Particle Filter SLAM

Varun Pawar
Electrical and Computer Engineering
University of California, San Diego
vpawar@ucsd.edu

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

In this project, a particle filter SLAM(Simultaneous Localization and Mapping) algorithm will be implemented to find the trajectory and create an occupancy map of a self-driving car. Particle filters are one of the most simple SLAM approaches. It creates multiple state hypotheses, enabling the robot to estimate under high noise conditions. The project aims at generating maps and estimating the trajectory using minimal sensor data. The application also focuses on understanding the challenges faced in state estimation under noisy conditions. There will also be a short discussion on the stereo image-based texture mapping in the end.

II. Problem formulation

A particle filter-based SLAM model is to be implemented in this project. The following section discusses in detail the related preliminaries.

A. Motion Model:

For this implementation, a differential drive model will be used for prediciting the vehicle trajectory. It is described as follows:
Consider x and y are the x and y coordinates of the robot respectively
ω_l and ω_r are the left and right wheel angular velocities,
It is given that the vehicle's axle has a lenght of Ł and the wheels' diameter are d_l and d_r respectively.
From here it follows,

$$\begin{aligned} v_l &= \frac{1}{2}\omega_l d_l \\ v_r &= \frac{1}{2}\omega_r d_r \\ v &= \frac{v_l + v_r}{2} \\ \omega_c &= \frac{v_l - v_r}{L} \end{aligned}$$

Hence the motion model is,

$$\begin{pmatrix}\dot{x}\\ \dot{y}\end{pmatrix} = \begin{pmatrix}cos(\theta) & -sin(\theta)\\ sin(\theta) & cos(\theta)\end{pmatrix}\begin{pmatrix}v\\ 0\end{pmatrix} \quad (1)$$

$$\dot{\theta} = \omega \quad (2)$$

Here θ is the yaw-heading of the vehicle. This model can be discretized as,

$$\begin{pmatrix}x_{t+1}\\ y_{t+1}\\ \theta_{t+1}\end{pmatrix} = \begin{pmatrix}x_t\\ y_t\\ \theta_t\end{pmatrix} + \tau \begin{pmatrix}v_t cos(\theta_t)\\ v_t sin(\theta_t)\\ \omega_t\end{pmatrix} \quad (3)$$

Here τ is the time-step size.

B. Sensors

There are four different sensors installed in the vehicle. They are as follows:

1) Wheel Encoder: The wheel encoder counts the number of ticks registered on a wheel. Consider a wheel of diameter d, which registers T ticks for δt seconds. One complete revolution of the wheel occurs after R ticks. Then the velocity of the wheel, i.e., v, is given as,

$$v = \frac{\pi d T}{R \delta t}$$

2) Fibreoptic gyroscope: Fibreoptic gyroscope measures the phase difference between two light sources passed through a coiled fiber optic cable fixed on the vehicle. The phase difference corresponds to the delta angle shift in every axis.

3) LIDAR: LIDAR is a range sensor that transmits a medium-range light beam only to be reflected and captured by the receiver installed in the sensor. By estimating the time of travel of the light beam, the sensor can effectively measure the depth of the scene in a particular direction. For this project, a 2D LIDAR will be used. A 2D LIDAR transmits light in a 2D plane. They are usually installed at a certain small pitch angle for the vertical understanding of the scene.

4) Stereo Camera: A stereo camera captures the images in the scene. On top of that, it also measures the disparity between the views of two cameras and could estimate the scene's depth.
A point in space (x, y, z) is projected in a camera pixel coordinates (u, v) using the following projection equation,

$$\begin{pmatrix}u\\ v\\ 1\end{pmatrix} = \begin{pmatrix}fs_u & 0 & 0 & c_u\\ 0 & fs_v & 0 & c_v\\ 0 & 0 & 0 & 1\end{pmatrix}\frac{1}{Z}\begin{pmatrix}X\\ Y\\ Z\\ 1\end{pmatrix}$$

where,

$$\begin{pmatrix}X\\ Y\\ Z\end{pmatrix} = R_o R^T \left(\begin{pmatrix}x\\ y\\ z\end{pmatrix} - p\right) \quad (4)$$

Where, R and R_o are rotation and axis flip matrix respectively. Pixel coordinates from two different views are used to calculate the disparity map and this disparity map helps us to calculate the depth of the scene.
It is described by the following equation,

$$\begin{pmatrix} u \\ v \\ d \end{pmatrix} = \begin{pmatrix} fs_u & 0 & 0 & c_u \\ 0 & fs_v & 0 & c_v \\ 0 & 0 & 0 & -fs_u b \end{pmatrix} \frac{1}{Z} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

where, d is the pixel disparity value. Further discussions on stereo cameras are in the solution and observation sections.

C. Sensor Data

- All parameters and static transformations can be found in param.zip. All transformations are provided from the sensor to the body frame, i.e., $T \in SE(3)$.
- The FOG provides relative rotational motion between two consecutive timestamps. The data can be used as $\Delta\theta = \tau\omega$, where $\Delta\theta$, τ and ω are the yaw-angle change, angular velocity, and time discretization, respectively.
- The sensor data from the 2D LiDAR, encoders, and FOG are provided in .csv format. The first column in every file represents the timestamp of the observation. For the stereo camera images, each file is named based on the picture's timestamp.

D. Objective

- **Mapping:** Try Mapping using the first LiDAR scan and display the map to make sure your transforms are correct before you start estimating the robot pose. Remove scan points that are too close or too far. Transform the LiDAR points from the LiDAR frame to the world frame. Use $bresenham2D$ or $cv2.drawContours$ to obtain the occupied cells and free cells corresponding to the LiDAR scan. Update the map log-odds according to these observations.
- **Prediction:** Implement a prediction-only particle filter at first. In other words, use the encoders and the FOG data to compute the instantaneous linear and angular velocities v_t and ω_t and estimate the robot trajectory via the differential drive motion model. Based on this estimate, build a 2-D map before correcting it with the LiDAR readings. In order to see if your prediction step makes sense, try dead-reckoning (prediction with no noise and only a single particle) and plot the robot trajectory.
- **Update:** Once the prediction-only filter works, include an update step that uses scan-grid correlation to correct the robot pose. Remove scan points that are too close or too far. Try the update step with only 3 - 4 particles to see if the weight updates make sense. Transform the LiDAR scan to the world frame using each particle's pose hypothesis. Using map correlation, compute the correlation between the world-frame scan and the occupancy map. Call map correlation with a grid of values (e.g., 9×9) around the current particle position to get a good correlation (see p2_utils.py). You should consider adding variation in the yaw of each particle to get good results.
- **Texture map:** Compute a disparity image from stereo image pairs using the provided script in p2_utils.py and estimate depth for each pixel via the stereo camera model. Project colored points from the left camera onto your occupancy grid to color it. Determine the depth of each RGB pixel from the disparity map and transform the RGB values to the world frame. Find the plane corresponding to the occupancy grid in the transformed data via thresholding on the height. Color the cells in the occupancy grid with RGB values according to the projected points that belong to its plane.

III. Technical Approach

A. Mapping

1) Calculating Occupied Cells: : Let, fov be the field of view of the lidar sensor. It transmits N light beams and is oriented at R_L^B at the position p_L^B. Then cells occupied in the grid map is given by,

$$\mathbf{x}_{occupied} = R_B^{WT} \left(R_L^{BT} \left(\begin{pmatrix} r_i cos(\theta_i) \\ r_i sin(\theta_i) \end{pmatrix} - p_L^B \right) - p_B^W \right) \quad (5)$$

Here, (R_B^W, p_B^W) is the pose of vehicle in the world frame. r_i is the range of i^{th} beam located at the yaw heading of $\theta_i = \frac{i-1}{N} * fov$

2) Estimating free cells:: For estimating free cells, Breshnham's algorithm is used. It gives the indices of grid cells give lie between the start and the end point of a light beam. This gives a comprehensive understanding of the scene and helps to identify the free cells in the map.

3) Updating Map:: Maps are updating using binary inidicators. Initially, all cells of the map are set to be occupied. Using the previous two steps, indices of free free are estimated. Wherever a free index is observed, a the cell value is updated to '0'.

B. Prediction

1) Dead Reckoning:: Dead reckoning adapts the motion model of differential drive as given in the previous section. The yaw evolution is linear and hence it is calculated by simple linear transformation $y_{new} = y_{old} + \delta y$, where δy is the delta-yaw change observed from the FOG sensor. Here the yaw is estimated independent of the linear velocity. For estimating the vehicle position, coordinates x and y are estimated as,

$$\begin{pmatrix} x_{new} \\ y_{new} \end{pmatrix} = \begin{pmatrix} x_{old} \\ y_{old} \end{pmatrix} + \tau \begin{pmatrix} v_l cos(\theta_{old}) \\ v_l sin(\theta_{old}) \end{pmatrix} \quad (6)$$

For increasing computational efficiency, the above code is implemented in parallellized manner using python numpy package's array class.

2) Noisy prediction::

- **Noise model:** For this implementation, a Gaussian noise model is assumed. From the motion model once can infer that the angular velocity and linear velocity are independent of each other, i.e. $P(\Omega, V) = P(\Omega)P(V)$, where Ω and V are angular and linear velocity random variables respectively.
Hence, $\Omega \sim \mathcal{N}(\omega, \sigma_\omega^2)$ and $V \sim \mathcal{N}(v, \sigma_v^2)$.
- **Motion model:** This approach follows the same differential drive model with some modification as given below,

$$\begin{pmatrix}\dot{x}\\ \dot{y}\end{pmatrix} = \begin{pmatrix}cos(\theta) & -sin(\theta)\\ sin(\theta) & cos(\theta)\end{pmatrix}\begin{pmatrix}v+\epsilon_v\\ 0\end{pmatrix} \quad (7)$$

$$\dot{\theta} = \omega + \epsilon_\omega \quad (8)$$

where, $\epsilon_\omega \sim \mathcal{N}(0, \sigma_\omega^2)$ and $\epsilon_v \sim \mathcal{N}(0, \sigma_v^2)$
- **Prediciton Step:** A new prediction is made as follows, $x_{new} \sim \mathcal{N}(x_{old}, (v_l \tau \sigma_v cos(\theta_{old}))^2)$, $y_{new} \sim \mathcal{N}(y_{old}, (v_l \tau \sigma_v sin(\theta_{old}))^2)$ and $\omega_{new} \sim \mathcal{N}(\omega_{old}, (\delta\theta_\omega)^2)$. Since, the prediction step is 10 times faster than the update step, the above method is parallelized in the code for 10-step interval. The rest of the sampling procedure is same.

C. Map Update:

1) Particle conditioned lidar map:: The lidar point cloud $r_i | i \in [N]$, at some timestep is transformed into the world-coordinate of given a k^{th} particle with pose (p_W^k, R_W^k), where $k \in [no\ of\ particles]$. The transformed coordinates are given as $\mathbf{x}^k_{occupied}$,

$$\mathbf{x}^k_{occupied} = R_W^{k\ T}\left(R_L^T\left(\begin{pmatrix}r_i cos(\theta_i)\\ r_i sin(\theta_i)\end{pmatrix} - p_B^k\right) - p_W^k\right) \quad (9)$$

Here, R_L^T and p_B^k are the body frame orientation and position of the lidar sensor.

2) Lidar-Map correlation:: In order to validate if a given point cloud belongs to the map, a map correlation metric is created, which finds the number of points where the map matches the point cloud. i.e., $_k\mathbf{I}(m_k == 1)$. If a point cloud has the highest agreement with the map, then it is certain that the particle responsible for it is closer to the true motion state. Hence, this information is used to re-estimate the particle filter's α weights.

$$\bar{\alpha}^k_{t|t} = \frac{CR(k, Map)}{\sum_j CR(j, Map)}\bar{\alpha}^k_{t|t}$$

. In this problem additioanl pertubation is added to the given-lidar point cloud, as would account for resolution error in the estimate and hence, consider a small neightbourhood around a particle and finds the one which has the highest agreement.

3) Map Update:: Following the previous section, map updates are made around the particle which has the highest map-correlation as a map with highest correlation value is more likely to be near the true position.

$$\begin{aligned} i_{max} &= argmax(CR(k, Map)) \\ map &= PC[i_{max}] \end{aligned}$$

here PC is the array of point cloud for different particles.

4) Particle Resampling:: For this proejct, sample imoprtance resampling method is used. Particulalrly, mutlinomial distribution is used to resample new particles from the given set of particles. Let, μ_k, then a new particle sample is given as, $\mu_{new} \sim multinomial(\mu_k, \alpha_k)$ and $\alpha_{new} = \frac{1}{N_{new}}$.

5) Computational Approach:: SLAM problem is challenging to train for the given problem. Insufficient memory is one of the major culprits. There is also a slow sequential update which hinders its performance. As discussed earlier, the prediction step is 10-times faster than the update step. So the prediction step is run in parallel for the ten-time step batch. This significantly improves. Skipping lidar updates is one more approach that has improved the performance without hurting the quality of the output. Since a few lidar updates are missed, a prediction update can be made for a considerable time step batch, thus improving its speed dramatically. Decent results have been observed for updates that are skipped by ten or more. Thus a slight compromise in update pays well overall.

D. Texture Mapping

1) Depth estimate:: Given a disparity map d, depth estimate of the map can be evaluated using the formula. $z = fs_u b\frac{1}{d}$. Here z is the camera frame depth, and d is the disparity value of a given pixel.

2) Texture Mapping:: For texture mapping, it is important to find the coordinates of the associated pixel in the world frame. So a new data tuple can be created $(x_i, y_i, z_i, v_r, v_g, v_b)$ consisting of cartesian coordinates and RGB values. Texture mapping follows the same approach as occupancy map generation. Hence it can be easily integrated with the SLAM model. For every occupancy map update, texture map update can be made to a new RGB map which stores the RGB value of a given grid point. Unlike occupancy maps, textures maps are agnostic to cell occupancy, and hence they do not provide any useful information for lidar point cloud correlation. Because of this limitation, they do not play any role in the traditional slam problem. More discussions on the current difficulty with the texture mapping are discussed in the results section.

IV. Results

- Dead-reckoning proves as the reference for the other particle filter problem. The trajectory closely resembles the one observed in the stereo images.
- Particle filter observations, trajectory, and occupancy map are corroborated by the dead revoking trajectory.
- Disparity maps generated by the stereo camera are a grain in nature. Even after gaussian blurs and other techniques, it was observed that the disparity maps produced poor quality occupancy maps. It further generated poor-quality bird's eye view textures.
- There are a few things to in play when it comes to poor quality disparity maps, 1. This is a predominantly background image. Hence, road texture is evenly distributed, and the light is evenly dispersed on the surface, which

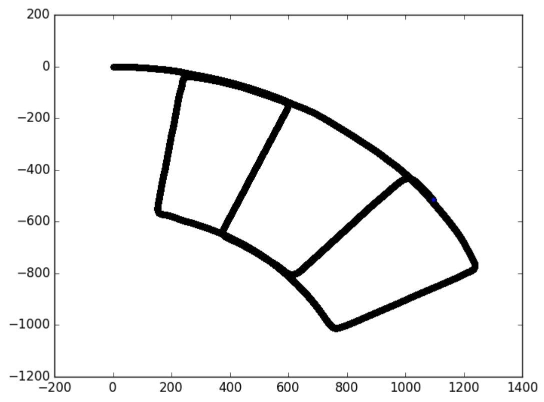
Fig. 1: Dead reckoning trajectory : This trajectory is observed for noiseless differential drive model. The trajectory can be coroborrated using the stereo image observations.

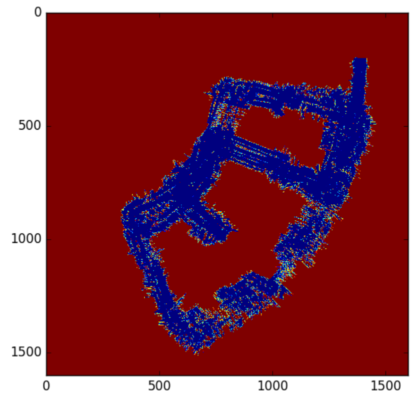
Fig. 2: Occupancy map generated by particle filter SLAM.

makes the portion of the road in both images hardly indifferent. This is why most of the pixels in the map are zeros. 2. The images are dark, and any white-level correcting will not help as stereo disparity maps use grayscale transformed images invariant white levels.

V. Conclusion

Particle filter SLAM is easy to implement and helps to give rich information of the agent trajectory and the map despite having very little data to work with. Nevertheless, this SLAM routine is prolonged and poses a significant challenge for online implementation. This problem is further exacerbated because parallel/GPU programming was not done for this implementation. Another challenge comes with the disparity maps and depth map generation. Disparity maps are grainy and are unreliable. It can be rectified using image segmentation, color thresholding, and a stream of stereo images instead of a pair. The deep learning-based approach is also a direction that can be explored. The future iterations of this work will try to proceed on the above points.

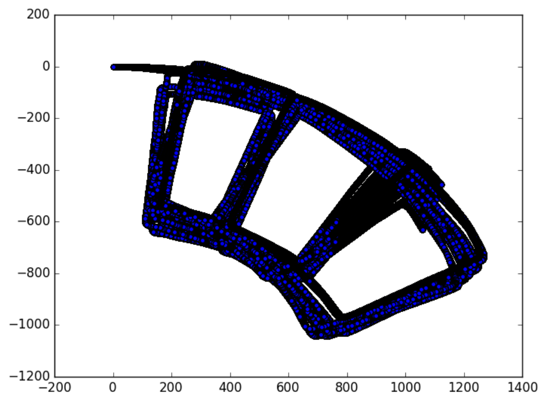
Fig. 3: Trajectory generated by particle filter.

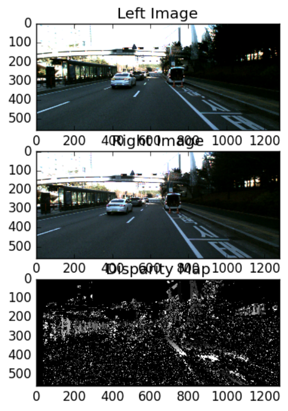

Fig. 4: Disaprity map generated by depth image

References

[1] Jdoepfert. "Jdoepfert/Roipoly.py: Select a Polygonal Region of Interest (ROI) with Python and Matplotlib, Similar to the Roipoly.m Function from Matlab." GitHub. Accessed February 7, 2022. https://github.com/jdoepfert/roipoly.py.git.

[2] 1. Van der Walt S, Sch"onberger, Johannes L, Nunez-Iglesias J, Boulogne, Franccois, Warner JD, Yager N, et al. scikit-image: image processing in Python. PeerJ. 2014;2:e453.