Project #2

Sensing & Estimation in Robotics

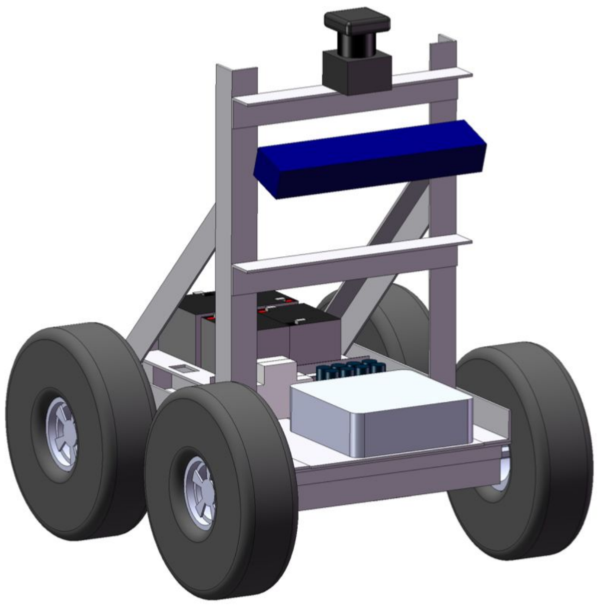
PARTICLE FILTER SLAM

Arshia Zafari
(A11167578)
ECE 276A
2/22/19

INTRODUCTION

The way intelligent systems perceive and understand the world around them has been an ongoing advancement in robotic technology, some of the key developments being the adoption of filtering techniques for sensing and localization. Simultaneous Localization and Mapping (SLAM) is a process by which a robot constructs a map of its unknown environment while simultaneously keeping track of it's own location within the map. One such method of SLAM has been the representation of the map under probabilistic techniques such as a particle filter. Particle filtering involves using random samples to represent a posterior distribution of a specific location in a map for the robot to judge where it is based on observations.

This report will explore the SLAM application with particle filtering using odometry, 2D laser ranges, and RGBD measurements from a differential-drive robot shown in **Figure (1)**. Using two different training sets as well as a test set, we will take readings from the IMU at 100Hz, encoder at 40Hz, and LIDAR at 40Hz to localize the robot and build a 2D occupancy grid map of the environment. We will analyze the map using strictly the sensor measurements ie. dead reckoning and compare the mapping with SLAM using different particle set sizes. Finally, RGBD information will be used from a Kinect disparity camera to color the floor of this map.



Figure(1). Differential-drive robot

PROBLEM FORMULATION

The robot may not always behave in a perfectly predictable way, so it generates random guesses of where it is likely to move. Therefore, each particle contains a full description of a possible future state and when the robot observes the environment, it discards particles inconsistent with this observation, and resamples them to be more closer to those which appear consistent. In this way, most particles will converge to where the robot actually is.

For this application, we will represent each particle as a delta function ($\delta_t = 1$) for the posterior probability of a location in the map. We will predict the position of the robot using a motion model of a system with differential drive, shown in **Eq.(1)**. Our motion model will give us the predicted "pose" or the x, y, and orientation θ (0-2π radians) using the input measurements from the encoder (the velocity v) and the yaw rate of the IMU (ω).

$$\begin{aligned} x_{t+1} &= x_t + \tau(v_t sinc(\tfrac{\omega_t\tau}{2}) cos(\theta_t + \tfrac{\omega_t\tau}{2})) \\ y_{t+1} &= y_t + \tau(v_t sinc(\tfrac{\omega_t\tau}{2}) sin(\theta_t + \tfrac{\omega_t\tau}{2})) \\ \theta_{t+1} &= \theta_t + \tau(\omega_t) \end{aligned} \quad \textbf{(1)}$$

The process of predicting our position involves applying the motion model individually to each particle μ as in **Eq.(2)**:

$$p_{t+1|t}(x) = \sum_{k=1}^{N_{t|t}} \alpha_{t|t}{}^{(k)} p_f(x \mid \mu_{t|t}{}^{(k)}, u_t) \quad \textbf{(2)}$$

where α specifies the weight of the particle. The weight is a characterization of how correlated the particle is compared to our current map. The process of updating in **Eq.(3)** corrects the predicted robot pose based on observed obstacles and their correlation to the current grid cells. The observations from LIDAR measurements are given by the distances to obstacles in meters in a field of view from -135° to +135°.

$$p_{t+1|t+1}(x) = \sum_{k=1}^{N_{t+1|t}} \frac{\alpha_{t+1|t}{}^{(k)} p_h(z_{t+1} \mid \mu_{t+1|t}{}^{(k)})}{\sum_{j=1}^{N_{t+1|t}} \alpha_{t+1|t}{}^{(j)} p_h(z_{t+1} \mid \mu_{t+1|t}{}^{(j)})} \delta(x; \mu_{t+1|t}{}^{(k)}) \quad \textbf{(3)}$$

Here we apply our observations to update the weights of each particle and using the particle with the highest correlation, we can build the map for that specific time epoch. In cases where the weights of the particles fall too low, we resample the particles by renormalizing their weights and updating their positions to the best particle. Once we have the map we can add texture to it by transforming a pixel from the image space to the world with standard optical transformations.

TECHNICAL APPROACH

SLAM

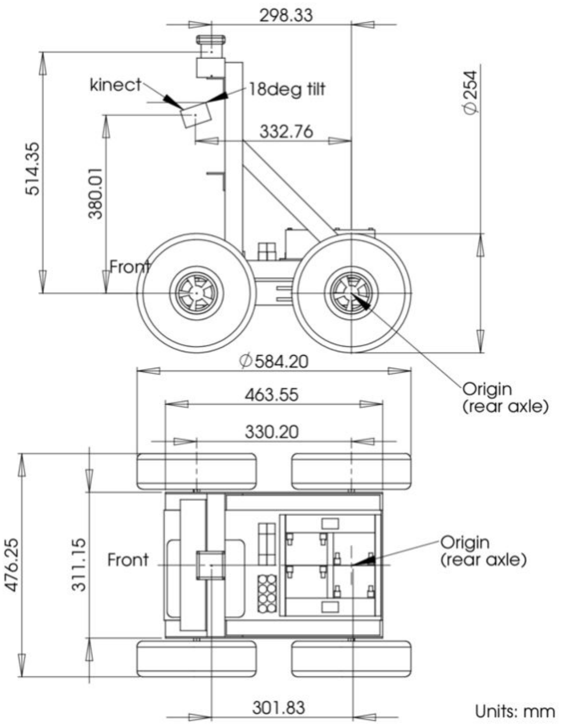
To begin SLAM, we must understand the transformations from our sensors to the world frame. This involves transforming from our sensor to the robot body and transforming the body to the world. Our model observes the world using the LIDAR scans, so the transformation starts at the LIDAR. Using the configuration in **Figure(2)**, we can see the LIDAR is about 0.133m from the center of the robot in the x direction (despite the origin being at the rear axle, we will set the origin at the dead center of the robot between the wheels). This gives us a simple negative displacement for the transformation to the robot frame. The transformation to the world frame is a bit more involved as it includes both a rotation and a displacement. Each coordinate in the world frame can be represented as in **Eq.(4)**:

$$s_W = R\, s_B + p \tag{4}$$

where p is the displacement of the body and R is given by the standard 2D rotation at current orientation θ in **Eq.(5)**:

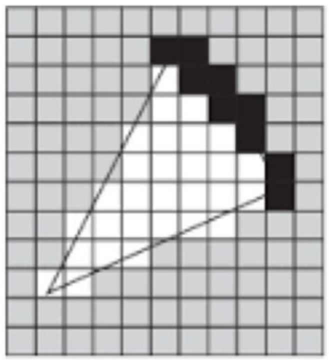
$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{5}$$

With these transformations we are able to implement the prediction step and verify the motion model by plotting the trajectory of the robot given the encoder readings and the IMU yaw rate. The encoders give us the displacement of the robot by the number of ticks from each wheel, each tick representing 0.022m. For every time step, we look at the left and right side, take the average of the front and rear wheels to get the distance travelled on both sides. Dividing each distance by the time step, we get the velocity from each side and we average them once more to get the linear velocity of the entire robot frame.



Figure(2). Robot configuration

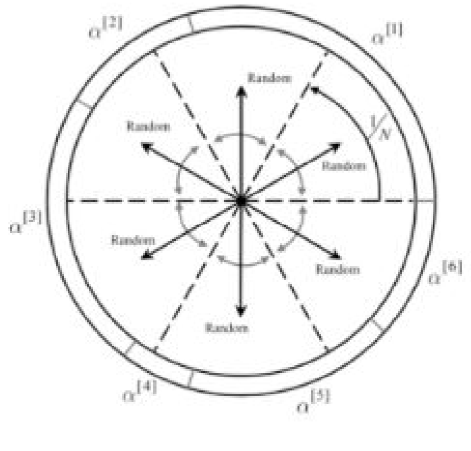
Because the data measurement comes at different timesteps, we must find a way to utilize the readings in a synchronous way. The heart of the motion of the robot comes from the encoders, so we will use the encoder timestep as our reference. Since the IMU data is sampled at a much higher frequency, we have a lot more IMU readings than encoder readings (100Hz vs 40 Hz), so for simplicity we can take the IMU reading to with the nearest timestep to that of the encoder; the accuracy might not be perfect, but at the frequency of data it does the job. The same is applied for the LIDAR measurements. Using the sensor measurements in this way we can predict the location of the robot for the next time iteration with some Gaussian noise added, because our predictions are not always perfect. The IMU data was initially sent through a 10Hz low-pass Butterworth filter to remove the extreme frequency vibrations. After, there was about $0.01 * N(0, 1)$ of white noise added back to the yaw readings (equivalent to 0.5° of angle speed noise) to model a more consistent imperfection of the prediction model.



Figure(3). Occupancy grid

During the update step, we use the particles to localize the robot and measure the correlation to the current map of the environment. Our map will be of size 1600x1600 with a resolution for 0.05 meters per cell. As mentioned before, the weight is a characterization of how correlated the particle is compared to our current map. We will be updating the weights of the particles using the one with the highest correlation with respect to the current map and the next LIDAR observation within a 9x9 grid. Conventionally, it is common to convert the map into a binary grid for occupied cells and calculate the correlation off a softmax of a binary representation, but here we will use the log-odds map to compute the correlations and scale the weights of the log-odds correlation exponentially (the weights are still scaled the same).

Once we select the best particle, we use its position to map the observations. This is done by estimating the probability density function of the map conditioned on the observation, which is equivalent to incrementing each cell of lidar readings with a log-odd ratio. This log-odds ratio gives us the confidence of the occupancy measurement of a specific cell. Here we implement a log-odds value of log(80/20) = log(4) meaning we are 80% confident in the occupancy or non occupancy for a specific cell in the map based on our observation. For simplification in calculations the log-odds is discretized and converted to an integer, so the value is really just 1. In **Figure(3)** we can see an example of the occupancy grid from a line of sight from the robot to the corresponding LIDAR hits. We will use the "bresenham2D" function to index the ray of cells between the robot and a specified endpoint of a LIDAR reading. With this function we will increment the endpoint of the ray with the LIDAR reading using our log-odds value and decrement the cells in between. With an upper log-odd threshold of 100 and a lower threshold of -50, the observations are mapped.



Figure(4). Stratified resampling

After the map is created for that time epoch, we will implement resampling to prepare for the next time epoch. For situations where the weights of these particles become too insignificant, we resample them using the process of "stratified resampling." Stratified resampling guarantees that samples with large weights appear at least once and those with small weights appear at most once as in **Figure(4)**. This way, particles that are more representative will be used to map the environment. **Eq.(6)** gives us the criteria for thresholding:

$$N_{eff} = \frac{1}{\sum_{k=1}^{N_{t|t}} (\alpha_{t|t}^{(k)})^2} \leq N_{threshold} \tag{6}$$

Here we will specify the thresholding value as 0.5*(number of particles), and we also threshold if any of the weight's are so small they become "NaN" values. The summary of the SLAM algorithm implemented in this project is as follows:

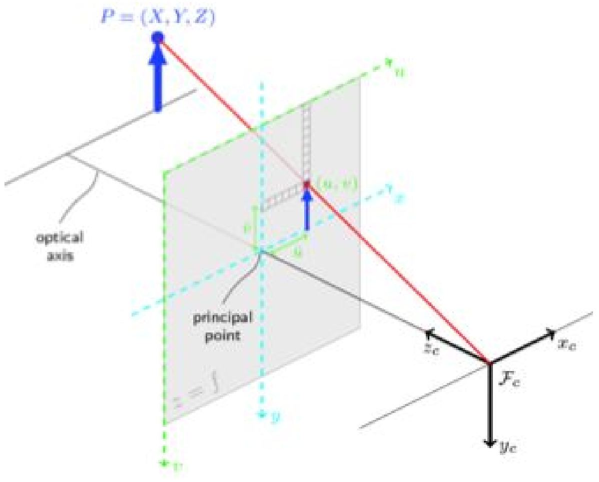
1. Randomly initialize particles around origin of map.
2. For every particle, we compute $\mu_{t|t}^{(k)} = f(\mu_{t|t}^{(k)}, u_t + \varepsilon_t)$ where f is the differential-drive model with inputs $u_t = (v_t, \omega_t)$ as the linear and angular velocity obtained by the encoders and IMU.
3. Feed the prediction $\varepsilon \sim 0.01 * N(0, 1)$ of additive white Gaussian noise to the yaw rate (to model the noise of our prediction).
4. To update, we use our observation data, namely the LIDAR readings for the next time step, z_{t+t}, and transform them to the world frame.
5. Then for every particle $\mu_{t+1|t}^{(k)}$ we find all the cells $y_{t+1}^{(k)}$ in the map corresponding to the scan. Update the particle weights using the correlation model:
$$p_h(z_{t+1} \mid \mu_{t+1|t}^{(k)}, m) \propto exp(corr(y_{t+1}^{(k)}, m))$$
6. After choosing the particle with the highest correlation to our previous map, we actually map the observations of the best particle with a log-odds value of int(log(4))
7. Finally check for resampling if Neff falls below 0.5*(number of particles) or our weights are insignificantly small.

The entire SLAM process using 100 particles takes about an hour and a half; see the *Results* section. The scripts needed for the SLAM implementation are *SLAM.py* and *SLAM_functions.py*, each with a more detailed and step-by-step explanation of the process.

Texture Mapping

Along with creating an occupancy grid of the map, we can also add texture to it. This section is not as complex as it may seem, it is just a few transformations. This involves using the disparity Kinect RGBD camera to transform the RBGD information of a specific pixel in the image of the environment to the world frame. The pixel represents a point in the environment so the corresponding location in our map will be colored using the color values of that pixel. The depth camera and RGB camera are not in the same location (there is an x-axis offset between them) and so it is necessary to use a transformation to match the color to the depth. The pixel coordinates z of a point m in the world frame observed by a camera at position p and orientation R with intrinsic parameters K is given by **Eq.(7)**:

$$z = K\pi(R_{oc}R^T(m-p)) \qquad \textbf{(7)}$$



Figure(5). Pinhole camera model

Since our objective is to go in the opposite direction, from pixel to world, we solve for m using **Eq.(8)**:

$$m = RR_{oc}^{-1}(K\pi)^{-1}z + p \qquad \textbf{(8)}$$

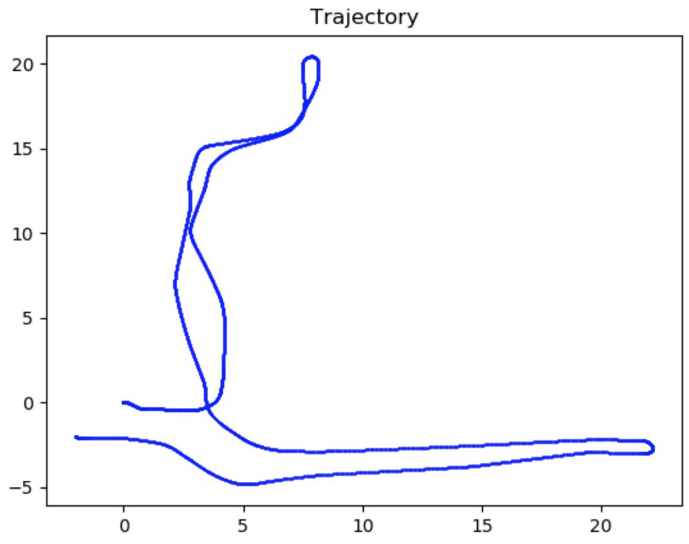
First for every trajectory position, we need to index the correct disparity and RGB image. During the SLAM process we use the same process of finding the nearest timestamp index. The correct indices for the disparity and RGB images are saved during the SLAM process so that must be run first before texture mapping can work. Using the disparity images we transform the indices along with the depth value to the optical frame as in **Figure(5)**. And then from the optical frame we transform to the body of the robot, and transforming once more from the body to the world. We use a camera with the intrinsics as follows: fs_u = 585.05108211 fs_v = 585.05108211 fs_θ = 0 and the optical origin (cu,cv) = (242.94140713, 315.83800193).

After transforming it into the optical and body frame we can use the same functions from the SLAM algorithm to transfer these pixels from body to world and place these RGB pixels in the corresponding coordinates. The map had to be copied two more times to create a 3 channel map for the RGB pixels to fit in. The *Results* section show the texture mapping for a SLAM map using 5 particles.

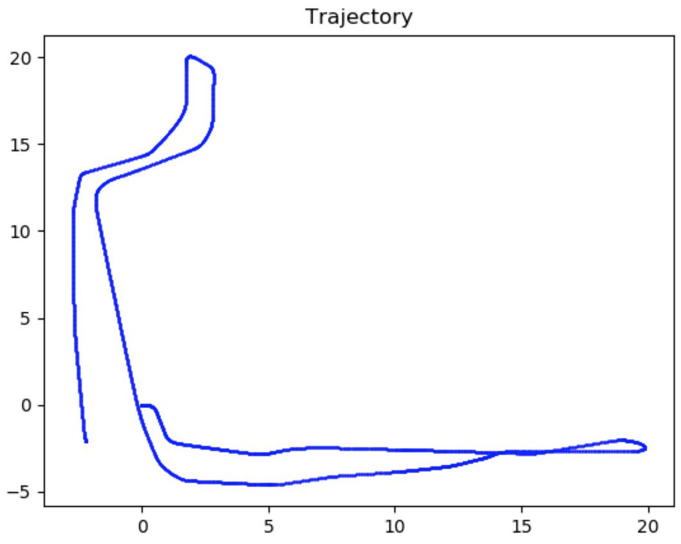
The texture mapping takes roughly 10 minutes. The scripts needed for the texture mapping implementation are *texture_mapping.py* and *textmap_functions.py*, each with a more detailed and step-by-step explanation of the process.

RESULTS

In our first attempt to implement the motion model for the prediction step, we plot the trajectories starting at (0,0) for each of the training sets below:



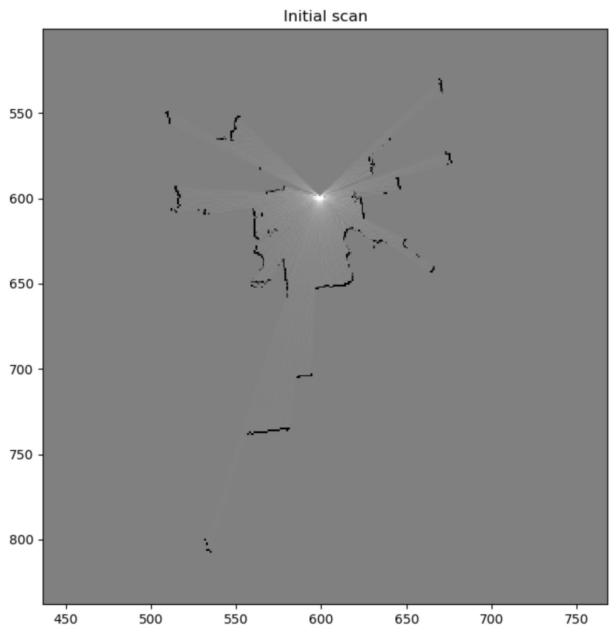
Figure(6a). Trajectory for dataset 20



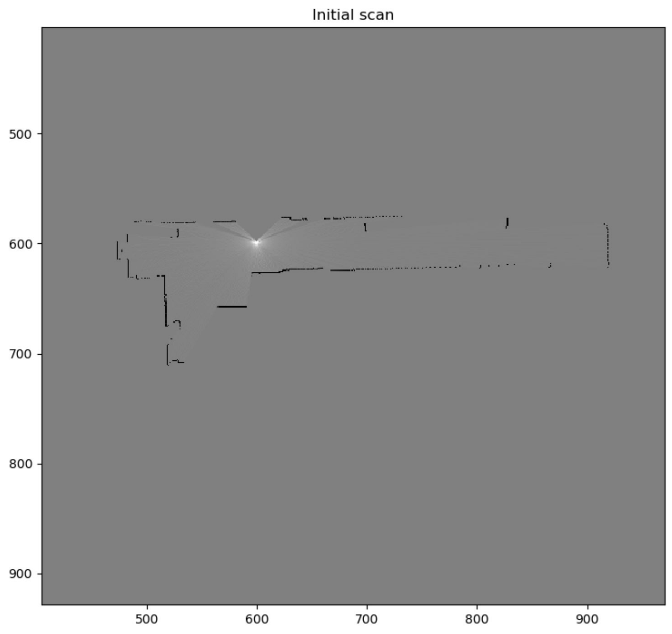
Figure(6b). Trajectory for dataset 21

Using the image set to visualize the robot's movement we can verify that these are indeed the correct trajectories. The robot starts and ends in the same location, traversing down a hallway, to the stairwell and back down another hallway.

Below are first LIDAR scans of the environment, thresholded between 0.2 and 30 meters. It was recommended to threshold the lidar readings between 0.1 and 30 meters, however due to some sort of mechanical intrusion, there were consistent false positives being mapped along the trajectory. Since the robot never gets too close to an obstacle, 0.2m is a valid change.

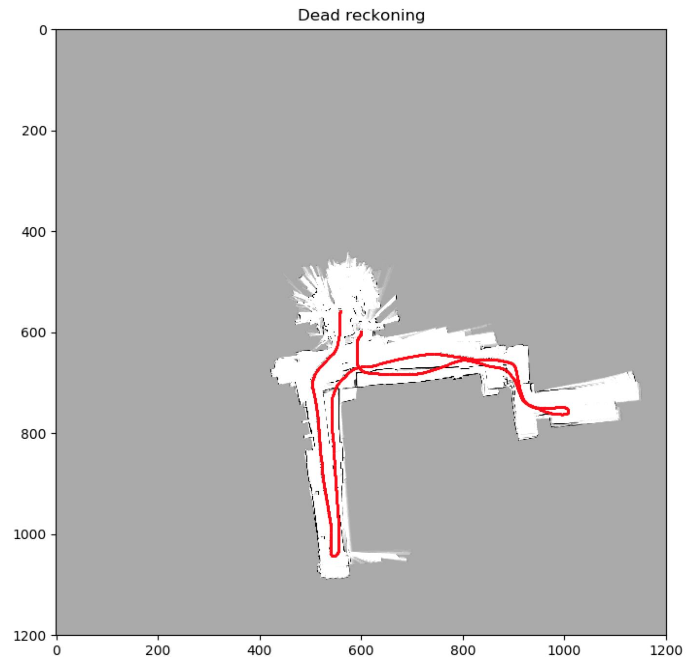


Figure(7a). First LIDAR scan for dataset 20

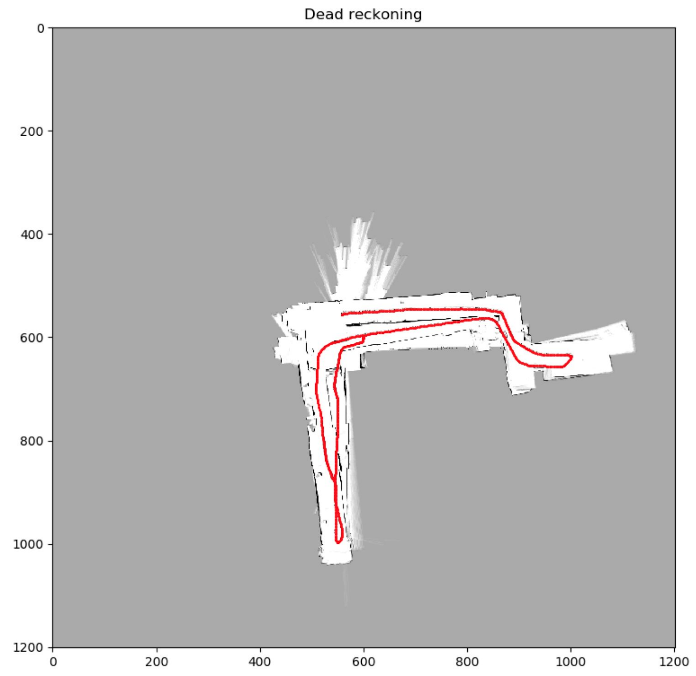


Figure(7b). First LIDAR scan for dataset 21

Using the sensor measurements, the map of the environment is built per dead reckoning:

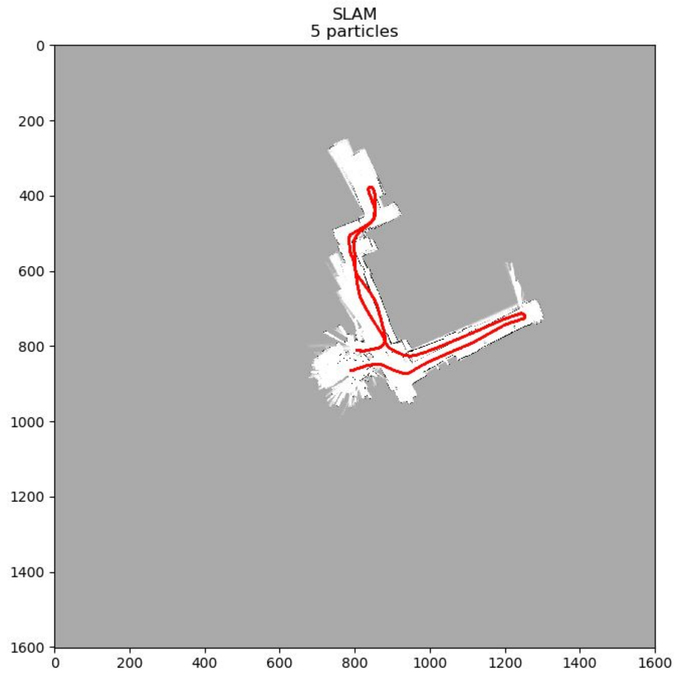


Figure(8a). Dead reckoning for dataset 20

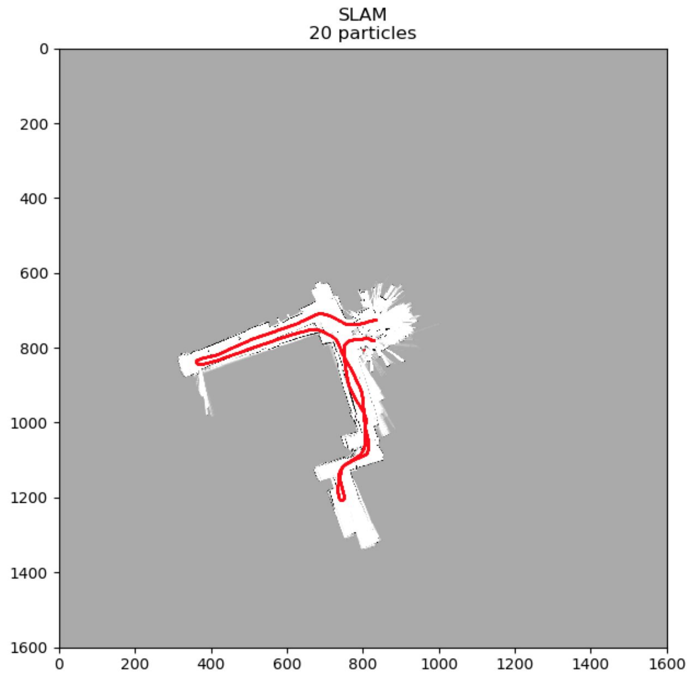


Figure(8b). Dead reckoning for dataset 21

Notice that the trajectories above (in red) follows the trajectory from **Figure(6a)** and **Figure(6b),** with the map of the hallways shown in white. This verifies our transformation from the sensors to the world frame. Now to implement the particle filtering, we analyze the SLAM using 5 and 20 particles. For the comparison between particle numbers we will look at only dataset 20:



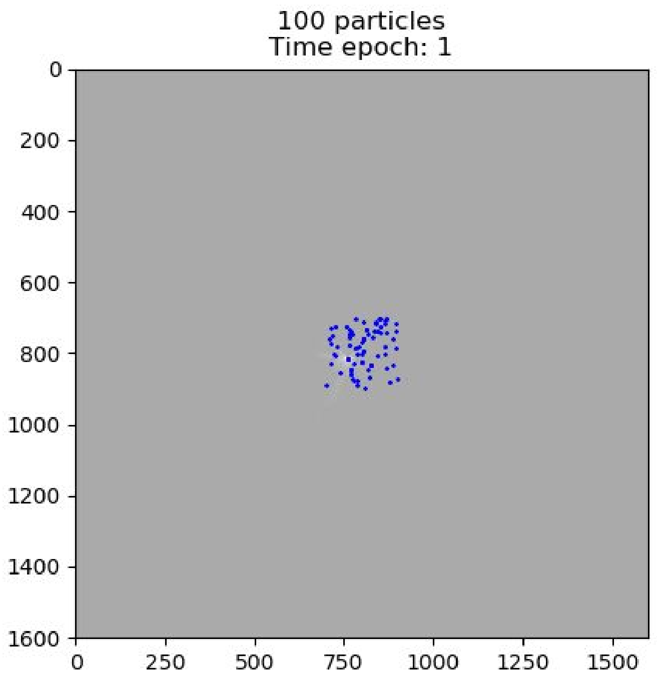
Figure(9a). SLAM using 5 particles (data 20)

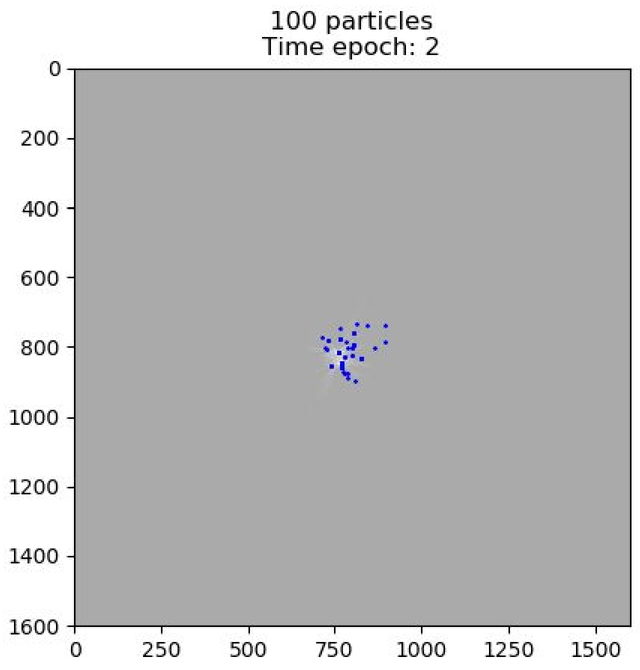


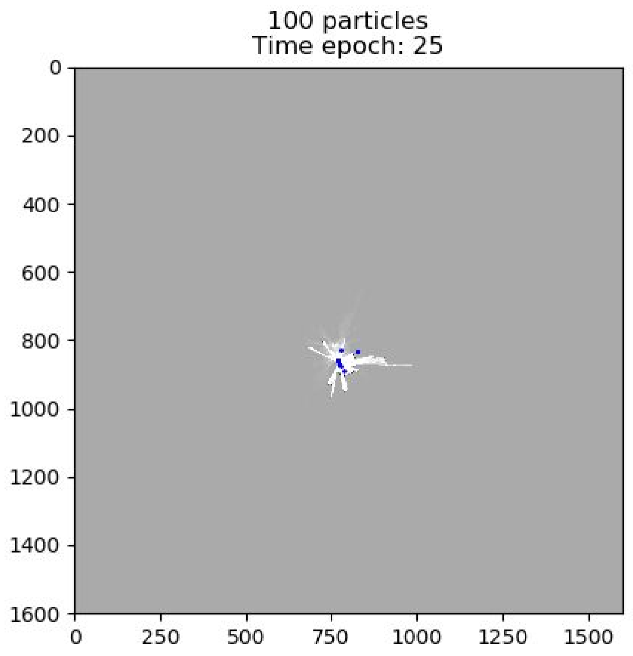
Figure(9b). SLAM using 20 particles (data 20)

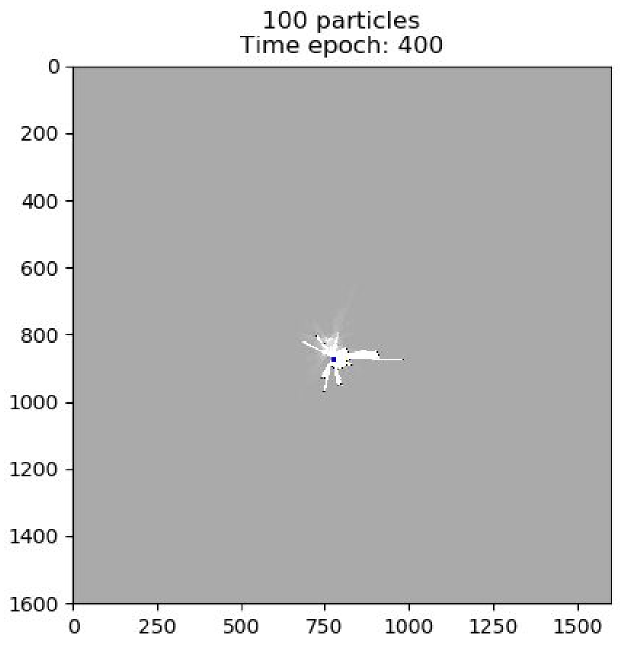
Firstly, the orientations of the maps will always differ because we start with a random initialization of pose for the particles, which includes a random orientation, so once the best particle is chosen in the beginning the map will be built based on that orientation. Looking at the maps themselves, as minute as it is to see, there is a slight difference in the mapping. Paying close attention to the trajectory, we can see that it is just slightly different when the robot enters the stairwell. The walls are slightly more defined along the hallways for 20 particles compared to 5.

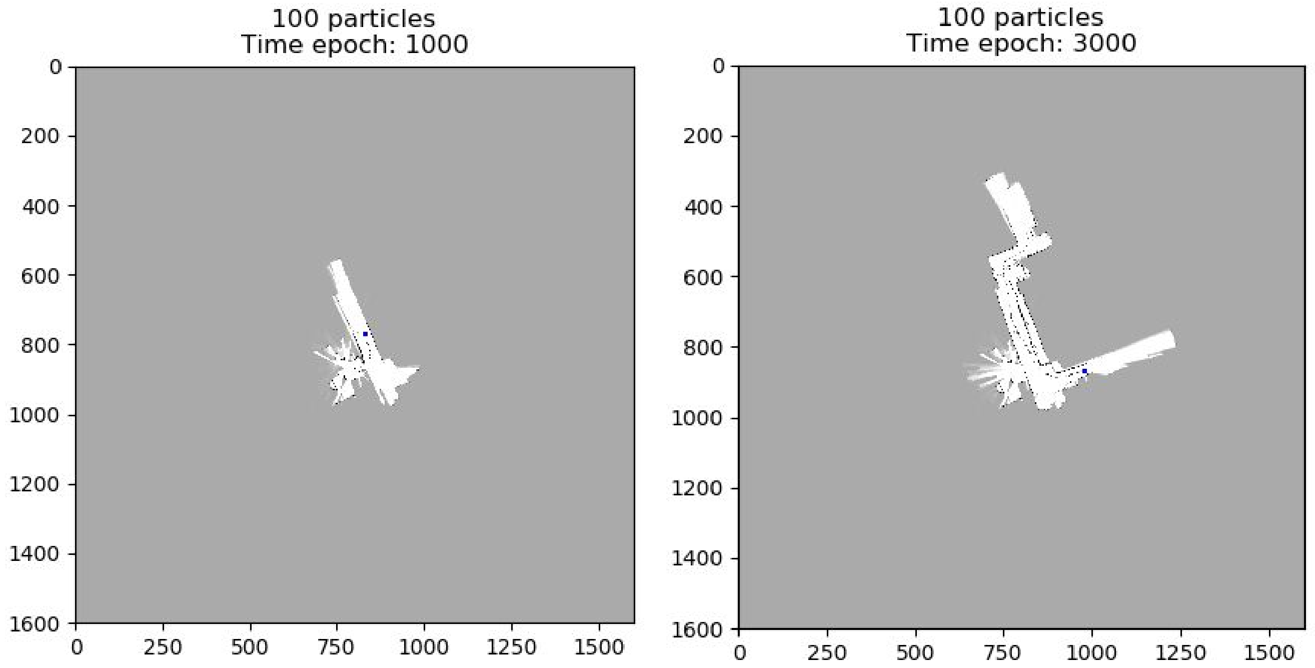
Now to implement 100 particles, lets actually visualize the behavior of the particles between each time epoch defined by the encoder:







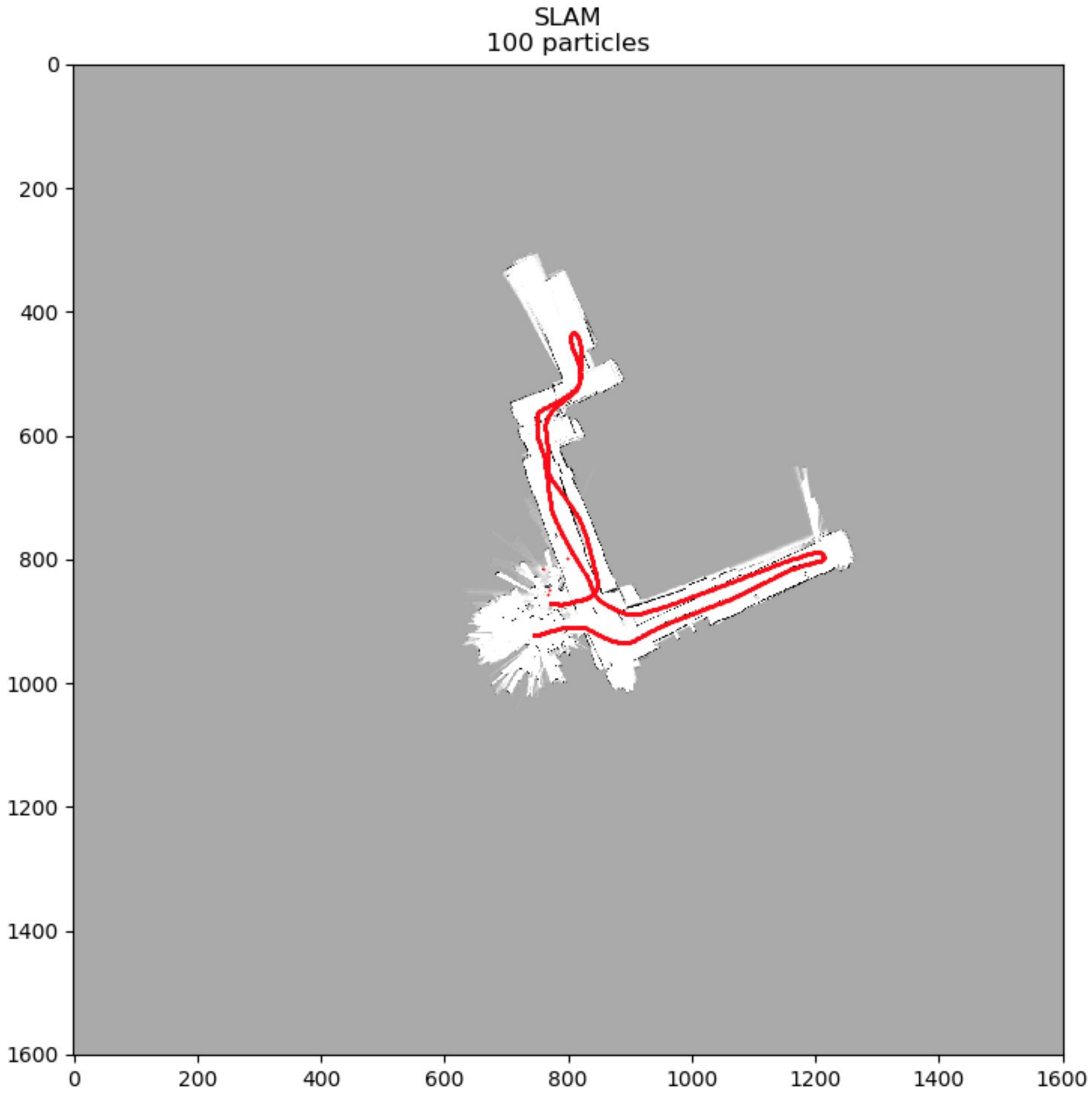




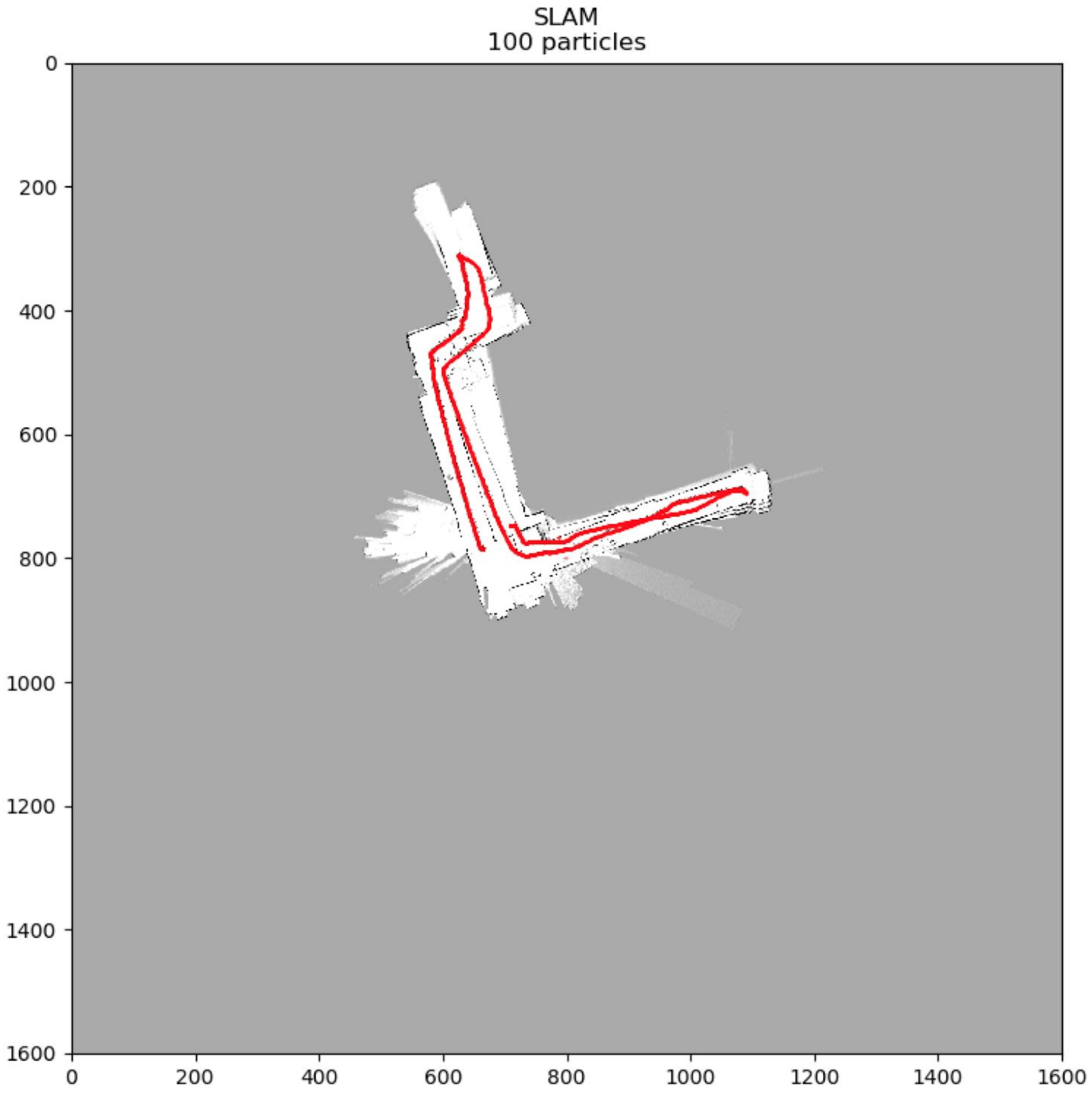
Figure(10). Visualization of SLAM using 100 particles

Looking at **Figure(10)** we can see the initial distribution of particles around our initial lidar reading. Each particle is initialized randomly, within ± 5m of the the origin. Resampling seems to happen right away after the second time epoch, as almost half of the poorly initialized particles are moved toward the best particle. Visually we can see this happens again at around time epoch 25 and eventually 400. Although it seems like there is only 1 point, there is still 100 particles each with slight variation of pose from the prediction step, but the noise is quite small so it is hard to notice. The command line prints each time epoch as well as when resampling occurs, and as the robot moves it looks like resampling happens pretty often, every few time epochs**.** This means that there is enough noise in our prediction step to deviate the particles, and that the weights are constantly updating as most correlated particle is quite evident within each time epoch.

Now that we know that the algorithm implements the particles correctly, lets looks at the final map and trajectories using a 100 particle SLAM.



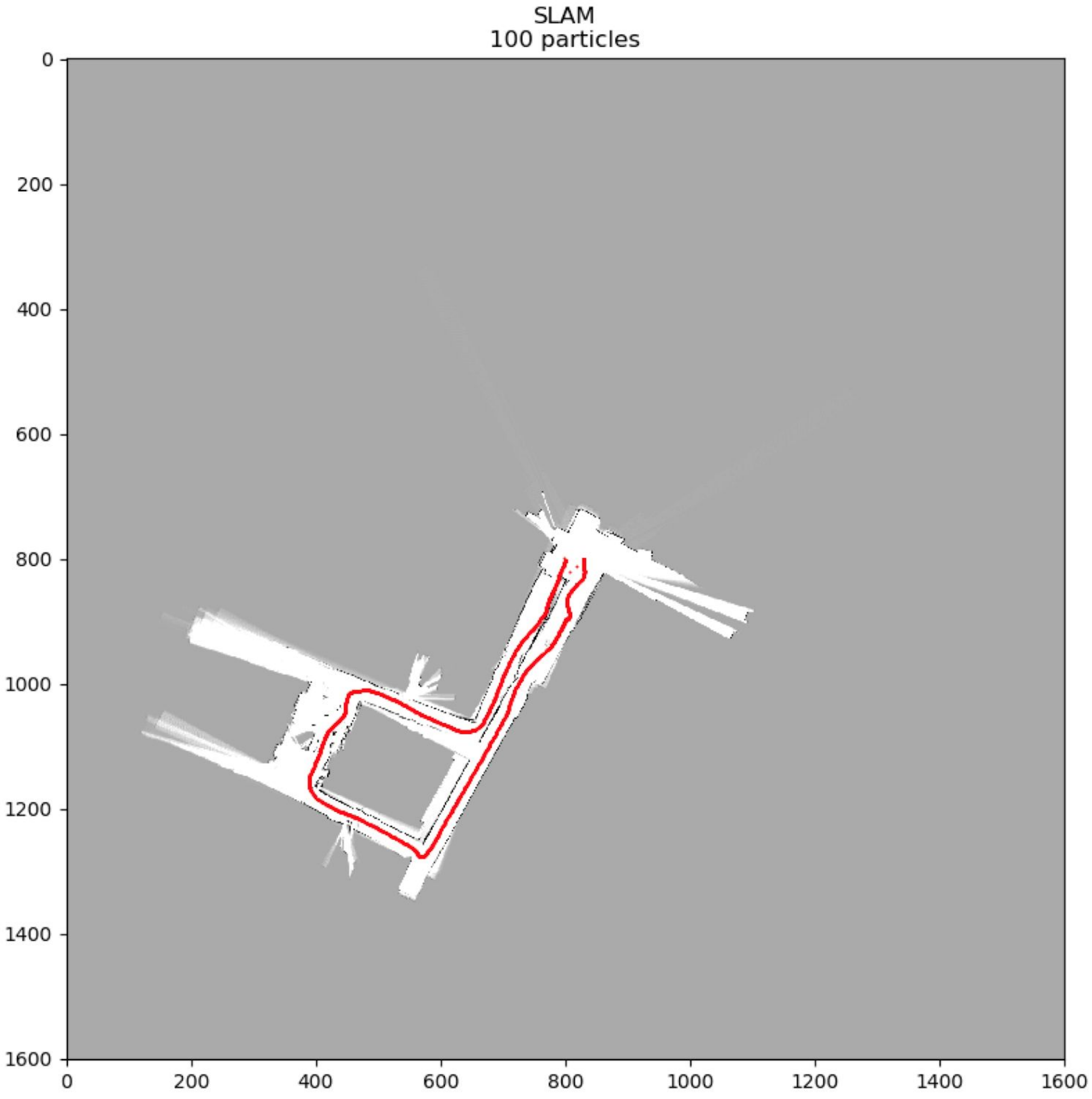
Figure(11). SLAM using 100 particles (data 20)



Figure(12). SLAM using 100 particles (data 21)

The results in **Figure(11)** and **Figure(12)** show a slight improvement from dead reckoning, at least with dataset 20. The walls are more well defined and there is less leakage of lidar scans beyond the walls. The trajectory remains consistent as well. Dataset 21 shows a few false readings but the hallway pointing to the right looks more defined than the one used by dead reckoning in **Figure(8b),** so it still does fairly well for the algorithm not being trained on it.

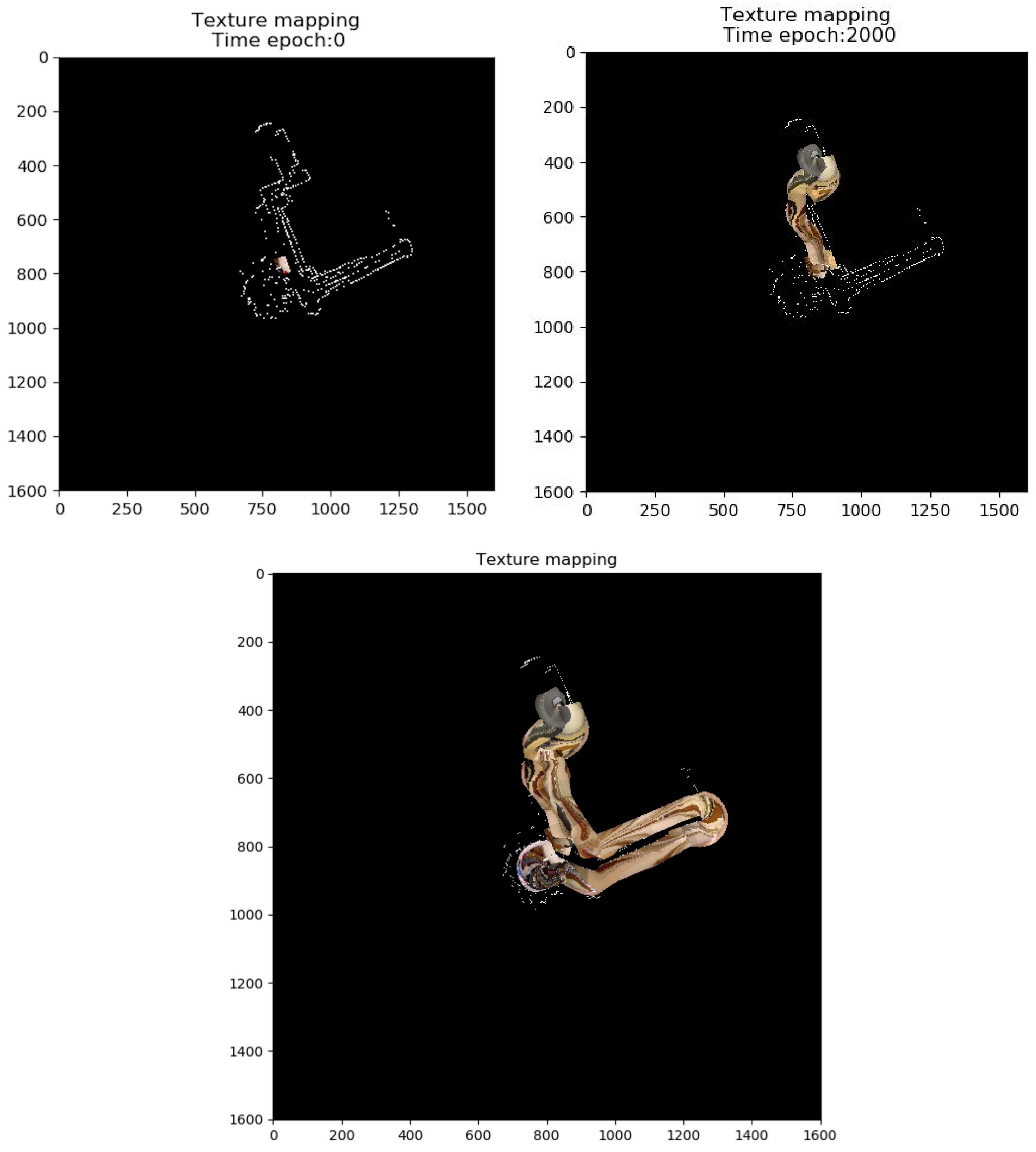
Now to analyze the performance of the algorithm on the test set:



Figure(14). SLAM using 100 particles (data 21)

The map in **Figure(14)** shows the 100 particle filter SLAM on the test set, and again it performs quite well. It looks like the robot is traversing down a hallway and makes a loop around the connecting hallways and ends up right where it started. A gif/video was created for this test case and is attached with the code, see "test_case.mov." Notice how around time epoch 300 the particles diverged and came back together at the best particle, showing that the update step is working properly.

Lastly, to visualize the texture mapping, let's take an example of our map from 5 particles in **Figure(9a)**. The results of texture mapping on that map is demonstrated below:



Figure(15). Example of texture mapping

Although it is not perfect the results are quite impressive. We can see from **Figure(15)** that the pixels in the image space were able to be mapped to the world frame. Looking closely at the colors of the map, we can see that the RGB pixels have been successfully transformed from the image space to the world space and also the correct images are being indexed. The lightish brown represents the hallways, the gray at the top represents the stairwell, and the lighter/colorful part in the center represents the room the robot starts and ends in. This demonstrates the validity of our transformations.

CONCLUSION:

This project has shown that using particle filtering and measurements from odometry sensors, we are abe to construct an occupancy map of an unknown environment and localize our robot within it. We first looked at dead reckoning to build a map based off sensor measurements alone. The maps themselves didn't seem terrible but there were still lots of scans incorrectly being plotted, which was improved using particle filtering. As we increased the number of particles we can deduce slightly better results compared to our dead reckoning. This is because the movement of the robot is more probabilistic and dependent on the current map. Lastly, with texture mapping we can see the success in our transformation from the image space to the world space, as the correct set of images are being mapped to the corresponding location in the occupancy grid.

In reality, we want to design intelligent systems that can withstand the real world, but the real world is never just idle all the time, it is always changing. Using more modern SLAM applications, robots are able to map and remember its location within a dynamic environment which is pretty extraordinary. There are so many more complex types of SLAM and filtering, since mapping is not always 2-dimensional like in this project. Nevertheless, the results shown here demonstrate just how advanced robotic sensing technology is becoming.