Robot Localization using Particle Filters

Kelvin Kang (kelvinka), Gerald D'Ascoli (gdascoli), Jonathan Lord-Fonda (jlordfon)

Table of Contents

Approach

To start, the assignment was nicely separated into three distinct parts: the motion model, the sensor model, and the resampler. To develop the initial build, we separated efforts based on these sections with Jonathan taking the motion model, Kelvin the sensor model, and Gerald the resampler. We finished our initial builds and met to review each other's code. Once we reasoned out the logic and ironed out the bugs that arose during integration, we started tuning.

To aid in the debugging process, the team built several visualization tools to unit test each component of the particle filter, ensuring that individual functions were correct before integrating, tuning, and testing. The tools and processes we used will be discussed in the subsequent sections.

Without optimization and vectorization of the code, the sensor model update and resampling step took around 20 seconds to complete on a 3GHz CPU. This is prohibitively slow as it took more than 2 hours to complete one run, limiting our tuning and experimentation capacity. However, we did not optimize the code until we ensured the correctness of our algorithm because optimization can make debugging harder. We sped up our code by vectorizing the motion model and adding a precomputed lookup table for the ray-casting algorithm, reducing the sensor model step from 20 seconds to around 2-3 seconds, which greatly helped us in tuning the model.

Implementation

Motion Model

The motion model takes in the state of the odometry readings both before (u_{t0}) and after (u_{t1}) the step of motion, as well as the belief of where the system was before the motion (x_{t0}) and outputs a belief as to where the system should be after the motion (x_{t1}), with includes noise jitter provided by α_1, α_2, α_3, and α_4.

The motion model first converts the expected motion provided by the odometry readings into a rotation (δ_{rot1}), a translation (δ_{trans}), and a final rotation (δ_{rot2}). This conversion allows us to isolate the error caused by translating and the error caused by rotating. Now that we have separated the translation and rotation halves, we can add specific noise to our system based on our four different alpha's. We adjust our rotation, translation, and rotation by adding Gaussian noise with the following standard deviations:

$$\hat{\delta}_{rot1} = \delta_{rot1} - N(0, \hat{\boldsymbol{\sigma}}_{rot1}) \qquad \hat{\boldsymbol{\sigma}}_{rot1} = \sqrt{\alpha_1 * \delta_{rot1}^2 + \alpha_2 * \delta_{trans}^2}$$

$$\hat{\delta}_{trans} = \delta_{trans} - N(0, \hat{\boldsymbol{\sigma}}_{trans}) \qquad \hat{\boldsymbol{\sigma}}_{trans} = \sqrt{\alpha_3 * \delta_{trans}^2 + \alpha_4(\delta_{rot1}^2 + \delta_{rot2}^2)}$$

$$\hat{\delta}_{rot2} = \delta_{rot2} - N(0, \hat{\boldsymbol{\sigma}}_{rot2}) \qquad \hat{\boldsymbol{\sigma}}_{rot2} = \sqrt{\alpha_1 * \delta_{rot2}^2 + \alpha_2 * \delta_{trans}^2}$$

We then convert our new, jittered rotation, translation, and rotation back into an x and y translation and a total rotation via the following equations:

$$x_{t1} = x_{t0} + \hat{\delta}_{trans} * cos(\theta_{t0} + \hat{\delta}_{rot1})$$
$$y_{t1} = y_{t0} + \hat{\delta}_{trans} * sin(\theta_{t0} + \hat{\delta}_{rot1})$$
$$\theta_{t1} = \theta_{t0} + \hat{\delta}_{rot1} + \hat{\delta}_{rot1}$$

These values are then passed back into main.py as the current belief of the robot's position. Since the motion model is supposed to add noise to our system, tuning of the alpha parameters is important to ensure that our system doesn't go haywire, but adds the correct amount of uncertainty to give us a good spread of possible locations. In order to tune these values, we set all of the alphas to zero and then went through each of them one-by-one, assessing their effect on the dispersion of particles starting at roughly the ground truth and not resampled, merely spreading out slowly from the uncertainty imposed by the particular alpha. Since we had no information regarding the accuracy of the odometry sensors, we merely watched the cloud disperse and adjusted alpha until it represented a slow drifting into the anticipated shape. An example of one such shape is shown below in Figure 1.

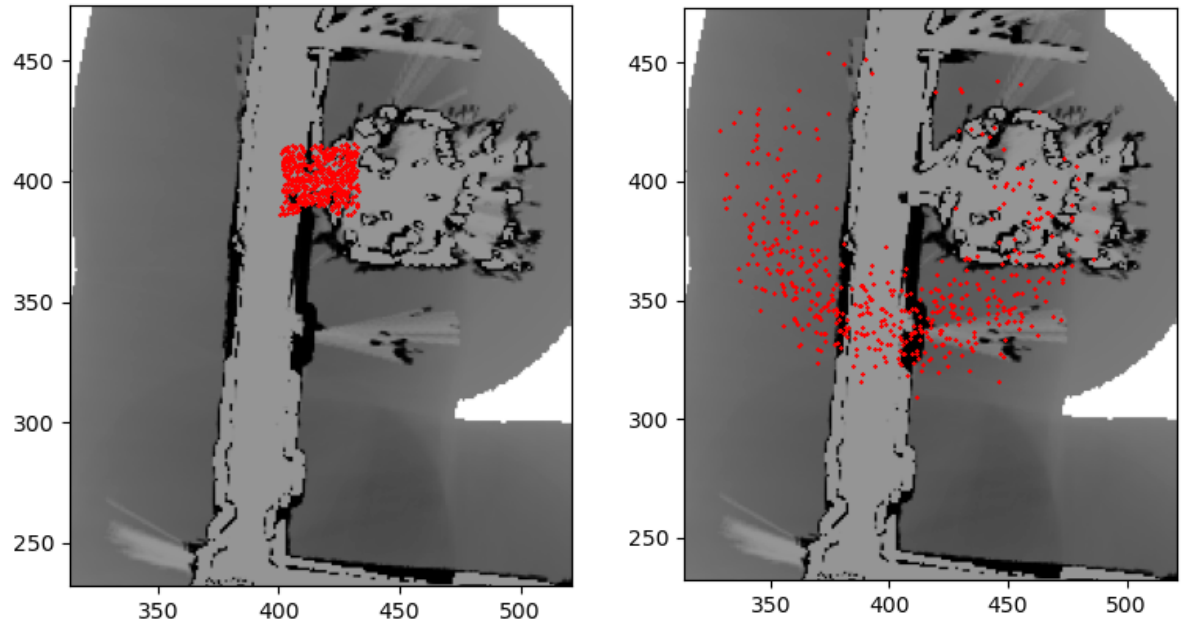

Figure 1: An example of the anticipated shape when tuning α_1, the variance of the rotation from the robot's rotation. Here we were looking to see that the expected robot position would fan out after it turned to travel down the hallway. A reasonable value for α_1 results in a wide interpretation of thetas (roughly 180 degrees for a 90 degree turn). The sensor model will devalue the particles that are further from the ground truth thereby recentering the clump of particles on the true position.

Tuned Parameters - Motion Model

Param	Description	Value
scale	Scaling of all alpha parameters to keep values consistent relative to each other.	0.75
alpha1	Variance of rotation due to rotation	4e-4 * scale
alpha2	Variance of rotation due to translation	1e-4 * scale
alpha3	Variance of translation due to translation	0.1 * scale
alpha4	Variance of translation due to rotation	0.5 * scale

Sensor Model

The sensor model takes in the laser range finder range measurements z_t, and one of the particles state belief x_t and outputs the probability of the range measurements given the state belief, hence it is giving $p(z_t|x_t)$.

To speed up the runtime, we precomputed as much as we could and stored the pre-computed values to a member array variable that acted as a lookup table. The sensor model first created an array of unit vectors of length map_resolution in the laser frame from 0 to 180 degrees with an increment of the subsampling parameter. The unit vectors were then projected until we hit max_range distance, with an increment of map_resolution. We then stored all these values to a 3D numpy array (x_ray_in_l), hence it was nominally of size 3 x 100 x 180.

The unit vector and the particle state belief were sent as params to our ray-casting function, which output the predicted range that was measured $z_{t,pred}$. The ray-casting algorithm incremented the unit vector step by step, then transformed that point (x_ray_in_l) from the laser frame to the world frame and checked if the occupancy_map values at those points met the minimum threshold. The distance from the robot to the raycasted unit vectors were returned as $z_{t,pred}$ to calculate the $p(z_t|x_t)$ with the sensor model.

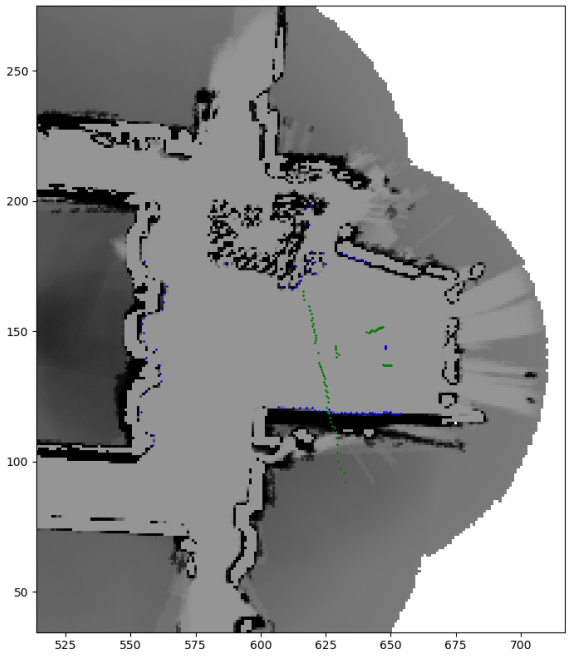

Figure 2: ray-casting debugging visualization, blue dots are the raycast points, green dots are sensor measurements

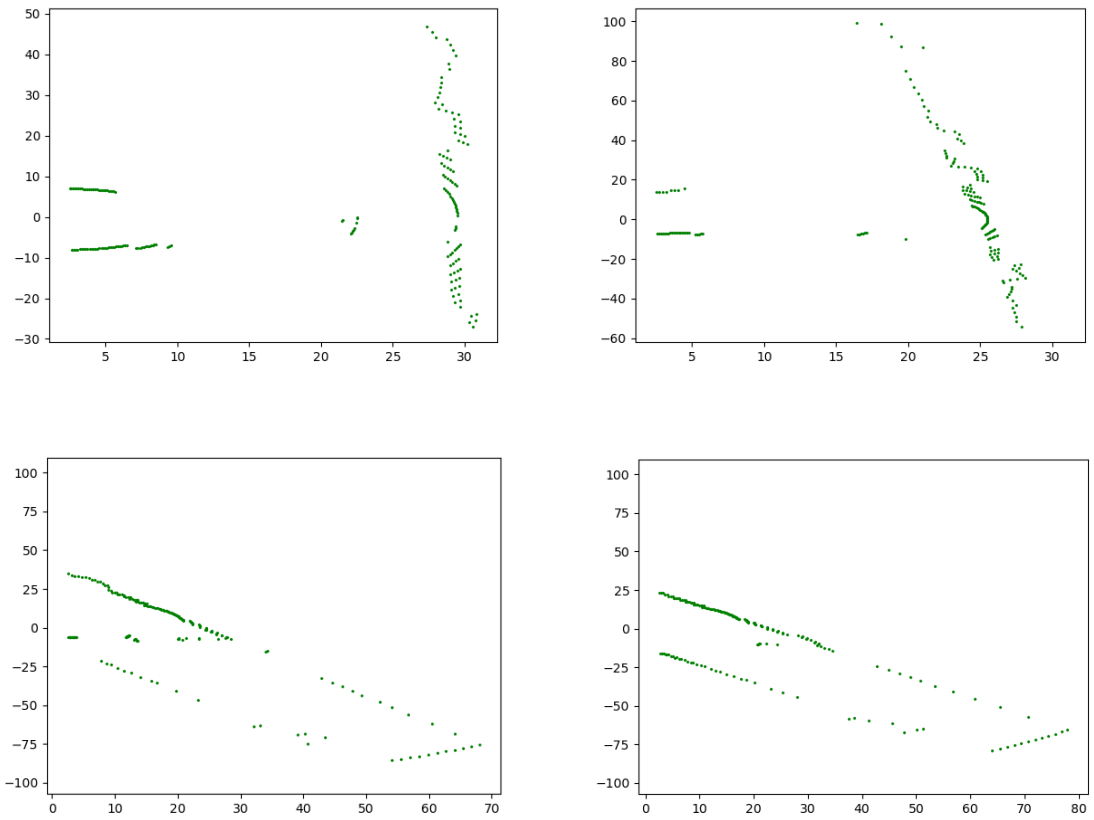

Figure 3: Sensor measurements visualization, this plays back the lidar scans from the robot's viewpoint

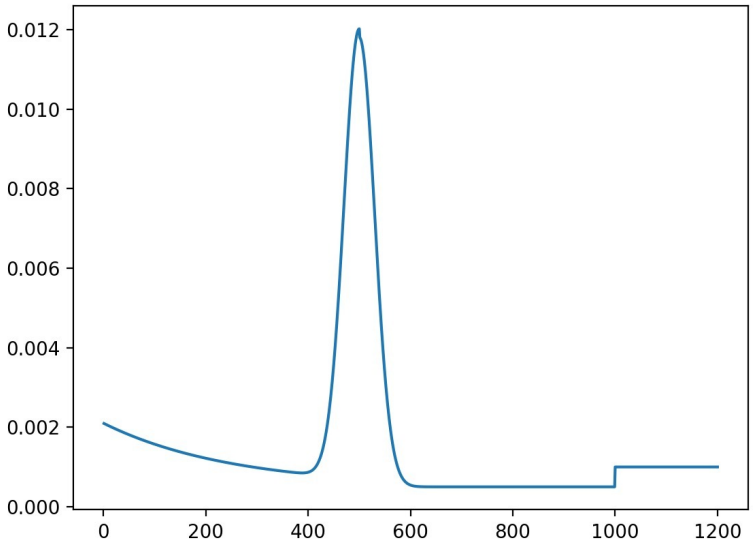

Figure 4: Sensor model probability distribution function (pdf)

The ray-casting algorithm is complex and hard to debug by looking at raw values only. Hence we developed a visualization tool as seen in 2, whereby the check if the raycast algorithm can detect the walls. We initialized a single particle, located at the right side of the large room pointing to the left. The blue dots, which are calculated from $z_{t,pred}$ are aligned nicely with the walls. We also displayed the actual sensor measurements as green dots. We collected a few of these scenarios and note their $p(z_t|x_t)$. This is to ensure that raycasts and sensor measurements that are close to each other should have a higher $p(z_t|x_t)$.

The recording of the laser scans are also visualized. As seen in Figure 3, we used this to check if our transform from laser frame to world frame is correct. We noticed that based on the laser scans playback, the robot turned right at the beginning. However, this is counter to what the animations and odometry data show. We found a bug in our axes transform and fixed it accordingly.

Lastly, we visualized the probability distribution function of the sensor model as seen in Figure 4 to ensure that we have a good shape according to the reference textbook. It also aided in tuning the sensor model. For the initial set of parameters for the sensor model, we changed the parameters until the general shape matches that of the reference shape. Then during testing, we encountered issues such as jittering of the state belief when lots of people walked near the robot, we increased the z_short and lowered the lambda_short to increase the probability of obstacles that are not part of the map when they are near the robot. This helps in stabilizing the robot when there are obstacles near the robot.

Another parameter tuning that helped was in reducing the z_hit and increasing the sigma_hit, this increases the uncertainty of the sensor model, allowing particles that do not align nicely to have some weightage. Essentially this slows down the convergence of the particles, which we have found to be useful as it expands the ‘search area’ of the particles when it does its search at the beginning of the run.

Tuned Parameters - Sensor Model

Param	Description	Value
z_hit	Weight of the main curve of the $z_{t,pred}$ part of the model	0.85
z_short	Weight of the heavy short tail part of the model	0.4
z_max	Weight of the peak at the end of the model	0.003
z_rand	Weight of the uniform probability of the model	2
sigma_hit	The standard deviation of the z_hit part of	50

	the model	
lambda_short	The lambda of the exponential distribution of the z_short part of the model	0.004
min_probability	Minimum probability of the occupancy map to be considered an obstacle	0.35
subsampling	Number of subsampling of the laser scans to speed up computation	2

Resampling Method

Multinomial Sampler

The multinomial_sampler function samples from the particles by randomly selecting the particles by priority based on their weights. The weights are first normalized to sum to 1, then a for loop iterates for *n* particles through generating a uniform random number, sampling a particle based on that number, and adding the particle to the new particle set. The particle is selected by adding the weights one by one until the sum of the weights is greater than or equal to the uniformly generated random value. Since greater weights add more to the sum, there is a higher chance that the great weight values will push the sum of weights over the random threshold, therefore giving higher priority to the higher weighted particles. The new set of particles sampled based on weight is then returned for the next filter iteration.

The multinomial sampler must generate a new random number for each sampled particle and must iterate from the start of the original particle set for each sampled particle.

Low Variance Sampler

The low_variance_sampler function samples particles based on weight using a sort of number line method. This method only has to generate a single random number so therefore it is a “low variance” sampler. The for loop to iterate sampling *n* particles starts with setting the next threshold value using $U = r + mM^{-1}$, where *r* is the single random value, *m* is the loop iterator for the index of the next particle, and M^{-1} is the sum of weights divided by the number of particles i.e. the average weight across all of the particles. Each iteration, *U* increases by M^{-1} (the average particle weight) as *m* increases. Then the original set of particles are iterated through, building up a sum of weights, until the sum of weights passes *U*. Once it passes, the particle that set it over the limit is sampled for the new set, and the loop starts again by incrementing to the next *U* value. This way, the higher weighted particles are sampled repeatedly because they raise the sum of weights greater than the incremental addition of the average weight to *U* while the lower weights are either ignored or sampled once or twice because they increment the sum of weights less than the average particle weight and therefore won’t be sampled again when *U* is incremented.

The low variance sampler only generates a single random value and then iterates through the original particle set once, but maintains the same set size by resampling the higher weighted particles repeatedly.

Adaptive Number of Particles

Reducing the number of particles helps speed up the particle filter. Each particle requires motion and sensor model calculations so fewer particles mean less calculation time.
The need for more particles helps locate the robot initially, but as those particles converge to certain modes based on the weights assigned to them, the number of particles can be reduced. For a map like this one of the Wean Hall hallway, a good metric to detect the number of modes remaining is the x and y variance of the current set of particles. As the modes dwindle to the few modes with more confidence, the variance in the x and y locations drops. The variance drops slowly at first as the modes start to form, but once they're formed the variance drops significantly each time a mode is sampled out. This nonlinear variance change fits well into a linear particle scaling function with bounds for a maximum and a minimum number of particles. The function was set to only start decreasing the number of particles once the sum of the x and y variance dropped below 1500. This parameter was tuned to this value based on the size of this Wean Hall map. For the bounds, the upper bound is set to the initial number of particles, and the lower bound is set to 200 or 1/10 the initial number of particles (whichever is greater).

$$\#particles = int(lowerbound + min(\Sigma_{x+y}, 1500)/1500 * (upperbound - lowerbound)$$

Kidnapped Robot Dispersion

Solving the kidnapped robot problem involves being able to re-initialize the robot's position at any given point in the event that the robot is randomly teleported across the map. To solve this issue, we add a small number of particles to our belief, initialized with random states, at regular time intervals (Figure 5). It's not a perfect solution, but if the robot is teleported then the new particles have a decent chance of finding the correct location and being weighted highly if they succeed. The higher weight then translates into the particles quickly funneling into the new location, thereby solving the problem and relocalizing the robot.

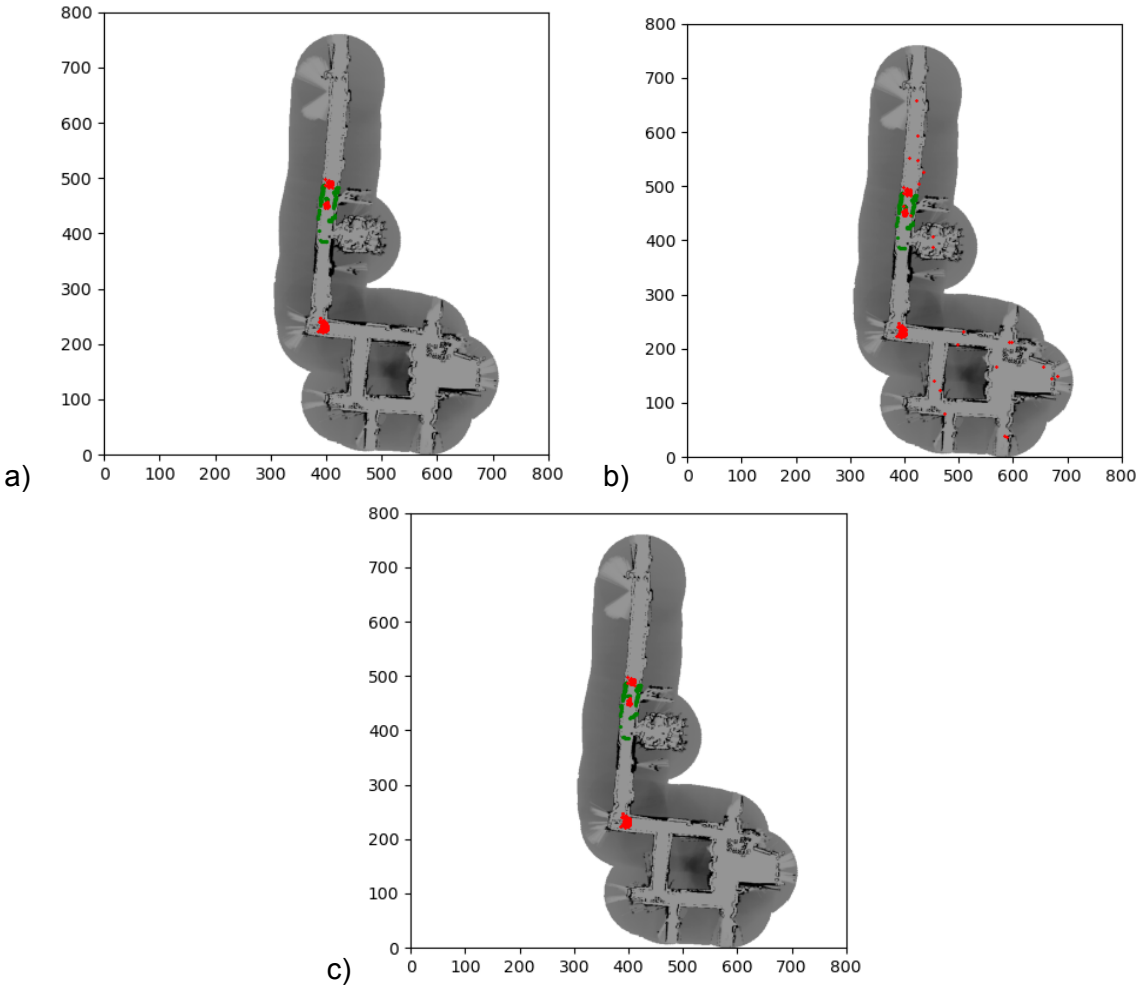

Figure 5: These are serialized time steps demonstrating the dispersion of new points (in b). As can be seen in c, none of the points matched well with the sensor input so the resampler threw them out. We wouldn't expect random points to match the sensor readings, but if the robot was dropped in a new location, then the sensor input may match one of these points, resulting in giving it a higher weight and having it likely chosen a few times by the resampler, thereby transferring particles over to the new point and learning its new location in the map.

Integration

We employed several techniques to aid us in the integration, tuning, and testing of the overall particle filter system

Random Seed

To remove randomness during tuning, we fixed a random seed to make the system deterministic. This helps to isolate changes to our tuning. The random seed was also helpful when we tested the system for repeatability, as we can change the seed value to change the random values.

Laser Scan Visualization

We overlaid the laser scan on the map, shown as green dots on the highest weight particle during the run. This helps to see if our filter is converging properly to the right location, visualize the scan matching against the walls and features, and see the heading of the particle filter cloud. The last point is especially important when checking if the filter is able to turn 180 degrees appropriately.

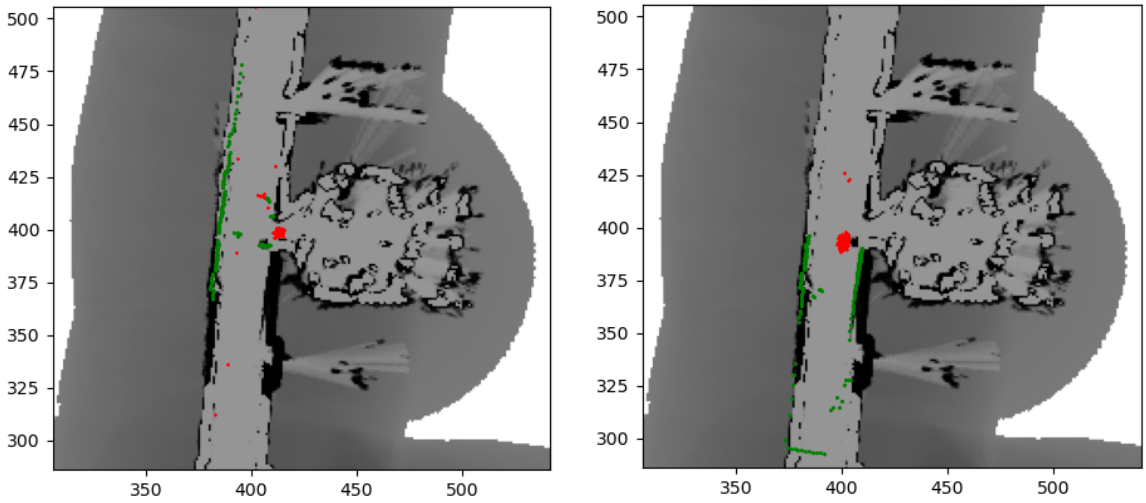

Figure 6: Overlay of the laser scan on the highest weight particle (green dots)

Description of performance

The particle filter is able to converge to the correct location if it has good particle initialization, i.e. there are sufficient particles close to the correct pose of the robot. After it converges, the tracking performance is good, even when the walls of the hallways look very similar around the map. This is because the motion model and sensor model have been tuned well to trust the odometry just enough to get both convergence and accurate tracking in areas with similar features.

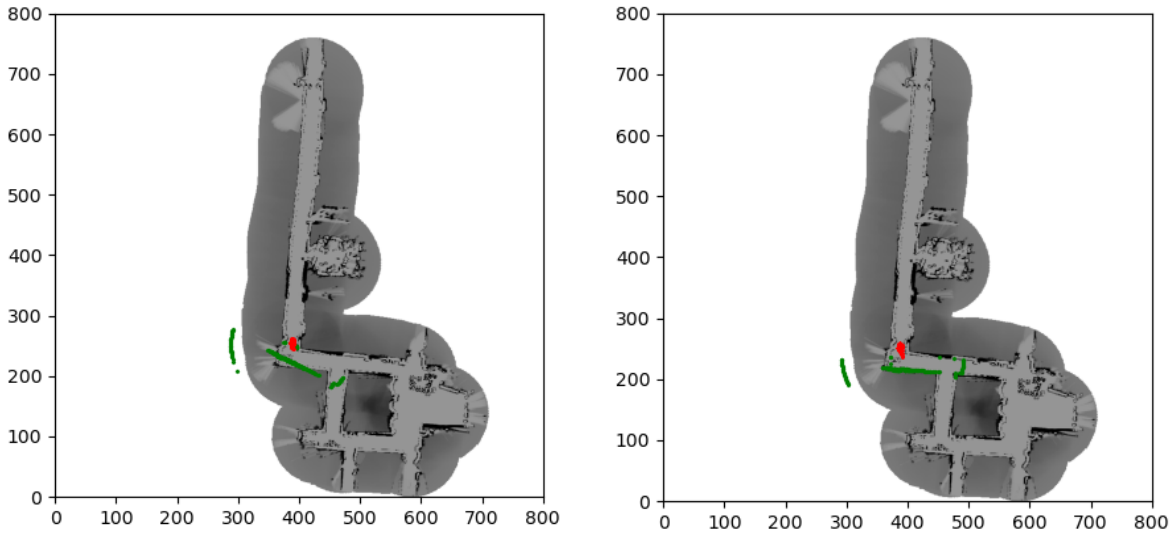

Figure 7: Example of the sensor model scan matching correcting the pose of the robot

Sensor model is able to rectify errors when it sees distinct features, the straight wall in front of the robot in the case seen in Figure 7. This is a desirable trait of the particle filter, however, since the map and the sensor measurements are imperfect, we faced issues whereby the particle filter struggles to move the belief forward when the robot is traveling through a corridor with similar scan matching over the interval. This is mitigated by tuning the motion model to put more trust in the odometry reading, especially the translation component.

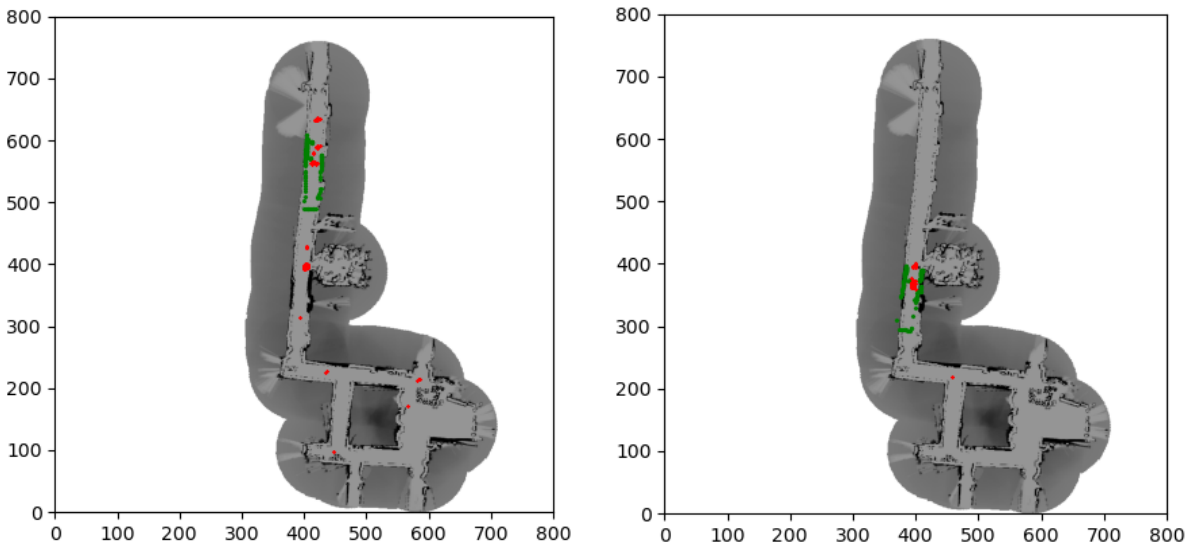

Figure 8: Highest belief particle switches between modes

During the first few steps when there are still multiple modes, the highest weight particle shifts around the map (Figure 8). However, this is acceptable and desirable as we are still unsure where the robot is. One disadvantage of a particle filter is that after the system converges, it does not check other possible poses around the map. Hence, once it converges to the wrong pose, it is unlikely that it can rectify itself to find the correct pose, even when the sensor measurements proved otherwise. This issue is slightly mitigated by our kidnapped robot dispersion algorithm.

Robustness / Repeatability

To test various conditions of the particle filter, we experimented with different random seeds. After running many different seeds, we noticed a low success rate due to reasonable failure conditions. If there are particles close enough to the initial state, the particle filter will work well, otherwise, the true location can be lost as particles converge to different locations with similar sensor measurements. As shown in our videos in the Results section, when particles are generated near the correct initial state, the model correctly localizes the robot and properly tracks the robots' course. However, as seen in the figures below, if the initial conditions are not close enough to the correct state, no particles converge to the exact correct location and instead, particles with similar sensor readings are selected by the sensor model.

Some seeds had a successful performance with properly initialized particles. These seeds correctly started in the open office, tracked down towards the square hallway, turned around and tracked back up the hallway, and so on. This is shown in Seeds 10 and 18 shown in Figure 9.

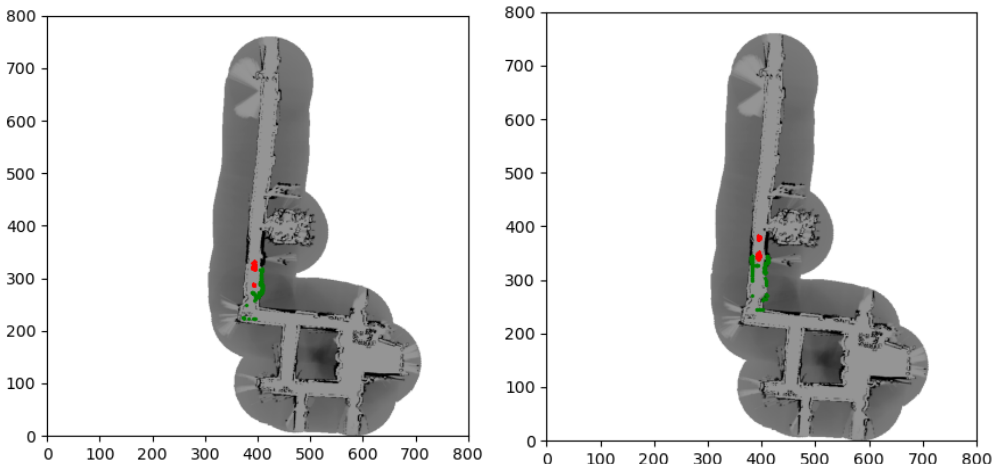

Figure 9: Seed 10 (left) and 18 (right)

A couple of seeds found roughly the correct location, but couldn't converge on the correct orientation such as Seed 0 shown in Figure 10.

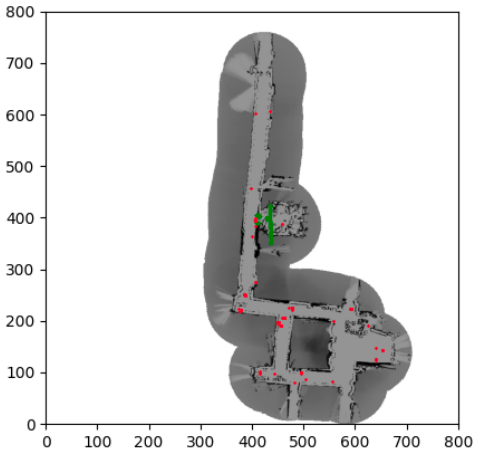

Figure 10: Run with Seed 0

Seeds often generate a particle in the top left of the square hallway where there is an open path facing a wall with a viable left turn, very similar to where the actual robot path looks out of the open office door. This is shown in Figure 11 for seeds 1, 69, and 420.

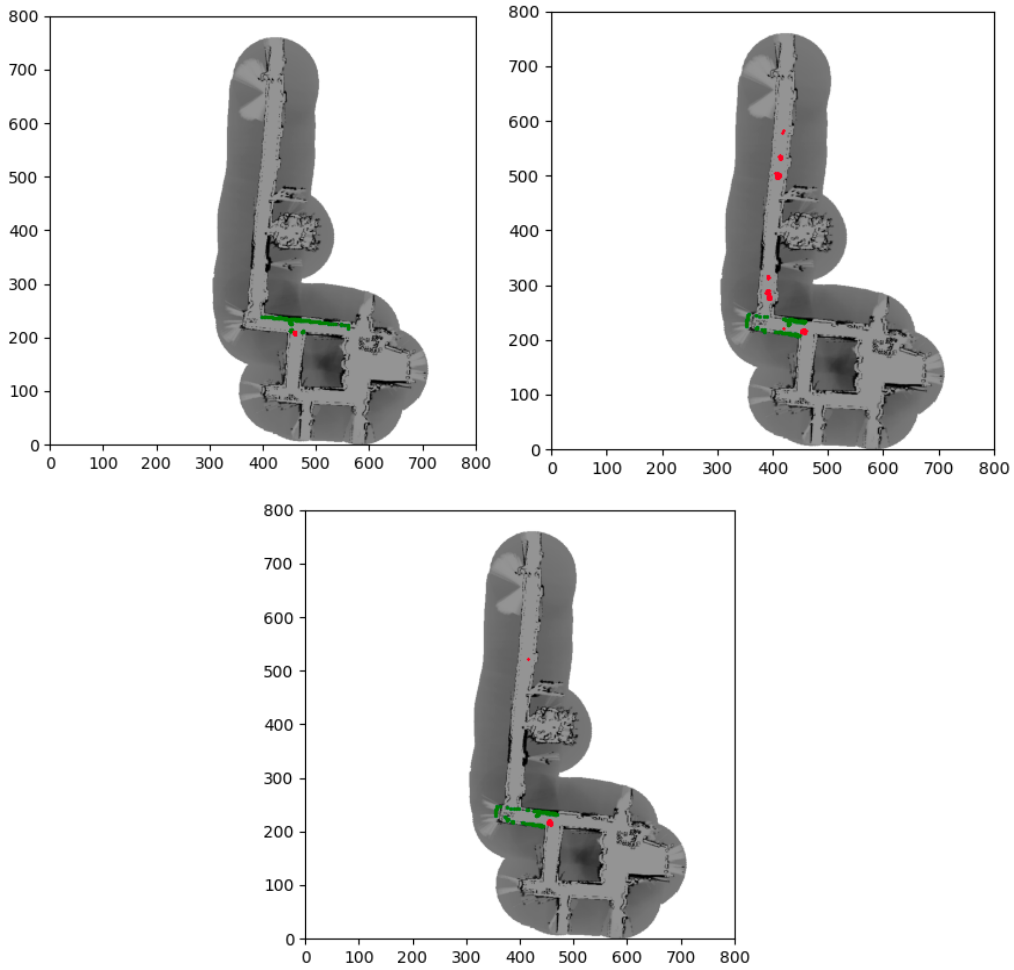

Figure 11: Run with Seed 1 (top left), 69 (top right), & 420 (bottom)

Another trend that arose was particles selected within the correct hallway with varying location and orientation. Some were just outside the office door facing the right way and some were at the bottom of the hallway facing the wrong way. The sensor model seemed to favor these particles because they properly registered a wall in front of them. This is expressed in Seeds 3, 13, and 22 in Figure 12.

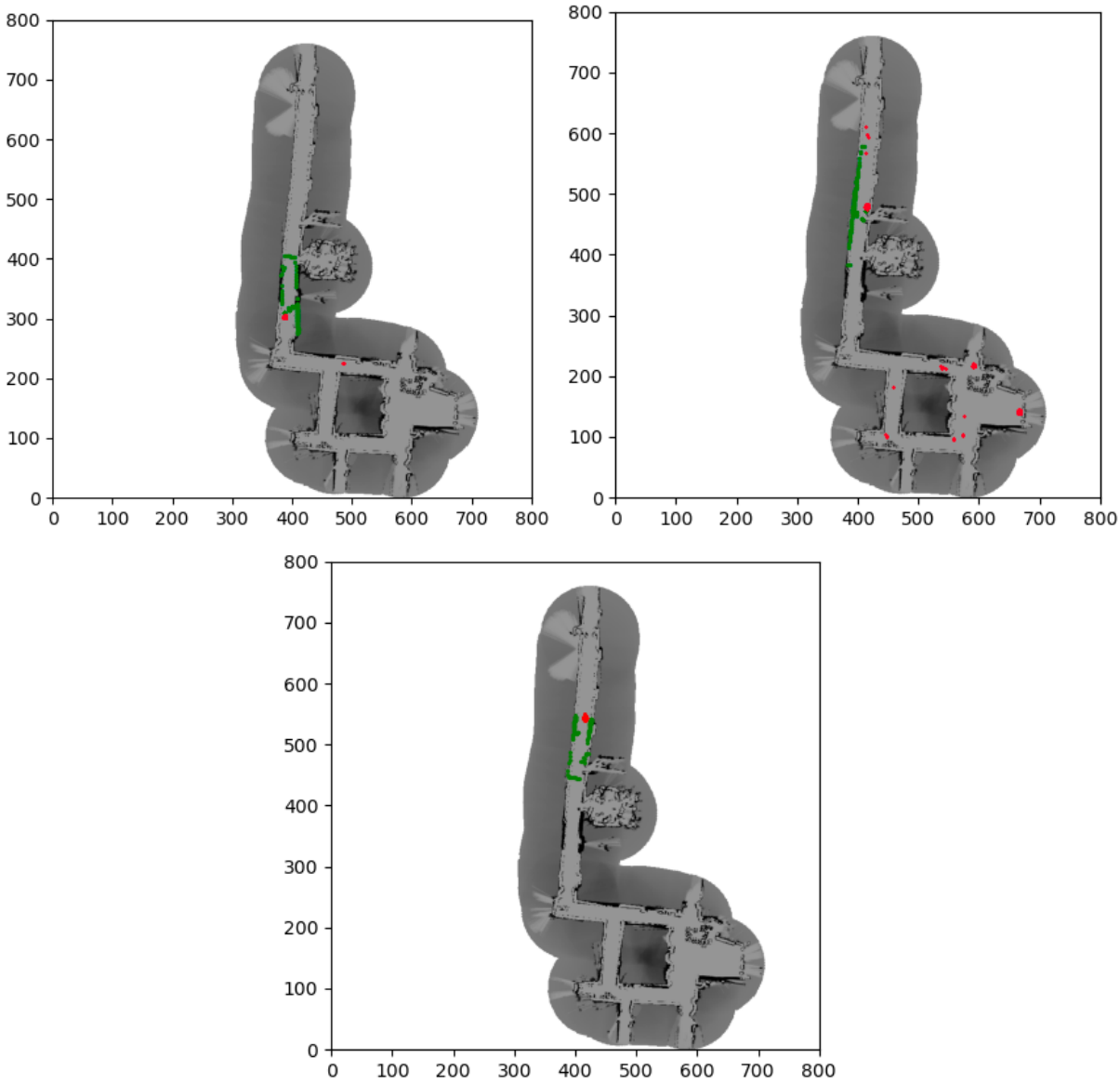

Figure 12: Run with Seed 3 (top left), 13 (top right), and 22 (bottom)

Overall, our model was not very robust to random seeds with set parameters and number of particles at 500. We found the particle filter's success was heavily reliant on the initial particle generations. With many more particles, the particle filter would have a greater chance to generate an initial particle under these conditions, but it wasn't consistent enough at 500 initial particles.

Results

Particle Filter with Random Freespace Initialization on robotdata1

https://www.youtube.com/watch?v=JYSXgS6zjL0&ab_channel=KelvinKang

Particle Filter with Random Freespace Initialization on robotdata2

https://www.youtube.com/watch?v=uxF2Qi30b18&ab_channel=KelvinKang

Future Work

There are two different aspects that we would improve if we were to continue this project. First of all, we would spend extra effort vectorizing all of our code so that we could test many simulations in the same amount of time. Along with this, we would also spend more time fine-tuning the sensor model so that it could quickly converge to the appropriate location with more consistency. Vectorization would enable us to run far more tests and be that much more efficient with our tuning process.

Secondly, if our filter was better at locking onto the correct location with more precision, we would implement a second solution to the kidnapped robot problem. We would track the sum likelihood of the particles over the course of the operation and if we saw a large drop in our match with the sensor input that was consistent over a number of time steps, we would reinitialize the particle filter, having ascertained that the robot was most likely teleported to a new location. However, without finer tuning of our system, this feature may cause more problems than it solves so for now we did not implement it.