# Particle filter SLAM using odometry, 2-D laser scans, and RGBD measurements

Yunhai Han
*Department of Mechanical and Aerospace Engineering*
*University of California, San Diego*
y8han@eng.ucsd.edu

***Abstract*—This paper represented approaches for simultaneous localization and mapping problem(also known as SLAM). In this project, we are provided with sensor data, odometry, 2-D laser scans, and RGBD measurements from a humanoid robot. However, all these sensor data are not perfect because they are contaminated by random noise to some extent. For this reason, instead of accumulating raw data without any modification, I introduced particle filter to generate enough number of particles at different locations and decide which are the better particles according to the current scan. Then, we select the particle with the largest probability and update the map assuming the scan data was obtained at that pose.**

***Index Terms*—simultaneous localization and mapping(SLAM), Particle filter, Odometry, Sensor noise**

## I. Introduction

Robotics is the science of sensing and manipulating the physical world through computer-controlled devices. Examples of successful robotic systems include mobile robots for planetary exploration, industrial robotics arms in assembly lines, autonomous driving cars and manipulators that assist surgeons. Robotics systems are situated in the physical world, obtain information from their local environments through onboard sensors, and make reactions. While all of these examples sound amazing, there are still some really challenging problems, which make these applications unavailable right now. First of all, robot environments are inherently unpredictable and dynamically changes. While the degree of uncertainty in well-structured environments is small, environments such as highways and private homes are highly dynamic, thus highly unpredictable. The uncertainty is particularly high for robots with people nearby. Second, sensors are limited in what they can perceive. Limitations arise from several factors. The range and resolution of a sensor is subject to physical limitations. For example, cameras cannot see through walls, and the spatial resolution of a camera image is limited. Sensors are also subject to noise, which makes sensor measurements unpredictable and hence limits the information that can be extracted. Third, robot actuation involves motors that are also unpredictable. Uncertainty arises from effects like control noise or mechanical failure. Some actuators, such as heavy-duty industrial robot arms, are quite accurate and reliable. Others, like low-cost or legged mobile robots, can be extremely flaky(like the humanoid robot in this project). How to handle uncertainty is indeed the most important step towards robust real-world robot systems [1].

During the decades of the development of Bayes filters, there are various efficient algorithms including Extended Kalman Filter, Unscented Kalman Filter and Particle Filter. The main advantage of particle filter over the other two filters is that it is a nonparametric method, which means it could approximate the posterior more accurately. In other way, particle filter do not make strong assumptions on the real posterior density. they are suited to represent complicated multimodel beliefs. However, this advantage comes with extra computational complexity for particle filter which is larger than the other algorithms and the running time is also proportional to the number of particles. If the number of particles goes to infinity, particle filter would converge to the correct posterior(which is unknown in most cases) and it also takes infinite time. But, although the number of particles is limited in reality, they could still provide with some reasonable results and the running time could be acceptable. That's one of the reasons why it becomes immensely popular in robotics.

## II. Problem Formulation

Given a set of sensor data including odometry, lidar scans and RGBD images, the problem is to build a map and mark the robot's trajectory on the map. However, as I mentioned in the **Introduction** section, there are random noise in these data, which requires me to implement particle filter to estimate the best trajectory and build the occupancy map corresponding to this trajectory.

- The first task to synchronize all the data. From the time tags of different sensor data, I could find that they were captured at different time. Hence before I use them, we need to match them based on their own time tags. The most simple way it to compare all the time tags of different sensor data set in the time domain and select the pairs of data which are closest to each other in $t$ as a new set. For example, suppose $x_{t_1}$ represents the time tag value of the first data in the set $x$ and $y_{t_k}$ represents the time tag value of the $kth$ data in the set $y$, I want to find a $k$, such that:

  $$y_{t_k} \leq x_{t_1} < y_{t_{k+1}}$$

  For each sensor data$x_i$ in $x$, I find the sensor data $y_k$ in $y$ which is closest to $x_i$ in the time domain.
- The second task is to transform the lidar sacns from its own frame to the world frame. Since all the angles

including robot's neck angle, head angle and orientation and robot's physical sizes are provided, I could use Euler angles:the **roll($\phi$)**,**pitch($\theta$)**,**yaw($\psi$)** to specify the rotation. Also, there exist some translations for the origins of different reference frames along the $z-axis$, so we need to add the distance between lidar and robot's head and robot's head between robot's mass center(translation vectors) into the coordinates. The rotation matrix could be defined as follow:

$$
\begin{aligned}
R &= R_z(\psi)R_y(\theta)R_x(\phi) \\
&= \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \\
&\quad * \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix}
\end{aligned}
$$

- The third task is to decide which grids are occupied based on the robot's current pose and the synchronized lidar scan data. I introduce the lidar-based occupancy gird mapping method to determine the cells that lidar beams pass through. Because this grid map is not continuous in 2-D dimensional plane, I need to use Bresenham's line rasterization algorithm to pick up the pixels on the line. For each observed cell $i$(the endpoints of the lines) or free cell $j$(cells on the lines), I increase or decrease the log-odds of each cell respectively. The initial log-odds for each cell could be set as zero since I have no idea of whether it is occupied or not. The update could be described as follow:

$$\lambda_{i,t+1} = \lambda_{i,t} + \log g_h(Z_{t+1}|m_i, x_{t+1})$$

  In this case, $g_h(Z_{t+1}|m_i, x_{t+1)}$ could be considered as a constant which specified how much we trust the observation $z_t$ and since all the sensor data were captured in a short time from the same sensor, I could assume it would not change.
  At the $i$th iteration, if I want to see what the map looks like, I could simply select all the grids with log-odds larger than zero. Because it means the probability of being occupied is larger than being free.
- The forth task is to update the pose of each particles. I simply add the **deltaPose** and random noise with certain magnitude to each particle to compute its next pose. During this step, the weight of each particle would not change.
- The fifth task is to update the weight of each particle. Using laser correlation model, I could compute the probability of the current laser scan $z$ being obtained from each particle's pose $x$ by modelling the correlation between $z$ and the local map $m$ around $x$. In short, I need to compute the correlation like this:

$$\textbf{corr}(z, m) = \sum \mathbf{1}\{m_i = z_i\}$$

  Then, the weights can be converted to probabilities via the **softmax** function:

$$p_h(z|x, m) = \frac{e^{corr(z,m)}}{\sum_v e^{(v,m)}} \propto e^{corr(z,m)}$$

  Then, I could use all the probabilities at each particle's pose $x_i$ with the same laser scan $z$ to update each particles's weight:

$$\alpha^k_{t+1|t+1} = \frac{\alpha^k_{t+1|t} p_h(z_{t+1}|\mu^k_{t+1|t})}{\sum_{j=1}^K \alpha^j_{t+1|t} p_h(z_{t+1}|\mu^j_{t+1|t})}$$

  In the above equation, $K$ represents the number of the particles and $k$ represent a specific particle. $\mu^k_{t+1|t}$ and $\mu^j_{t+1|t}$ both represent the particle's pose with respect to $k$ and $j$ particles. You can see after this update, the sum of weights is still one, which approximate the probability distribution over all possible poses. Besides, in the next section **Technical Approach**, the derivative of this update formula and an important assumption will be shown for the verification.
- The sixth task is to resample among all the particles. I have to set a threshold $N_{threshold}$ and compare it with $N_{eff} = \frac{1}{\sum_{k=1}^K (\alpha^k_{t|t})^2}$ and determine whether or not to create new set of particles from the previous one by the following rule:

  **Resample if** $N_{eff} \leq N_{threshold}$

  The most important thing is to find a appropriate threshold because it really has a big effect of the algorithm's performance. Also, in the next section, I will talk more about it.

I implement all the above steps iteratively except the first one which is only required once at the beginning. Hopefully, I will obtain a good occupancy grid map and the robot's trajectory. Just for clarification, I did not implement them in the same order as I describe here. I would tell more details in the next section of how they work and how I manage the structure.

What's more, you could see there are some parameters for us to determine the values. Indeed, these parameters play a significant role and it is reasonable to impart them with some intuitively right values.

The next part of this project is to color the map from the RGBD sensor data. It could also be decomposed into several tasks.

- The first task is the same as the previous part, that is to synchronize the RGBD images and the robot's pose data. Since I already have the map and I assume the map is static in this short time, I don't have to worry about it. However, for the trajectory, I need to know the time tags for the each grid on the trajectory and match them with the cameras' data by the **timestamp.txt**. This could be done by the same way I described before.
- The second task is to read the depth image and transform the depth information from camera's own frame to the world frame. At the beginning, I have to transfer the

depth value(unit:mm) to the x,y,z coordinates in its own frame. The principle behind this is very simple. I first assume the skew coefficient, distortion coefficients and other uncertainties are all zero, which means the camera is perfect. Thereby, the camera model could be described as:

$$\begin{bmatrix} u \\ v \\ \mu \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

The function of $\mu$ in the above equation is to scale $u$ and $v$ up to one. $u, v, f_x, f_y, c_x, c_y$ and $Z$ are known, I could obtain $X$ and $Y$ by only one computation:

$$X = \frac{(u - c_x) * Z}{f_x}$$
$$Y = \frac{(v - c_y) * Z}{f_y}$$

In practice, the z-axis of the camera model represent its principal axis, which is equivalent to the heading direction of the robot. However, in the case, the heading direction is defined as y-axis and the z-axis is defined as vertical axis.

the robot's neck angle, head angle and orientation and the robot's physical sizes are required for the correct transformation from camera frame to the world frame. Since the method here is also similar to the previous one, the x,y,z coordinates of each pixels in the would frame could be easily obtained. The height of the robot's mass center is 0.93m, any pixels with z value less than -0.93m should be considered as floor. I find all the pixels from all depth images and record their row,column indexes in each image in order to extract their RGB values from the corresponding RGB images.

- The third task is to extract the RGB value of each pixel which is considered to represent floor in the previous task. Since the x,y,z coordinates or each pixel is known in the camera frame, I just project them into the image plane to find the $u$ and $v$ and read the RGB value. Then, I draw the map with these RGB values.

Hopefully, If I do all of these steps perfectly and I am lucky enough to figure out the right parameters, I should have a good texture map right now.

## III. Technical Approach

In the **Problem Formulation** section, I briefly describe the framework of how I deal with this project. In this section, I will tell more details about each tasks. Besides, in the last, I will provide with the flowchart.

### A. Data synchronization

Suppose I have successfully read all the sensor data, I am aware of the time tag of each piece of sensor data. I first compare the size of different data sets and select the first piece of sensor data from the data set the size of which is smaller. Then, I compare the time tag of it $x_1$ with the time tag $y_i$ of the sensor data belonging to other data sets. I set $i$ start from 1 and make the comparison and there are two possible results:

- $x_1$ is smaller than $y_1$. For this case, I need to discard this piece of sensor data and make $x_1$ to $x_2$, and compare $x_2$ with $y_1$ again until find the $x_i$ which is larger than $y_1$.
- $x_1$ is larger or equal to $y_1$. For this case, I don't have to discard $x_1$ and I store this piece of sensor data into $X$ and increase $y_1$ to $y_2$ and compare $x_1$ with $y_2$ until find the $y_i$ which is larger than $x_1$, thus storing the sensor data with time tag $y_i$ into $Y$.

I implement method iteratively until one of them goes to the end of its whole set. To the five data sets, the length of joint data is almost three times larger then it of lidar scans. This makes sense since the joint sensors works with much higher frequencies than lidars. After the synchronization is done and it only needs to be implemented once at the beginning of the program, I could step towards other tasks.

### B. Generate particles

The initialization of all the particles is also implemented only once. In this step, I generate $K$ particles and all of them have the pose (0,0,0) and share the same probability $\frac{1}{K}$. For the next iteration, they would be added with the **deltapose** and random noise $(\delta x, \delta y, \delta\theta)$ which would make each particle represent different pose.

### C. Transformation from lidar frame to the world frame

From the specification provided by the manufacturer(**Specifications-UTM-30LX.pdf**), I could first transform the length of laser range into the $x, y$ coordinates in the laser laser plane($z = 0$). The angular resolution is 0.25°, so I generate a

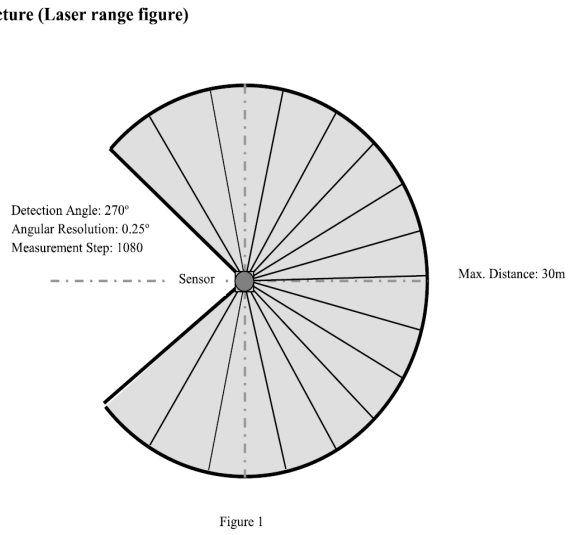


Fig. 1. Laser range figure from specification

angle list from -135° to 135° with the step as 0.25°. Suppose $l_i$ represents the length of each laser range and $\alpha_i$ represents

the angle from 135° to 135°, the $x, y$ coordinates could be computed as follow:

$$x_i = l_i * \cos\alpha_i$$
$$y_i = l_i * \sin\alpha_i$$

There are in total 1081 laser ranges for each scan, but some of them may have values larger than 30m or less then 0.1m. I remove these outliers before computing $x_i$ and $y_i$ because it is very unrealistic to trust them. If the laser range detects obstacle within 0.1m, the laser beam probably contacts some parts of the robot body instead of walls or windows in the environment.

I know the lidar is fixed on the head, so the y-axis of lidar frame and head frame are parallel. In this case, the point $x_i, y_i, z_i = 0.15$ is the corresponding point in the head frame. The next task is to transform the points into the world frame. Here, this could be done by three steps:

- **Transform from moving head frame to fixed head frame**
  I assume there is a fixed head frame. The head angle could be considered as pitch angle $\theta$ and the neck angle could be considered as yaw angle $\psi$ between the two different frames. For the convenience, I relist the Euler angles extrinsic matrix:

$$R = R_z(\psi)R_y(\theta)R_x(\phi)$$
$$= \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix}$$

  Here, only the $\theta$ and $\psi$ are needed and $\phi$ is always zero. After this operation, the lidar points have been transformed to the fixed head frame.
- **Transform from head frame to body frame**
  Here, there is only an offset along z-axis and it is the distance between head and robot's center of mass. Hence, this transformation is very simple:$z+ = 0.33$.
- **Transform from body frame to world frame**
  Since the roll and pitch angles around the robot's center of mass could be ignored, there is only yaw angle. The yaw angle could be obtained by accumulating the angle data from **deltapose**. Hence I successfully transform all the points from lidar frame to the world frame.
  I assume the height of the robot's center of mass is fixed at 0.93m, so any points with $z$ smaller than 0.93m are totally unreliable:they could be the contact points with the floor or just some outliers. It is safe to remove all such points.

### *D. Lidar-based Occupancy Grid Mapping*

From the previous operation, all the points are represented in the world coordinate but the robot could move in the world. Given the robot pose at time $t$: $x_t$, the real position of the points in the world coordinate are:$x_i+ = x_t^x, y_i+ = y_t$. I could believe that all the grids corresponding to these points are occupied, but in reality, it is not the case due to the noise. Hence, the more general method is to model the map cells $m_i$ as independent Bernoulli random variables:

$$m_i = \begin{cases} \text{Occupied(1)} & \text{with prob.}\gamma_{i,t} = p(m_i = 1|z_{0:t}, x_{0:t}) \\ \text{Free(-1)} & \text{with prob.}1 - \gamma_{i,t} \end{cases}$$

From the above equation, to represent a probabilistic map, I only need to keep a vector of the occupancies probabilities of all the grids at time $t$:$\gamma_t^i$ Using Bayes Rules:

$$\gamma_{i,t} = \frac{1}{\eta_t} p_h(z_t|m_i = 1, x_t)\gamma_{i,t-1}$$

Here, $\eta_t$ is a constant but I don't know its real value. However, I can use a trick a cancel out this parameter.

$$1 - \gamma_{i,t} = \frac{1}{\eta_t} p_h(z_t|m_i = -1, x_t)(1 - \gamma_{i,t-1})$$

If I define the odds ratio of a binary random variable $m_i$:

$$\begin{aligned} o(m_i|z_{0:t}, x_{0:t}) &= \frac{p(m_i|z_{0:t}, x_{0:t})}{p(m_i = -1|z_{0:t}, x_{0:t})} \\ &= \frac{\gamma_{i,t}}{1 - \gamma_{i,t}} \\ &= \frac{p_h(z_t|m_i = 1, x_t)}{p_h(z_t|m_i = -1, x_t)} \frac{\gamma_{i,t-1}}{1 - \gamma_{i,t-1}} \end{aligned}$$

Hence, I could conclude that to estimate the pdf of $m_i$ conditioned on $z_{0:t}$ is equivalent to accumulating the log-odds ratio:

$$\begin{aligned} \lambda(m_i|z_{0:t}, x_{0:t}) &= \log o(m_i|z_{0:t}, x_{0:t}) \\ &= \log(g_h(z_t|m_i, x_t)o(m_i|z_{0:t-1}, x_{0:t-1})) \\ &= \lambda(m_i|z_{0:t-1}, x_{0:t-1}) + \log g_h(z_t|m_i, x_t) \\ &= \lambda(m_i) + \sum_{s=0}^{t} \log g_h(z_s|m_i, x_s) \end{aligned}$$

In the above equation, $\lambda(m_i)$ represents the initial guess of occupancy state of grid $m_i$. Since I have no idea of how the map should be, I just set it as zero. It is natural to tell the probabilistic occuancy mapping reduces to keeping tracking of the cell log-odds $\lambda_{i,t} = \lambda_{i,t-1} + \Delta\lambda_{i,t-1}$. To perform the update described above, I need to specify the the observation model, that is what is really $\Delta\lambda_{i,t} = \log g_h(z_t|m_i, x_t)$.

Using Bayes rule again, I could simplify the observation log-odds ratio:

$$\begin{aligned} g_h(z_t|m_i, x_t) &= \frac{p_h(z_t|m_i = 1, x_t)}{p_h(z_t|m_i = -1, x_t)} \\ &= \frac{p(m_i = 1|z_t, x_t)p(m_i = -1)}{p(m_i = -1|z_t, x_t)p(m_i = 1)} \end{aligned}$$

$$\begin{aligned} \Delta\lambda_{i,t} &= \log g_h(z_t|m_i, x_t) \\ &= \log g_h(z_t|m_i, x_t) = \log \frac{p(m_i = 1|z_t, x_t)}{p(m_i = -1|z_t, x_t)} - \lambda(m_i) \end{aligned}$$

Here, the second term $\lambda(m_i)$ represents the prior log-odds ratio of the grid being occupied over being free. In this case,

since I have no idea of what the map is, it could be safely set as zero. The first term specifies how I trust the observation $z_t$. For example, if $z_t$ indicates that $m_i$ is occupied, the log ratio of the true positive vs false positive:

$$\frac{p(m_i|m_i\text{is observed occupied at time}t)}{p(m_i=-1|m_i\text{is observed occupied at time}t)}=4$$

For each observed cell $i$, decrease the log-odds if it was observed free or increase the log-odds if the cell was observed occupied:

$$\lambda_{i,t+1}=\lambda_{i,t}+\log g_h(z_{t+1}|m_i,x_{t+1})$$

Finally, at time $T$, if the log ratio for cell $i$ $\lambda_{i,T}$ is larger than 0, I could consider it as the occupancy grid and draw it on the map.

### *E. Particle filter for lidar-based localization*

The main advantage of particle filter over bayes filter is that instead of closed-form solution, it uses a certain number of particles to approximate the probability distribution.

To be more specific, the particle filter uses a mixture of delta functions(particles):

$$\delta\left(\mathbf{x};\boldsymbol{\mu}^{(k)}\right):=\begin{cases}\infty & \mathbf{x}=\boldsymbol{\mu}^{(k)}\\ 0 & \text{else}\end{cases}\qquad\text{for } k=1,\ldots,N$$

with weights $\alpha^{(k)}$ to represent $p_{t|t}$ and $p_{t+1|t}$, such that:

$$p\left(\mathbf{x}_t|\mathbf{z}_{0:t},\mathbf{u}_{0:t-1}\right)=p_{t|t}\left(\mathbf{x}_t\right)=\sum_{k=1}^{N_{t|t}}\alpha_{t|t}^{(k)}\delta\left(\mathbf{x}_t;\boldsymbol{\mu}_{t|t}^{(k)}\right)$$

$$p\left(\mathbf{x}_{t+1}|\mathbf{z}_{0:t},\mathbf{u}_{0:t}\right)=p_{t+1|t}\left(\mathbf{x}_{t+1}\right)=\sum_{k=1}^{N_{t+1|t}}\alpha_{t+1|t}^{(k)}\delta\left(\mathbf{x}_{t+1};\boldsymbol{\mu}_{t+1|t}^{(k)}\right)$$

Also, the sum of the weights of all the particles is always one.

*1) Particle filter Prediction:* I could approximate the prediction step as follow:

$$\begin{aligned}p_{t+1|t}(\mathbf{x})&=\int p_f\left(\mathbf{x}|\mathbf{s},\mathbf{u}_t\right)\sum_{k=1}^{N_{t|t}}\alpha_{t|t}^{(k)}\delta\left(\mathbf{s};\boldsymbol{\mu}_{t|t}^{(k)}\right)d\mathbf{s}\\&=\sum_{k=1}^{N_{t|t}}\alpha_{t|t}^{(k)}p_f\left(\mathbf{x}|\boldsymbol{\mu}_{t|t}^{(k)},\mathbf{u}_t\right)\approx\sum_{k=1}^{N_{t+1|t}}\alpha_{t+1|t}^{(k)}\delta\left(\mathbf{x};\boldsymbol{\mu}_{t+1|t}^{(k)}\right)\end{aligned}$$

since $p_{t+1|t}(\mathbf{x})$ is a mixture pdf with components $p_f\left(\mathbf{x}|\boldsymbol{\mu}_{t|t}^{(k)},\mathbf{u}_t\right)$, I may approximate it with particles by drawing samples from it. This conversion is significant because it could help us avoid lots of integration and just use motion model to each $\overline{\boldsymbol{\mu}}_{t|t}^{(k)}$ and obtain $\boldsymbol{\mu}_{t+1|t}^{(k)}$:

$$\boldsymbol{\mu}_{t+1|t}^{(k)}\sim p_f\left(\overline{\boldsymbol{\mu}}_{t|t}^{(k)},u_t\right)\text{ and set }\alpha_{t+1|t}^{(k)}=\bar{\alpha}_{t|t}^{(k)}$$

Besides, during each iteration, I have to compute $N_{eff}$ and decide whether or not to resample all the particles. This would be described later.

*2) Particle filter Update:*

$$\begin{aligned}p_{t+1|t+1}(\mathbf{x})&=\frac{p_h\left(\mathbf{z}_{t+1}|\mathbf{x}\right)\sum_{k=1}^{N_{t+1|t}}\alpha_{t+1|t}^{(k)}\delta\left(\mathbf{x};\boldsymbol{\mu}_{t+1|t}^{(k)}\right)}{\int p_h\left(\mathbf{z}_{t+1}|\mathbf{s}\right)\sum_{j=1}^{N_{t+1|t}}\alpha_{t+1|t}^{(j)}\delta\left(\mathbf{s};\boldsymbol{\mu}_{t+1|t}^{(j)}\right)d\mathbf{s}}\\&=\sum_{k=1}^{N_{t+1|t}}\left[\frac{\alpha_{t+1|t}^{(k)}p_h\left(\mathbf{z}_{t+1}|\boldsymbol{\mu}_{t+1|t}^{(k)}\right)}{\sum_{j=1}^{N_{t+1|t}}\alpha_{t+1|t}^{(j)}p_h\left(\mathbf{z}_{t+1}|\boldsymbol{\mu}_{t+1|t}^{(j)}\right)}\right]\delta\left(\mathbf{x};\boldsymbol{\mu}_{t+1|t}^{(k)}\right)\end{aligned}$$

In the above equation, the right term is still the delta function, which represents each particle. Hence, like the operation in the prediction step, I only have to update the weights of each particle and this is the meaning of the left part. Since I already have known the values of $\alpha_{t+1|t}^{(j)}$ for all $j\in N_{t+1|t}$, only $p_h(z_{t+1}|\mu_{t+1|t}^{(k)})$ is unknown. For lidar-based localization, this could be computed via **softmax** function:

$$p_h(\mathbf{z}|\mathbf{x},\mathbf{m})=\frac{e^{\text{corr}(\mathbf{z},\mathbf{m})}}{\sum_{\mathbf{v}}e^{\text{corr}(\mathbf{v},\mathbf{m})}}\propto e^{\text{corr}(\mathbf{z},\mathbf{m})}$$

The correlation **corr** is computed as:$\text{corr}(y,m)=\sum_i 1\{m_i=y_i\}$. There are two important tips here:

- The denominator $\sum_{\mathbf{v}}e^{\text{corr}(\mathbf{v},\mathbf{m})}$ represents all possible observations at the current state $\mu_{t+1|t}^{(k)}$. Intuitively, it could not be the same for all the particles at different positions. However, as professor told on piazza, if I introduce the assumption that $\sum_v\exp\left(\text{corr}\left(\text{lidar2world}\left(v,x_i\right),m\right)\right)=\sum_v\exp\left(\text{corr}\left(\text{lidar 2world}\left(v,x_j\right),m\right)\right)$, the sum over all possible lidar observations at two different particle locations $x_i$ and $x_j$ for the same map are equal. Professor also gave the reason:our map and observations (after transforming to world and finding the cells) are binary and even if the local map around the two different locations are different the sum of the errors with respect to all possible scans will be equal. After that, the update of the weights of each particle is simple:

$$p\left(z|x_k\right)=\frac{e^{f(z,x_k)}}{\sum_v e^{f(v,x_k)}}$$

$$\begin{aligned}\alpha_k'=\frac{p\left(z|x_k\right)\alpha_k}{\sum_\ell p\left(z|x_\ell\right)\alpha_\ell}&=\frac{e^{f(z,x_k)}\alpha_k}{\sum_v e^{f(u,x_k)}\sum_\ell\frac{e^{f(z,x)}(z_\ell)}{\sum_u e^{f(u,x_\ell)}}}\\&=\frac{e^{f(z,x_k)}\alpha_k}{\sum_\ell\frac{\sum_\nu e^{f(v,x_k)}}{\sum_u e^{f(u,x_\ell)}}e^{f(z,x_\ell)_{\alpha_\ell}}}\end{aligned}$$

$$\alpha_k'=\frac{\text{softmax}_k(\mathbf{c})\alpha_k}{\sum_\ell\text{softmax}_\ell(\mathbf{c})\alpha_\ell}$$

Based on these derivatives, I could safely update the weights of each particle by the ratio of $\text{corr}(y,m)$ and the sum of them.

- In order to improve the accuracy, I should add some small shift or rotation into each particle. The main reason behind this operation is that the probability of all possible

particles is enlarged and that really plays an important role for the success of the particle filter.

$$\text{corr}_{\text{new}}\ (\mathbf{z}, \mathbf{m}, \mathbf{x}) = \max_{\Delta\mathbf{x} \in D} \text{corr}(\text{lidar2world}(\mathbf{z}, \mathbf{x}+\Delta\mathbf{x}), \mathbf{m})$$

where $D$ is the set of small deviations for the particle pose $\mathbf{x}$. After finding the $corr_{new}$ value for each particle, I can change the particle positions to $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$. In practice, it could greatly improve the accuracy especially for Lidar4 dataset. On the other hand, I could simply increase the number of particles and it could provide with the same improvement.

*3) Particle filter Resample:* I need to compute $N_{eff}$ and if it is smaller than $N_{threshold}$, resample the particle set $\left\{\boldsymbol{\mu}_{t|t}^{(k)}, \alpha_{t|t}^{(k)}\right\}$ via stratified or sample importance resampling.

$$N_{eff} = \frac{1}{\sum_{k=1}^{N}\left(\alpha_{t|t}^{(k)}\right)^2} \leq N_{threshold}$$

In this project, I choose sample importance resampling method, and it is very for easy for implementation by python's function **np.random.choice**. There are two procedures:

- Draw $j \in \{1, \ldots, N\}$ independently with resplacement with discrete probability $\alpha_{t|t}^{(j)}$
- Add the sample $\mu_{t|t}^{(j)}$ with weight $\frac{1}{N}$ to the new particle set

Here, how I pick up such $N_{threshold}$ is very important. If I select a very large value, the requirement for resampling would be satisfied for most cases. It also comes with some other effects:it would converge to the b̈estp̈article very quickly. On the other hand, if I set it as a very small value, it would take much longer time to converge, but keep as many candidates as possible. This could work well if some sequence of sensor data are incorrect because it would allow to keep some b̈adp̈articles at current state, but indeed the correct ones. If $N_{threshold}$ is set too large, these b̈adp̈articles would be removed immediately and it would be less possible to find these particles.

### F. Flowchart

### G. Texture map

All the above sections describe how I implement particle filter to obtain the optimal robot's trajectory and occupancy map. And after that, I could use them and RGB and depth cameras to color the map. This section could also be decomposed into several tasks.

*1) Data synchronization:* This task is the same of synchronization task for particle filter, except the fact that this time, I synchronize the JointData with the images.

*2) Convert depth into X,Y,Z coordinates:* After reading each depth image from dataset, the first step is to convert each pixel

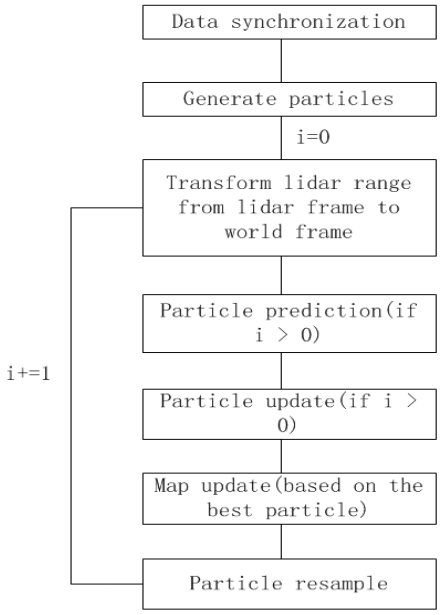


Fig. 2. Flowchart

value into $X, Y, Z$ coordinates. I rewrite the formula which could finish this:

$$X = \frac{(u - c_x) * Z}{f_x}$$
$$Y = \frac{(v - c_y) * Z}{f_y}$$

$u, v, f_x, f_y, c_x, c_y$ and $Z$ are known, I could obtain $X$ and $Y$ as shown above. Here I have to define $Y$ as the direction of the camera's principal axis instead of $Z$, which would not contradict to the representation of the world frame.

*3) Transform from camera frame to the world frame:* Using the same transformation method I describe before, I could transform the points from camera frame into world frame. Based on the assumption that the robot's center of mass keeps at 0.93m, any points in world frame with $z$ smaller than -0.93m are supposed to be considered as floor. I mark these points and record their $X, Y, Z$ coordinates in **camera frame**(Depth camera).

*4) Extract RGB values for points marked as floor:* Assume I already know the $X, Y, Z$ coordinates in the depth camera frame, I need to transform these points from depth camera frame into RGB camera frame. In the codes, the extrinsic matrix is given from the function *getExtrinsicsIRRGB*. With one step of transformation, these points could be represented in the RGB camera frame as $X_c, Y_c, Z_c$. Then, the next step is to find the corresponding pixel coordinates in the image plane:

$$\begin{bmatrix} \mu \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f & c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Here, $\mu$ and $\nu$ represent the row and column index in the image plane. I could extract the RGB value at $(\mu, \nu)$.

However, due to the camera's radial distortion, $\mu$ and $\nu$ I compute are not the real ones. The simplest effective model for radial distortion:

$$x = x_d \left(1 + a_1 r^2 + a_2 r^4\right)$$
$$y = y_d \left(1 + a_1 r^2 + a_2 r^4\right)$$

where $(\mu, \nu)$ are the pixel coordinates of distorted points and $r^2 = \mu^2 + \nu^2$ and $a_1, a_2$ are additional parameters modeling the amount of distortion. The $x, y$ are the ideal points with distortion effect removed. Hence, the more reasonable way is to extract the RGB values at $(x, y)$ instead of at $(\mu, \nu)$.

*5) Color the map:* Since I know which points are considered to be ground floor and their RGB values, then I simply set the value of corresponding grid cell on the occupancy map as these RGB values.

### *H. Done!*

Enjoy the results and the codes!

## IV. Results

### *A. Dead-reckoning*

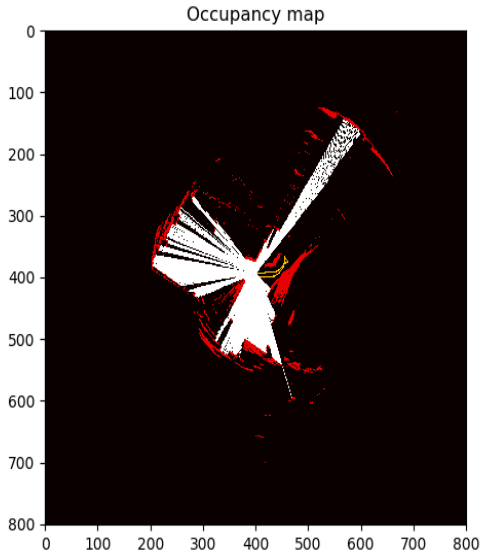


Fig. 3. Lidar0

From the above five figures, you could see the noise inside sensor data would make it impossible to build the map or localize the robot accurately.

### *B. Particle filter*

In the first, I didn't realize the importance of the number of particles on the final results. Besides, for the purpose of the computational efficiency, neither the shift nor the rotation have been added into the correlation function for each particle. I tried three particles and ten particles, but none of them works well: For each number, I have tried different sets of parameters but it doesn't work. Hence, I decided to increase the number of particles to 100 and this time, I obtained some good results. You could see the quality of the occupancy map is much better than the previous ones, which makes me strongly believe the number of particles plays the most important role in particle filter. And I think to add shift or rotation into the computation of correlation function indeed has the same effect:to add more diversity of particles(keep more effective candidates). Maybe if I have a unbelievably powerful computing machine, I could set the number of particles as infinity and that could

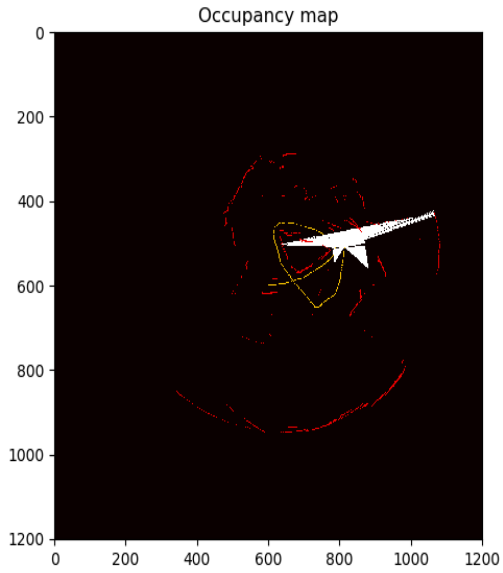


Fig. 4. Lidar1

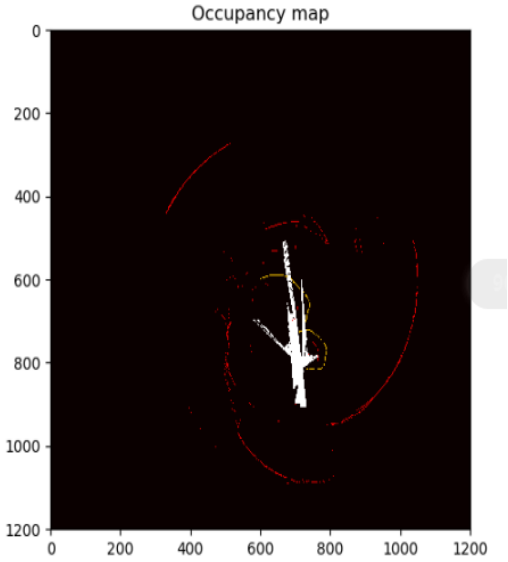


Fig. 5. Lidar2

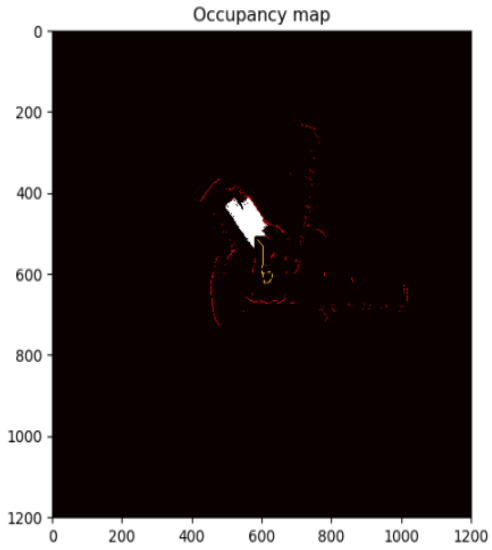


Fig. 6. Lidar3

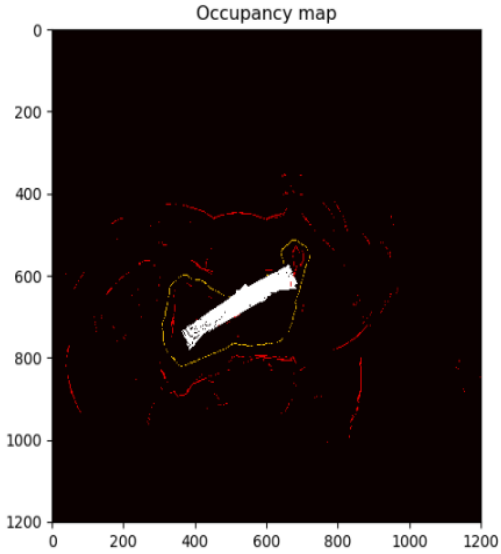


Fig. 7. Lidar4

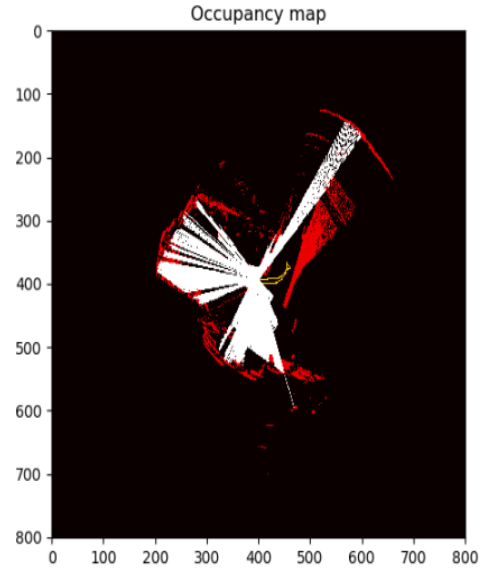


Fig. 8. Lidar0(three particles)

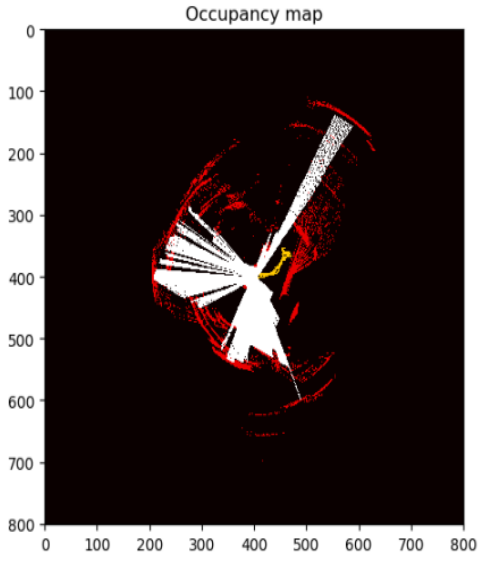


Fig. 9. Lidar0(ten particles)

provide with the 100% accurate solution as **bayes filter** could give(but it is not possible for implementation and that's why we have to make some linear Gaussian assumption or use some nonparametric methods). For the dataset2 and dataset3, I could also obtain some reasonable results. This means only 100 particles are in fact not enough even for some not so bad results in these two cases.

However, at the beginning, what I tried to do is to try noise with different magnitudes of noise. Indeed, it doesn't work no matter how I tune these parameters.

Right now, the possible reason I think is that the robot itself perform more worse in these four cases, which requires more particles. So there are two methods to overcome this problem:

- Increase the number of particles
- Add shift or rotation when computing the correlation function

However, due to the bad computing power of my laptop and limited time left to me before the due time, none of them are possible for implementation, which is a sad news.

### *C. Texture map*

I first obtained the whole occupancy map and the robot's trajectory, and then try to color the map by the depth and

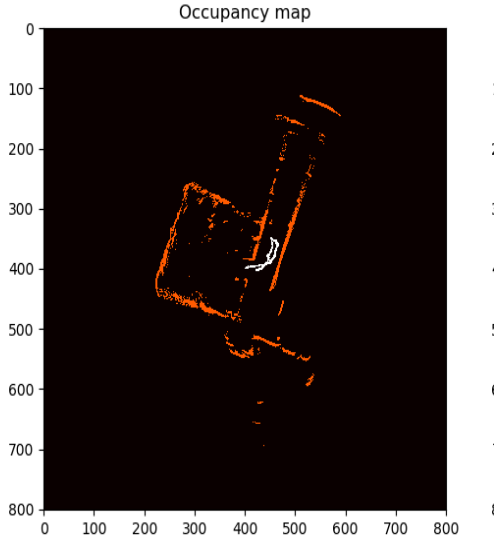


Fig. 10. Lidar0(100 particles)

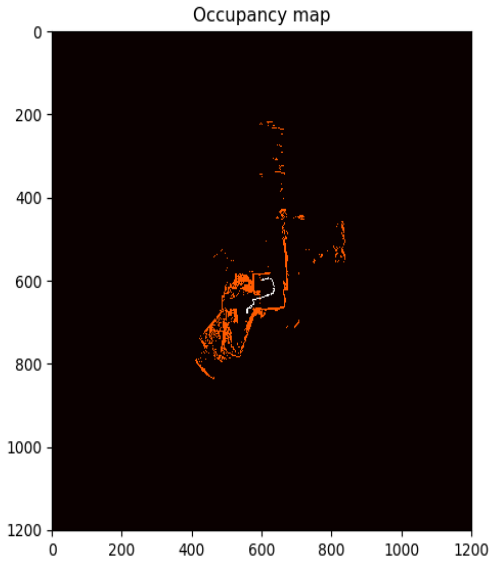


Fig. 12. Lidar3(100 particles)

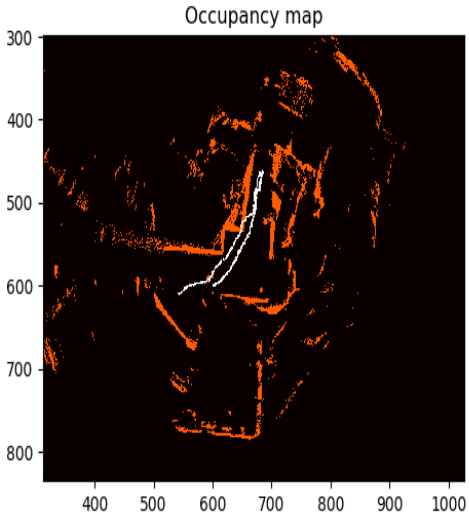


Fig. 11. Lidar2(100 particles)

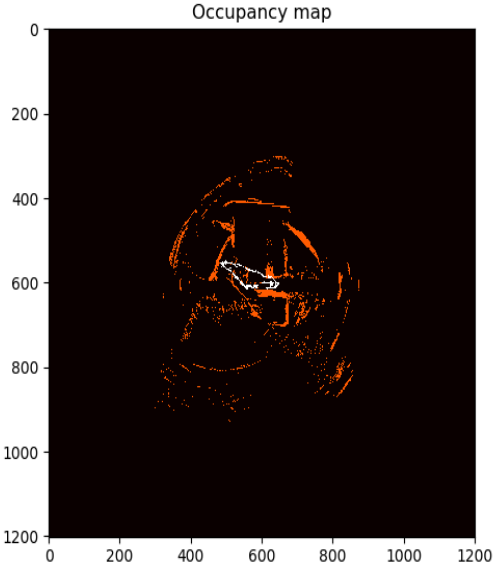


Fig. 13. Lidar1(100 particles)

RGB images. All the procedures I described detailly in the previous section, but my results are not accurate enough. I don't have time for extracting RGB values from the images, so I just pick up some ground candidates on the map. However, it seems something goes wrong in my codes since the result is not ideal.

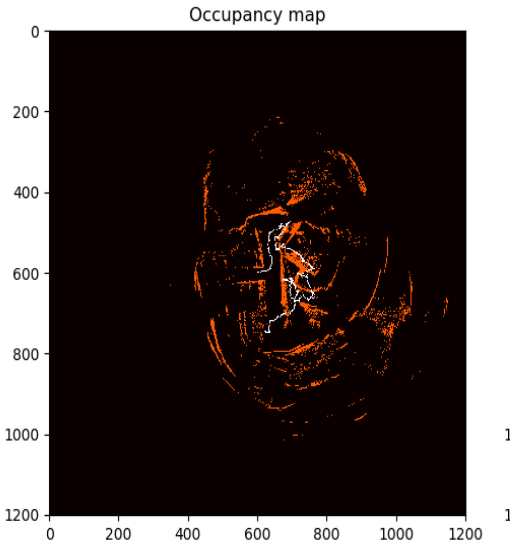


Fig. 14. Lidar4(100 particles)

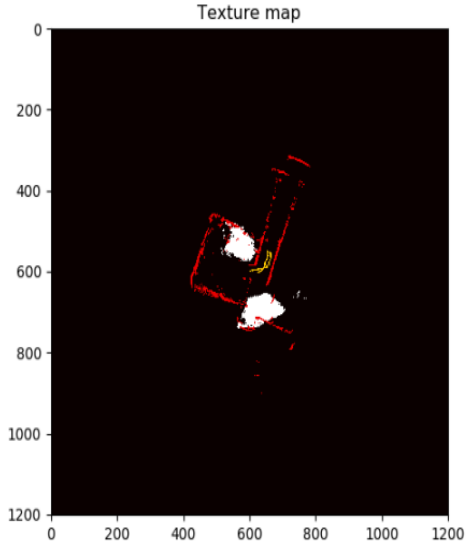


Fig. 15. Texture map for Lidar0

## V. Acknowledgment

I am grateful to the whole Python communities for providing us with so many powerful tools. Besides, I am grateful to all classmates who shared so many great ideas and posted them on piazza.

## References

[1] Sebastian Thrun, Wolfram Burgard and Dieter Fox, Probabilistic robotics, MIT Press, 647 pp