CSE 571 - Robotics
Homework 1 - Particle Filters for Localization

Due Tuesday April 16th @ 11:59pm

The key goal of this homework is to get an understanding of the properties of Particle filters for state estimation. In the programming assignment, you will implement a Particle Filter (PF) for landmark based localization, and a learned observation model from camera images. You will also analyze their performance under various conditions. The zip file containing the starter code for this homework can be found at `https://github.com/liyi14/cse571_24sp_hw1`.

Useful reading material: Lecture notes, Chapters 3,4,5,7 & 8 of Probabilistic Robotics, Thrun, Burgard and Fox (pdf shared with class).

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. If you have used up your 6 late days this quarter, there will be a penalty of 20% of the maximum grade per day.

1 Environment Setup

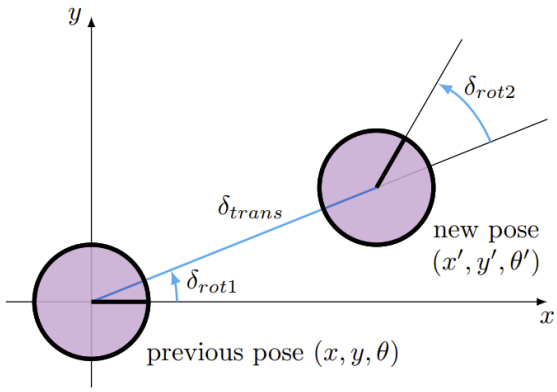

Figure 1: Odometry-based motion model

1.1 Motion Model

Figure 1 describes a simple motion model. The state of the robot is its 2D position and orientation: $s = [x, y, \theta]$, where x and y are it's location on the 2D ground plane, and θ is the angular direction the robot is facing. The control to the robot is $u = [\delta_{rot1}, \delta_{trans}, \delta_{rot2}]$. This means that the robot controls itself at each time step by first rotating by δ_{rot1}, then driving forward some distance δ_{trans}, then rotating again by δ_{rot2}.

Given the current state $s_t = [x_t, y_t, \theta_t]$ and controls $u = [\delta_{rot1}, \delta_{trans}, \delta_{rot2}]$, we can use this motion model to describe an estimate for the next state $s_{t+1} = [x_{t+1}, y_{t+1}, \theta_{t+1}]$ using the following equations:

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

We refer to these functions as the motion model g such that $g(u, s_t) = s_{t+1}$. For more details about odometry motion model, please refer to Section 5.4 of Probabilistic Robotics [1].

1.2 Observation Model

Assume there is a landmark l at position (l_x, l_y). Suppose the robot receives measurement of the landmark in terms of the bearing angle ϕ relative to the direction it is currently facing. The estimation for ϕ in terms of the landmark's position (l_x, l_y) and the robot's state (x_t, y_t, θ_t) follows

$$\phi = \tan^{-1} \frac{l_y - y_t}{l_x - x_t} - \theta_t$$

For more information about landmark-based observation model, see Sec. 6.6.2 of Probabilistic Robotics [1].

2 Programming assignments

Environment Requirement

We strongly encourage you to use a Linux machine for coding assignments. For training Neural Networks, we have provided colab notebook. Please see detailed instructions below.

(a) Linux / Mac

Virtual environment is strongly recommended! To set up a virtual environment, first install conda from `https://www.anaconda.com/`, then run:

```
conda create -n cse571 python=3.9
conda activate cse571
pip install numpy
pip install pybullet
pip install matplotlib
pip install torch # optional
```

(b) Windows

If you have a windows machine and would like to use Ubuntu, please go to: `https://www.vmware.com/products/workstation-player.html` and download VMware Workstation Player for Windows. You will also need to download both the .vmdk and the .ovf files. However, VM is not mandatory.

To set up the VM on your Windows machine, click the .exe file and follow installation steps. Once the VMware player is installed, open VMWare Workstation 17 Player, and choose "Open a Virtual Machine" and select "ros-noetic-22wi.ovf" and import. You should be good to go! Once you set up your VM, log in "robotics" with the password: "robotics".

Inside your VM machine, open a terminal, and please follow (a) Linux/Mac to set up the virtual environment on your VM machine.

(c) CoLab

CoLab is a free Jupyter notebook environment that runs entirely in the cloud hosted by Google. Please go to `https://colab.research.google.com` and click "Upload". Select your ".ipynb" jupyter notebook and you are all set!

Problem Overview

In the programming component of this assignment, you will implement an Particle Filter (PF) for localizing a robot based on landmarks. Then you will also implement a learned observation model which estimates landmark bearing angles from images. In our environment, the robot is driving in a path that would cause it to follow a rectangle. However, because it's motion is noisy, it's true path will deviate substantially from the intended path. The goal is to use the PF to find out where the robot actually is, given information about how the robot is moving and the landmark observations.

We will use the odometry-based motion model in question 1.2. We assume that there are landmarks present in the robot's environment. The robot receives the bearings (angles) to the landmarks and the ID of the landmarks as observations: $(\text{bearing}, \text{landmark ID})$.

We assume a noise model for the odometry motion model with parameters α as shown below([1] Table 5.6), and a separate noise model for the bearing observations with parameter β (see [1], Section 6.6). Note that for the current problem set, data association has already been assumed to be correct. Therefore, there is no need to worry about the data association problem while attempting the questions. However, it is important to keep in mind that in real-world scenarios, data association can be a significant challenge in multi-object tracking and should be carefully considered in developing any tracking algorithm.

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

2: $\quad \delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3: $\quad \delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$
4: $\quad \delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5: $\quad \hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$
6: $\quad \hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot1}} + \delta_{\text{rot2}}))$
7: $\quad \hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

8: $\quad x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$
9: $\quad y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$
10: $\quad \theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

11: $\quad$ *return* $x_t = (x', y', \theta')^T$

Table 5.6 Algorithm for sampling from $p(x_t \mid u_t, x_{t-1})$ based on odometry information. Here the pose at time t is represented by $x_{t-1} = (x\ \ y\ \ \theta)^T$. The control is a differentiable set of two pose estimates obtained by the robot's odometer, $u_t = (\bar{x}_{t-1}\ \ \bar{x}_t)^T$, with $\bar{x}_{t-1} = (\bar{x}\ \ \bar{y}\ \ \bar{\theta})$ and $\bar{x}_t = (\bar{x}'\ \ \bar{y}'\ \ \bar{\theta}')$.

Figure 2: Table 5.6 from [1]. The **sample** function samples a single random value with mean 0 and variance equal to the function argument.

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

the robot over the whole time sequence with a PF.

The result will be evaluated on the metric of mean position error, mean mahalanobis error and ANEES(Average Normalized Estimation Error Squared). In short words, the lower, the better.

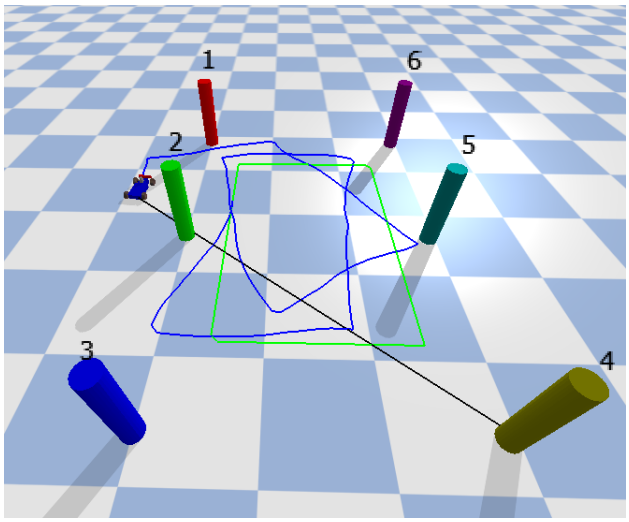

Figure 3: The test environment in pybullet. The blue line traces out the robot's true position, which is a result of noisy actions. The green line traces the robot's position assuming that actions weren't noisy. The black line indicates the bearing angle to the current landmark.

Code Overview

The starter code is written in Python and depends on matplotlib, numpy, pybullet and torch. This section gives a brief overview of each file.

- `localization.py` – This is your main entry point for running experiments.
- `train_localization_model.ipynb` – A jupyter notebook that will train your observation model once you have implemented it.
- `utils.py` – This contains a useful function for normalizing an angle between $[-\pi, \pi]$.
- `policies.py` – This contains a simple policy, which you can safely ignore.
- `racecar.urdf` – The robot description of the racecar.
- `meshes/` – Meshes used by the racecar.
- `make_data.py` – A script which generates a dataset to use for training your learned observation model.
- `requirements.txt` – A list of pip packages necessary to run the assignment code.
- `soccer_field.py` – This implements the dynamics and observation functions, as well as the noise models for both. It also contains the interface to display the robot in pybullet.
- `pf.py` – **Add your particle filter implementation here!**
- `observation_model.py` – **Add your observation model 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. The black line indicates the bearing angle to the current landmark. After you implement a filter, the filter's estimate of the robot's position will be drawn in red.

To check the implementation of particle filter, run

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

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

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

Data Format

- state: $[x, y, \theta]$
- control: $[\delta_{rot1}, \delta_{trans}, \delta_{rot2}]$
- observation: $[\theta_{bearing}]$

Hints

- Make sure to call `utils.minimized_angle` any time an angle or angle difference could exceed $[-\pi, \pi]$.
- Make sure to use the low-variance systematic sampler from lecture. It gives you a smoother particle distribution and also requires only a single random number per resampling step.
- Switch pybullet from GUI to DIRECT (headless) for a significant speedup. This can be done by omitting the `--plot` flag when running `localization.py`

2.1 PF Implementation [60 points]

Implement the particle filter algorithm in `pf.py`. You will need to fill in `ParticleFilter.update` and `ParticleFilter.resample`.

```
python localization.py pf --seed 1
...
Mean position error: 7.4464918380817995
Mean Mahalanobis error: 7.018081273538514
ANEES: 2.339360424512838
```

```
python localization.py pf --seed 1 --data-factor 16 --filter-factor 16
...
Mean position error: 39.67240074809783
Mean Mahalanobis error: 6.610035061476349
ANEES: 2.2033450204921166
```

```
python localization.py pf --seed 1 --data-factor 0.0625 --filter-factor 16
...
Mean position error: 9.72115743733739
Mean Mahalanobis error: 0.3513419286184077
ANEES: 0.1171139762061359
```

```
python localization.py pf --seed 1 --data-factor 16 --filter-factor 16 --num-particles 200
...
Mean position error: 49.14857442421298
Mean Mahalanobis error: 2.4300966694407857
ANEES: 0.8100322231469286
```

```
python localization.py pf --multi_run 10 --seed 0

Data factor: 1
Filter factor: 1
Mean MPE: 8.275366675257736
Standard deviation of MPE: 1.6082609959005343
Mean ANEES: 2.7597135229244274
Standard deviation of ANEES: 1.3569377486950798
```

Note that these numbers are included for reference and you are not expected to exactly replicate these numbers. Depending on your implementation, you may get different results than this. When reporting numbers of position error and ANEES, make sure to include param `--multi_run 10 --seed 0` like

```
python localization.py pf --multi_run 10 --seed 0 --data-factor 16 --filter-factor 16
```

to run 10 trails with different seeds and report the mean and standard deviation of position error and ANEES.

(a) Plot the real robot path and the filter path under the default parameters.

(b) Plot the mean and standard deviation of the position error and ANEES as both the data and the filter noise factors range over $[1/16, 1/4, 1, 4, 16]$. Discuss anything interesting you observe.

(c) Plot the mean and standard deviation of the position error and ANEES as the filter noise factors vary over $[1/16, 1/4, 1, 4, 16]$ while the data noise factor is kept as default. How does the results compare with what you get using the default parameters? Discuss anything interesting you observe.

(d) Plot the mean and standard deviation of the position error and ANEES as both the data and the filter noise factors range over $= [1/16, 1/4, 1, 4, 16]$ (as 2.1.b) and the number of particles varies over $n = [20, 100, 500]$. Discuss anything interesting you observe.

(e) Discuss any challenges you faced and describe your debugging process.

2.2 Observation Model Learning [40 points]

In this section you will learn a model which takes panoramic images of the scene around the racecar, and predicts the bearing angles to each landmark. The racecar has four evenly spaced cameras that have been stitched together into a single panoramic image that is 32 pixels tall and 128 pixels wide that look like Figure 4.

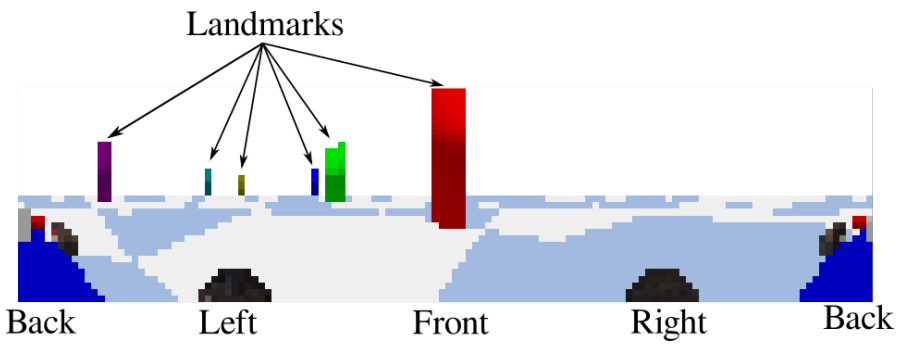

Figure 4: An illustration on the observation input of the car

(a) **Generate Datasets:** The first step in any learning problem is to aquire your dataset. We have provided the script `make_data.py` to generate data for you. This will randomly place the car in the scene, capture a panoramic image, and record the ground-truth bearing angle to each landmark. You can generate the train and test dataset using the following code:

```
$ python make_data.py --split train --seed 1234 --size 10000
$ python make_data.py --split test --seed 12345 --size 1000
```

Do not change `--seed` if you want consistent result. This will generate the folders `hw1_train_dataset` and `hw1_test_dataset` containing `rgb_XXXXXX.png` and `label_XXXXXX.npy`. The rgb images in each should look like 4. Zip these folders as `hw1_train_dataset.zip` and `hw1_test_dataset.zip`.

(b) **Implement Neural Network:** Next you will build a neural network to predict landmark heading directions from visual observations. To do this, it will be easiest to work in the provided colab notebook. Go to colab and upload `train_observation_model.ipynb`. Then click the folder icon next to the notebook shown in Figure 5 to open the file browser of your colab runtime. From here, upload `observation_model.py`, `hw1_train_dataset.zip` and `hw1_test_dataset.zip`.

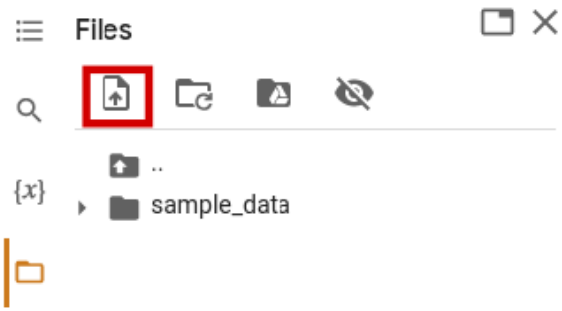

Figure 5: Click the file upload icon to upload your files.

We have already implemented a basic neural network in `observation_model.py` but you are encouraged to change it and check the various results. Your network should take a single `output_channels` argument that allows you to change how many values the network predicts (we will use this to change the prediction format in the next section). The forward function should take a batch of images in as a [`batch_size x 3 x 32 x 128`] tensor and output a `batch_size x output_channels` tensor. Note that

this assignment should not require a complicated model. Feel free to use external sources for network ideas, but please mention which ones you used in your writeup.

WARNING: Make sure you download your `observation_model.py` from colab frequently, and include it in your writeup when you are finished! If your runtime disconnects, you may lose the changes you made to this file!

(c) **Train Models:** In this section you will train two different versions of your model, one that predicts the bearing angles ϕ directly, and the other which predicts $\cos(\phi)$ and $\sin(\phi)$ instead.

(i) At the top of the notebook, a variable `supervision_mode` is defined. Set this to `"phi"` and run all the cells in the training notebook. This will train your network to predict the bearing angles of each landmark directly (6 channels, one for each landmark). After each epoch of training, the script will generate a set of debugging images that look like Figure 6. The first row of lines under the image shows the ground truth heading of the landmarks, while the second row shows the network's current predictions.

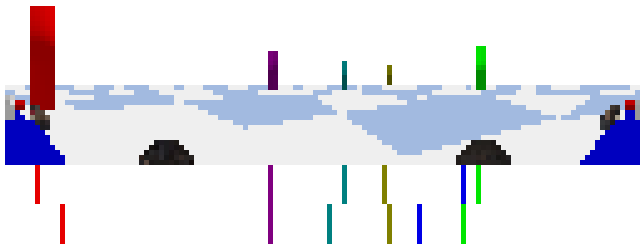

Figure 6: Debug images produced by the evaluation.

Report the error mean and standard deviation of your trained model.

(ii) Change `supervision_mode` to `"xy"`. This will train your network to predict $\cos(\phi)$ and $\sin(\phi)$ instead of the bearing angles directly (12 channels, two for each landmark). Report the mean and std of your model.

For reference, you should be able to get a model with less than 0.07 mean error for at least one of the supervision modes. Figure 7 shows the loss curve for a fairly simple convolutional neural network (using `"xy"` as `supervision_mode`) that achieves less than 0.05 mean error after 20 epochs.

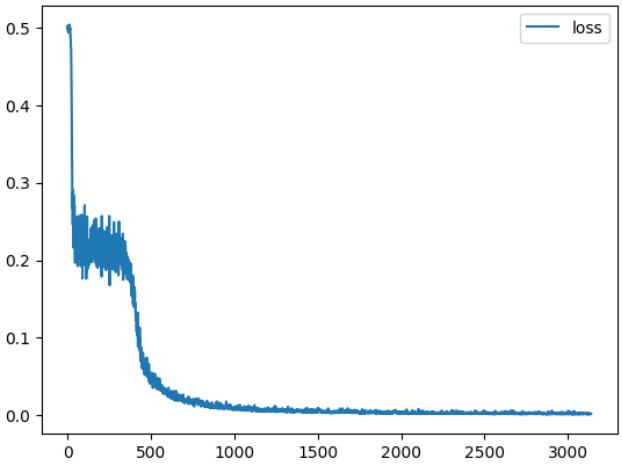

Figure 7: Loss curve for a simple convolutional network.

(iii) Did one of these work better than the other? Why do you think one might be superior?

(iv) Run particle filter with the learned observation model using the following command.

```
python localization.py pf --use-learned-observation-model \
checkpoint.pt --supervision-mode xy
```

Report the position error output in the writeup across 10 different random seeds.

(d) Discuss any challenges you faced and describe your debugging process.

(e) **Extra Credit: Analysis [10 points]**: Train your network again, with (1) an increased training dataset of 50000 examples (2) make the network deeper and/or wider (3) adjust training parameters such as learning rate (4) more epochs. What modifications are most helpful? Summarize your observation in the writeup.

3 Submission

We will be using the Canvas for submission of the assignments. Please submit a PDF writeup and a zip file of your entire working directory, excluding the training and test data for 2.2.

References

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