COMP7308: Introduction to unmanned systems (2024 Fall)

Assignment 2: EKF Localisation Algorithm

:

1 Introduction

In this project, you will start to implement EKF localisation algorithm of a robot. The code for this project includes the following files, available on the Moodle.

Files you will edit and submit:

`/src/ekf_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/ekf_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 EKF localisation algorithm for two-wheeled robot (100 points)

One way to model the motion of a two-wheeled robot is:

$$s_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = g(s_{t-1}, u_t) = \begin{pmatrix} x_{t-1} + u_{t,1}\cos(\theta_{t-1})\Delta t \\ y_{t-1} + u_{t,1}\sin(\theta_{t-1})\Delta t \\ \theta_{t-1} + u_{t,2}\Delta t \end{pmatrix}$$

1. Simulation of the robot dynamics (10 points)

Before implementing the EKF localisation algorithm, first we need to accomplish the simulation system, i.e., the simulation of robot's dynamics. Here, the simulation will consider noise when robot moves. Therefore, the motion function will be $s_t = g(s_{t-1}, u_t) + \epsilon$, where $\epsilon = \mathrm{N}(0, \hat{R})$ is a Gaussian noise for the state. Implement it within the function `dynamics()`.

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`. To specify simulation iteration steps for fast debugging, you can run like: `python test/student_grader.py --graphics -q q1 --iter-step 20`. The simulation path is green.

Note:

1. Please check the comment within each function for variables' details and the requirement.
2. Be aware of the shape of each variable, either given by us or defined yourself. For example, as to `numpy` array `vel`, you can print it by `vel.shape` during debugging.

Next we will implement the EKF localisation algorithm, please refer to attached slides segments of lecture 4 for more details.

2. EKF algorithm: prediction step (30 points)
For the noise in the motion model, we assume

$$R_t = \begin{pmatrix} 0.02 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.005 \end{pmatrix},$$

which is already defined for your self-testing.

Implement it within the function `ekf_predict()`.

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 20`. Because there is no correction step, you will see that the uncertainty region keeps growing, which means the eigenvalues of the covariance matrix is increasing. The robot is predicted moving without noise.

3. EKF algorithm: correction step with range measurements (30 points)

First, for correction step, at each step we consider using **only** range measurements

$$\hat{z}_{t,k} = h(\bar{s}_t, l_k) = \left(\sqrt{(\bar{\mu}_{x,t} - l_{x,k})^2 + (\bar{\mu}_{y,t} - l_{y,k})^2} \right),$$

where $(l_{x,k}, l_{y,k})$ is the known position of the landmark k. Correspondingly, the form of H will also change. You should update these measurements **individually** as described in the last two pages of the attached slides segments. For the noise in the measurement model we assume

$$Q_t = \left(0.2\right),$$

which is already defined for your self-testing.

Implement it within the function `ekf_correct_no_bearing()`.

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`. To specify simulation iteration steps for fast debugging, you can run like: `python test/student_grader.py --graphics -q q3 --iter-step 20`. With correction step, the uncertainty region is restricted, and the predicted position is closer to the ground truth.

4. EKF algorithm: correction step with range and bearing measurements (30 points)

Here, for correction step, at each step we consider using range and bearing measurements

$$\hat{z}_{t,k} = \begin{pmatrix} \sqrt{(\bar{\mu}_{x,t} - l_{x,k})^2 + (\bar{\mu}_{y,t} - l_{y,k})^2} \\ \arctan(\frac{l_{y,k}-\bar{\mu}_{y,t}}{l_{x,k}-\bar{\mu}_{x,t}}) - \bar{\mu}_{\theta,t} \end{pmatrix},$$

where $(l_{x,k}, l_{y,k})$ is the known position of the landmark k. You should update these measurements **individually** as described in the last two pages of the attached slides segments. For the noise in the measurement model we assume

$$Q_t = \begin{pmatrix} 0.2 & 0 \\ 0 & 0.2 \end{pmatrix},$$

which is already defined for your self-testing.

Implement it within the function `ekf_correct_with_bearing()`.

Note: The detective range of the sensor in our project is $[-\pi, \pi]$, so the measurement value is within this range. You should wrap the estimated angle value in $\hat{z}$ to $[-\pi, \pi]$ and you can use function `WrapToPi(angle)` which we have implemented for you. Please use `numpy.arctan2(y, x)` or `math.arctan2(y, x)` to calculate the angle $\arctan(\frac{l_{y,k}-\bar{\mu}_{y,t}}{l_{x,k}-\bar{\mu}_{x,t}})$ rather than `numpy.arctan(y/x)`, because the later will return the angle within $[-\pi/2, \pi/2]$.

To test your implementation, run the autograder: `python /test/student_grader.py -q q4`. You can also visualise your trajectory result by running `python /test/student_grader.py -q q4 --graphics`, and compare it with the result `./hint/q4.gif` and `./hint/q4.png`. To specify simulation iteration steps for fast debugging, you can run like: `python test/student_grader.py --graphics -q q4 --iter-step 20`. With correction step, the uncertainty region is restricted, and the predicted position is closer to the ground truth.