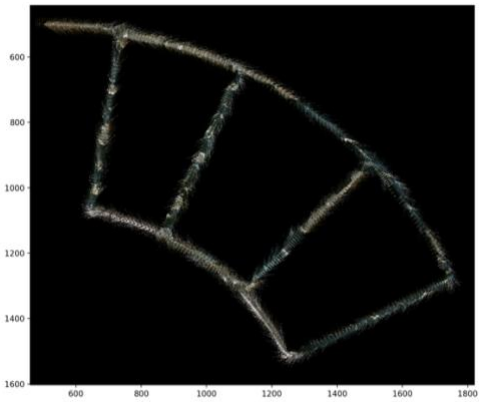
Timestamp: Iteration 80000 (800 seconds)

Occupancy Grid Map

The evolution in occupancy grid map with time can be seen with the following four images. We can see than the LiDAR reading works correctly we can see in the following two photos, the place where there is a bus, our occupancy grid map also, has a boundary, occupied cells.



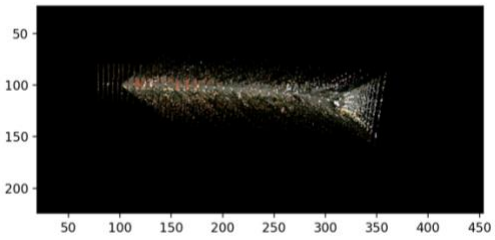
Timestamp: Iteration 115865 (1158 seconds)



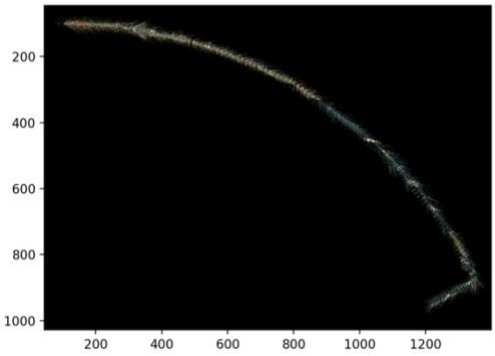
Timestamp: Iteration 115865 (1158 seconds)

Texture Mapping

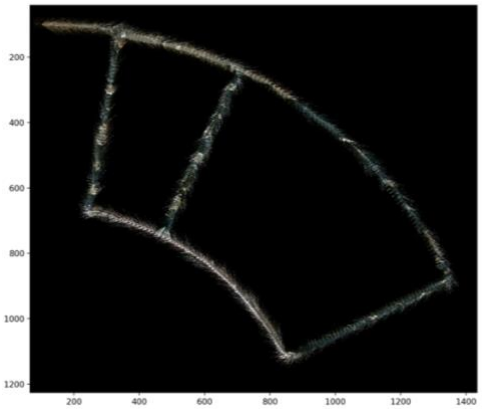
The evolution in texture mapping with time can be seen with the following four images.



Timestamp: Iteration 10000 (100 seconds)

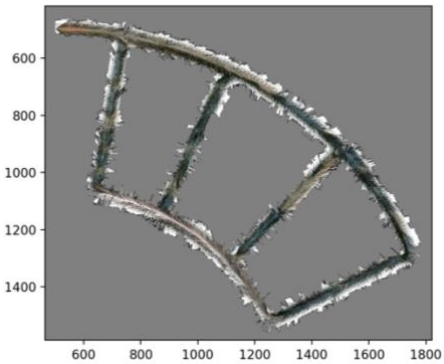


Timestamp: Iteration 40000 (400 seconds)



Timestamp: Iteration 80000 (800 seconds)

Finally, the RGB values from the texture mapping is used to fill the Occupancy grid map, and both of the two overlaps perfectly. Thus the two algorithms works decent.



In this project, the following areas were explored but could not make into the final report. There is distortion in the image, and to get even better results, we need to remove distortion first, and then perform the Texture Mapping operation.

Hyperparameters used:

No. of particles - 300

Resolution of the map - 0.5