ESE 650 Learning in Robotics, Spring 2018
Project 4 Simulataneous Localization and Mapping

Hantian Liu

Abstract— **The report presents an approach for constructing a map of an unknown environment while simultaneously localizing the robot within it (SLAM), using a 2D laser scanner (LIDAR) and onboard IMU. The poses of robots are then used to generate a 2D texture map, using the RGB-D data from the onboard Kinect sensor.**

I. INTRODUCTION

In this project, SLAM was implemented for an indoor environment, by integration of the IMU orientation and odometry information from a walking humanoid with LIDAR. A 2D occupancy grid map was constructed, and robot poses were predicted and updated by Monte Carlo Localization (MCL), also called Particle Filter Localization. Then, given camera and depth imagery, a 2D texure map for the floor was built based on estimated robot poses.

II. METHODOLOGY
SLAM

A. SLAM Algorithm

Given the data from 2D laser scanner and onboard IMU, the SLAM algorithm could be summarized in a simple way as follows.

1) Initializate occupancy grid map and particles
2) Find the best particle with highest weight
3) Update occupancy grid map given the pose of the best particle
4) Propagate all particles
5) Update weights of all particles
6) Resample the particles when necessary
7) Repeat 2)-6)

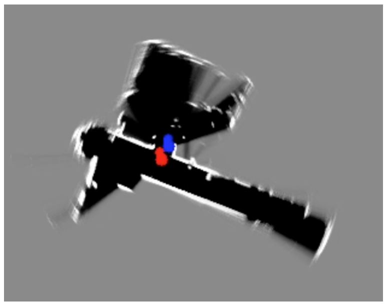
B. Data Pre-processing

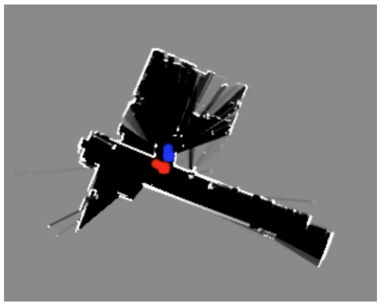
The ranges of all LIDAR scans were in LIDAR frame, so transformation was made to convert the LIDAR readings into the ground/world frame. The homogeneous transformation matrices, all in 4×4, are as shown below

$$T = T_{\text{b2g}} \cdot T_{\text{b2rotb}} \cdot T_{\text{h2b}} \cdot T_{\text{L2h}}$$

where Rot represents the SO(3) rotation matrices around the certain axes, and further

$$T_{\text{LIDAR to head}} = \begin{bmatrix} Roty(\text{head pitch}) & \begin{bmatrix} 0 & 0 & d_{\text{LIDAR to head}} \end{bmatrix}^T \\ 0 & 1 \end{bmatrix}$$





$$T_{\text{head to body}} = \begin{bmatrix} Rotz(\text{head yaw}) & \begin{bmatrix} 0 & 0 & d_{\text{head to CoM}} \end{bmatrix}^T \\ 0 & 1 \end{bmatrix}$$

$$T_{\text{body to rotated body}} = \begin{bmatrix} Roty(\text{body pitch}) \cdot Rotx(\text{body roll}) & 0 \\ 0 & 1 \end{bmatrix}$$

$$T_{\text{body to global}} = \begin{bmatrix} Rotz(\text{body yaw}) & \begin{bmatrix} 0 & 0 & h_{\text{CoM}} \end{bmatrix}^T \\ 0 & 1 \end{bmatrix}$$

Note that here the body yaw was the unbiased IMU yaw readings, instead of the yaw angle in odometry. This change improved the performances of mapping a lot, since IMU is generally more accurate than the odometry in incremental angles. Figures showing comparisons were shown above, where the upper map was built using odometry yaw while the lower map was using the yaw from IMU.

C. Occupancy Grid Mapping

Occupancy grid algorithms use a log-odds representation of the probability that each grid cell is occupied.

Map Update
The Log-odds values and saturation thresholds determine how many scans it needs to occupy a grid cell, when the cell is part of an obstacle. Also, it controls how fast it is to clear a cell, when the obstacle detected at first is a moving obstacle, which actually needs to be identified as free space.
Normally, it is more confident for the obstacles/walls that LIDAR detects, over detecting the free space found based on rays' ends. So I set the occupied update value to be 4 times larger than the free update value.

Free Cells Detection
According to the LIDAR readings of the obstacles' ranges, the area between robot and the rays' ends were considered free space. For a single raycast. Bresenham's line algorithm is often used to find the cells in between. However, to speed up the algorithm, I used opencv for graph computation.
In detail, $cv2.drawContour$ was used to fill up the area defined by the rays' ends, and thus find a mask of the free cells in the area, which would be directly used for updates in free space. Note that $opencv$ would also count the rays' ends as inside the area, so the edges needs a subtraction to avoid double counting.
It turned out to improve the speed of the algorithm a lot, where for mapping, a single scan only took 7 to 8 ms, to updated a 801×801 map from raw laser data .

D. Monte-Carlo Localization

MCL represents the belief and the posteriors over robot poses, by a random collection of weighted particles, which approximates the desired distribution.

Propagation
Motion update was fulfilled by first convert the delta poses in raw global frame to local frame, then transform the delta pose in the current global frame, and finally adding random noise from Gaussian distribution.

$$\begin{bmatrix}\Delta x \\ \Delta\theta\end{bmatrix} = \begin{bmatrix} R'(\hat{\theta_t})(\hat{P_{t+1}} - \hat{P_t}) \\ \hat{\theta_{t+1}} - \hat{\theta_t}\end{bmatrix}$$

$$\begin{bmatrix}x_{t+1} \\ \theta_t\end{bmatrix} = \begin{bmatrix} R(\theta_t)\Delta x + x_t \\ \theta_t + \Delta\theta\end{bmatrix}$$

where

$$\text{raw odometry} = \begin{bmatrix}\hat{P_t} \\ \hat{\theta_t}\end{bmatrix}$$

$$\text{global odometry} = \begin{bmatrix}x_t \\ \theta_t\end{bmatrix}$$

The noise in the motion model shows how much we trust the raw odometry over the match of laser scans. They were of vital significance in the performance of the particle filter.
Large noise would put more confidence in scan matching, where for a straight path, the updated motion would not propagate much, since the scan matching would only cover its two sides. A large noise in yaw would probably cause a wrong turn in motion. Small noise could not seperate the particles much enough to get the best particle apporoaching the true odometry.
Here the standard deviation I used for the Gaussian noise had a minimum of 0.01m for x and y poses, and 0.05 for θ yaw angles, would increase with the $\Delta pose$ when they became larger the the minimum values.

Weights Update
The weights for each particle was updated by multiplying the correlation between the scans of the particle, and the current map. Then the weights were normalized by their sum. Best particle was the one with highest weight.
One method for updating weights was to construct a sliding window for each scan, for example, a 9×9 grid. The corresponding occupancy grids value was added to the window cells, by putting the window around each cell of ray's end. And the maximum value inside the 9×9 grid was seen as the correlation for the scan. However, a larger sliding window would cause much more time to compute. To speed up the algorithm, I counted matching cells between the map and the scans. In detail, given rays' ends of each scan, newly discovered occupied cells and free cells could be obtained. Using the new occupied cells' indices, I counted the cells in current map with a value greater than 1. Same applies to the new free cells. The counts shows the overlap between the existing map and the new map from the scan. Since this method covers the whole map, it has better perfomance not only in speed, but also in distinguishment in similar particles.
It turned out to improve the speed of the algorithm a lot, where for SLAM, a single scan only took around 0.22 seconds, to updated the robot pose as well as a 1001×1001 map from raw laser data .

Resampling
Resampling of particles would be necessary if most weights are put on several particles, which to some extent decreases the number of effective particles, since the other particles already have large error.

$$N_{eff} = \frac{(\sum_i w_i)^2}{\sum_i w_i^2}$$

When the effective number, calculated as above, got lower than the threshold, which I set as 0.4 of particle numbers, resampling would be perfomed.
The cumulative sum of weights represented the weights

correspondingly. A random number, drawn from a uniform distribution between 0 to 1, was picked for each particle. And new particle was updated as the old particle with the weights where the number fell in.
Finally all particles got equal weights again, as they had initially.

III. METHODOLOGY TEXTURE MAPPING

A. Texture Mapping Algorithm

The algorithm to update the texture map could be summarized in a simple way as follows.

1) Transform the 2D depth pixels to 3D points in depth/IR camera frame
2) Keep the points that fit the ground plane
3) Transform the 3D points in depth/IR camera frame into RGB camera frame
4) Given robot pose, transform the 3D points in RGB camera/body frame to global/world coordinates
5) Transform the 3D points in RGB camera frame to 2D RGB pixels, get the color in image, and interpolate into corresponding global/world coordinates

B. Transformation Between 2D Pixels and 3D Points

Given the camera intrinsic parameters, focal lengthes f and principal points p, the conversion between 2D pixels and 3D point under camera frame, was shown as follows.

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Here the Z refers to the depth values.

C. Ground Detection

First, the invalid depths were removed, i.e. shorter than 50cm or longer than 5m. In camera frame, the axes aligned with the depth was Z-axis. The one pointing vertically down was Y-axis. Thus the surface normal of the ground plane was obtained as follows

$$\overline{n} = Rotx(\text{head pitch}) \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$$

For a 3D point (X, Y, Z) in camera frame, the ground plane was

$$a_0 X + a_1 Y + a_2 Z + a_3 = 0$$

where

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \overline{n}$$

And

$$a_3 = -h_{\text{head}} - d_{\text{Kinect to head}} \cdot \cos(\text{head pitch})$$

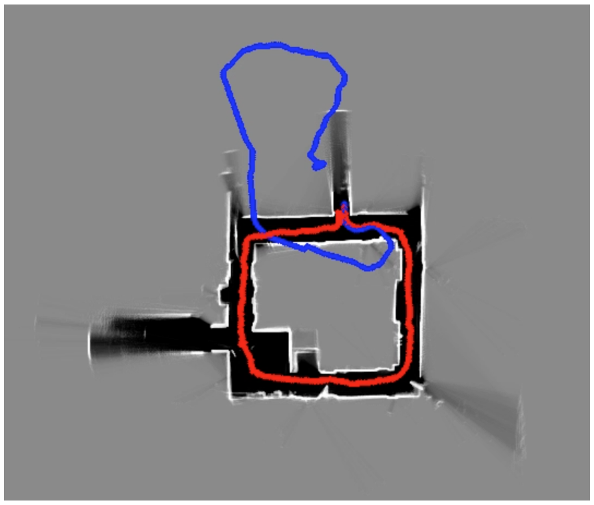


Fig. 1. SLAM result of test data

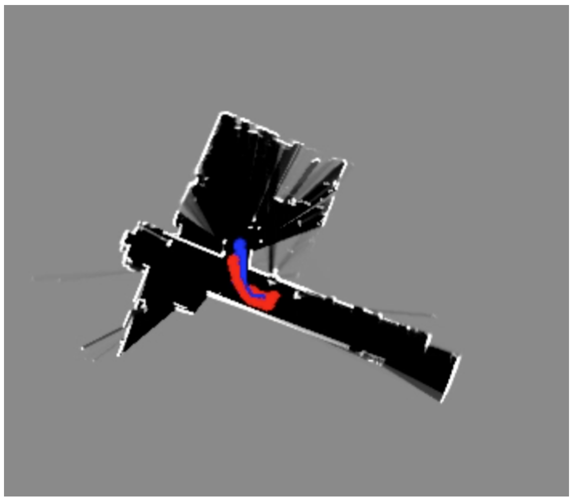


Fig. 2. SLAM result of training data 0

By simple thresholding the left-hand-side, whether a point belongs to the ground plane could be determined.

$$|a_0 X + a_1 Y + a_2 Z + a_3| \leq \epsilon$$

After trials with several numbers, I picked the best and went with a threshold of 0.1.

IV. RESULTS

A. SLAM

Here only the final occupancy grid maps and the odometry in red (raw odometry in blue) were presented, please refer to the videos submitted for the whole process.
For the test data, the SLAM results are as follows
Results of the texture mapping please refer to the figures submitted.

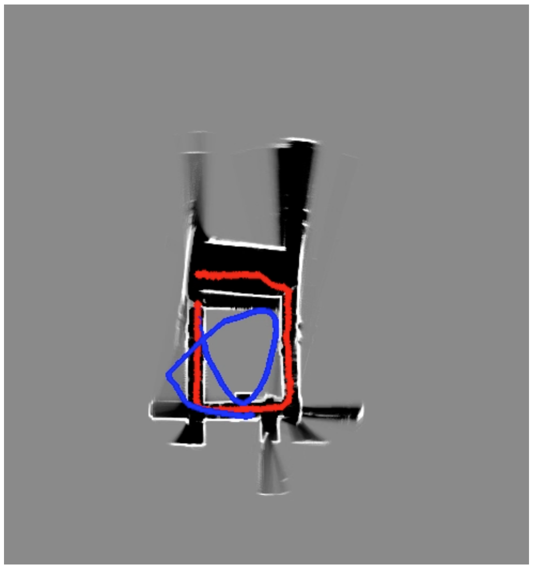


Fig. 3. SLAM result of training data 1

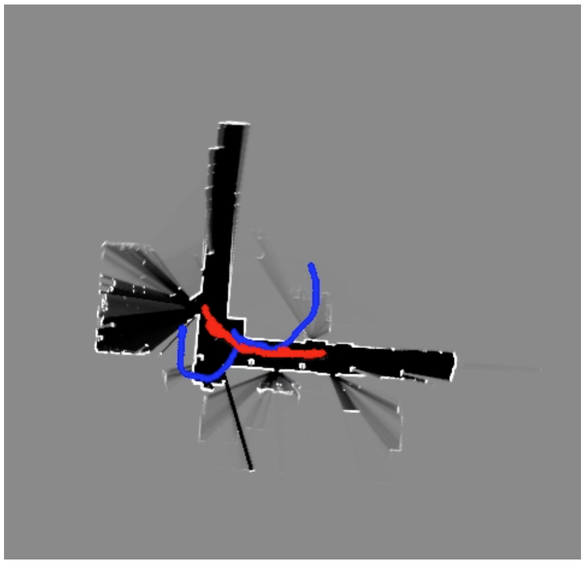


Fig. 4. SLAM result of training data 2

V. DISCUSSION

My SLAM algorithm has a pretty satisfactory results and strongly competitive computation speed. But given more time, I could do more improvements on it.

First of all, the resampling method I mentioned above is not gauranteed to improve the particle distributions. It still has the probability that the random number keeps coming from the particles with lowest weights. So to improve that, low-variance resampling could be used. On top of the random number for each particle. A uniform incremental values from 0 to 1 could be added to the random number, and the sum needs to be if larger than 1. New particle with regard to the sequence

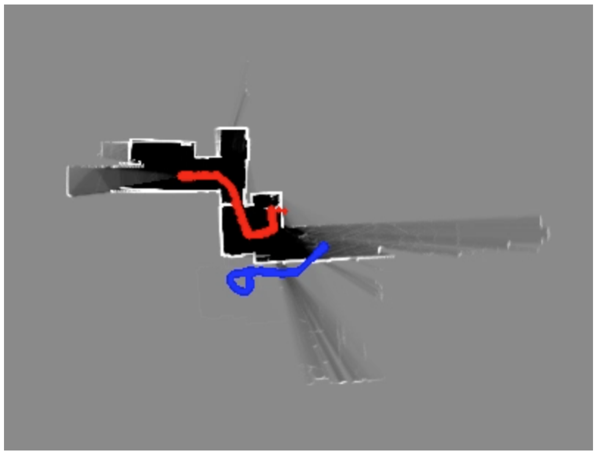


Fig. 5. SLAM result of training data 3

of this incremental value, would be updated as the old particle with the weights that the sum, instead of the random number only, falls in. In this manner, the best particle is gauranteed to be picked for updates.

Secondly, the noise for motion model could be further refined with regard to the motion. Since the motion model is very sensitive to the noise, using the same noise could not suit all odometry. Specific noise for specific odometry would make more sense. For example, when the robot poses tell a straight routine of robot, the noise could be lower, in that odometry could be trusted more over the scan matching. Especially the noise in yaw, because we do not want the particles to vary in yaw when the robot is heading forward only.

For texture mapping, my code runs relatively slow since the files are too large. Therefore, I did not have enough time to improve it in the ground detection part. I suppose that RANSAC would have better performance in ground detection instead of simple thresholding. I found that the threshold was tricky to determine, and it was important for selecting the ground points. RANSAC would save more efforts in that it only used the threshold for choosing inliers, and would generate the plane by inliers at the end.