1.5em 0pt

16-833: Robot Localization and Mapping, Fall 2024

Homework 1
Robot Localization using Particle Filters

Venkata Nagarjun Pudureddiyur Manivannan (vpudured) and Jonathan Lee (jlee6)

1 Motion Model

1.1 Implementation of Motion Model:

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

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

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

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

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot1}} + \alpha_2 \delta^2_{\text{trans}})$

$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta^2_{\text{trans}} + \alpha_4\ \delta^2_{\text{rot1}} + \alpha_4\ \delta^2_{\text{rot2}})$

$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot2}} + \alpha_2 \delta^2_{\text{trans}})$

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

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

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

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

Figure 1: The sampling algorithm used for the Motion Model from [1]

1.1.1 Non-Vectorized Version

The motion model function was called on the position vector x_{t-1} for every particle via a loop.

The variables as mentioned in the algorithm above were initialized and calculated for every particle to return its future position as x_t based on the odometry.

The sampling method used in the motion model used a gaussian distribution whose mean was 0 and a standard deviation as the square of the variance passed as a parameter to the sampler.

The motion model returned a single vector representing the updated position of a particle with a size **3 * 1**

1.1.2 Vectorized Version

The vectorized version eliminated the loop for all the particles. The position of the particles at time step t-1 are passed a vector of size **n x 3** where n are the number of particles.

Numpy broadcasting was used to handle the individual element wise operations on each position vector. The gaussian sampler was altered to give a gaussian noise vector of size **n * 1** to ensure that each particle was given a different noise instead of broadcasting the same noise to all the particles.

2 Sensor Model

2.1 Implementation of the Beam Range Finder Sensor Model

2.1.1 Non Vectorized Version - Part 1

The main function calls the sensor model function n times - one iteration for each particle where the inputs to the sensor model are the sensor measurement $\mathbf{z_t}$ of size **k x 1** and the estimated position vector of the particle from the motion model $\mathbf{x_{t1}}$ of size **3*1**.

Within the sensor model call for a particle, a for-loop run **k** times, one time for each measurement in the $\mathbf{z^k_t}$

Within each iteration the four probabilities as mentioned in the algorithm in 2 is computed. The equations governing the four particles are given in part 2.

2.1.2 Ray Casting

The ray-casting algorithm for each sensor measurements aims to compute the true sensor measurement $\mathbf{z^{k*}_t}$ for the particle had it been in that particular position and orientation. This is used to compare with the given sensor measurement. This comparisons helps us determine how close the current particle we are evaluating is to the actual position and orientation of the robot.

Realtime ray casting for a given position and orientation in all possible directions is extremely computation heavy. Therefore, we build a look-up-table whose inputs are **x,y,theta** and the output is the corresponding sensor measurement as measured from the map had the particle been there. We build the ray-caster using the Amanatide and Woo's Raycasting Algorithm where we determine the closest occupancy from a particular position and a particular orientation by casting a ray to find the first point of occupancy intersection in a voxelized map.

2.1.3 Non Vectorized Version - Part 2

By retrieving the $\mathbf{z^{k*}_t}$, we compute the probabilities of the particle accurately describing the position of the robot with four probabilites mentioned below to come

up with a scalar weight q. This q is cumulatively multiplied across all the sensor measurements for the particle.

The sensor model returns a scalar weight of the particle based on the various probabilities computed.

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*

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

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

$q = q \cdot p$

return q

Figure 2: The algorithm for the sensor model from [1]

- P_{hit}

$$p_{\text{hit}}(z_t^k \mid x_t, m) = \begin{cases} \eta\, \mathcal{N}(z_t^k; z_t^{k*}, \sigma_{\text{hit}}^2) & \text{if } 0 \le z_t^k \le z_{\text{max}} \\ 0 & \text{otherwise} \end{cases}$$

where z_t^{k*} is calculated from x_t and m via ray casting, and $\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{\text{hit}}^2)$ denotes the univariate normal distribution with mean z_t^{k*} and standard deviation σ_{hit}:

$$\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{\text{hit}}^2) = \frac{1}{\sqrt{2\pi\sigma_{\text{hit}}^2}}\, e^{-\frac{1}{2}\frac{(z_t^k - z_t^{k*})^2}{\sigma_{\text{hit}}^2}}$$

The normalizer η evaluates to

$$\eta = \left(\int_0^{z_{\text{max}}} \mathcal{N}(z_t^k; z_t^{k*}, \sigma_{\text{hit}}^2)\, dz_t^k \right)^{-1}$$

Figure 3: Equations used for p_{short} from [1] the normalizer was taken as 1

- P_{short}

$$p_{\text{short}}(z_t^k \mid x_t, m) = \begin{cases} \eta\, \lambda_{\text{short}}\, e^{-\lambda_{\text{short}} z_t^k} & \text{if } 0 \le z_t^k \le z_t^{k*} \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

As in the previous case, we need a normalizer η since our exponential is limited to the interval $[0; z_t^{k*}]$. Because the cumulative probability in this interval is given as

$$\begin{aligned} \int_0^{z_t^{k*}} \lambda_{\text{short}}\, e^{-\lambda_{\text{short}} z_t^k}\, dz_t^k &= -e^{-\lambda_{\text{short}} z_t^{k*}} + e^{-\lambda_{\text{short}} 0} \\ &= 1 - e^{-\lambda_{\text{short}} z_t^{k*}} \end{aligned} \tag{6.8}$$

the value of η can be derived as:

$$\eta = \frac{1}{1 - e^{-\lambda_{\text{short}} z_t^{k*}}} \tag{6.9}$$

Figure 4: Equations used for p_{hit} from [1]

- $P_{\text{Probability of sensor failure}}$ The most common failure of a sensor is it outputting a measurement of as the max range. To model this, we use binary probability where $\mathbf{p_{max}=1}$ if the sensor measurement is greater than or equal to the max value and **0** for any other sensor measurement.

- $P_{\text{Probability of Random Measurements}}$ Crosstalk, ghost measurements, and any other unexplainable measurements are modelled p_{rand} which is binary probability. It

is **1/max range** if the sensor reading is between 0 and max range. It is **0** for any other sensor readings.

2.1.4 Vectorized Version

The vectorized version deviates from the non-vectorized version and is described below:

1. The input to the vectorized version are the sensor measurements $\mathbf{z_t}$ of size **k * 1** and the estimated position vector of the all the particles from the motion model $\mathbf{X_{t1}}$ of size **n * 3**.

2. A vector of size **n * k * 4** to describe the analogous q vector in the non-vectorized version

3. Instead of looping over the **k** sensor measurements and **n** particles, we compute the 4 probabilities simultaneously by using the matrix mentioned above.

4. We then sum it along the last dimension to reduce the **q vector** to a size **n*k** which is analogous to computing probability for all sensor measurements for a single particle and broadcasting the same operation to all n particles.

5. We then perform a product along the last dimension again to reduce it to size **n * 1** which represents multiplying the q iteratively over all the sensor measurements to.

6. The final vector of size **n*1** represents the scalar weight for all the particles after the sensor model is called on them.

3 Resampling

3.1 Implementation of Low Variance Resampler

Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):

$\bar{\mathcal{X}}_t = \emptyset$
$r = \text{rand}(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for $m = 1$ *to* M *do*
  $U = r + (m-1) \cdot M^{-1}$
  while $U > c$
    $i = i + 1$
    $c = c + w_t^{[i]}$
  endwhile
  add $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$
endfor
return $\bar{\mathcal{X}}_t$

Figure 5: Resampling Algorithm as in [1]

Given a vector χ_t of size **n*4** where the each row contains the x,y,theta,weight for each particle after the sensor and motion model update, the resampling model

performs the resampling process over the n particles to assign them new positions and orientations based on the weights from the sensor model.

Resampling is also essential for preventing particle degeneracy and maintaining diversity among particles. It redistributes particles based on their weights, ensuring that particles with higher weights contribute more to pose estimation. This prevents the algorithm from becoming dominated by a few particles and improves its accuracy.

This process helps the particle to converge to those positions which are more likely to be where the robot is actually present.

4 Particle Initialization in Freespace

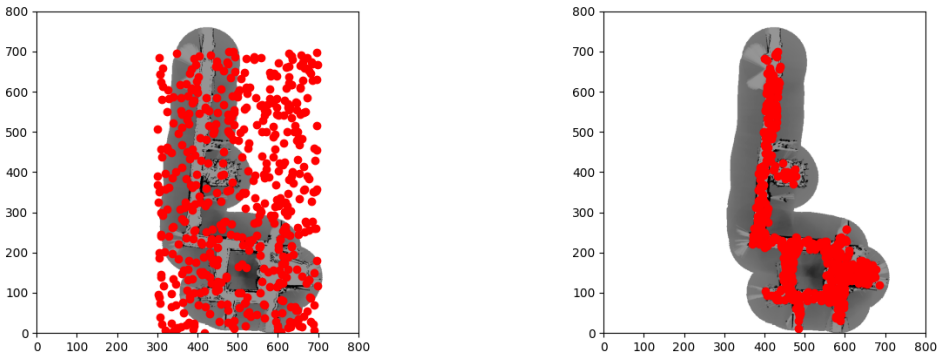

Figure 6: Particle initialization (A) uniformly and (B) in free space

Algorithm 1 Initialize Particles in Free Space

function INIT_PARTICLES_FREESPACE(n, map)
  $\bar{\mathbf{X}}_{init} \leftarrow \mathbf{0}_{n\times 4}$
  $\mathbf{x} \leftarrow \text{Unif}(3000, 7000, n)$
  $\mathbf{y} \leftarrow \text{Unif}(0, 7000, n)$
  $X_{\text{invalid}} \leftarrow \{\langle i, j\rangle | \text{map}[\lfloor \mathbf{y}(i)/10 \rfloor, \lfloor \mathbf{x}(j)/10 \rfloor] > 10^{-6}, \forall i, j < n\}$
  while $|X_{\text{invalid}}| > 0$ **do**
    $\mathbf{y}(X_{\text{invalid}[0,:]}) \leftarrow \text{Unif}(0, 7000, |X_{\text{invalid}}|)$
    $\mathbf{x}(X_{\text{invalid}[1,:]}) \leftarrow \text{Unif}(3000, 7000, |X_{\text{invalid}}|)$
    $X_{\text{invalid}} \leftarrow \{\langle i, j\rangle | \text{map}[\lfloor \mathbf{y}(i)/10 \rfloor, \lfloor \mathbf{x}(j)/10 \rfloor] > 10^{-6}, \forall i, j < n\}$
  end while
  $\theta \leftarrow \text{Unif}(-\pi, \pi, n)$
  $\mathbf{w} \leftarrow \mathbf{1}_n \cdot \frac{1}{n}$
  $\bar{\mathbf{X}}_{init} \leftarrow \text{HSTACK}([\mathbf{x}, \mathbf{y}, \theta, \mathbf{w}])$
  return $\bar{\mathbf{X}}_{init}$
end function

Particle initialization helped with initial particle convergence as the initial particle distribution better covers the actual robot starting distribution (in the hallways and rooms).

5 Tuning

Parameter	Value	Description	Tuning notes
α_1	0.0001	(Motion model) scales rotational variance due to rotation	A very small value works well to prevent particle from flipping direction in long hallways
α_2	0.0001	(Motion model) scales rotational variance due to translation	A very small value works well to prevent particle from flipping direction in long hallways
α_3	0.005	(Motion model) scales translational variance due to translation	Translational variance is slightly larger than rotational
α_4	0.005	(Motion model) scales translational variance due to rotation	Translational variance is slightly larger than rotational
z_{hit}	15	(Sensor model) weight of normal distribution	Not too high since ray caster can return a very different range
z_{short}	2	(Sensor model) weight of exponential distribution	Lower since few rays are blocked by people
z_{max}	1.5	(Sensor model) weight of max range distribution	Lower since few rays reach max range
z_{rand}	500	(Sensor model) weight of uniform distribution	Keep artificially high to prevent distribution of particles from collapsing
σ_{hit}	100	(Sensor model) Standard deviation of normal distribution	Increased due to low resolution of map and ray caster
λ_{short}	15	(Sensor model) scales exponential distribution	Found default value worked well
Subsampling	2	(Sensor model) downsampling factor for number of rays	Good balance between speed and accuracy

Table 1: Parameters, description of their effect, and tuned values

We tuned parameters one at a time and observing the effect on the particles. We used a single log file for testing where we knew the approximate starting location of the robot was in the long hallway. First, we tuned the z_{hit}, z_{rand}, and σ_{hit} parameters for the sensor model, as we noticed the particle weighting was the most sensitive to these values. As noted in Table 1, these parameters also effect the particle convergence which we don't want happen to fast or too slow. Then we found tuning the motion model helped prevent the particles from switching directions 180 degrees where the sensor scans are symmetric (such as in corridors).

6 Results

Log	Log Duration	Alg Runtime	Video	Speedup	Video
1	135s	438s	57s	7.7x	https://youtu.be/pqc2zz75PE4
3	132s	733s	76s	9.6x	https://youtu.be/ShBwDDDnaYY
5	102s	584s	116s	5x	https://youtu.be/agrVBpdqf28

Table 2: Videos of different log files and runtimes (measured with 10000 particles and no plotting).

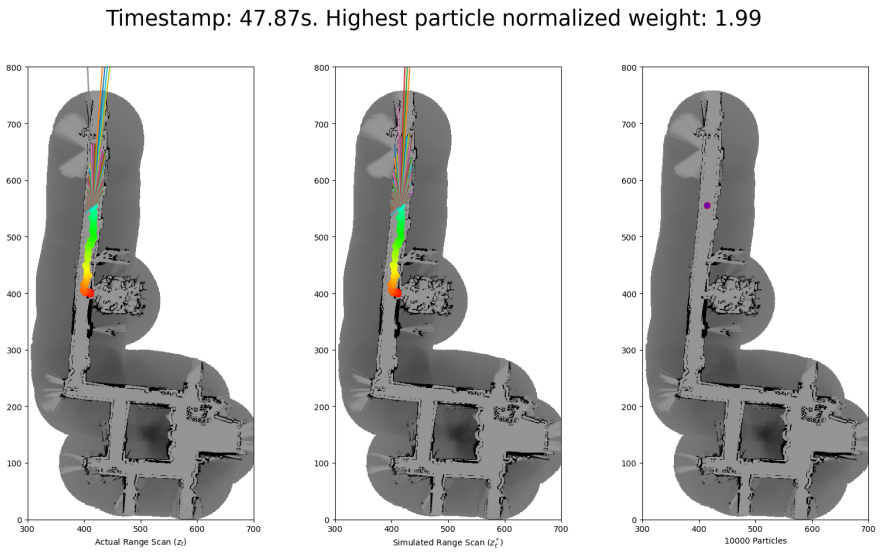

Figure 7: Screenshot of custom debugging tool taken midway throught robotlog5.txt. Left: the actual robot range data overlayed on the particle with highest weighting. Center: the simulated range scan overlayed on the particle with highest weighting. Right: all particles overlayed on the map. The history of best particles is shown using the "hsv" colormap.

7 Performance and Future Work

In terms of time complexity, the sensor model is $O(nm)$ and the motion model is $O(n)$ where n is the number of particles and m is the number of measurements. Therefore the overall computation time is $O(nm)$. Note that our ray casting is pre-computed in a lookup table, so the size or resolution of the map is not included in the time complexity.

In terms of runtime, Table 2 reports speeds that are $\sim$3-5 times slower than the runtime of the log file. However, this is with 10000 particles, which we used in order to ensure the highest accuracy performance of the particle filter. With 2000 particles, the algorithm can run at a realtime rate and could potentially run onboard the robot since it only consumes a single thread of a laptop grade CPU.

Further improvements can be made to the implementation, such as using an adaptive number of particles. One simple solution is to reduce the number of particles over time after the particle filter has converged. Additionally, addressing the kidnapped robot problem is another extension. We could re-initialize the particles if the average likelihood of the scan range matching is below some threshold.

References

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.