Particle Filter Localization – Report

Matteo Villani – 12343046

Initialize particles

In order to initialize the Particle set randomly within the boundaries of the map, I iterate `num_particles_` times: at each iteration a Particle object is instantiated, I compute each particle's pose by generating at random `x` and `y`, aligning with the map's offset by adding the map's origin position. If the particle's cell is occupied (value >0 in the gridMap) I skip this cell, since we can already tell that the robot is not in an occupied cell. I set each property of the Particle object assigning each particle the same orientation: the identity quaternion for zero rotation. Then I add the particle to the set.

Figure 1 Particle set

Likelihood field

To pre-integrate the likelihood field of the environment I considered the **Beam-based Proximity Model**: it calculates the probabilities based on the distance between the expected measurement (`z_exp`) and the actual measurement obtained from the laser sensor (`z`).

Firstly, for each cell of the 'map', I compute **the distance to the closest obstacle** using the Breadth-First Search Algorithm. This distance can be seen as (`z_exp-z`) if the actual scan beam (`z`) falls into the cell. Having that value it's possible to compute:

$$\boldsymbol{p_{hit}} = 1 * e^{-0.5\left(\frac{(z-z_{exp})}{\sigma_{hit}}\right)^2} \qquad \boldsymbol{p_{rand}} = \frac{1}{z\_max}$$

Where: $\boldsymbol{p_{hit}}$ it's essentially a measure of how well the predicted measurement matches the actual measurement obtained from the sensor, $\boldsymbol{\sigma_{hit}}$ represents the standard deviation of the hit probability distribution, $\boldsymbol{p_{rand}}$ is the (uniform) random measurement, $\boldsymbol{z\_max}$ is the max range for the laser. By combining these values in a weighted sum, we obtain the total weighted probability for each cell that is stored in a `nav_msgs::OccupancyGrid` message for visualization and in *likelihood* variable to keep the un-casted double value.

Motion Model

This task required to update all the particles whenever a new odometry has been received by the callback function. In my first approach I decided to update each particle's pose based on its own previous pose: this means that each particle will move independently based on its history of motion leading to a spread out over time. I then opted to consider the robot's previous odometry: all particles move in unison with the robot's motion, so they are more likely to converge towards the robot's true pose. I implemented the Motion Model only if the robot was moving (thus, if its absolute value of **linear or angular velocity** obtained from the odometry is > 0.001). Then, I retrieve the **Robot's previous odometry** (x, y, θ), the **Robot's new odometry** (x', y', θ') and the **particle's previous odometry** (`x_prev_part, y_prev_part,` θ`_prev_part`). With these quantities I can compute $\boldsymbol{\delta_{rot1}}$, $\boldsymbol{\delta_{trans}}$, $\boldsymbol{\delta_{rot2}}$, they represent the **incremental motion** performed by the robot between consecutive time steps.

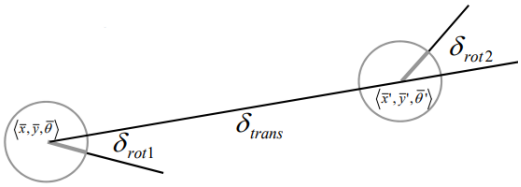

Figure 2 Incremental motion

$$\boldsymbol{\delta_{rot1}} = atan2(y' - y, x' - x) - \theta; \qquad \boldsymbol{\delta_{rot2}} = \theta' - \theta - \delta_{rot1}; \qquad \boldsymbol{\delta_{trans}} = \sqrt{(x' - x)^2 + (y' - y)^2}$$

At this point I compute **the predicted incremental motion with added noise**, incorporating uncertainties in the robot's motion, namely: $\widehat{\boldsymbol{\delta_{rot1}}}, \widehat{\boldsymbol{\delta_{trans}}}, \widehat{\boldsymbol{\delta_{rot2}}}$.

$$\widehat{\boldsymbol{\delta_{rot1}}} = \delta_{rot1} + \varepsilon_{\alpha_1 |\delta_{rot1}| + \alpha_2 |\delta_{trans}|}; \qquad \widehat{\boldsymbol{\delta_{trans}}} = \delta_{trans} + \varepsilon_{\alpha_3 |\delta_{trans}| + \alpha_4 |\delta_{rot1} + \delta_{rot2}|};$$

$$\widehat{\boldsymbol{\delta_{rot2}}} = \delta_{rot2} + \varepsilon_{\alpha_1 |\delta_{rot2}| + \alpha_2 |\delta_{trans}|};$$

Note: that the $\boldsymbol{\varepsilon}$ function introduces some variability sampling from a normal distribution according to the $\boldsymbol{\alpha_1, \alpha_2, \alpha_3, \alpha_4}$ parameters:

- $\boldsymbol{\alpha_1}$: How rotation affects rotation variance. A higher $\alpha_1$value indicates higher rotational motion noise, leading to a wider spread of particles and more uncertainty in the robot's rotation.
- $\boldsymbol{\alpha_2}$: How translation affects rotation variance. Increasing α_2 can result in a broader distribution of particles, since it will lead to more uncertainty in the robot's translation.
- $\boldsymbol{\alpha_3}$: How translation affects translation variance. increasing α_3 widens the distribution of possible positions for the robot after it has moved, reflecting the increased uncertainty in its final location due to translational motion.
- $\boldsymbol{\alpha_4}$: How rotation affects translation variance. Increasing this means that the localization algorithm will consider a wider range of possible positions for the robot after it has rotated, leading to a broader spread of estimated poses.

After computing the predicted motion, I am ready to compute the **update pose** (`x_update, y_update, θ_update`) for each particle according to these formulas:

$$\boldsymbol{x_{upd}} = x_{prev_{part}} + \widehat{\delta_{trans}} * \cos\left(\theta_{prev_{part}} + \widehat{\delta_{rot1}}\right);$$

$$\boldsymbol{y_{upd}} = y_{prev_{part}} + \widehat{\delta_{trans}} * \sin\left(\theta_{prev_{part}} + \widehat{\delta_{rot1}}\right); \qquad \boldsymbol{\theta_{upd}} = \theta_{prev_{part}} + \widehat{\delta_{rot1}} + \widehat{\delta_{rot2}}$$

Finally update each particle according to above's values.

```
particles_[i].updatePose(x_update, y_update, theta_update);
```

Laser Updates

This task's purpose is to define each particle's weight using the laser scan received by the method *updateLaser()*. I use this information to calculate the probability of the **beam proximity model** for each range measurement, that I combine, with the use of the pre-computed likelihood, into a single weight for each particle.

To achieve this goal, firstly, I compute **the pose of the virtual laser** for each particle: since each one represents a pose hypothesis of the robot I retrieve the **particle's pose**; then, I apply the inverse transformation to the laser scan data from the robot's frame to the particle's frame. At this point, using every transformed laser measurement I can compute the probability $\boldsymbol{p_{unexp}, p_{rand}, p_{max}}$:

$$\boldsymbol{p_{unexp}} = \lambda e^{-\lambda z} \quad if\ z < z_{exp}; \qquad \boldsymbol{p_{rand}} = \frac{1}{z_{max}}; \qquad \boldsymbol{p_{max}} = 1 \quad if\ z = z_{max}$$

- $\boldsymbol{p_{unexp}}$**:** This is the probability of receiving a measurement that is not expected.
- $\boldsymbol{p_{max}}$: This is the probability of obtaining a maximum range measurement, which occurs when the laser beam doesn't encounter any obstacles within its maximum range.
- $\boldsymbol{p_{rand}}$: is the same one explained before.

So, for each particle, for each laser measurement I compute these probabilities and combine them into the weighted sum `tot_prob`. **`Likelihood_val`** represents the pre-computed likelihood value of the **hit cell** (the cell that the single simulated laser scan hit), thus **p_hit**.

Note: for the probability I use the *log* function in order to avoid numerical instability; so the probability have to sum each other for each laser measurement (instead of multiplying).

```
tot_prob += log((z_hit_w * likelihood_val) + (z_unexp_w * p_unexp) + (z_rand_w * p_rand) +
                                        (z_max_w * p_max));
```

After the iterations for all the scan measurements for a single particle, I update the weight of the *i-th* particle.

```
particles_[i].updateWeight(tot_prob);
```

I encountered many problems during this task: at the end of the for-loops I printed the particles weights and I noticed that the most important weights are the ones of particles very close to the obstacles, inferring that the robot is more likely to be present on an obstacle than in free space (it should be the opposite). Backtracking the computation, I realized that the problem was the likelihood computation: it associate high values for the cells where obstacles are present and lower values, so lower weights, for free-space cells. To solve this problem, I looked at some references and I compute the likelihood with different methods and using a greater σ to obtain a wider distribution of values.

Particle Weight Normalization

To normalize the particle weights, I Iterate through all the particle weights and compute their **sum** into `sum_weights`. Then, I divide each particle's weight by the sum computed in the previous step.

This implementation ensures that after normalization, the sum of all particle weights will be equal to 1, representing a valid probability distribution.

```
// Calculate the sum of all particle weights
double sum_weights = 0.0;   // Normalization factor (eta)
for(int i = 0; i < particles_.size(); ++i) {
    sum_weights += particles_[i].weight_;
}

// Normalize the weights_ using the sum as the normalization factor
for(int i = 0; i < particles_.size(); ++i) {
    particles_[i].weight_ /= sum_weights;
}
```

Figure 3 - Particle Weight Normalization

Particle Resampling

To implement the *Systematic Resampling* for my particle set according to the pseudocode in Fig. 4 I start to compute the **cumulative sum** of the particle weights **cs** (c_i into the pseudocode*)* to generate the PDF.

I generate a **starting point u** (initialize the threshold u_i) by generating a random number between 0 and $\frac{1}{n}$ where **n** is the number of particles in the set. To **draw the samples,** I iterate *n-random_sample_size* times and use the threshold to determine which particles to select; If `u` exceeds the cumulative weight of a particle, we skip to the next particle. The selected particle is added to the ***S'*** array of resampled particles. During these iterations I compute the **mean pose** of all the particles to spawn a ***random_sample_size*** of random particles set around the mean pose. The size of the latter set is defined by a percentage (I set 10%) of the whole particle set.

1. Algorithm **systematic_resampling**(*S*,*n*):
2. $S' = \emptyset, c_1 = w^1$
3. **for** $i = 2 \dots n$ — *Generate pdf*
4. $c_i = c_{i-1} + w^i$
5. $u_1 \sim U(0, \frac{1}{n}], i = 1$ — *Initialize threshold*
6. **for** $j = 1 \dots n$ — *Draw samples …*
7. **while** ($u_j > c_i$) — *Skip until next threshold reached*
8. $i = i + 1$
9. $S' = S' \cup \{< x^i, \frac{1}{n} >\}$ — *Insert*
10. $u_{j+1} = u_j + \frac{1}{n}$ — *Increment threshold*
11. **return** S'

Figure 4 Systematic Resampling

Note: According to the reference I implemented the Resampling only when the robot is moving[1]. At the beginning, I didn't consider this and thus, the robot was iteratively resampling (even without input commands) changing its particles and resampling its position diverging very fast from the ground truth. With the modification the algorithm improved the performance by far, but in some simulations the resampling doesn't happen properly (only "once in a while").

Calculate Robot Pose

To estimate the robot's pose using the particle set distribution, I can calculate the mean pose of all particles and update the robot's pose estimate with this average pose.

```
//calculate mean of given particles
double x_mean = 0, y_mean = 0, yaw_mean = 0
int n = particles_.size();

// Robot pose from the particles
for (int i = 0; i < n; i++)  {
  Particle p = particles_[i];
  x_mean += p.getX();
  y_mean += p.getY();
  yaw_mean += p.getTheta();
}
```

Figure 5 Robot pose calculation

While the average of all the particles' pose is correct, it doesn't resemble the robot's pose precisely: there are some possible improvements possible for my Particle Filter implementation. One could be to **fine-tune the alphas** that define the variability of translation/rotation in the motion model in order to have the proper variance introduced for this case-study. Also, it's possible to **improve the likelihood** implementation (maybe setting the standard deviation optimally) since the pre-computed values affect a lot the particle weights, and thus, the resampling algorithm. The more accurate particles the algorithm draws from the sample set the more accurate will be the localization.

References

[1] - THRUN, S., BURGARD, W., & FOX, D. (2000). *PROBABILISTIC ROBOTICS*. Lecture. EARLY DRAFT—NOT FOR DISTRIBUTION Sebastian Thrun, Dieter Fox, Wolfram Burgard, 1999-2000

[2] - *Uncertainty & Localization_I* slides.

[3] - *Uncertainty & Localization_II* slides.

[4] – Jeroen D. Hol, Thomas B. Schon, Fredrik Gustafsson. (2006). *ON RESAMPLING ALGORITHMS FOR PARTICLE FILTERS*. Division of Automatic Control Department of Electrical Engineering Linkoping University ¨ SE-581 83, Linkoping, Sweden