Robot Localization using Particle Filter

Akshay Pramod Roy* and Dhaneshwaran Jotheeswaran*

School of Electrical and Computer Engineering

E-mail: apr8@gatech.edu; dhaneshwaran@gatech.edu

Abstract

Robot Localization is one of the fundamental problems in Robotics. In this Lab, we explored a particular algorithm, called the Particle Filter, to Localize our robot in a given map with just the Laser Range data and the Odometer data. These are the videos of the robot converging for *robotdata1.log* and for *robotdata4.log*.

Introduction

Particle Filter is a type of Non-parametric filter. Non-parametric in the sense that the algorithm as such doesn't depend on any parameters like mean, co-variance, or other higher order moments. Of course, individual modules within the algorithm, like the sensor model and the motion model are parameterized, but the core algorithm is not constrained by any parameter except the number of Particles used. In short, the posterior that the Particle Filter produces doesn't have a functional form. The are a lot of parts that went into making this project happen. We will go over these individual parts one by one, and discuss how that affects the Algorithm as a whole, and how it was tweaked to get the right parameters.

The Map

The given map was a $(800 * 800)$ map of the Wean Hall at Carnegie Mellon University. The map's resolution was 10cm/pixel. Each pixel in the given map ranged from 0 to 1 for known locations and -1 for unknown locations. The map had probability of the pixel being free space as opposed to it being occupied, as mentioned in the Instruction.txt. We just ignored all the -1 pixels and considered them as being 0 (Occupied with probability 1). We initially started with using the map as it was, but then, since it was flipped, we flipped it as well and worked on the map in the following orientation.

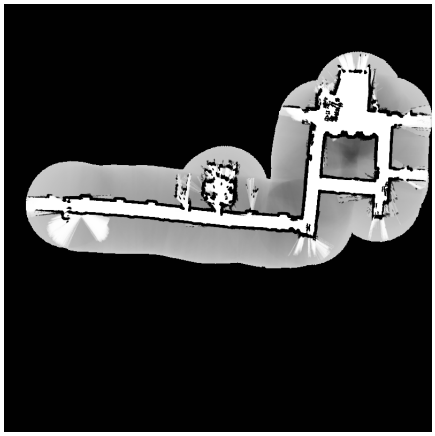

Figure 1: Map

The Log Files

We were given 5 log files, each showing the Odometry and the laser range data. The Odometry gives out the Odometer reading, which is a set of x, y, θ values, and the time stamp. The Laser gives out the Odometer x, y, θ and the laser x, y, θ, as it is 25cm offset from the Robot center. Also, it gives the 180 range readings in cms from the 180 lasers in the sensor, which are spaced 1 degree from each other.

We used *robotdata1.log* and *robotdata4.log* for checking the rightness of our algorithm, and recorded it.

The Particle Filter

Now let's jump into the core of the algorithm. The overall idea behind the algorithm is as follows.

1. Create initial particle set.
2. Run the motion model on each particle based on the log.
3. Run the measurement model to find out the weight of each particle in the particle set.
4. Normalize the weights and resample particles based on the weights of each particle.
5. Repeat steps 2 through 5 till the end of the log.

So, let's go into the individual parts.

The Initial Particle Set

One of the important things was to choose to high enough number for the number of particles in the particle set. Only if we choose to have many particles in the initial set, we will have

guarantees for convergence. We spawned 20,000 particles uniformly randomly, all over the map, except at unknown places on the map and places having obstacles with probability '1'. We also limited our range in sampling particles along each axes. We dealt in the image space, meaning, our robot pose was in the range $([0:800],[0:800],[-\pi:\pi])$. Our initial particle set looks something like this.

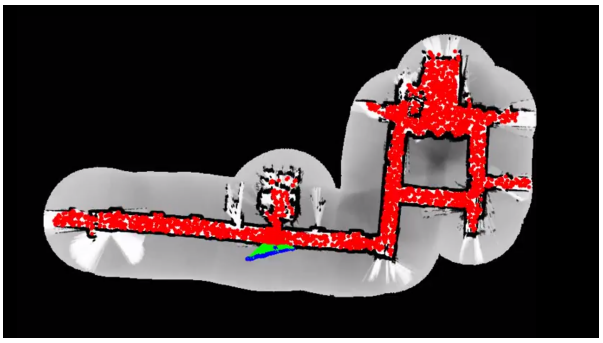

Figure 2: Initial Particle Set

The Motion Model

After we initialize the particles, we have to run the motion model on each particle. The log, as mentioned before, has 2 types of readings - The Odometer reading and the Laser Reading. Each one carries with it the position as measured in the robot odometry frame. We chose to do 2 things. Either to run the motion model for both the readings, or just do it for the laser readings. Depending on which log data we used, we did both. For *robotdata1.log* we used just the laser readings for running the motion model, and for *robotdata4.log* we used both the readings to run our motion model.

We used the algorithm **sample_motion_model_odometry** from the book. The idea behind the motion model is simple. The robot says it has moved so much, but due to the inherent errors in the reading, we have to incorporate stochasticity in our model. The errors can be from wheel slip, irregular floor surface, limited resolution of the wheel encoders, etc. The stochasticity is adding in three flavors- the initial turn of the robot, the translation of the robot, and the final turn of the robot.

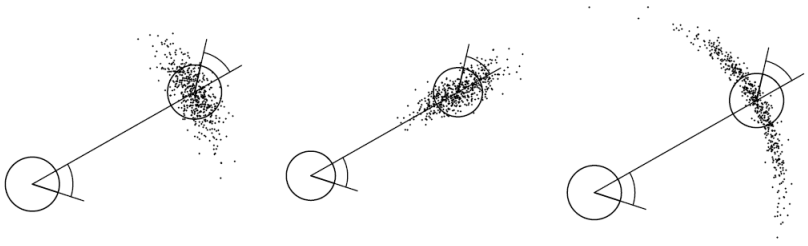

Figure 3: Motion Model

Hyperparameters

As you can see from the figure, the spread of the particle can be tweaked to make it more (or less) random. This can be done using the 4 hyperparameters in the motion model code. They are $\alpha1, \alpha2, \alpha3, \alpha4$. After tweaking it a bit, we found that the best parameters for *robotdata1.log* was

$$\alpha1 = 0.0004, \alpha2 = 0.0007, \alpha3 = 0.0007, \alpha4 = 0.0004$$

and for *robotdata4.log* was

$$\alpha1 = 0.0004, \alpha2 = 0.0007, \alpha3 = 0.0007, \alpha4 = 0.0004$$

Here, $\alpha1$ and $\alpha2$ directly affects the stochasticity in the two rotations and $\alpha3$ and $\alpha4$ affects the stochasticity in the translation.

The following image shows two particles being run from the *robotdata1.log* data and using the motion model. This almost mimics the error in the odometry data from the file. But, as we sample thousands of particles, there will be many particles which will land in the spot where the robot actually was.

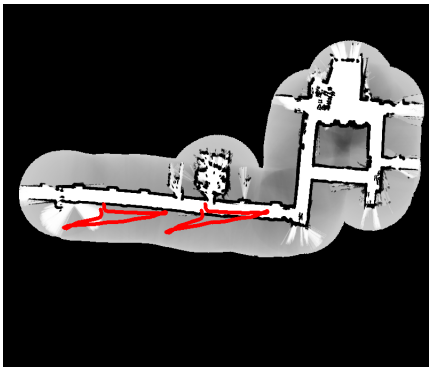

Figure 4: Particles moving

The Measurement Model

So far, we have initialized the particles, and then ran the motion model with the log data on each particle. Then we see the laser data from the log, and find out the probability of the the measurement coming from each particle. There are multiple parts involved in making the measurement model work.

The Distance Table

First, we need the ground truth- what would have been the distance measured from the laser sensor at each point on the map, for valid points. We computed the distance metric for points which are not obstacles and known in the map. Since this was one of the most

computationally expensive steps in the entire stack, we decided to store it before running the particle filter. The distance table was a $800 * 800 * 360$ array computed by doing ray casting. We decided to have a threshold for an obstacle at **0.15** (probability of it being an obstacle is **0.85**). The following image gives us a better understanding of what's happening under the hood.

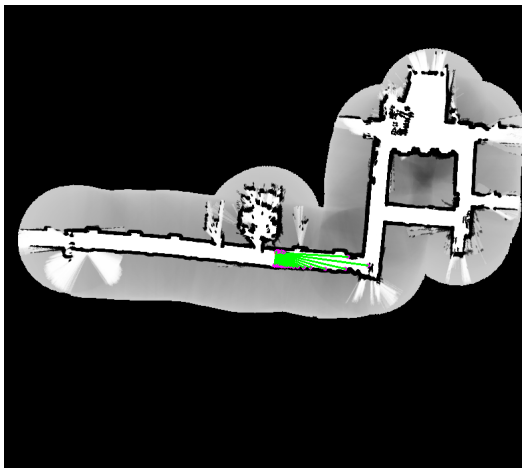

Figure 5: Ray Casting

Hyperparameters: The Four Parts to the Measurement Model

As discussed in the book, the measurement from the laser sensor can be from 4 different sources.

- The actual Obstacle.
- Dynamic objects in the world.
- Random noise.
- Max reading from the sensor.

These four parts have individual parameters to tweak. The variance of the gaussian that parameterizes the reading due to the obstacle, the λ_{short} that parameterizes the probability of seeing an object in front of the obstacles, and the four mixing coefficients for the four modes.

We gave the maximum weight to the mode that corresponds to the obstacle (Z_{hit}) for both the log data. But careful consideration and parameter tweaking went into making this work.

A few things we noted with *robotdata1.log* were:

- There were Dynamic Obstacles (Humans) around the robot, which gave many short readings. To tackle this, we gave a large value to Z_{short} which models these kinds of readings.

- We also needed high variance for the Gaussian.
- Because of symmetry in the corridor, we needed to tweak a lot these parameters so that the particles converge to the right place.

And, the following set of parameter values gave us the best results for *robotdata1.log*

$$Z_{hit} = 500, Z_{short} = 2, Z_{rand} = 200, Z_{max} = 0.001, \sigma^2 = 60, \lambda = 0.1$$

A few things we noted with *robotdata4.log* were:

- There were a lot of random values thrown out by the sensor, which caused the robot to not converge properly. To tackle this, we gave a large value to Z_{rand} which models these kinds of readings.
- We also needed high variance for the Gaussian.
- Because of symmetry in the corridor, we needed to tweak a lot these parameters so that the particles converge to the right place.

For *robotdata4.log*, these values gave us the best results:

$$Z_{hit} = 500, Z_{short} = 1, Z_{rand} = 300, Z_{max} = 0.001, \sigma^2 = 70, \lambda = 0.005$$

The plot for $Z_{star} = 1000$ gave us the curve below for the measurement model.

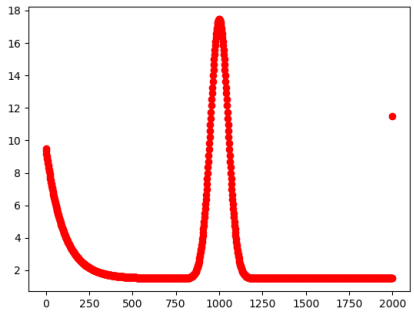

Figure 6: Measurement Model

The Measurement Probability

Notice that the pdf doesn't integrate to 1 in the above curve. This is because when we compute the probability, we multiply a lot of probabilities and that results in the measurement probability becoming very tiny. And since we normalize anyway in the Particle Filter, we decided to weigh them such that they don't drop too low. Another thing to note is that we didn't use all 180 readings from the laser. We decided to use only every 5th laser reading so that we compromise on computational efficiency and accuracy.

Normalize and Resample

So, we have run the motion model, and found the weight for each particle in the particle set using the measurement model. Since the pdf didn't integrate to 1 in our measurement model, we normalize the weights for all the particles, such that they all sum to 1. And then comes the most interesting part of the algorithm. The Resampling procedure in the Particle Filter is an instance of the Importance Sampling technique. We used the **low_variance_sampler** algorithm from the book. This just takes in one pseudo-random number, and then it uses that to resample particles, still proportional to their weights. This algorithm is efficient and doesn't lose as many particles as a normal resampler, which generates M random numbers to resample, would. Another thing that we didn't do was to resample when the robot didn't move. This is because, it would be better to keep all the particles in our set when the robot doesn't move, so that we don't end up with a single pose which has the highest weight in the particle set. Also, another important thing that we did was to resample at a slower rate, because of the symmetry in the corridor. Slower in the sense that we didn't sample when the robot only moved by a small amount, which was **15cm** in our case. This helped us a lot in making the particle set converge.

Lather, Rinse, Repeat

So, with everything in hand now, we just have to keep running the above loop over and over till the end of the log file. This is how the particle set looks like when it converges on the *robotdata1.log* file towards the end of the file:

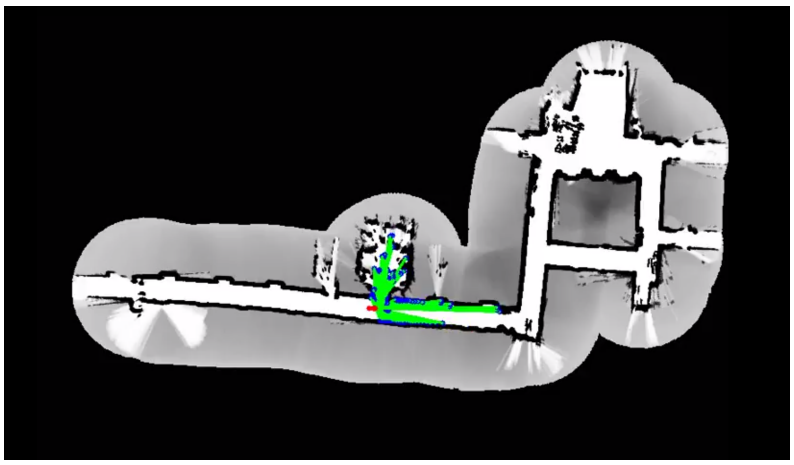

Figure 7: Particles Converging on *robotdata1.log*

And the same on the *robotdata4.log* file towards the middle of the file.

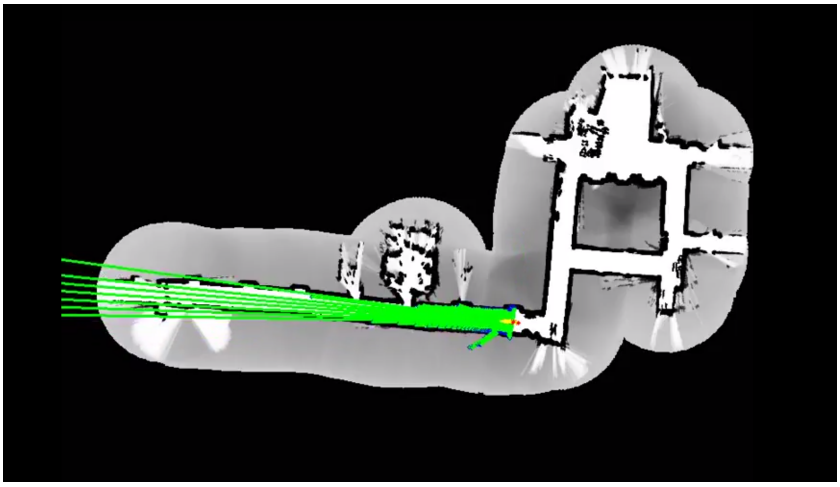

Figure 8: Particles Converging

Future Work

There were many things that we noticed along the way when we were tackling this problem. A few improvements to our algorithm is listed below.

- The particles don't converge well when the number of particles in less. Only when we had over 20,000 particles the particles converged to the right location.
- The code as written is very slow. We could have parallelized the code more, and vectorized in many more places than there already is. We just vectorized the measurement model. The functions within the **particle_filter.py** that apply the motion model and the sensor model itself can be vectorized. Using a lot of for loops has downgraded our performance. If we were to work on this further, we will definitely look into making it faster. Given enough time, we would consider moving the code-base to C++ and running highly optimized code.
- We hand-tuned all the parameters in the models. We would definitely consider using Expectation-Maximization to learn the intrinsic parameters of the models.
- Since we use the same number of particles throughout the algorithm, and it takes a lot of time to do one run, it is highly inefficient to have the same number of particles throughout the algorithm's run. Instead, we can have adaptive number of particles at each iteration, depending on our confidence levels and the spread of the particle set.
- We would also like to solve the kidnapped robot problem, which seems really interesting to us.

There could still be more advances and things we could consider for our Particle Filter. But, owing to space and time, we end our discussion here.