COMP7308: Introduction to unmanned systems (2024 Fall)

Assignment 3: FastSLAM Algorithm

:

1 Introduction

In this project, you will start to implement the FastSLAM algorithm. The code for this project includes the following files, available on the Moodle.

Files you will edit and submit:

`/src/fastslam_robot.py`: The location where you will fill in portions during the assignment. When you finish, you should submit **ONLY** this file without modifying its name.

Files you should NOT edit:

`/src/fastslam_robot.yaml`: Information of the simulation environment.

`/hint/`: Some reference results you can utilise to compare with yours.

`/test/student_grader.py`: Project autograder of student version. Pass it can make sure that you get **most of** the grades.

Evaluation

Your code will be autograded for technical correctness. Please **DO NOT** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Academic Dishonesty

We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. Instead, contact the TA if you are having trouble.

Prerequisite

To install the simulation package, please check Assignment 0. You should be familiar with `numpy` and we name it `np` in our file.

2 FastSLAM algorithm for two-wheeled robot (100 points)

0. Simulation of the robot dynamics (0 points)

This time we implement the motion function of the robot for you: `motion_model()`. You can run the autograder: `python /test/student_grader.py -q q0 --graphics` to view the trajectory of the robot. The simulation path is green.

Next we will implement the FastSLAM algorithm. The **key idea** of FastSLAM algorithm is that each particle represents an independent robot, and we use weight to evaluate its importance. Each particle contains the state, its own map, and the weight. We update their states and their maps individually.

Please refer to the slide of the SLAM lecture. Here we provide a detailed version for you.

1: **Algorithm FastSLAM 1.0_known_correspondence(z_t, c_t, u_t, Y_{t-1}):**

2: *for* $k = 1$ *to* M *do* // *loop over all particles*

3: *retrieve* $\left\langle x_{t-1}^{[k]}, \left\langle \mu_{1,t-1}^{[k]}, \Sigma_{1,t-1}^{[k]} \right\rangle, \ldots, \left\langle \mu_{N,t-1}^{[k]}, \Sigma_{N,t-1}^{[k]} \right\rangle \right\rangle$ *from* Y_{t-1}

4: $x_t^{[k]} \sim p(x_t \mid x_{t-1}^{[k]}, u_t)$ // *sample pose*

5: $j = c_t$ // *observed feature*

6: *if feature* j *never seen before*

7: $\mu_{j,t}^{[k]} = h^{-1}(z_t, x_t^{[k]})$ // *initialize mean*

8: $H = h'(x_t^{[k]}, \mu_{j,t}^{[k]})$ // *calculate Jacobian*

9: $\Sigma_{j,t}^{[k]} = H^{-1}\ Q_t\ (H^{-1})^T$ // *initialize covariance*

10: $w^{[k]} = p_0$ // *default importance weight*

11: *else*

12: $\hat{z} = h(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ // *measurement prediction*

13: $H = h'(x_t^{[k]}, \mu_{j,t-1}^{[k]})$ // *calculate Jacobian*

14: $Q = H\ \Sigma_{j,t-1}^{[k]}\ H^T + Q_t$ // *measurement covariance*

15: $K = \Sigma_{j,t-1}^{[k]}\ H^T\ Q^{-1}$ // *calculate Kalman gain*

16: $\mu_{j,t}^{[k]} = \mu_{j,t-1}^{[k]} + K(z_t - \hat{z})$ // *update mean*

17: $\Sigma_{j,t}^{[k]} = (I - K\ H)\Sigma_{j,t-1}^{[k]}$ // *update covariance*

18: $w^{[k]} = |2\pi Q|^{-\frac{1}{2}}\ \exp\left\{-\frac{1}{2}(z_t - \hat{z}_n)^T\ Q^{-1}\ (z_t - \hat{z}_n)\right\}$ // *importance factor*

19: *endif*

20: *for all other features* $j' = j$ *do* // *unobserved features*

21: $\mu_{j',t}^{[k]} = \mu_{j',t-1}^{[k]}$ // *leave unchanged*

22: $\Sigma_{j',t}^{[k]} = \Sigma_{j',t-1}^{[k]}$

23: *endfor*

24: *endfor*

25: $Y_t = \emptyset$ // *initialize new particle set*

26: *do* M *times* // *resample* M *particles*

27: *draw random* k *with probability* $\propto w^{[k]}$ // *resample*

28: *add* $\left\langle x_t^{[k]}, \left\langle \mu_{1,t}^{[k]}, \Sigma_{1,t}^{[k]} \right\rangle, \ldots, \left\langle \mu_N^{[k]}, \Sigma_N^{[k]} \right\rangle \right\rangle$ *to* Y_t

29: *endfor*

30: *return* Y_t

Figure 1: On page 450, Probabilistic Robotics by Thrun, Burgard, and Fox

1. FastSLAM: sampling from motion model (5 points)

For particle k with state $\boldsymbol{x}_{t-1}^{[k]} = (x_{t-1}^{[k]}, y_{t-1}^{[k]}, \theta_{t-1}^{[k]})^{\mathbf{T}}$, we should generate sample $\boldsymbol{x}_t^{[k]}$: $\boldsymbol{x}_t^{[k]} \sim p(\boldsymbol{x}_t | \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t)$. To achieve this goal, you can utilise the motion model and then add a noise, i.e., $\boldsymbol{x}_t^{[k]} = f(\boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t) + \boldsymbol{\epsilon}$. Directly utilise the motion model `motion_model()` and parameters we provide.

Implement this part within the function `fastslam_particles_motion_predict()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q1`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q1 --graphics`, and compare it with the result `./hint/q1.gif` and `./hint/q1.png`.

Because there is no correction step, you will see that the distribution of particles is growing, which means the uncertainty keeps growing. But the average will cancel most of the influence of the noise, so you will see the predicted trajectory is like trajectory without noise. The predicted path is yellow.

2. FastSLAM: landmarks and weights update (60 points)

On page 71, for k-th particle, we utilise its state $\mathbf{x}_t^{[k]} = (x_t^{[k]}, y_t^{[k]}, \theta_t^{[k]})^{\mathbf{T}}$ to update its map, i.e., landmarks' positions, and its own weight based on range and bearing measurements $\mathbf{z}$. Here the measurement model is

$$\hat{\boldsymbol{z}}_j^{[k]} = h(\boldsymbol{\mu}_{j,t-1}^{[k]}, \mathbf{x}_t^{[k]}) = \begin{pmatrix} \sqrt{(\mu_{x,j,t-1}^{[k]} - x_t^{[k]})^2 + (\mu_{y,j,t-1}^{[k]} - y_t^{[k]})^2} \\ \arctan(\frac{\mu_{y,j,t-1}^{[k]} - y_t^{[k]}}{\mu_{x,j,t-1}^{[k]} - x_t^{[k]}}) - \theta_t^{[k]} \end{pmatrix},$$

where $\boldsymbol{\mu}_{j,t-1}^{[k]} = (\bar{\mu}_{x,j,t-1}^{[k]}, \bar{\mu}_{y,j,t-1}^{[k]})$ is the estimated position of the landmark j. Nevertheless, when you have not seen landmark j before t and do not have the estimated position information, given your position $\mathbf{x}_t^{[k]}$ and the measurements data $\boldsymbol{z}_j^{[k]}$, you can directly initialise the mean $\boldsymbol{\mu}_{j,t}^{[k]}$ and covariance $\Sigma_{j,t}^{[k]} = H^{-1} Q_t (H^{-1})^{\mathbf{T}}$.

First, utilise the measurement model in the inverse way for special initialisation as the mean at time t: $\boldsymbol{\mu}_{j,t}^{[k]} = h^{-1}(\boldsymbol{z}_j^{[k]}, \mathbf{x}_t^{[k]})$, which is mentioned in the lecture of the EKF SLAM. How the sensor works in 2D space is shown in Figure 2. Then use the mean to derive the form of the Jacobean $H = h'(\boldsymbol{\mu}_{j,t}^{[k]}, \mathbf{x}_t^{[k]}) = \frac{\partial h(\mathbf{m}, \mathbf{x})}{\partial \mathbf{m}}\Big|_{(\boldsymbol{\mu}_{j,t}^{[k]}, \mathbf{x}_t^{[k]})}$, where $\mathbf{m}$ means the position of the landmark.

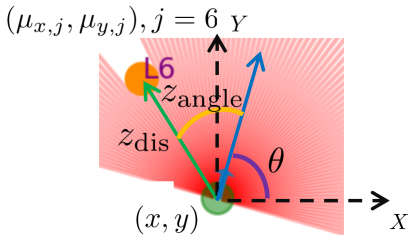

Figure 2: An example showing how sensor get measurements.

Finish two cases for each measurement: the landmark is newly observed, or the landmark has been observed before.

Implement this part within the function `fastslam_particles_measurement_update()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q2`. You can

also visualise your trajectory result by running `python /test/student_grader.py -q q2 --graphics`, and compare it with the result `./hint/q2.gif` and `./hint/q2.png`. To specify simulation iteration steps for fast debugging, you can run like: `python test/student_grader.py --graphics -q q2 --iter-step 10`.

Here we can update the map, and our predictions are plotted using blue crosses $\times$. With correction step, the predicted position is closer to the ground truth. However, we only change the weight of each particles, and these particles still separate like before. Without the resampling, we cannot get rid of those 'bad' particles.

Note:

1. WrapToPi(angle) and `arctan2(y, x)` are still necessary.
2. You can refer to EKF localisation algorithm to see how to calculate the Jacobian H.
3. You can read the code of `class particle` at line 10-28 to get a better understanding of the data structure for each particle.
4. Be aware of the shape of each variable, either given by us or defined yourself. For example, as to `numpy` array `state`, you can print it by `state.shape` during debugging.

3. FastSLAM: resampling (35 points)

Though resampling makes sure the we can get rid of 'bad' particles, resampling too many times can kill some 'good' particles and reduce the particles diversity, which is so-called "particle depletion". Therefore, we will reduce the resampling frequency, and we consider resampling when this condition is met:

$$n_{\text{eff}} = \frac{1}{\sum_k (w_t^{[k]})^2} < \frac{M}{D},$$

where M is the number of particles and D is a divisor ($D = 1.5$ in our question).

A sampling algorithm is provided for you shown in Figure 3. Here $\mathcal{X}$ means the sample set, $\mathcal{W}$ means the corresponding weight set, M^{-1} is the sampling step between two samples, r represents the randomly-chosen start position, and c accumulate the weight.

Here is a figure showing how this sampling algorithm works.

Implement this part within the function `fastslam_particles_resample()`.

To test your implementation, run the autograder: `python /test/student_grader.py -q q3`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q3 --graphics`, and compare it with the result `./hint/q3.gif` and `./hint/q3.png`. We will not recommend you to use `--iter-step` here because you may miss resampling.

You will see that resampling indeed improve the performance of our algorithm.

Note:

1. In our code $\mathcal{X}$ and $\mathcal{W}$ are defined together as a particle set.
2. In our code the id of sample i is counted from 0.
3. You can use `copy.deepcopy(old_particle)` to generate a new particle.
4. When add new particles to the new set $\bar{\mathcal{X}}$, you should set the weight of this particle to the default weight, i.e., $1/M$, so that we can make sure that every new particle is the same important as each other.

1: **Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):**
2: $\bar{\mathcal{X}}_t = \emptyset$
3: $r = \text{rand}(0; M^{-1})$
4: $c = w_t^{[1]}$
5: $i = 1$
6: *for* $m = 1$ *to* M *do*
7: $U = r + (m-1) \cdot M^{-1}$
8: *while* $U > c$
9: $i = i + 1$
10: $c = c + w_t^{[i]}$
11: *endwhile*
12: *add* $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$
13: *endfor*
14: *return* $\bar{\mathcal{X}}_t$

Figure 3: On page 110, Probabilistic Robotics by Thrun, Burgard, and Fox

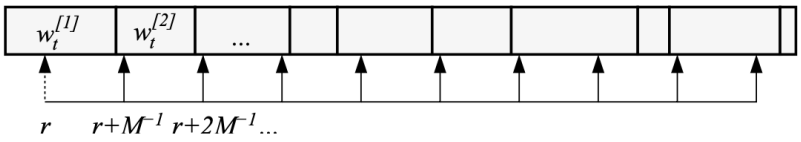

Figure 4.7 Principle of the low variance resampling procedure. We choose a random number r and then select those particles that correspond to $u = r + (m-1) \cdot M^{-1}$ where $m = 1, \ldots, M$.

Figure 4: At page 111, Probabilistic Robotics by Thrun, Burgard, and Fox