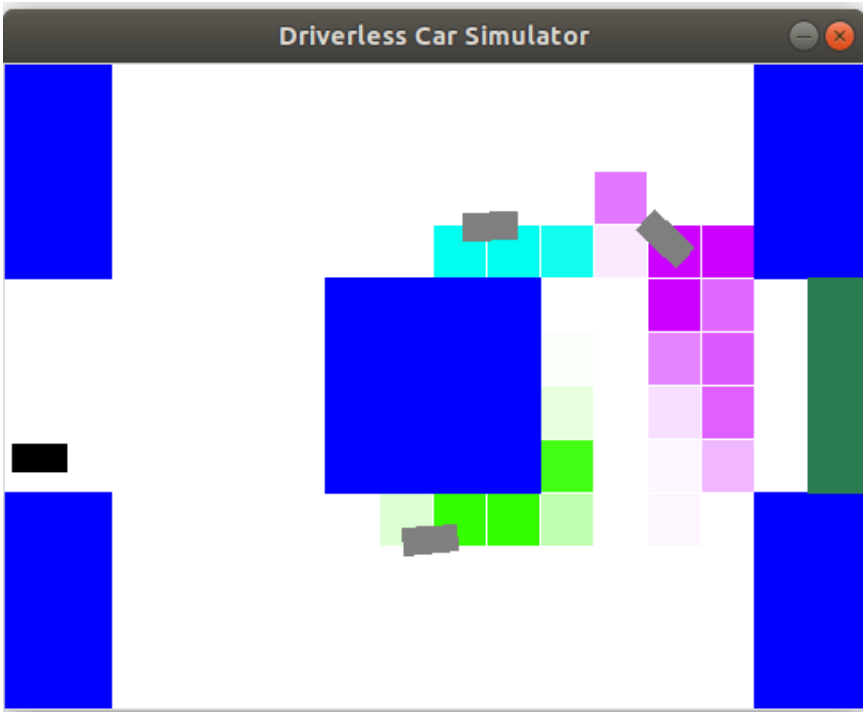
Spring 2021
Introduction to Artificial Intelligence

Homework 4: Car Tracking

Due Date: 2021/6/1 23:55



Introduction

Building an autonomous driving system is an incredibly complex endeavor. In this assignment, you will focus on the sensing system, which allows us to track other cars based on noisy sensor readings.

Getting started

This code base is a modified version of the Driverless Car written by Chris Piech at Stanford University.
(https://stanford-cs221.github.io/winter2021/assignments/car/index.html)

You can only run the code base on a local machine. Google Colab cannot execute it because it has GUI. Please install python 3 on your own machine and be familiar with the code with CLI.

Let's start by trying to drive manually, run the following command:

```
python drive.py -l lombard -i none
```

You can steer by either using the arrow keys or 'w', 'a', and 'd'. The up key and 'w' accelerates your car forward, the left key and 'a' turns the steering wheel to the left, and the right key and 'd' turns the steering wheel to the right. Note that you cannot reverse the car or turn in place. Quit by pressing 'q'. Your goal is to drive from the start to finish (the green box) without getting in an accident.

Arguments for python drive.py:

- -a: Enable autonomous driving (as opposed to manual).
- -i <inference method>: Use none, exactInference, particleFilter to (approximately) compute the belief distributions over the locations of the other cars.
- -l <map>: Use this map (e.g. small or lombard). Default is small.
- -d: Debug by showing all the cars on the map.
- -p: All other cars remain parked (so that they don't move).

The drive.py file is not used for any grading purposes, it's just there to visualize the code you will be writing and help you gain an appreciation for how different approaches result in different behaviors (and to have fun!).

Autograding

In this assignment, TAs will use a grader to grade your implementation. The grader has been included in the code base, and is different from homework 3. You can use the following command to test by yourself.

```
python grader.py
```

You can see the test cases in grader.py. The grader will check your code to determine whether it has correct outputs. After that, it will show the score you get.

To run a single test (e.g., part1-1), run the following command:

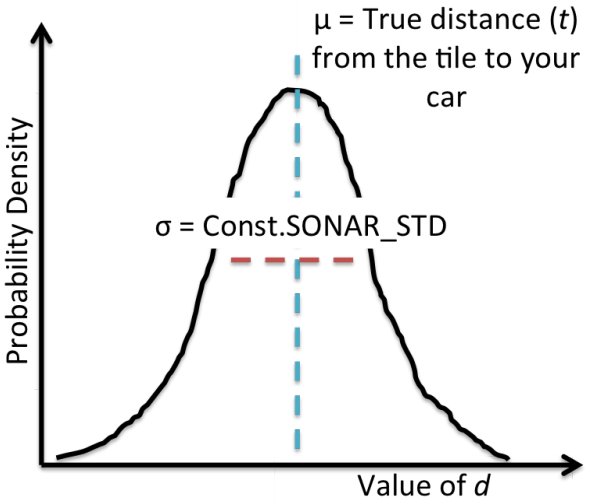
```
python grader.py part1-1
```

Modeling car locations

We assume that the world is a two-dimensional rectangular grid on which your car and K other cars reside. At each time step t, your car gets a noisy estimate of the distance to each of the cars. As a simplifying assumption, we assume that each of the K other cars move independently and that the noise in sensor readings for each car is also independent. Therefore, in the following, we will reason about each car independently (notationally, we will assume there is just one other car).

At each time step t, let $H_t \in \mathbb{R}^2$ be a pair of coordinates representing the actual location of the single other car (which is unobserved). We assume there is a local conditional distribution $p(h_t|h_{t-1})$ which governs the car's movement. Let $a_t \in \mathbb{R}^2$ be your car's position, which you observe and also control. To minimize costs, we use a simple sensing system based on a microphone. The microphone provides us with E_t, which is a Gaussian random variable with mean equal to the true distance between your car and the other car and variance σ^2 (in the code, σ is Const.SONAR_STD, which is about two-thirds the length of a car). In symbols, $E_t \sim N(\|a_t - H_t\|_2, \sigma^2)$.

For example, if your car is at $a_t = (1, 3)$ and the other car is at $H_t = (4, 7)$, then the actual distance is 5 and E_t might be 4.6 or 5.2, etc. Use util.pdf(mean, std, value) to compute the probability density function (PDF) of a Gaussian with given mean and standard deviation, evaluated at value. Note that evaluating a PDF at a certain value does not return a probability (densities can exceed 1), but for the purposes of this assignment, you can get away with treating it like a probability. The Gaussian probability density function for the noisy distance observation E_t, which is centered around your distance to the car $\mu = \|a_t - H_t\|_2$, is shown in the following figure:



Your job is to implement a car tracker that (approximately) computes the posterior distribution $\mathbb{P}(H_t|E_1 = e_1, \cdots, E_t = e_t)$ (your beliefs of where the other car is) and update it for each $t = 1, 2, \cdots$. We will take care of using this information to actually drive the car (i.e., set a_t to avoid a collision with h_t), so you don't have to worry about that part.

To simplify things, we will discretize the world into **tiles** represented by (row, col) pairs, where 0 <= row < numRows and 0 <= col < numCols. For each tile, we store a probability representing our belief that there's a car on that tile. The values can be accessed by: self.belief.getProb(row, col). To convert from a tile to a location, use util.rowToY(row) and util.colToX(col).

Requirements

Please modify the codes in submission.py between # Begin your code and # End your code. In addition, do not import other packages.

In part 1 and 2, you will implement ExactInference, which computes a full probability distribution of another car's location over tiles (row, col).

In part 3, you will implement ParticleFilter, which works with particle-based representation of this same distribution.

A few important notes before you get started:

- This assignment will be one of the most conceptually challenging assignments. Please start early!
- The code portions of this assignment are short and straightforward -- no more than about 40 lines in total -- but only if your understanding of the probability concepts is clear!
- As a notational reminder: we use the lowercase expressions $p(x)$ or $p(x|y)$ for local conditional probability distributions, which are defined by the Bayesian network. We use the uppercase expressions $\mathbb{P}(X = x)$ or $\mathbb{P}(X = x|Y = y)$ for joint and posterior probability distributions, which are not predefined in the Bayesian network but can be computed by probabilistic inference. Please review the lecture slides for more details.

Part 1: Emission probabilities (20%)

- In this part, we assume that the other car is stationary (e.g., $H_t = H_{t-1}, \forall t$). You will implement a function observe that upon observing a new distance measurement $E_t = e_t$ updates the current posterior probability from
$$\mathbb{P}(H_t|E_1 = e_1, \cdots, E_{t-1} = e_{t-1})$$
to $\mathbb{P}(H_t|E_1 = e_1, \cdots, E_t = e_t) \propto \mathbb{P}(H_t|E_1 = e_1, \cdots, E_{t-1} = e_{t-1})p(e_t|h_t)$,
where we have multiplied in the emission probabilities $p(e_t|h_t)$ described earlier under "Modeling car locations". The current posterior probability $\mathbb{P}(H_t|E_1 = e_1, \cdots, E_{t-1} = e_{t-1})$ is stored as self.belief in ExactInference.
- Once you implement the observe function, you should be able to find the stationary car by driving around it, run the following command:

```
python drive.py -a -p -d -k 1 -i exactInference
```

Note:

- Read through the Belief class in util.py before you get started. you'll need to use this class for several of the code tasks in this assignment.
- Read through the comment of the observe function in the ExactInference class of submission.py before you get started. Make sure you understand what to do.

- Remember to normalize the posterior probability after you update it.

Part 2: Transition probabilities (20%)

- Now, let's consider the case where the other car is moving according to transition probabilities $p(h_{t+1}|h_t)$. We have provided the transition probabilities for you in self.transProb. Specifically, self.transProb[(oldTile, newTile)] is the probability of the other car being in newTile at time step $t+1$ given that it was in oldTile at time step t.
- In this part, you will implement a function elapseTime that updates the posterior probability about the location of the car at a current time step t

$$\mathbb{P}(H_t = h_t | E_1 = e_1, \cdots, E_t = e_t)$$

to the next time step $t+1$ conditioned on the same evidence:

$$\mathbb{P}(H_{t+1} = h_{t+1} | E_1 = e_1, \cdots, E_t = e_t) \propto \sum_{h_t} \mathbb{P}(H_t = h_t | E_1 = e_1, \cdots, E_t = e_t) p(h_{t+1}|h_t)$$

Again, the posterior probability is stored as self.belief in ExactInference.

- Finish ExactInference by implementing the elapseTime function . When you are all done, you should be able to track a moving car well enough to drive autonomously by running the following command:

```
python drive.py -a -d -k 1 -i exactInference
```

Note:

- Read through the comment of the elapseTime function in the ExactInference class of submission.py before you get started. Make sure you understand what to do.
- You can also drive autonomously in the presence of more than one car:

```
python drive.py -a -d -k 3 -i exactInference
```

- You can also drive down Lombard:

```
python drive.py -a -d -k 3 -i exactInference -l lombard
```

On Lombard, the autonomous driver may attempt to drive up and down the street before heading towards the target area. Again, focus on the car tracking component, instead of the actual driving.

Part 3: Particle filtering (40%)

- Though exact inference works well for the small maps, it wastes a lot of effort computing probabilities for every available tile, even for tiles that are unlikely to have a car on them. We can solve this problem using a particle filter. Updates to the particle filter have complexity that's linear in the number of particles, rather than linear in the number of tiles.
- For a great conceptual explanation of how particle filtering works, check out this video (https://www.youtube.com/watch?v=aUkBa1zMKv4) on using particle filtering to estimate an airplane's altitude.

- In this part, you will implement observe and elapseTime functions for the ParticleFilter class in submission.py.
- Some of the code has been provided for you. For example, the particles have already been initialized randomly.
- When you're finished, your code should be able to track cars nearly as effectively as it does with exact inference. Run the following command:

```
python drive.py -a -i particleFilter -l lombard
```

Note:

- Read through the comment of the observe and elapseTime functions in the ParticleFilter class of submission.py before you get started. Make sure you understand what to do.
- To debug, you might want to start with the parked car (argument -p) and the display car (argument -d).

Report (20%)

- **A written report is required.**
- The report should be written in **English**.
- Save the report as a **.pdf** file.
    - font size: 12
- For part 1 ~ 3, please take some screenshots of your code and explain how you implement codes **in detail**.
- Describe problems you meet and how you solve them.

Submission

Please prepare your submission.py and report (.pdf) into `STUDENTID_hw4.zip`.

e.g. 309123456_hw4.zip

Late Submission Policy

20% off per late day