16-833 SLAM HW1: Monte Carlo Localization

Jeremy Kilbride (Andrew ID: jkilbrid) Ryan Wu (Andrew ID: weihuanw)

February, 17 2024

In this report, we investigated a practical implementation of the Monte Carlo Localization (MCL), also known as particle filter. We will be discussing the team's approach, implementation, performance, and future work. Overall, our results were satisfactory considering the given data and computation resources available.

1 Our Approach

We completed this assignment by first researching the details of the algorithm in the course textbook: *Probabilistic Robotics* by Thrun, Burgard, and Fox. Our implementation is largely based on pseudo-code and equations provided in this book. After researching the implementation details we first implemented all components in Python. We wrote vectorized code in Python as much as possible. Each component was tested individually, and once all components were functional independently, we integrated them to all work together in the main script. After getting the visualization to work by running the main script, we rewrote our ray casting code in C++ and used the package pybind11 to bind our C++ code to Python. Once this was complete, we were able to run the main script in near real-time. Having a reasonably fast implementation allowed us to tune our implementation reasonably quickly. Below is a high-level pseudo-code from *Probabilistic Robotics* for the MCL algorithm.

1: **Algorithm Particle_filter**($\mathcal{X}_{t-1}, u_t, z_t$)**:**
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: for $m = 1$ to M do
4: sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M do
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to $\mathcal{X}_t$
11: endfor
12: return $\mathcal{X}_t$

Figure 1: **Table 4.3** of *Probabilistic Robotics* for the MCL algorithm.

1.1 Parameter Tuning

There are several parameters to consider when trying to implement an MCL. These parameters are number of particles initialized, four motion model's hyper-parameters α_1-α_4, and six sensor model's intrinsic parameters z_{hit}, z_{short}, z_{max}, z_{rand}, σ_{hit}, and λ_{short}. In our implementation, we used the trial and error

method until the particle's convergent is satisfactory. Our parameter values are: α_1= 0.01, α_2= 0.01, α_3= 0.001, α_4= 0.001, z_{hit}= 1.22, z_{short}= 0.15, z_{max}= 0.1, z_{rand}= 550, σ_{hit}= 63.5, λ_{short}= 0.1, with 10,000 particles initialized in the map's free space.

2 Implementation

2.1 Motion Model

The first component of our MCL program that we implemented was the motion model. Our implementation was based on the pseudo-code provided in **Table 5.6** of *Probabilistic Robotics* shown below:

1: **Algorithm sample_motion_model_odometry**(u_t, x_{t-1})**:**

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

3: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$

4: $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5: $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$

6: $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot1}} + \delta_{\text{rot2}}))$

7: $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

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

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

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

11: *return* $x_t = (x', y', \theta')^T$

Figure 2: **Table 5.6** of *Probabilistic Robotics* for motion model framework.

Our motion model is vectorized using NumPy, so we modify all particles simultaneously. Our motion model first calculates the change in heading for each particle before movement δ_{rot1}, then calculates the change in position for each particle δ_{trans}, and finally calculates the change in heading for each particle after movement δ_{rot2}. These calculations are purely based on odometry data from the log files. Next, each of the three calculated δ values for each particle has some noise added to it by sampling from three different normal distributions to give new values $\hat{\delta}_{rot1}$,$\hat{\delta}_{rot2}$, and $\hat{\delta}_{trans}$. The standard deviation of each distribution is calculated based on four hyper-parameters α_1-α_4. After noise is added, a new x, y, and θ is calculated for each particle based on $\hat{\delta}_{rot1}$,$\hat{\delta}_{rot2}$, and $\hat{\delta}_{trans}$. These newly calculated particle state beliefs are then returned as a single NumPy array.

2.2 Sensor Model

Our Sensor Model has two main parts: ray casting and likelihood estimation.

2.2.1 Ray casting

Our ray casting algorithm is based on a blog post from Joel Schumacher found at `https://theshoemaker.de/posts/ray-casting-in-2d-grids`. We used what Schumacher calls the "exact" method for ray casting. Our pseudo-code based on this blog post is below:

Algorithm 1 Ray Casting

1: initialize empty array $distances$ of size $[NumParticles, 180/subsample]$
2: **for** i in range $NumParticles$ **do**
3: get $particle.heading$, $particle.coordinates$
4: $direction = particle.heading - \frac{\pi}{2}$
5: $gridX, gridY = floor(paricle.coordinates, map.resolution)$
6: $CurrentCoordinate \leftarrow particle.coordinates$
7: **for** j in range 0 to $(180/subsample)$ **do**
8: **if** $cos(direction) \leq 0$ **then**
9: $Xincrement = -1$
10: **end if**
11: **if** $sin(direction) \leq 0$ **then**
12: $Yincrement = -1$
13: **end if**
14: $RayLength = 0$
15: **while** $map[gridX, gridY] \leq 0.01$ **do**
16: find dtX distance to next edge in X
17: find dtX distance to next edge in Y
18: $dt = min(dtX, dtY)$
19: $RayLength+ = dt$
20: **if** $dt = dtX$ **then**
21: $gridX+ = Xincrement$
22: **else**
23: $gridY+ = Yincrement$
24: **end if**
25: **end while**
26: $distances[i, j] = RayLength$
27: $direction+ = \frac{\pi}{180} * subsample$
28: **end for**
29: **end for**
30: **return** $distances$

Our code loops over each particle individually. In the above algorithm, lines 1 through 7 are specific to our case, and lines 9 through 26 make up the central logic of the algorithm based on Schumacher's article. For each particle the code loops over each beam. The heading of the rightmost beam is determined by subtracting $\frac{\pi}{2}$ from the particle heading. The heading for each successive beam is determined by adding $\frac{\pi}{180} * subsample$. $subsample$ is an integer representing how degrees are between each heading that we use for ray casting. Once the particle coordinates and the heading are obtained, we then begin the actual ray casting. In Schumacher's method, the grid coordinates are obtained by floor dividing the global coordinates of the particle by the grid size (in our case this is the map resolution). Next, we obtain the directions that we want to move based on the heading of the beam. Essentially this portion of the algorithm determines which quadrant of the X-Y plane the heading points towards and then decides which way the grid coordinate will be incremented. For example, if the heading is 120°, then the algorithm will determine that X needs to be incremented by -1 and Y needs to be incremented by $+1$ each time those coordinates are updated. Now the actual ray-casting starts. The current global coordinates and ray length are successively updated based on the distance to the closest edge in the grid along the current heading. The grid coordinates are also successively incremented based on whether the next edge is in the x direction or y direction. Each time the loop runs we check the value of the occupancy grid map. If the value is above 0.01, then we stop the while loop and add the ray length to the array of ray lengths. For our case, we also check if the value of the occupancy map is -1 as a break condition for the while loop

2.2.2 Likelihood Estimation

The likelihood of each particle is calculated by evaluating the four different probability distributions as described in *Probabilistic Robotics* chapter 6. The pseudo-code for this method is from **Table 6.1** of *Probabilistic Robotics* and shown below:

Algorithm beam_range_finder_model(z_t, x_t, m)**:**

$q = 1$

for $k = 1$ *to* K *do*

compute z_t^{k*} *for the measurement* z_t^k *using ray casting*

$p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$

$+ z_{\max} \cdot p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

$q = q \cdot p$

return q

Figure 3: **Table 6.1** of *Probabilistic Robotics* for sensor model's likelihood calculations.

The first distribution is a Gaussian distribution with the mean being z_t^*, which is an array of expected values for the sensor measurement obtained from ray casting. The second distribution is a uniform distribution to represent random process noise. The third distribution is an exponential distribution which represents the laser hitting an object that was not detected on the map. The final distribution is a point distribution at the maximum sensor range. This represents the beam going beyond the max range, in which case the measurement would just be the max range. Our implementations of these distributions were based on equations 6.2-6.10 from *Probabilistic Robotics*. The likelihoods from these four distributions are each multiplied by the hyperparameters z_{hit}, z_{short}, z_{rand}, and z_{max} and then summed to give us the likelihood for each measurement. Next, we take the logarithm of these likelihoods, sum them, and then take the exponential of them. Lastly, we normalize the likelihoods and return them to the main script.

2.3 Resampling

Our algorithm for resampling is based on pseudo-code for low-variance sampling from *Probabilistic Robotics* shown below:

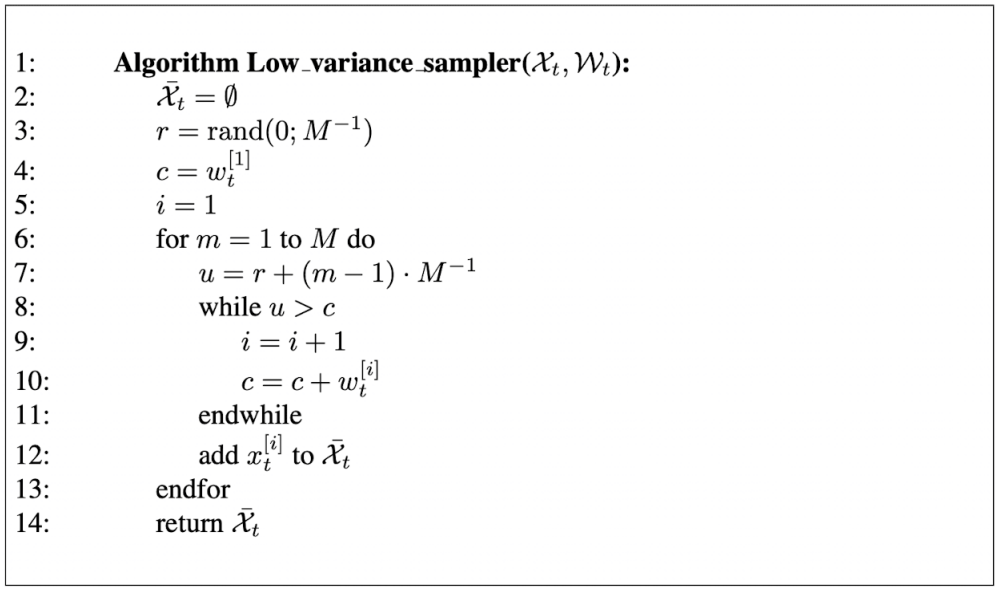

Figure 4: **Table 4.4** of *Probabilistic Robotics* for low-variance sampling.

This algorithm uses two cumulative sums to determine which particles to resample. The quantity u is incremented by M^{-1}, where M is the number of particles, each time a new particle is added to $\bar{\mathcal{X}}_t$ in the for loop; and c is incremented by the weight w_t of the i-th particle in $\mathcal{X}_t$ at each iteration of the while loop. M^{-1} represents the weight of a single particle if all particles had the same weight or in other words the average particle weight. The key to this algorithm is that each particle's weight is compared to the average particle weight. So particles with weights that are above average are more likely to be resampled, whereas the opposite is true for particles with weights that are lower than average. This means that if a certain particle's weight is much higher relative to the other particles, then there will be several iterations of the for loop wherein $c > u$ and i are not incremented. This algorithm is also advantageous because it has linear time complexity, whereas other sampling methods presented in *Probabilistic Robotics* have $Nlog(N)$ time complexity.

2.4 Particle initialization

In our code, the particles are initialized only if free space. Our algorithm for this is below:

Algorithm 2 Particle initialization

```
initialize empty list to store map indices
for i in range(map.height) do
    for j in range(map.width) do
        if 0 ≤ map[i, j] ≤ 0.01 then
            list.append([i,j])
        end if
    end for
end for
shuffle list of indices and select indices for the desired number of particles
```

In essence, our algorithm checks the value of every square in the grid map. If the value of the map is greater than 0 and less than a certain threshold(in this case 0.01), then that is a potential location to instantiate a particle. After checking the entire map we shuffle our list of potential particle locations and select n indices from that list at which to initialize particles, where n is the desired number of particles. The

rest of the process of particle initialization is virtually identical to the random particle initialization function that was provided with the base code.

2.5 Laser Visualization

At each time step our code plots the lasers from the measurement data at that time step, with the start point of the rays being the most likely particle. This is accomplished by first calculating the heading of each beam. The endpoint of each beam is calculated using the heading and distance of each beam along with some trigonometry. Lastly, the beams are plotted in an iterative manner using the start and end points. Finally, after a brief pause, the beams are cleared from the plot.

3 Performance

3.1 Overall Performance

With the deployment of pybind11, our program ran significantly faster compared to using only raw Python code. For example, when running robotdata1.log and robotdata4.log with 10,000 initialized particles, we got a run time of 9 min and 38 s and 7 min and 53 s respectively. We hyperlinked two videos to visualize what we believed to be our implementation properly localizing the robot for robotdata1.log and robotdata4.log. Please be aware both videos are speedup compared to real-time, with robotdata1.log and robotdata4.log having a speed multiplier of around 5 and 4 respectively.

3.2 Repeatability and Robustness

In terms of repeatability and robustness, our implementation was able to localize and converge to the same location approximately every 3 out of 5 runs. However, some run results are illogical with the particles localizing in non-free space. We suspect these outlier runs stem from faulty hyper-parameters, intrinsic parameters, or the given map data. The implementation's robustness, in our opinion, is sufficient for simulation but requires some further tuning and future work results that are reliable enough for real-life deployment.

4 Future Work

Various aspects can be improved in our program. To cut down run time, all calculation algorithms should be run in C++ or even the whole program should be deployed in C++ for simplicity. Furthermore, instead of trial and error during parameter tuning, we can utilize the Learn Instristic Parameters algorithm in **Table 6.2** of *Probabilistic Robotics* to improve robustness. Another interesting avenue could be localizing multi-agents using MCL. Nonetheless, with the given data and computation resources, our implementation proved to be satisfactory in our use case.

Videos

robotdata1.log: https://www.youtube.com/watch?v=c0KEJxJpUXs
robotdata4.log: https://www.youtube.com/watch?v=VU-m3kIBEyo

References

Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT press, 2005.