ECE 276A PR2: SLAM

Eric Megrabov (A12177906)

***Abstract*—The focus of this project is to implement Simultaneous Localization and Mapping (also known as SLAM) along with particle filtering by processing raw odometry as well as lidar data from a robot. This robot performs some movement and the objective is to map its environment while determining where the robot lies in that environment.**

I. Introduction

As the name implies, the purpose of Simultaneous Localization and Mapping is to develop and update a map of an environment while at the same time trying to determine where the robot is in that map. The problem lies in the fact that it is difficult to know where the robot is without knowing the map, and it is difficult to know the map without knowing where the robot is. As a result, the use of poses must be incorporated (also known as hypotheses of the robot's location) in order to do a chain of prediction and updating until convergence is found. SLAM is extraordinarily useful for a plethora of robotics tasks, such as autonomous or self-driving cars, ocean mapping research, drone swarms, and many others.

II. Problem Formulation

In this project, the input is provided as raw measurements from the robot called THOR. This robot has corresponding sensor information in the form of lidar scans, head joint angles, relative delta poses, and RGBD data. RGBD will be discussed only briefly as it is useful for texture mapping, which was not explored in the scope of this project. The data is outlined in depth below.

- Center of mass at 0.93 meters
- Head at 0.33 meters above center of mass
- Lidar sensor at 0.15 meters above head
- Kinect sensor at 0.07 meters above head

- Lidar sensor reads between 0.1 to 30 meters over a -135 to 135 degree scan range with a resolution of 0.25 degrees between readings. This yields 1081 measurements per scan in polar coordinates. These are coupled with a corresponding time at which each scan occurs
- A set of times corresponding to head and neck angles (yaw and pitch, respectively) for the robot head
- Odometry given in the form (x, y, theta), where (x,y) is a pose in the world frame in the x-y plane, and theta is the yaw of the robot. The robot has no pitch or roll
- Kinect V2 sensor that provides RGB images at resolution 1920x1080 and depth images at resolution 512x424

The objective is to determine where the robot is and determine a map of the environment. This can be approached by maximizing the joint distribution:

$$p(\mathbf{x}_{0:T}, \mathbf{m}, \mathbf{z}_{0:T}, \mathbf{u}_{0:T-1}) = \underbrace{p_{0|0}(\mathbf{x}_0, \mathbf{m})}_{\text{prior}} \prod_{t=0}^{T} \underbrace{p_h(\mathbf{z}_t \mid \mathbf{x}_t, \mathbf{m})}_{\text{observation model}} \prod_{t=1}^{T} \underbrace{p_f(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{u}_{t-1})}_{\text{motion model}}$$

The priors $\mathbf{p}_{0|0}$, observation model $\mathbf{p}_h$, and the motion model $\mathbf{p}_f$. The joint distribution **p** is over the robot states, observations, controls, and the map of the world. The variables in the probability mass function are as follows:

- $x_{0:T}$: defines robot state at steps 0 to T
- m: represents the map of the world
- $z_{0:T}$: defines observations at steps 0 to T
- $u_{0:T-1}$: defines control inputs at steps 0 to T-1

III. Technical Approach

The project code was written using Python 3, mainly in Jupyter Notebook.

A. Data Processing

Since most of the data is given in raw format, some pre-processing needed to be performed. First, I threshold all of my lidar scans to be in the range 0.1 to 30 meters, as this is what the specifications of the lidar sensor state in the specifications. If a scan distance is outside of this range, I simply remove it from the dataset. In addition, one set of times corresponds to a set of lidar scans while a different set of times corresponds to a set of joint angles for the head. In order to perform a lidar-to-world transformation, which is necessary to perform SLAM, I first found which joint angle times were closest to the lidar scan times and created a new dataset that maps the joint angles to the lidar scans that occur around the same time. For the lidar scans, it is necessary to convert them from polar to Cartesian coordinates. This is performed by using the following formula:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$$

Since these scans are performed in the lidar frame, it is necessary to convert them from lidar frame to head frame, head frame to body frame, body frame to world frame so that it can be properly used for SLAM. The transformation matrices are as follows:

$$rotation\ matrix\ R = R_z R_y R_x =$$
$$\begin{bmatrix} cosz & -sinz & 0 \\ sinz & cosz & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cosy & 0 & siny \\ 0 & 1 & 0 \\ -siny & 0 & cosy \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cosx & -sinx \\ 0 & sinx & cosx \end{bmatrix},$$

$$translation\ vector\ p = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \text{and}\ \ transform\ T = \begin{bmatrix} R & p \\ 0^T & 1 \end{bmatrix}$$

Here, the lidar to head transformation is a simple translation downwards.

The head to body transformation has a rotation in yaw and in pitch. For this application, the neck angle corresponds to yaw (z) and the head angle corresponds to the pitch (y). There is no change in roll so R_x is identity. The *p* vector is another translation downwards, similar to the lidar to head transformation.

Lastly, the body to world transformation is performed by using the robot's absolute pose (obtained by creating a running sum of the delta poses) in the form (x, y, theta). The only nonidentity matrix is R_z which uses the robot's yaw theta. P is $(x, y, 0.93)^T$ since the robot is 0.93 m off of the ground. The ultimate transformation for each time step lidar scan is:

$$scan' = {}_{world}T_{body} * {}_{body}T_{head} * {}_{head}T_{lidar} * scan$$

Where *scan* is the vector *(a, b, c,* **1***)* that we want to transform to the world frame. After obtaining this transformed scan vector, I remove any vector in the data that is close to the ground. This is done by deleting any vectors that have a *c* value less than 0.01 meters. At this point, the data is fully processed and ready to be used for SLAM.

B. Mapping

Now that the data is clean and converted for use, it is possible to create a map of the environment in the form of an occupancy grid. This grid is of the form 25x25 square meters. The distance between each adjacent cell in the grid is uniformly 0.05 meters. This grid is large enough to accommodate all of the datasets without causing data to go outside of the boundaries given by the map. In this map, I chose each cell to be modeled by a Bernoulli random variable as discussed in class. The probability that a cell is occupied by the robot is said to be γ while the probability that it is not occupied is 1- γ. To create a log-odds map, which will be useful for determining whether or not a cell is occupied in the map, it is necessary to add the odds ratio to each cell in which the lidar scan hits an object, and to subtract the log odds ratio from each cell in which the lidar scan passes through. For cells that do not interact at all with a scan, they are left untouched. The cells that lidar scans pass through are determined by using cv2.line, which is similar to Bresenham2D. This means that over time, it is possible to estimate the probability density function of a given cell by simply accumulating the log-odds ratio at each time step. The log-odds ratio as well as the log-odds can be seen below.

$$\begin{aligned} o(m_i \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t}) :&= \frac{p(m_i = 1 \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t})}{p(m_i = -1 \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t})} = \frac{\gamma_{i,t}}{1-\gamma_{i,t}} \\ &= \underbrace{\frac{p_h(\mathbf{z}_t \mid m_i = 1, \mathbf{x}_t)}{p_h(\mathbf{z}_t \mid m_i = -1, \mathbf{x}_t)}}_{g_h(\mathbf{z}_t \mid m_i, \mathbf{x}_t)} \underbrace{\frac{\gamma_{i,t-1}}{1-\gamma_{i,t-1}}}_{o(m_i \mid \mathbf{z}_{0:t-1}, \mathbf{x}_{0:t-1})} \end{aligned}$$

$$\begin{aligned} \lambda(m_i \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t}) :&= \log o(m_i \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t}) = \log\left(g_h(\mathbf{z}_t \mid m_i, \mathbf{x}_t) o(m_i \mid \mathbf{z}_{0:t-1}, \mathbf{x}_{0:t-1})\right) \\ &= \lambda(m_i \mid \mathbf{z}_{0:t-1}, \mathbf{x}_{0:t-1}) + \log g_h(\mathbf{z}_t \mid m_i, \mathbf{x}_t) \\ &= \lambda(m_i) + \sum_{s=0}^{t} \log g_h(\mathbf{z}_s \mid m_i, \mathbf{x}_s) \end{aligned}$$

In addition, I also constrained the log odds of each cell to have a minimum and maximum value if -10 and 10 respectively. This was in order to ensure that there would not be many estimates that were overly confident, which should lead to a more accurate map.

As part of hyperparameter tuning, I experiment with many values of γ, but ultimately I chose 0.8 to be the one for building my log-odds map as it yielded good results. This means that on every scan, I add log(4) to each occupied cell and I subtract log(4) from each unoccupied cell. From here, I am able to use my log-odds map to build an occupancy grid. Originally, I did this by simply assigning any value in the grid greater than some threshold to be 1, and any value less than some threshold to be 0. For the sake of plotting, the occupancy grid has 3 values:

$$OccPlot(i) = \begin{cases} 0\ if\ unobserved \\ 1\ if\ occupied \\ -1\ if\ unoccupied \end{cases}$$

When used for map correlation, the occupancy grid is

$$Occ(i) = \begin{cases} 0\ if\ unobserved/unoccupied \\ 1\ if\ occupied \end{cases}$$

The occupancy map will be useful in determining the best particle in the particle filter. However, I found that it is more useful to recover the probability mass function for each cell by using

$$\gamma_{i,t} = p(m_i = 1 \mid \mathbf{z}_{0:t}, \mathbf{x}_{0:t}) = 1 - \frac{1}{1 + \exp(\lambda_{i,t})}$$

This allows me to threshold on a probability value rather than arbitrarily at a guessed threshold. If the probability is greater than 90%, I count it as occupied. If it is less than 10%, I count it as unoccupied. Otherwise, I assume that a conclusion cannot be made.

C. Localization

In order to perform localization, it is possible to use a Bayes filter along with the Markov assumption to create a particle filter. In the prediction step,

$$p_{t+1|t}(\mathbf{x}) \approx \sum_{k=1}^{N} \alpha_{t+1|t}^{(k)} \delta\left(\mathbf{x}; \boldsymbol{\mu}_{t+1|t}^{(k)}\right)$$

Note that these summation terms are mixtures delta functions for the states given a pose, where α represents particle weights and μ represents particle poses for the particle filter. This will update the particles with the next known delta pose with noise added. After prediction, we perform the update step.

In the update step,

$$p_{t+1|t+1}(\mathbf{x}) = \sum_{k=1}^{N} \left[\frac{\alpha_{t+1|t}^{(k)} p_h \left(\mathbf{z}_{t+1} \mid \boldsymbol{\mu}_{t+1|t}^{(k)} \right)}{\sum_{j=1}^{N_{t+1|t}} \alpha_{t+1|t}^{(j)} p_h \left(\mathbf{z}_{t+1} \mid \boldsymbol{\mu}_{t+1|t}^{(j)} \right)} \right] \delta \left(\mathbf{x}; \boldsymbol{\mu}_{t+1|t}^{(k)} \right)$$

The update state will allow us to update the robot position by using the particle with the highest laser correlation of the particles which have random noise sampled from a Gaussian distribution. After performing update, we will again perform predict. These combined steps being repeated together over timesteps will help obtain particle locations and weights.

To begin the algorithm, I initialize a set of N particles (for most of my data sets, this will be 50). Each of these particles have a corresponding pose (x, y, theta) in the world frame, obtained from using delta poses provided by the odometry. After much analysis, it seemed as though there was very little noise in the odometry for the (x, y) position of the particle, but the provided yaw angle was somewhat inaccurate. To start, I spawned all N particles at (0, 0, 0) which is the origin in the world frame, obtained a log odds map and occupancy grid using the initial lidar scan that was converted to the world frame, initialized each particle weight to be 1/N, and began the SLAM algorithm. For further time steps, I added the corresponding delta pose at that timestep to each particle (predict step), and I added some amount of noise to the x and y coordinates as well as a smaller amount of noise to the theta to account for inaccurate measurements. Originally, I had only used noise in theta, but to further ensure that I obtained a proper value for theta, I created a fixed range of theta values from -3 degrees to 3 degrees with a resolution of 0.25. From here, I added each possible theta value to a given particle, calculated the map correlation using that particle with all theta values, and only kept the one had the highest correlation. Given this, the SLAM algorithm combined with the Bayes particle filter will select which of these particles is *most likely* to be the correct robot position. Note that this map correlation is a grid of size 9x9 surrounding a given particle. This correlation was calculated using the following equation:

$$\mathbf{corr}(y, m) := \sum_{i} \mathbb{1}\{m_i = y_i\}$$

where y_i is a transformed scan and m_i is a cell in the occupancy grid. Thus, I would overwrite the inaccurate theta measurements with my newly obtained ones. After repeating this for all particles, I perform a softmax operation on the list of correlations, multiplied with the particle weights, and normalized. For the softmax function, it was necessary to use a different form for softmax due to numerical conditioning:

$$softmax(z) = softmax(z - max_i z)$$

After doing so, the particle that had the highest weight was used for updating the log odds map. The best particle is recorded and plotted on the occupancy grid to indicate the robot's trajectory over time, which completes localization for a given time step.

At this point, I also check to make sure that I have enough effective particles (in other words, avoiding particle depletion) to make sure that my algorithm performs well. If this number goes below 70%, I resample the particles using stratified resampling. The higher the threshold value, the more often resampling will occur since a high threshold means that the algorithm requires a large number of effective particles. This makes sure that the algorithm does not simply put too much weight on a single particle over time, but rather splits it up into multiple particles around the same point. The reason I used stratified resampling over other strategies is because theoretically, it guarantees that really small weighted samples appear at most once while really high weights must appear at least once, thus preserving the integrity of the data. This threshold was obtained as follows:

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

From here, the log odds map is recalculated using the newly determined best particle, which takes the algorithm back to the mapping step. This is SLAM and is repeated for all given time steps until convergence occurs.

As a sanity check, I also include dead reckoning to compare against SLAM. Dead reckoning assumes that the odometry readings are perfectly accurate with no variation in any of the measurements. Theoretically, removing all noise and using a single particle in SLAM should replicate dead reckoning.

Unfortunately, performing so much processing causes extreme slowdown. This makes the runtime of the SLAM algorithm take several hours per data set. To combat this, I chose to downsample, in which I use every 50th time step to process rather than every single one. I did this by calculating the difference in odometry pose from time t to time t-50 and using that as my new delta pose. This sped the algorithm up very much while preserving similar results.

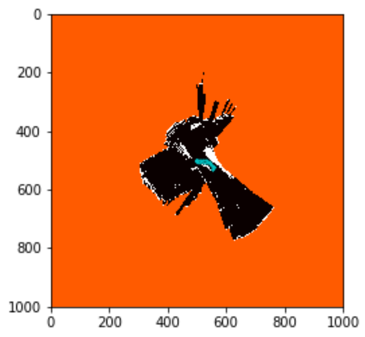
IV. Results

Here, I will discuss the outcome of my dead reckoning results as well as my SLAM results. Please note that all plots are flipping along the Y-axis due to the functionality of Python's plt.show function. I will first showcase my dead reckoning results for each dataset followed by my SLAM results for each dataset with a no-sweeping yaw variation strategy. Lastly, I will display my SLAM results using the algorithm which sweeps over several yaw variations.

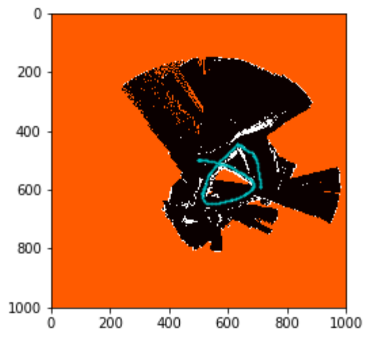
A. *Dead Reckoning*

The dead reckoning results can be seen below.

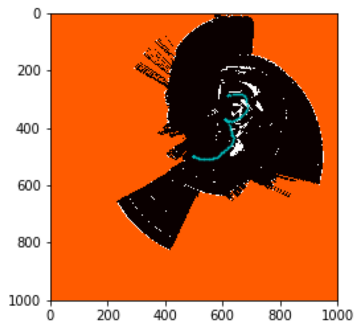
Dataset 0



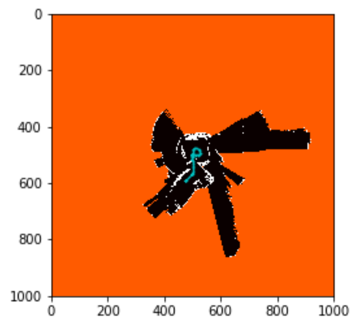
Dataset 1



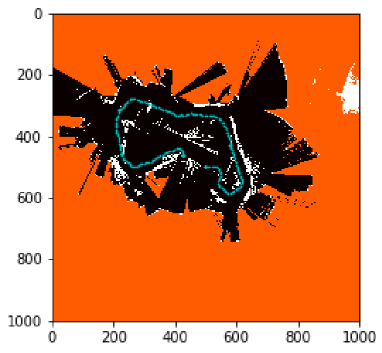
Dataset 2



Dataset 3



Dataset 4

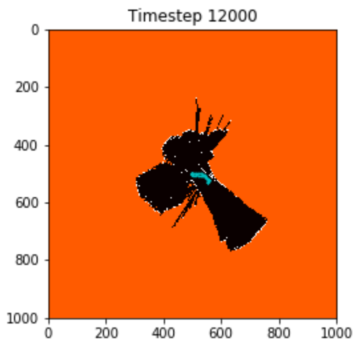


Notice how the dead reckoning results have a very large amount of rotational noise, even for the cleanest dataset, 0. In addition, the occupied areas are very thick, which is a result of setting a threshold between -10 and 10 in the log-odds minimum and maximum values for the sake of avoiding overly confident estimates. In the next section, we will explore if there was any improvement to this in SLAM.

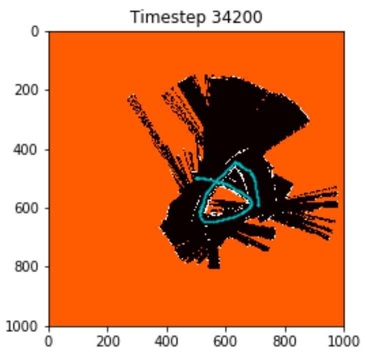
B. *SLAM: Sanity Check – Dead Reckoning Approximation*

The first results I obtained were used as a sanity check to ensure that my SLAM works properly. I created only one particle, and did not add any noise. Theoretically, this should approximate SLAM, which it successfully did. This indicates that the SLAM algorithm works correctly. These results are shown below.

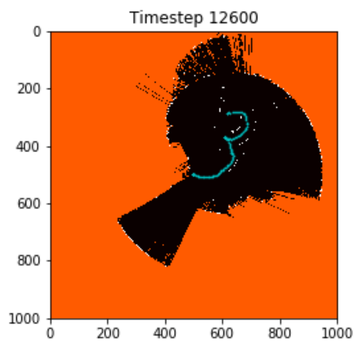
Dataset 0



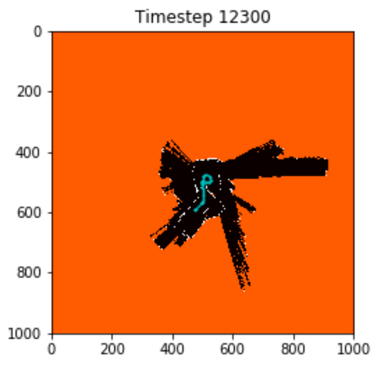
Dataset 1



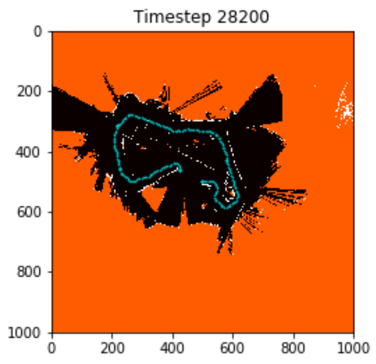
Dataset 2



Dataset 3



Dataset 4



By inspection, it is clear that these results are very similar to the dead reckoning results. The most notable differences are that the occupied points (in white) are much more prevalent in dead reckoning than in my SLAM with no noise and one particle. This might mean that SLAM is under-confident in those particle estimates. In general, there should not be large connected blobs of occupied zones in the SLAM results, which is a good sign. In addition, another difference is that SLAM with no noise and one particle provided much less rotational noise than dead reckoning did for dataset 1. Overall, these results indicate that SLAM is working as expected for the base case.

C. SLAM: No Varied Yaw

After establishing that the SLAM algorithm has generally correct functionality with no noise and one particle, I tuned hyperparameters with a focus on obtaining a good result from dataset 0. The hyperparameters tuned were:

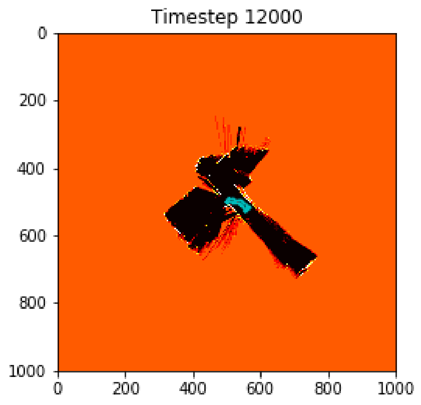
- log odds ratio: log(0.8/0.2) = log(4)
- log odds threshold: -10 to 10
- map pmf confidence threshold (Occ(i) shown in technical approach):
    - >0.9 → 1, < 0.1 → 0, in between unchanged
- Noise in x coordinate of robot: 1e-2
- Noise in y coordinate of robot: 1e-2
- Noise in theta yaw angle of robot: 1e-5
- Number of particles: 50
- Downsampling frequency: 25 (ie this skips 25 steps at a time)
- Resampling threshold: 70% of particle count

Note that these were all determined experimentally. For this strategy, I do not sweep over a range of thetas for each particle; the strategy in which I do will be discussed in the next section. Changing the log odds ratio did not affect the results very much. The log odds threshold significantly filtered out overly confident estimates. Increasing this threshold allowed more to remain while decreasing it got rid of many. The map pmf confidence threshold allowed for less rotational noise when using a stringent threshold like 0.9 as opposed to 0.7 and thus improved results. Adding noise in x and y coordinates for the robot allowed the particles to disperse further from each other and helped SLAM find better estimates of the true robot position. When the noise is too large, the trajectory becomes jagged and disconnected, but when it is small, it follows too closely to the inaccurate odometry readings. Adding too much noise to theta causes huge rotational artifacts, but too little noise seems to have a tendency to approximate dead reckoning. Increasing the number of particles certainly improves accuracy, but it also decreases runtime significantly. Downsampling by too large of jumps causes SLAM to lose information on the localization and mapping, but too little causes the runtime to be prohibitively long. Lastly, the resampling threshold will resample very often if the threshold is set to something high, like 95% of the particle count, or less frequently if the threshold is set very low to 5%. The most time-consuming part of this project was homing in a combination of these hyperparameters that would provide the cleanest looking results.

The plots without the sweeping theta hyperparameter can be seen below. The values that I plot are not thresholded to be -1, 0, and 1 but rather the true values that correspond to the probability mass function. The closer to white a point is, the more likely it is to be occupied. The closer to black a point is, the less likely it is to be occupied. For example, darker shades of red and orange are closer to black while yellow dots are closer to white.

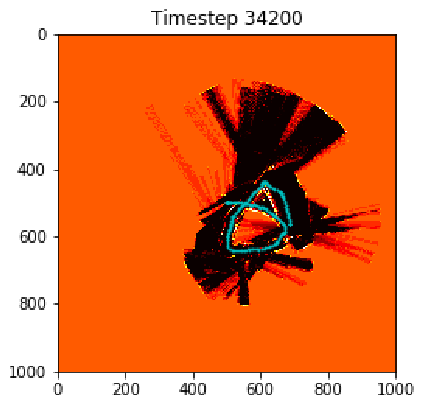
Note: The following are the results on the *final iteration* of each dataset, but I have included five .gif files with my submission that displays a range of iterations corresponding to each dataset. Please look at these files if you would like to see the evolution of SLAM over time.

Dataset 0



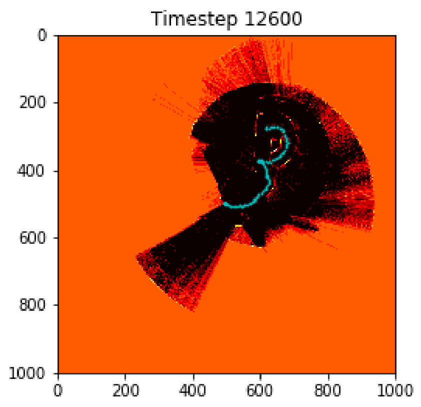
The result for dataset 0 has a very significant improvement over dead reckoning. The rotational noise is almost entirely gone, and is only midly visible at the ends of the hallway facing diagonally down and to the right. The algorithm is able to map the occupied points that make up the doorways on the left and right sides (around [500, 500] and [575, 475]. This indicates that it was able to properly map the environment since it detected walls nearby, two openings into rooms, and a straight hallway. In addition, the trajectory closes the loop, showing that the movement of the robot is also behaving correctly.

Dataset 1



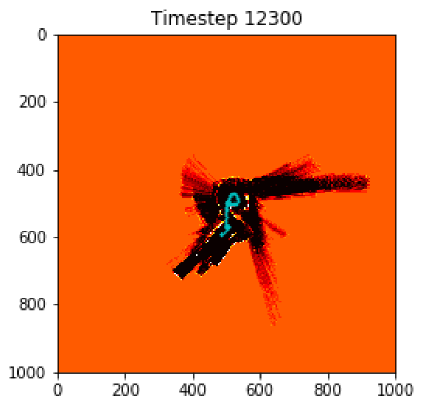
SLAM on dataset 1 shows good improvement over dead reckoning in that it reduces a great deal of the rotational noise on the upper left side and the right side, but there is still rotational noise visible on the top. It also does a decent job of detecting that the middle of the image is unobserved (orange island). The trajectory does not quite close the loop, which could be caused by too drastic of a change in the particle position as well as misdirection due to incorrect theta. Generalizing to all datasets with one set of parameters is quite difficult.

Dataset 2



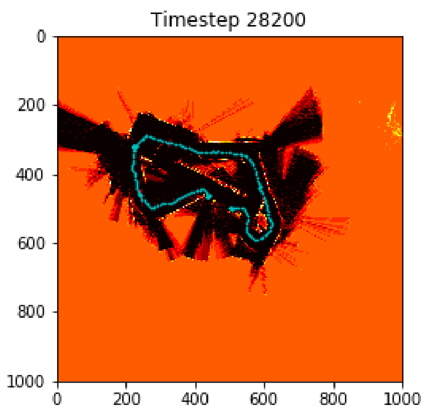
Here, SLAM begins to break down. While not quite as bad as the dead reckoning case, there is still far too much rotational noise present. Some occupied cells are detected, but not as much as expected. This is again likely due to difficulty in generalizing hyperparameters. This could also be due to downsampling as there seems to be a very sharp turn at a high rate of speed in this dataset.

Dataset 3



There is some reduction in rotational noise, especially in the downwards direction. A general mapping is obtained with relative accuracy. In addition, the trajectory seems to be overall correct after comparing to the Kinect RGB images in the provided dataset.

Dataset 4



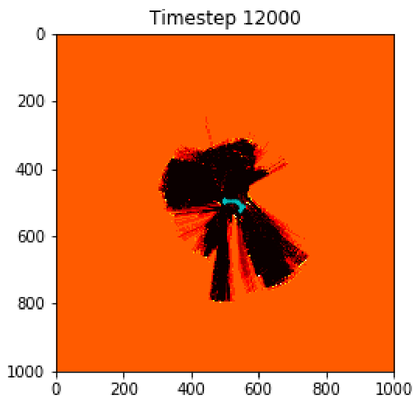
Here, SLAM is very close to completing the loop but does not quite make it to the end. In addition, it is able to detect that the region enclosed by the trajectory is unobserved. Although not very clean, it still performs much better than the dead reckoning results, which do not even detect that the middle area is unobserved. It is necessary to account for sharp turns with better hyperparameter tuning.

Overall, my SLAM yielded significant improvements over dead reckoning. Although the results for datasets 1-4 are not ideal, they still show promise that with individual hyperparameter tuning, their results might be cleaned up significantly. Since I overfit to dataset 0, it is to be expected that SLAM does not yield perfect plots on other datasets.

D. *SLAM: Varied Yaw*

In this strategy, I add a series of yaw values between -3 to 3 degrees with a resolution of 0.25. This creates 19 additional theta values to explore. As a result, slam takes about 19 times longer to execute. Therefore, I was unable to perform this strategy for all datasets. The result shown below is for dataset 0.

Dataset 0



This unfortunately did not form any improvement over dataset 0. Although theoretically it should have decreased rotational inaccuracies, it did not seem to help for this case. It is possible that given more time, it would be feasible to run it on other datasets and observe improvement.

E. *Conclusion*

It is clear that SLAM with Bayes particle filter is extremely sensitive to hyperparameter tuning. There are many pros and cons to setting any given hyperparameter to high or low values. Ultimately, generalizing a set of hyperparameters to be used across more than one dataset is quite difficult due to the delicate nature of the strategy described. While I have exhausted several techniques to get good results such as hyperparameter tuning, variation in yaw, resampling, downsampling, and others, there are still other techniques to try, such as different filters or fitting to different datasets. This project has also highlighted the long runtimes that are required in order to run simultaneous localization and mapping. Without downsampling or reducing particle count, the runtime of this project is prohibitively long. In summary, my algorithm worked well for slow movement and smooth turning, but did not generalize well to situations in which the robot has rapidly changing yaw or position.

Acknowledgment

I used Professor Atanasov's slides extensively throughout the report as well as the TA's office hours. Thank you for the excellent resources provided throughout the duration of this project.