CSE 571 - Robotics
Homework 2 - EKF and RRT

Due Tuesday May 14th @ 11:59pm

This homework consists of two parts. In the first part, you will derive and implement Extended Kalman Filter (EKF) to perform landmark-based localization and mapping. In the second part, you will implement Rapidly-exploring Random Tree (RRT), a sampling-based motion planning method, to generate trajectories for a robot arm. The code for this homework can be found at `https://github.com/liyi14/cse571_24sp_hw2`.

Collaboration: Students can discuss questions, but each student MUST write up their own solution, and code their own solution. We will be checking code/PDFs for plagiarism.

Late Policy: This assignment may be handed in up to 5 days late (Tuesday May 19th @ 11:59pm), at a penalty of 10% of the maximum grade per day.

1 Extended Kalman Filter

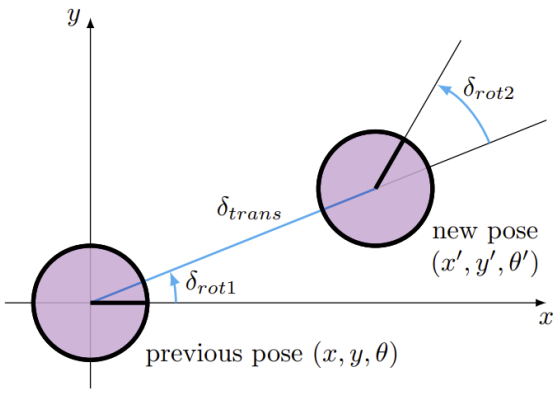

Figure 1: Odometry-based motion model

1.1 Jacobian Derivation

Hint

1. Take a look at the relevant sections (7.4, 10.2) in the book [1].
2. Make sure you get the derivations correct before proceeding to implementation.

1.1.1 Motion Model Jacobian [5 points]

We will reuse the odometry motion model from HW1 (see Figure 1 and Sec. 5.4 in the book [1]). The state of the robot is its 2D position and orientation: $s_t = [x_t, y_t, \theta_t]$. The control to the robot is $u_t = [\delta_{rot1}, \delta_{trans}, \delta_{rot2}]$, i.e. the robot rotates by δ_{rot1}, drives straight forward δ_{trans}, then rotates again by δ_{rot2}.

The equations for the motion model $s_t = g(u_t, s_{t-1})$ are as follows:

$$\begin{aligned} x_t &= x_{t-1} + \delta_{trans} * cos(\theta_{t-1} + \delta_{rot1}) \\ y_t &= y_{t-1} + \delta_{trans} * sin(\theta_{t-1} + \delta_{rot1}) \\ \theta_t &= \theta_{t-1} + \delta_{rot1} + \delta_{rot2} \end{aligned}$$

Your task is to derive the Jacobian matrix G, a matrix consisting of the first-order derivative of s_t with respect to the previous state s_{t-1}, as well as V, a matrix consisting of the first-order derivative of s_t with respect to the control input u_t. Since s_t, s_{t-1} and u_t are all 3-dimensional. G and V are both 3×3 matrices.

In short, $s_{t+1} = [x_{t+1}, y_{t+1}, \theta_{t+1}]$ is the prediction of the motion model. Derive the Jacobians of g with respect to the state $G = \dfrac{\partial g}{\partial s}$ and control $V = \dfrac{\partial g}{\partial u}$:

$$G = \begin{pmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \theta} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \theta} \\ \frac{\partial \theta'}{\partial x} & \frac{\partial \theta'}{\partial y} & \frac{\partial \theta'}{\partial \theta} \end{pmatrix} \qquad V = \begin{pmatrix} \frac{\partial x'}{\partial \delta_{rot1}} & \frac{\partial x'}{\partial \delta_{trans}} & \frac{\partial x'}{\partial \delta_{rot2}} \\ \frac{\partial y'}{\partial \delta_{rot1}} & \frac{\partial y'}{\partial \delta_{trans}} & \frac{\partial y'}{\partial \delta_{rot2}} \\ \frac{\partial \theta'}{\partial \delta_{rot1}} & \frac{\partial \theta'}{\partial \delta_{trans}} & \frac{\partial \theta'}{\partial \delta_{rot2}} \end{pmatrix}$$

1.1.2 Observation Model Jacobian [5 points]

Assume there is a landmark m at location (x_m, y_m). The robot receives two measurements of the bearing angle ϕ and the landmark, the range r , where

$$\begin{aligned} \phi &= \text{atan2}(y_m - y_t, x_m - x_t) - \theta_t \\ r &= \sqrt{(x_m - x_t)^2 + (y_m - y_t)^2} \end{aligned}$$

Derive Jacobian matrix H_s, derivative of the measurements with respect to the robot state $s_t = [x_t, y_t, \theta_t]$ and the Jacobian matrix H_m, derivative of the measurements with respect to the landmark position (x_m, y_m).

1.2 EKF Localization and Mapping [40 points]

In the programming component of this assignment, you will first implement an Extended Kalman Filter (EKF) (Fig. 2) to localize a robot based on landmarks. Then, you will add landmarks to the state and build a map while localizing the robot, thus performing simultaneous localization and mapping (SLAM).

We will use the odometry-based motion model you derived in 1.1. We assume that there are a set of landmarks present in the robot's environment. At each time step, the robot receives the range r, the bearing angle ϕ and the ID of a particular landmark as observations.

We assume a noise model for the robot's motion with parameters α (Sec 5.4 [1]) and a separate noise model for the observations with parameter β (Section 6.6 [1]). See the provided code for implementation details.

At each timestep, the robot starts from the current state and moves according to the control input. The robot then receives a landmark observation from the world. You will use EKF to combine information from the

1: **Algorithm Extended_Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)**:**
2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
3: $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$
4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$
5: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$
7: *return* μ_t, Σ_t

Figure 2: Extended Kalman Filter

noise model, the control inputs and the landmark observations to estimate the robot's trajectory as well as the positions of the landmarks.

Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib. The conda environment you installed for HW1 should suffice for this homework. This section gives a brief overview of each file.

- `localization.py` – This is your main entry point for running experiments.
- `soccer_field.py` – This implements the dynamics and observation functions, as well as the noise models for both.
- `utils.py` – This contains assorted plotting functions, as well as a useful function for normalizing an angle between $[-\pi, \pi]$.
- `policies.py` – This contains a simple policy to control the robot, which you can safely ignore.
- `ekf_slam.py` – **Add your Extended Kalman Filter implementation here!**

Command-Line Interface

To visualize the robot in the soccer field environment, run

```
$ python localization.py --plot none
```

The blue line traces out the robot's position, which is a result of noisy actions. The green line traces the robot's position assuming that actions weren't noisy. After you implement a filter, the filter's estimate of the robot's position will be drawn in red.

```
$ python localization.py --plot ekf_slam
```

Typical commands would be like:

```
# problem a
$ python localization.py ekf_slam --plot
# problem b
$ python localization.py ekf_slam --multi_run 10 --data-factor 0.1 --filter-factor 0.1
# problem c
$ python localization.py ekf_slam --multi_run 10 --motion-factor 0.1 --observation-factor 1.
```

To see other command-line flags available to you, run

```
$ python localization.py -h
```

Hints

- Make sure to call `utils.minimized_angle` any time an angle or angle difference could exceed $[-\pi, \pi]$.
- Turn off plotting for a significant speedup.
- Check this tutorial for more hints

Task

1. Fill in the Jacobian matrices `G`, `V`, `R` and `H` in `ekf_slam.py` using your derivation in 1.1.
2. Implement the EKF update (Fig. 2) in `ExtendedKalmanFilterSLAM.update` in `ekf_slam.py`. Make sure to keep track of the set of landmark IDs that the robot has observed.

Answer the following questions in your writeup.

(a) Under the default noise parameters, $\alpha = \beta = 1$, plot the robot's path and the landmark locations estimated by EKF and compare it with the ground truth path and landmarks.

(b) Plot the position error of the robot state and the landmark positions as the data factor (`--data-factor`) and the filter noise factor (`--filter-factor`) **both** changed over $[0.01, 0.03, 0.1, 0.3, 1.0]$. You should run 10 trials per value with different random seeds (`--multi_run 10`). Does EKF estimates the map accurately? How does the localization accuracy compare with the EKF that does not estimate landmark positions? Discuss anything interesting you observe.

(c) Plot the position error of robot state and the landmark positions as the motion factor (`--motion-factor`) and observation factor (`--observation-factor`) **varies** over $[0.1, 0.3, 1.0]$. You should run 10 trials per value with different random seeds (`--multi_run 10`). Discuss anything interesting you observe. A typicall command would be like

2 Rapidly-exploring Random Tree

In this part, you are provided with a robot arm that has 2 links is able to move in a 2D plane only. Your task is to implement Rapidly-exploring Random Tree (RRT), a classical sampling-based motion planning algorithms named after its data structure, and study the parameters that govern its behaviors.

Code Overview

The starter code is under the `rrt` folder. Here is a brief overview.

- `plan.py` - Contains the main function. Run `python plan.py -h` to see the command-line arguments that you can provide.
- `ArmEnvironment.py` - Environment and utility functions for the 2D robot arm.

- `RRTPlanner.py` - RRT planner. Algorithm to be filled in by you.
- `RRTTree.py` - Contains the tree data structure that is essential for your implementation of RRT.
- `2dof_planar_robot.urdf` - URDF (`http://wiki.ros.org/urdf`) file for the 2D robot arm.

2.1 RRT Implementation [50 points]

Implement a RRT planner for the 2D robot in `RRTPlanner.py` by filling in `Plan` and `extend` functions. Your results from a successful RRT implementation should be comparable to the following results.

```
$ python plan.py -o 0 --seed 0
...
cost: 198.84383834015168

$ python plan.py -o 2 --seed 0
...
cost: 170.48992200228264
```

If you turn on the `-v` flag, i.e. `python plan.py --seed 0 -v`. You should see a plot similar to Fig. 3.

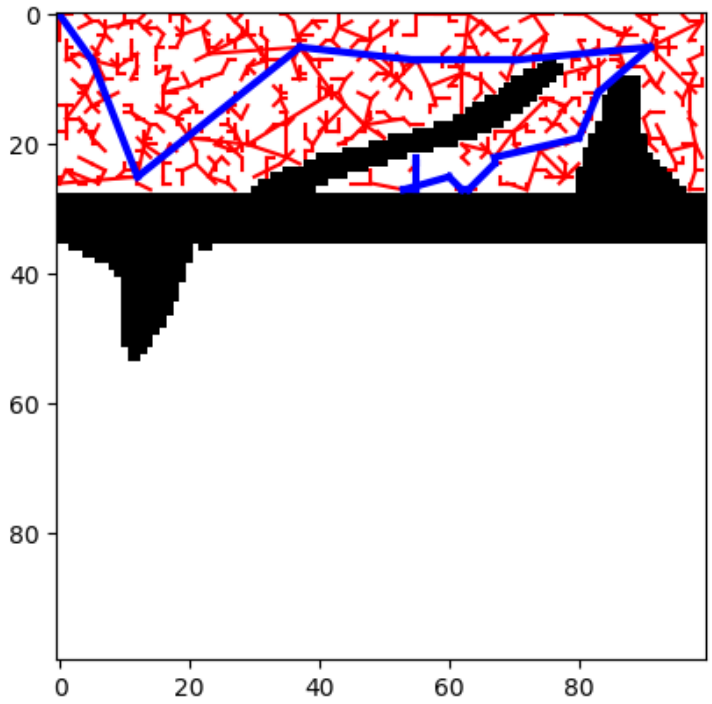

Figure 3: Example RRT path in configuration space.

Answer the following questions.

Note that since RRT is non-deterministic, you will need to provide statistical results, i.e. mean and standard deviation over **at least 5 runs** with different random seeds specified by `--seed`.

The `-o` flag specifies the number of obstacles in the environment. Please report your results with `-o 2`. You can use other values for debugging.

1. Bias the sampling to pick the goal with 5%, 20% probability. Report the performance (cost and time). For each setting, include at least one figure like Fig. 3 showing the RRT tree in configuration space.

2. Implement two versions of the `extend()` function:

   - the nearest neighbor tries to extend to the sampled point only by a step-size η. Set $\eta = 0.5$ and report results in your write-up.
   - the nearest neighbor tries to extend all the way till the sampled point (i.e. $\eta = 1$).

   You can assume the point robot to be able to move in arbitrarily any direction in configuration space, i.e. states can be interpolated via a straight line (see Fig. 4 for an illustration).

   As before, report the performance (cost, time) and include at least one figure showing the final state of the tree for each `2dof_robot_arm` setting. Which strategy would you employ in practice?

3. Discuss any challenges you faced and describe your debugging process.

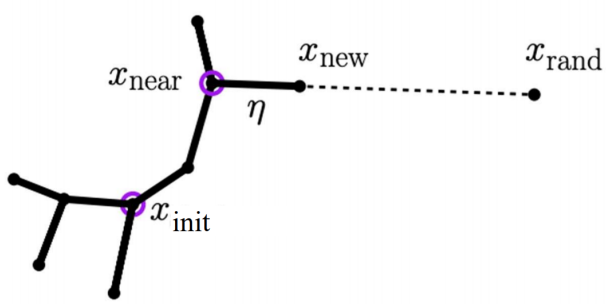

Figure 4: Visualization of `extend()` with step size η, which controls the ratio of the distance from x_{new} (the state to be extended) to x_{near} and the distance from x_{rand} (a sampled state) to x_{near}. If η is set to 1, $x_{\text{new}} = x_{\text{rand}}$.

Hint

Check out these useful functions that you should use to simplify your implementation.

- In `RRTTree.py`
  - `AddVertex`
  - `AddEdge`
  - `GetNearestVertex`
- In `ArmEnvironment.py`
  - `compute_distance`
  - `goal_criterion`
  - `edge_validity_checker`

3 Submission

We will be using the Canvas for submission of the assignments. Please submit the written assignment answers as a PDF. For the code, submit a zip file of the entire working directory.

References

[1] Thrun, Burgard and Fox (2005), Probabilistic Robotics.