The Infamous Particle Filter
16-831 HW3: Fall 2011

Natasha Kholgade, Heather Knight, and Marynel Vázquez

1 Introduction

In this assignment, we address the problem of localizing a lost robot within a known map, given its odometry and laser rangefinder data. We implement a particle filter that (1) initializes N particles, (2) uses odometry measurements as principal input to the robot's motion model, (3) uses laser ray casting to assess particle quality with respect to real world laser measurements, (4) resamples the particles when the weight variance rises about a threshold. Particle filters are great for situations in which we have sensor and motion data from a known map but no ground truth. We observe good localization performance with a choice of $800 - 1000$ particles and noise parameters of $\alpha_1 = .03, \alpha_2 = .03, \alpha_3 = .01, \alpha_4 = .01$ for the motion model. As the accompanying videos demonstrate, our implementation worked successfully on the first and second data logs.

2 Implementation

We implement almost our entire system in MATLAB, except for the ray casting algorithm. We implement the latter in C++, since it is computationally intensive. We use MATLAB Executable (MEX files) to bridge the gap between the two programming environments.

2.1 Particle Initialization

Given that the dataset depicts a robot traveling through a building, we can assume that the robot was always inside either a room or the hallway. Thus, we generate our initial samples as a uniform distribution over all pixels that are known but unoccupied with a probability of one (see Fig. 1).

2.2 Motion Model

We use the motion model described in Thrun et al. [1] in this assignment. Our motion model uses the odometry data only to estimate the motion relative to the current particle frames. also incorporating Gaussian noise to model the presence of inaccuracies in the odometry sensing. Figure 2 depicts results from our motion model testing phase.

2.2.1 The Motion Equations

To get motion estimates from one odometry configuration $(\tilde{x}, \tilde{y}, \tilde{\theta})$ to another $(\tilde{x}', \tilde{y}', \tilde{\theta}')$ on a two dimensional plane, the robot needs to perform a rotation to align with the direction of travel, a translation along that direction, and a rotation to adjust itself to the final orientation (Figure 3).
The initial change in rotation can be specified as:

$$\delta_{rot1} = \text{atan2}(\tilde{y}' - \tilde{y}, \tilde{x}' - \tilde{x}) - \tilde{\theta}$$

The change in translation is the distance travelled between the initial and final locations:

$$\delta_{trans} = \sqrt{(\tilde{x}' - \tilde{x})^2 + (\tilde{y}' - \tilde{y})^2}$$

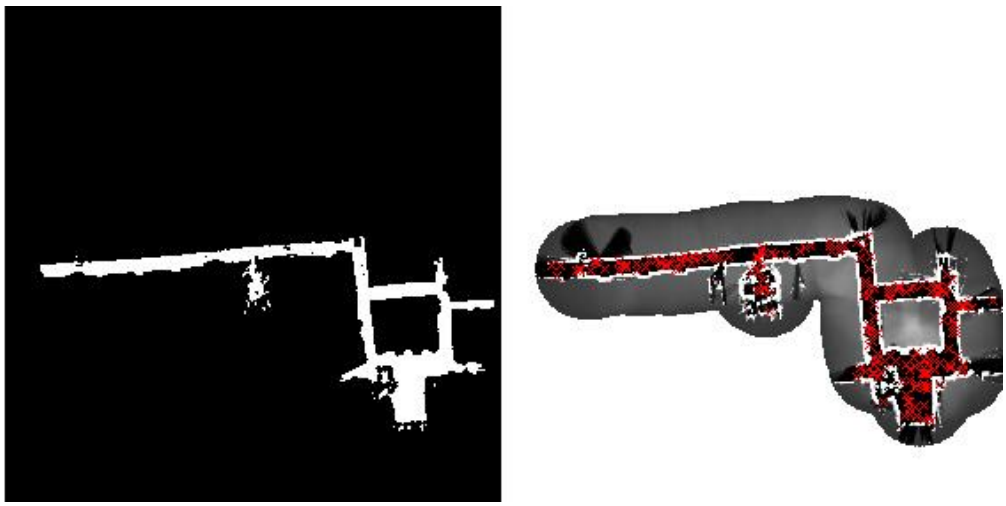


Figure 1: Particle Initialization: (left) map of all pixels that are known but unoccupied with a probability of one, (right) full map with initialization with 200 particles

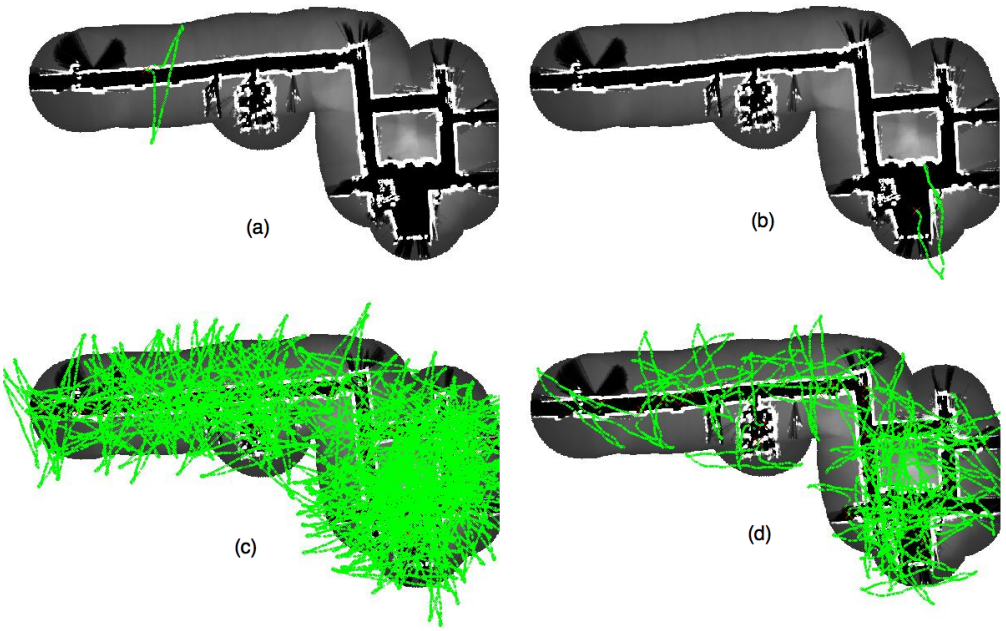


Figure 2: Motion Model for (a) one particle with no noise, (b) one particle with noise, (c) 200 particles no noise, (d) 200 particles with noise

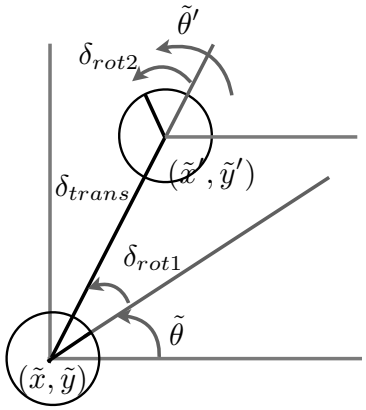


Figure 3: Motion model for a robot moving from configuration $(\tilde{x}, \tilde{y}, \tilde{\theta})$ to $(\tilde{x}', \tilde{y}', \tilde{\theta}')$

The final change in rotation is given as

$$\delta_{rot2} = \tilde{\theta}' - \tilde{\theta} - \delta_{rot1}$$

2.2.2 Incorporating Noise

We assume that the actual values of rotation and translation can be described by removing independent noise ϵ_b with zero mean and variance b:

$$\hat{\delta}_{rot1} = \delta_{rot1} - \epsilon_{\alpha_1|\delta_{rot1}|+\alpha_2|\delta_{trans}|}$$

$$\hat{\delta}_{trans} = \delta_{trans} - \epsilon_{\alpha_3|\delta_{trans}|+\alpha_4|\delta_{rot1}+\delta_{rot2}|}$$

$$\hat{\delta}_{rot2} = \delta_{rot2} - \epsilon_{\alpha_1|\delta_{rot2}|+\alpha_2|\delta_{trans}|}$$

The updates to the translation and rotation are given by:

$$x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1})$$

$$y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1})$$

$$\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$$

2.2.3 Sampling Distribution

We obtain all our samples from a normal distribution centered around the quantity of interest:

$$x \sim \mathcal{N}(\overline{x}, \sqrt{b})$$

To obtain a random sample according to the normal distribution, we randomly generate a set of 12 values from the uniform distribution between -1 and 1, sum them, and multiply by the factor $b/6$ where b is the variance of the distribution.

2.2.4 Training the $\alpha's$

We found that having larger noise for the rotation parameters α_1 and α_2 versus the translation parameters α_3 and α_4 was important. The rotation space for all possible $\alpha's$ given the number of particles we have is very large, thus we found small rotation parameters made it unlikely that our particles would happen to be at the best orientations. Small horizontal and vertical translation errors were easier to recover from by comparison.

2.2.5 Validation

We tested the motion model individually before integration. As shown in Figure 2, first we generated a single particle and ran the the motion model without noise. Its motion mimicked the motion from the odometry data as expected. Next, we added noise to the motion model, and increased the number of particles. When the particles were initialized at the same position and the motion model ran with noise, we obtained a satisfactory banana-shaped spread.

2.3 Sensor Model

We implemented three different methods for assessing the sensor model. First, we focused on a simple probabilistic map model. Consider a particle with laser position $[x\ y\ \theta]$, and a laser measurement $L = [l_0\ \ldots\ l_{179}]$. We computed where the laser beams fall in the map as follows:

$$\big(\forall_i \mid i \in \mathbb{N}^0 \wedge 0 \leq i < 180\ |\alpha = i\pi/180\ \wedge\ map\_pos_i = [x + l_i\cos(\alpha),\ y + l_i\sin(\alpha)]\big)$$

and then we extracted the occupancy probabilities for all these places:

$$o_i = smoothed\_map(round(map\_pos_i * scale)), \text{ with } scale = 1/10$$

Note that a smoothed version of the occupancy map (*smoothed_map*) was used instead of the original values. This typically helps beams that fall close to an occupied space in the map, but not exactly.

The weight of the particle was finally obtained in this approach by summing and exponentiating the probabilites o_i for all beams.

Particles tended to move outside of the free space in the map with our first implementation. We considered a reduced number of uniformly distributed beams from the full measurement to make the independence assumption more reasonable, but our results did not improve. To try to accommodate this situation, we decided to use the number of hits for a laser measurement, instead of the occupancy probabilities directly. This meant transforming o_i to either 0 or 1, depending on whether beam i falls in a free or occupied space in the map:

$$hit_i = \begin{cases} 1, \text{ if } o_i >= H \\ 0, \text{ if } o_i < H \end{cases}$$

typically with $H = 0.9$. We then summed up and exponentiated the number of hits to compute the weight of the particle.[1]

After several tests, we could not get our second approach to work. Particles did not remain in the unoccupied space of the map, even after ignoring laser beams with values outside the expected range. The maximum expected value for a laser measurement was always set to 8000 cm, while the minimum varied from 0 to 50.

After limited success, we elected to implement a ray tracing based method, first in MATLAB and then in C++.

2.3.1 Ray tracing Model

The weight for a particle is computed from the difference between the laser measurement L (captured in the real world by the robot) and an *ideal* measurement $\hat{L}$ taken from the position of the particle in the map. The process goes as follows: (1) we ray cast laser beams from the laser position $[x\ y\ \theta]$ of a particle, then (2) we find the location where the rays hit for the first time an obstacle in the map, and (3) finally use the distances from the the laser position to the hit locations as the components of the ideal laser measurement.

[1] Exponentiation was useful to avoid zero weights.

We process all laser beams i, with $i = 1 \dots 179$, as detailed below:

$beamAngle_i = \theta + i * \pi/180 - \pi/2$ // beams start from right to left
$orientation_i = [cos(beamAngle_i), sin(beamAngle_i)]$
$curr\_step = step$
while $curr\_step < maxL$:
    $beamEnd_i = laser\_position + orientation_i * currStep$
    $o_i = map(floor(beamEnd_i * scale))$
    if isUnknown(o_i) :
        $beamEnd = laser\_position + orientation * maxL$
        $break$
    elif $o_i > H$:
        $break$
    endif
    $curr\_step+ = curr\_step$
end
$distance_i = norm_2(laser\_position - beamEnd_i)$

The variables $scale$, $step$, $maxL$, and H represent the map scale, the step to move forward along the beam,[2] the maximum allowed range value and the occupied probability threshold, respectively. The function isUnknown(.) tells wheter the occupancy probability o_i is outside the allowed $[0, 1]$ range.[3]

The values $distance_i$ compose the ideal laser measurement $\hat{L} = [distance_0 \ \dots \ distance_{179}]$. Typically, $\hat{L}$ would have 180 components, but if some values in the real laser measurement L are outside the expected range, then their respective beams are ignored both for L and $\hat{L}$.

To obtain a value for the weight of a particle we first subtract L and $\hat{L}$. This operation gives us the differences in range per beam. Then we compute probabilities for the differences under a Normal distribution with zero-mean and variance σ_{beam}. Ideally, all differences would be zero, hence the zero mean, but to overcome measurement noise we allow some variation in the readings using σ_{beam}. Finally, we sum all probabilities and exponentiate. In other words,

$$w = exp(\sum_i (\mathcal{N}(L_i - \hat{L}_i; 0, \sigma_{beam})))$$

2.3.2 Validation

We created an interactive point and click interface to validate the ray casting model. Using the application, we positioned different particles on the map, and run our ray casting algorithm for each of them. Images such as the right-most in Figure 4 were obtained during this testing phase.

We assume laser beams are independent, and designed our code to allow for measurement subsampling. We ran tests of the sensor model using all 180 beams, but also using every other beam (90 in total) and less. We corroborated later that skipping too many beams (e.g., more than 4) in favor of more independence between measurements did not improve our results.

Once we were confident in the sensor model itself, integrating this step into the overall particle filtering functioning involved many iterations of parameter testing. Challenged we faced were choosing how often to resample (discussed in the next section), evaluating α values to fit the sensor model, and matching up coordinate systems across the code of the various project contributors.

[2]The step is chosen small enough so that no cell in the occupancy map is skipped along the ray.
[3]Some occupancy values in the map are unknown.

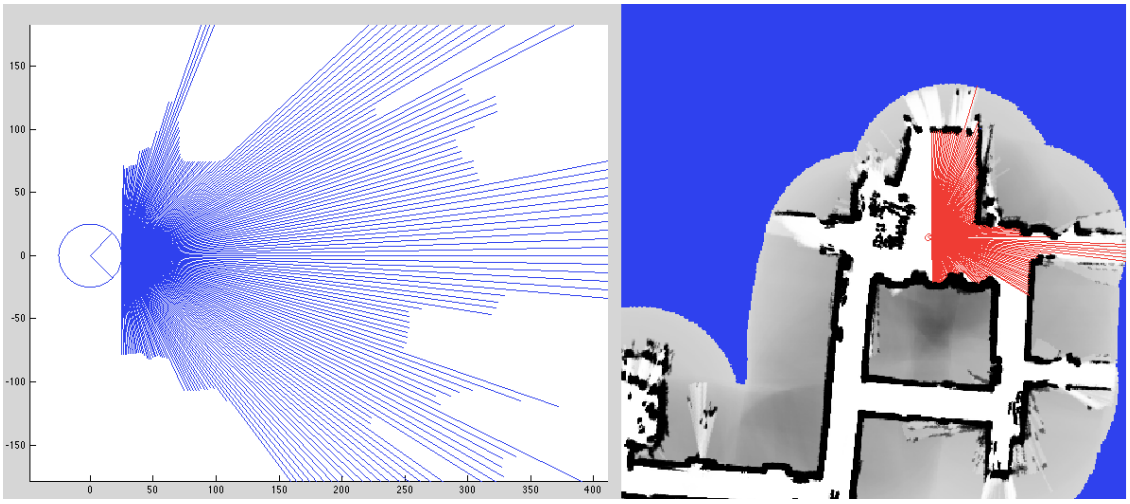


Figure 4: Laser data visualization: (left) real laser measurement, (right) ray casting result

2.4 Resampling Step

As discussed in class, we used Normalized Importance Resampling to redistribute the particles over the previous particle space in a manner that reflects their overall probability levels as expressed by the particles most recent weights.

2.4.1 Methodology

Specifically, for the N particles we create N bin indices by using their normalized weights to create a continuous PDF. We accomplish that by creating a cumulative sum of weights for each sequential particle. Next, we generate a random number between zero and one as the first resampling position, using the continuous PDF index of that location to assign the first particle in our new set of particles. After that we iterate $N-1$ times, moving our sampling position forward by $1/N$ with a circular buffer. When resampling is complete, we match our new particles with the stored indices and reset all weights.

2.4.2 Resampling Frequency

We do not resample the particles at explicit timestep, but rather track the overall variance in the weights and only perform resampling when this number has surpassed a certain threshold. We experimentally determining a good variance threshold by tracking how long it took unlikely particles or particle clusters to die out, while maintaining enough diversity in the particle distribution to allow reliable convergence.

Resampling too often can lead to lack of particle diversity and thus loss of information, not resampling enough means the algorithm will not converge and means fewer additional particles will be added to the true particle location, which could make that true particle location less robust to noise.

3 Algorithm Summary

Our particle filter code consists of the following steps:

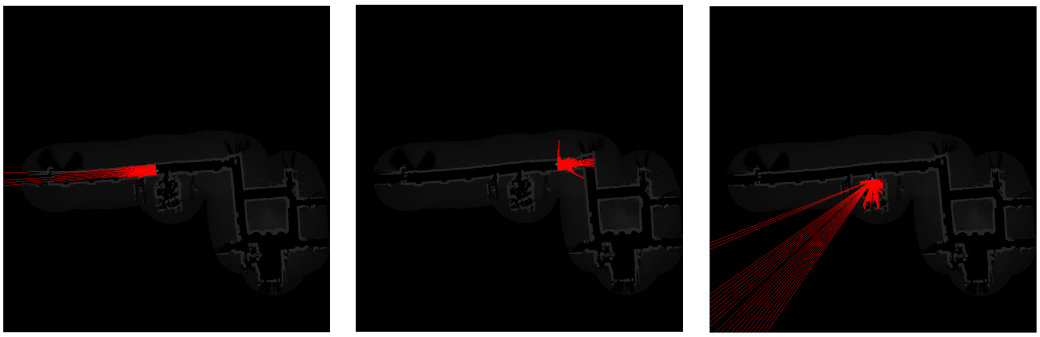
```
Generate N particles using a uniform distribution over the unoccupied \
  positions in the map and all possible headings.

While there is still data:
  If next reading is Odometry data:
     Sample particles using the motion model.
```

```
      Move any particle that goes off the map to the border.
   If next reading is Laser data:
      Filter the laser data (e.g, ignore beams outside of expected range)
      Compute weight for each particle using the ray casting approach
   If weights variance is above threshold:
      Resample particles (and set new particle weights to 1/N)
```

It also generates the map and sensor visualization for each step along the way.

4 Results



(a) (b) (c)

Figure 5: Plots of the robot's location for 'robotdata2.log' at frames (a) 270 (few frames into the start of the route), (b) 3670 (midway), and (c) 5360 (final frame)

As the accompanying videos demonstrate, our implementation worked successfully on the logs 'robotdata1.log' and 'robotdata2.log'. Note: to execute our accompanying code, use the script:

```
runParticleFilter
```

Before using the code, execute at the command prompt

```
./compile.sh
```

which is located in the same directory (change the include path in the 'compile.sh' script to point to your version of MATLAB), and then call

```
mex CXX=g++ -lm particle.o vector2D.o raycast.cpp -output raycast
```

before running the particle filter.

As the movies show, our particle filter algorithm converges to the correct global position within around 200 iterations. We depict the particles as blue dots. For the highest likelihood particle, we plot an oriented circle with triangle depicting laser location and use its pose to transform the filtered laser data into red beams in the world coordinates. For the first data log, we used 1000 particles, and for the second data log, we used 800 particles. We found that noise parameters $\alpha_1 = .03, \alpha_2 = .03, \alpha_3 = .02, \alpha_4 = .02$ worked best to produce the desired results.

References

[1] Thrun, S., Burgard, W., and Fox, D. *Probabilistic Robotics*, The MIT Press, 2005.