Particle Filter SLAM and Texture Mapping for an Autonomous Car

Shrey Kansal
Mechanical and Aerospace Engineering
University of California San Diego
skansal@ucsd.edu

I. Introduction

There has been rapid development in the field of self-driving cars, automated logistics, drones, and manufacturing techniques. All these areas can be justifiably grouped into what we call Robotics. Regardless of the area of application, sensing and estimation has been at the core of research and development for robotics. As the name suggests, sensing and estimation is essential for a robot to function with as much dexterity and perception as humans. The latter of the two, Estimation comprises of tracking the movement of the robot using sensors like encoders, IMU and gyroscope, and estimating its environment using LiDAR scans and stereo camera. It is crucial to understand how the pose of various sensors change in accordance with robot movement in order to accurately model them.

From an analytical point of view, SLAM (Simultaneous Localization and Mapping) is the computational problem of constructing or updating the map of an unknown environment while simultaneously keeping track of the robot's position and orientation within it. In more naïve terms, it can be considered as a chicken-and-egg problem as both the map of environment and the robot's pose are interdependent. While there exists algorithms such as Extended Kalman Filter and learning based approaches, in this paper we implement a Particle Filter based algorithm along with Occupancy Grid Mapping and Texture Mapping for the best particle so obtained. We use wheel encoder and Fiber Optic Gyro for the robot's pose, and 2D LiDAR scans for occupancy grid mapping. In the second part of the project, we use stereo camera mounted on the robot for texture mapping.

II. Problem Formulation

Our problem statement of Simultaneous Localization and Mapping (SLAM) comprises of estimating the robot's pose ($\boldsymbol{x}_t$) at each time step with respect to its changing environment ($\boldsymbol{m}_t$) and vice-versa. Hence, we define our problem in two parts: predicting the current position of the car, updating it using inputs from 2D LiDAR scans and implement texture mapping for the trajectory.

It is observed that data and images from all sensors is asynchronous and is pre-processed at various stages of the project.

A. Motion Model

Motion model is a nonlinear function f or equivalently a probability density function p_f that describes the motion of the robot to a new state $\boldsymbol{x}_{t+1}$ after applying control input $\boldsymbol{u}_t$ at state $\boldsymbol{x}_t$.

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{w}_t) \sim p_f(\cdot|\boldsymbol{x}_t, \boldsymbol{u}_t)$$
$$\boldsymbol{w}_t \; is \; motion \; noise$$

Motion model simply describes the movement of the robot, given the data collected from various sensors. We implement Euler discretization of the differential drive kinematic model to predict the position of robot with respect to time.

B. Observation Model

Observation model is a nonlinear function h or equivalently a probability density function p_h that describes the observation $\boldsymbol{z}_t$ of the robot depending on state $\boldsymbol{x}_t$ and map $\boldsymbol{m}_t$.

$$\boldsymbol{z}_t = h(\boldsymbol{x}_t, \boldsymbol{m}_t, \boldsymbol{v}_t) \sim p_h(\cdot|\boldsymbol{x}_t, \boldsymbol{m}_t)$$
$$\boldsymbol{v}_t \; is \; motion \; noise$$

Combining the Motion and Observation Models, we get a joint distribution as follows:

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

Bayesian Inference is used to maintain a posterior probability density function (pdf) for the parameters given the data. Using the prior knowledge from Particle

Filter theory and Occupancy Grid Mapping theory, we approach our given problem in the following steps:

Occupancy Grid Map: Each obstacle that the LiDAR rays ($\boldsymbol{z_t}$) encounter needs to be represented by a position in the virtual map ($\boldsymbol{m_t}$). So, we initialize an empty map, $\boldsymbol{m} \in \mathbb{R}^{mxn}$ (Occupancy Grid Map) and update the same after transforming LiDAR data to world-frame coordinates.

Texture Map: As part of problem formulation, we need to super impose the RGB color values from these images onto an initialized color map, in accordance with the timestamp of the images and the trajectory.

III. TECHNICAL APPROACH

SLAM can be described as parameter estimation problem for state, $\boldsymbol{x}_{0:T}$ and map $\boldsymbol{m}$ given a dataset of the robot inputs, $\boldsymbol{u}_{0:T-1}$ and observations $\boldsymbol{z}_{0:T}$. In the given case, we implement Particle Filter SLAM using Bayesian Inference (BI).

A. SLAM for given problem

Predict Step:

Wheel encoders: We have 116048 readings from two encoders at rear left and right wheels each. Each reading represents the total number of ticks up till that point of time. From these readings, we can determine the total distance ($\boldsymbol{x}$) traversed by the robot (given the sensor parameters).

$$meters\ per\ tick = \frac{\pi * (wheel\ diameter)}{resolution}$$

$$resolution = 4096\ ticks\ per\ revolution$$

$$left\ wheel\ dia. = \ 0.623479$$

$$right\ wheel\ dia. = \ 0.622806$$

Fiber Optic Gyro (FOG): We have 1160508 readings of Roll, Pitch and Yaw from the FOG sensor. The yaw readings representing the turning angle of the robot about the Z-axis (world). Each reading of FOG sensor gives the difference of angle for that timestep ($\Delta\theta$), which the robot has rotated about the respective axes.

Dead Reckoning: To start the process of SLAM, we first implement a prediction only filter. In other words, we predict the state of the robot at each time step using the given control measurements from wheel encoders and the FOG. This aids us in deciding the grid size for the update steps.

Both sensors are asynchronous, and the gyroscope outputs approximately 10 times more data than the encoder for any time interval. By simple observation, we take the sum of every 10 data points of yaw angle reading for every 1 reading of encoder.

By dead reckoning, we mean computing and plotting the trajectory of the robot, given 1 particle and without noise. As we are given readings from the encoder, we first calculate the difference of ticks in each timestep. Using the formula mentioned in Section II, we obtain the distance travelled by the robot in each time step.

To compute the trajectory, we are given the differential drive kinematic model, and we assume Euler discretization of the same. Mathematically, we define the robot's pose as $\{x_t, y_t, \theta_t\}$ at any given time t. The trajectory is initialized at 0 and is continuously updated with time. The motion model of the same is given as follows:

$$\boldsymbol{x}_{t+1} = \begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = f(\boldsymbol{x_t}, \boldsymbol{u_t}) \coloneqq \boldsymbol{x_t} + \tau \begin{bmatrix} v_t \cos(\theta_t) \\ v_t \sin(\theta_t) \\ \omega_t \end{bmatrix}$$

$$\theta_t = \omega_t \Delta t$$

where,

$$\tau = \Delta t\ (timestep),$$
$$v_t\ is\ the\ linear\ speed,$$
$$\omega_t\ is\ the\ angular\ speed,$$

The dead reckoning plot can be seen in Fig 1.

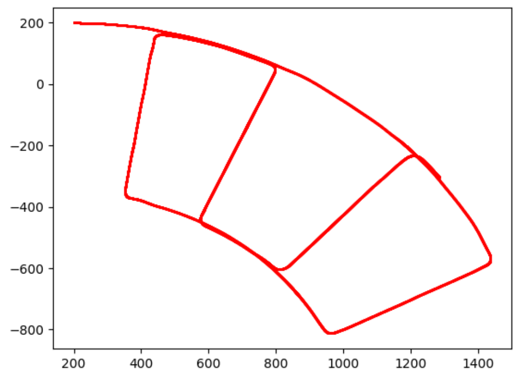

Figure 1 Dead reckoning Trajectory

Predict Only Filter: Continuing from the model built for dead reckoning, we add 100 particles for each point in time along with random gaussian noise (1% relative) for both Encoder and FOG readings. This trajectory is then plotted in a similar fashion and the results can be seen in Fig. 2.

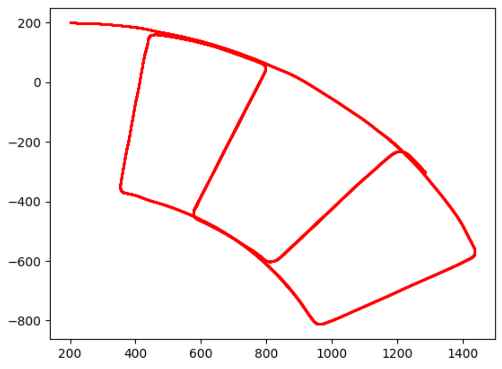

Figure 2 Prediction only with 1% Noise for N=100

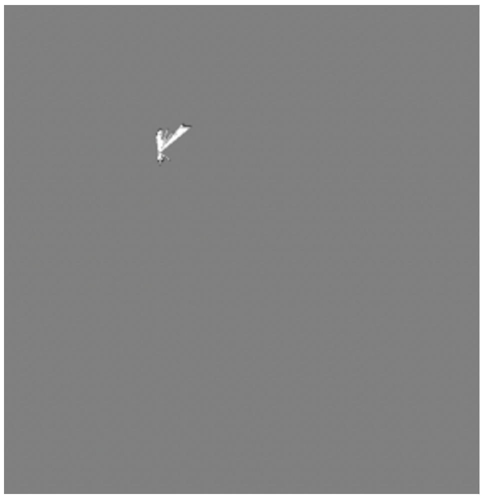

Figure 4 LiDAR Scan at 0 timestep

Update Step:

Initialize: To get started with the update step of SLAM, we first initialize a 2D grid map of size 1600 x 1600, with a resolution of 0.5. First, we initialize a set of equal weights (probabilities) for each particle and maintain the same in a vector form throughout the program.

2D LiDAR Scans: The LiDAR (Light Detection and Ranging) illuminates the scene with pulses laser light and measures the return times and wavelength of the reflected pulses. The readings are spread-out across 115865 timesteps, with 286 rays at a given timestep. Each data point corresponds to the distance travelled by each ray. The rays move from -5deg to +185deg at an interval of 0.666 deg in the sensor frame. Using the show_lidar() function in pr2_utils.py file, we can visualize the LiDAR rays in polar coordinates (Fig 3) and world coordinates (Fig 4).

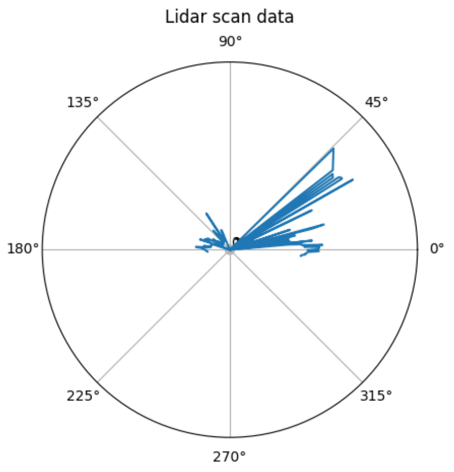

Figure 3 LiDAR scan for 0 timestep

LiDAR Transformation: As the prediction only filter starts working, we include an update step in the code, which takes into account the LiDAR scan Next, we filter the LiDAR readings such that the maximum distance of any light beam is 60m. LiDAR readings are then transformed from sensor to vehicle frame using the given rotation (R) and pose (p) parameters.

$$T_{vl} = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$$

For vehicle to world frame transformation of LiDAR data, we find the closest timestamp (post predict) in LiDAR readings using *bisect_right()* function in python and transform the corresponding data point on the fly in time loop.

Map Correlation: Next, we initiate a loop over the number of particles. The map correlation is computed between the world-frame scan and the occupancy map using the mapCorrelation() function in pr2_utils.py. The mapCorrelation() function returns a correlation matrix, from which we find the maximum weight of the particle. After obtaining the maximum weight of each particle, we find the particle with maximum weight, and call it as the best particle. During this process, the weights are normalized using Softmax() function.

Ray Tracing: Next, we initiate a loop over the number of LiDAR readings because the number of readings for each point in trajectory is different after filtering the LiDAR data. Within this loop, a ray tracing algorithm (Bresenham2D) is implemented to obtain an array of cell coordinates through which a given ray passes.

Log-odds Update: After we get the points through which a ray passes, we update the log-odds of these points received from ray tracing: decreasing the log odds of free space (-1.5*log4) and increasing the log odds (+log4) of occupied cells. Since, this is a recurring process, we cap the values of log-odds at both minimum and maximum before visualizing the same. The two values in case of free and occupied cells is different, as we get a discernible boundary at the end points of the LiDAR rays.

Resampling: Resampling is a fail-safe algorithm which works to re-initialize the particles and weights in the case when the effective number of particles ($N_{eff} = 1/\sum_{k=1}^{N} \alpha^{(k)2}$) is below a certain threshold ($N_{threshold} = 20\%\ of\ N$). We implement a stratified resampling algorithm to account for such situation. This algorithm guarantees that samples with large weights appear at least once and those with small weights – at most once.

Texture Mapping:

Stereo Camera Images: We are given 1161 images from left and right lens of the stereo camera. About 96% of the images are synchronous, while we must correct the last 4% of the images.

Synchronizing the data: We notice that the last 4% of the stereo images are not synchronous; however, the difference in time step is very small. We implement a code to sort the left images in increasing timestep order, and find the closest right image using argmin() function.

Disparity and Transformation: After arranging the images, we build a disparity (d) map using the given cv2 function in pr2_utils.py. Given the transformation and K (intrinsic parameters) matrices, we first find the depth (z) of each pixel in optical frame. This depth is then capped to be below 50m as we are not interested in a point beyond this limit for a given timestep. Then, we proceed to find the x and y coordinates of each pixel in optical frame. Next, using the given rotation matrix and pose, we transform the pixel coordinates to vehicle frame. Using the trajectory data, we then transform the pixel coordinated from vehicle frame to the world frame (like LiDAR transformation).

$$d = u_L - u_R = \frac{1}{z} f s_u b$$

$$\begin{bmatrix} u_L \\ v_L \\ d \end{bmatrix} = \begin{bmatrix} f s_u & 0 & c_u & 0 \\ 0 & f s_u & c_u & 0 \\ 0 & 0 & 0 & f s_u b \end{bmatrix} \frac{1}{z} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_{or} R^T (\boldsymbol{m} - p)$$

$u_L,\ v_L = pixel\ coordinates$ of left image

$u_R,\ v_R = pixel\ coordinates$ of right image

$f = focal\ length\ of\ the\ camera$

$b = baseline\ of\ the\ camera$

$c_u, c_v = coordinates\ of\ the\ principal\ point$

$s_u, s_v = pixel\ scaling\ (\frac{pixels}{meter})$

We have been given the inverse of $R_{or}R^T$, and transformation is done accordingly.

Mapping: We first initialize an empty color map of size 1500 x 1500. Inside a loop over sorted filenames in the left image directory, we get the RGB values of each pixel in each image. Using the world coordinates of pixels, we assign respective RGB values at the corresponding coordinates in the empty color map. Before assigning the color values, we cap the height of each pixel in world coordinates to be less than 1m.

IV. Results

We implemented our SLAM algorithm for the given data set and use case. The results can be visualized in the form of evolution of robot's state and environment for various timesteps in Fig 5 through Fig 8. The algorithm seems to work fairly good for the given data inputs.

Trajectory: I plotted the trajectory for both 1% noise and 5% noise, and the trajectory for 5% noise is distinct from the former. Since the use case is of an autonomous cars, many lives are at stake during the on-road testing. Hence, I decided to compute with 1% noise only. Moreover, I tried to compute the yaw angle of the robot using the vehicle geometry and encoder readings. However, the plot did not seem right and was not in accordance with the stereo images. I believe, this is the reason why we have much frequent readings from the FOG sensor than the encoder (to get yaw angle correct.

Map Correlation and Log-Odds: For the map correlation part, we are required to input a grid of values around the current particle position to get a good correlation. I tried to implement this with a 9 x 9 grid, and it increased the computation time by almost 2 times as compared to a 5 x 5 grid. Since, no benefit of a 9 x 9 grid was discernible, I decided to go ahead with a 5 x 5 grid size, which yielded good results. During the update of log-odds, it was seen that if we increase and decrease the log-odds of occupied and free cells by the same amount, we do not get a good enough map as output. Hence, I decided to change these values by a factor of 0.5.

Resampling: Resampling was not observed for until 5% noise. However, the algorithm is expected to perform well in case such a situation occurs.

V. Future Work

As part of future work, for texture mapping, we need to undistort the images, and project the RGB values on the log-odds grid map itself for better visualization. This will yield a better texture map result. We will also increase the noise in data obtained from various sensors and match with the actual noise measured.

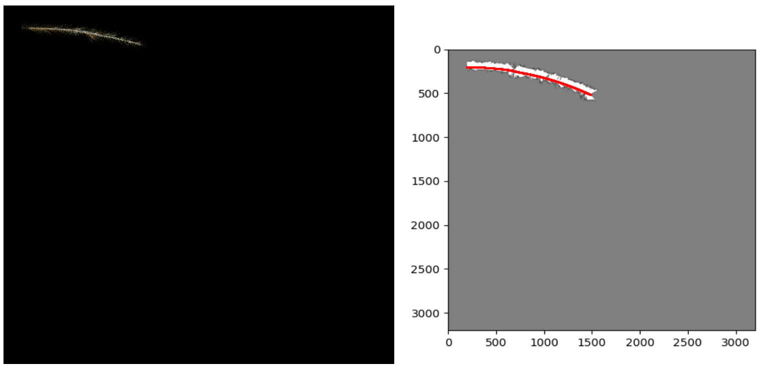

Figure 5 Texture Map and Grid Map at 20k timesteps

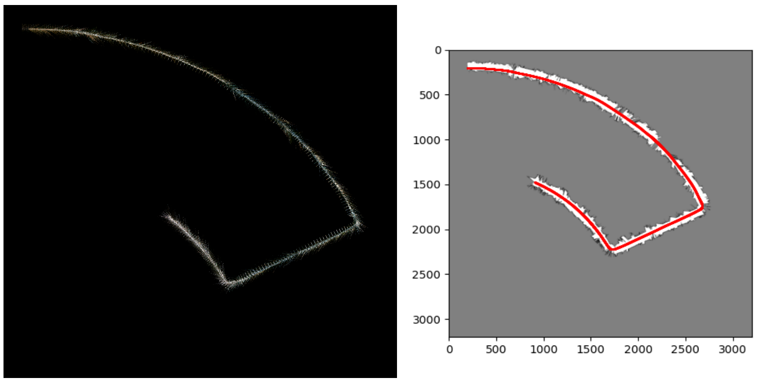

Figure 6 Texture Map and Grid Map at 50k timesteps

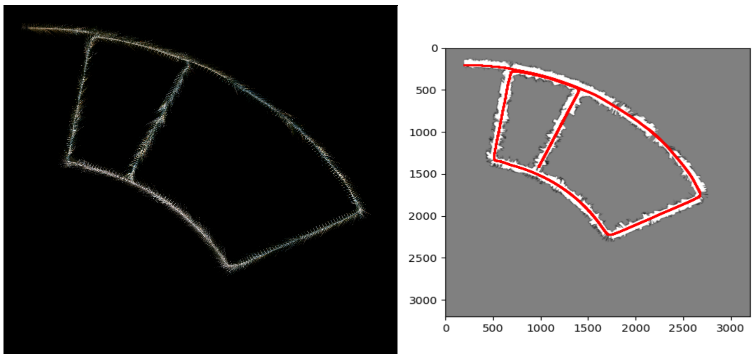

Figure 7 Texture Map and Grid Map at 80k timesteps

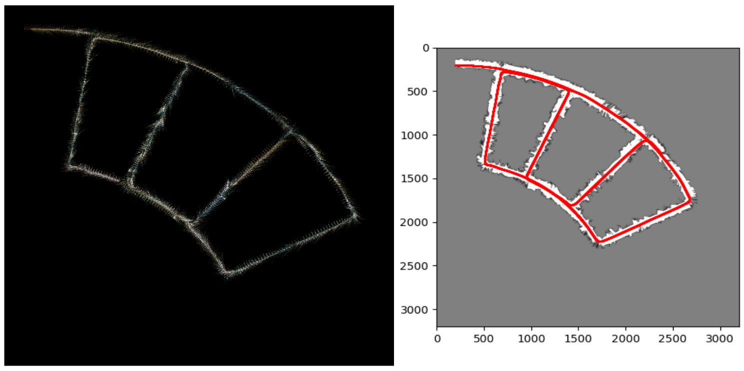

Figure 8 Texture Map and Grid Map at 115865 timesteps