Programming Assignment 3
Probabilistic Reasoning

CMSC 421 Fall 2020
Due date: 11:59 PM, December 14, 2020

Introduction

Goal

In this programming assignment, you will localize a car in a racetrack, and analyze a competitive situation on the racetrack and their effects on successfully overtaking an opponent, avoiding collision, and winning a race. In order to localize the car, you will create a particle filter (and a Kalman filter for extra credit), and in order to analyze situations, you will implement a Bayesian network to model conditional dependencies between situations and outcomes. We will then ask you some questions about your implementations and performance.

Setup

1. Clone the repo at https://github.com/jdkanu/cmsc421-p3
2. Try running `python3 gui.py` and driving around the track with your arrow keys. You should see a blue car at the bottom, and it should move according to your keyboard inputs.

Directory structure

Files you will edit:

`particle_filter.py`	Write your particle filter code in here
`kalman_filter.py`	Write your kalman filter code in here
`bayesian_network.py`	Write your bayesian network code in here

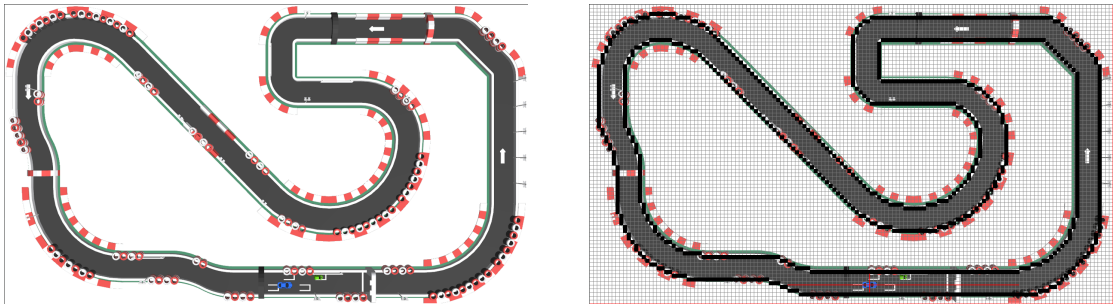
Files you will not edit:

`data`	Directory containing various data objects to define simulator
`agents.py`	Agents for `probability.py`
`aima_utils.py`	Utility functions for `probability.py`
`gui.py`	GUI for interacting with simulator
`car.py`	Car in simulator
`plots.py`	Generates plots for particle filtering and Kalman filtering
`probability.py`	File from AIMA code repo providing Bayesian network
`racetrack.py`	Racetrack in simulator
`simulator.py`	Simulator representing world with car and racetrack
`utils.py`	Utility functions

Simulator

Suppose for a moment that we want to set up an autonomous driving system to race a car around a racetrack, and we know the map of the racetrack ahead of time. In order to drive the car quickly around the track and avoid collisions, the system must know the current position of the car on the track (e.g., is the car approaching turn 3? is the car on the racing line?), as well as the direction in which the car is facing (e.g., is the car traveling straight down the road, or is it traveling toward a wall?), so that the system can generate the actions that are appropriate for the current state.

The problem of establishing the car's position and orientation in the map is the problem of localization. To simulate localization, we have set up a 2-D racetrack and car inside of a simulator.



(a) (b) label 2

Figure 1: Racetrack environment

The racetrack is located inside of a rectangular area 1400 units in width and 800 units in height. The car may move around the racetrack, but it can collide with the walls. The map is represented by an occupancy grid, where each cell is either 1 if a wall is present there, or 0 otherwise. The car has a throttle, brakes, and steering. The car also has range sensors that measure the direction in each of the 4 cardinal directions (North, South, East, West).

GUI

When using the GUI, you can control various aspects of the simulator. You will need to use the GUI to visualize your results. Listed below are the available keyboard inputs:

`↑ (ARROW UP)`	Press throttle
`↓ (ARROW DOWN)`	Press brakes
`← (ARROW LEFT)`	Steer left
`→ (ARROW RIGHT)`	Steer right
`o`	Toggle drawing of occupancy grid and sensor readings
`p`	Toggle particle filtering
`r`	Toggle drawing of particles
`k`	Toggle Kalman filtering
`d`	Toggle GPS noise distribution

Run `python3 gui.py -h` for info on how to specify parameters for particle filtering and Kalman filtering. Example usage:

```
python3 gui.py --num_particles 100 --max_sensor_range 100 --sensor_noise_std 2.0
```

plots.py

Run `plots.py` from the terminal to automatically generate plots for particle filtering and Kalman filtering using a pre-recorded drive around the track. Type `python3 plots.py -h` for info on how to specify parameters. Example usage for particle filtering (pf):

```
python3 plots.py -w pf --num_particles 100 --max_sensor_range 100 --sensor_noise_std
2.0 --filename pf.png
```

Example usage for Kalman filtering (kf):

```
python3 plots.py -w kf --gps_noise_dist gaussian --gps_noise_var 4.0 --filename
kf.png
```

Plots will be stored inside the plots directory.

Your tasks

1. Complete all tasks listed below as Task 1.*, Task 2.*, Task 3.*.
2. Submit your answers to Gradescope by the deadline.
3. Submit your code to the submit server.

Submission requirements

- Your final code and responses submitted should be your own work (though you are encouraged to collaborate, discuss and compare results with your classmates).
- Keep your answers brief but reasonably thorough. The best response is one that is accurate and rigorous, while using as few words as possible.
- For coding tasks, write all your code in the sections bounded by

```
# BEGIN_YOUR_CODE ######################################################
```

and

```
# END_YOUR_CODE ########################################################
```

Suggestions

You have 2 weeks to complete these tasks. Code for particle filtering should be less than 30 lines of code (approx), the coding portion for Bayesian networks is mostly counting, and Kalman filtering (extra credit) mostly involves setting the parameters of the Kalman filter from `filterpy`. I suggest you aim to finish particle filtering in the first week or early in the second week, and Bayesian networks in the second week. Continue to Kalman filtering (extra credit) if time permits.

1 Particle Filtering

In this section, you will create a particle filter to find the position and orientation of the car in the racetrack, given a known map of the racetrack and range sensor readings (distances to walls) from sensors on the car. You will not need to implement the entire algorithm from scratch—instead, we have provided some scaffolding in which you will write some important pieces of code.

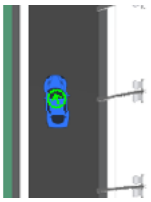


Figure 2: Estimated position and orientation after convergence of example particle filter.

Your particle filter will initialize a set of particles uniformly randomly within the known bounds of the map. Each particle represents a possible location and orientation given the information seen so far; collectively, the set of particles represents a distribution of possible locations and orientations. As new information comes in, your particle filter will update the particles to more accurately reflect the true distribution. At each moment, your particle filter will use its distribution to compute a single estimation of position and orientation of the car. Your particle filter is expected to eventually converge to a distribution which accurately reflects the true distribution, and estimate a position and orientation which is approximately equal to the true position and orientation, as shown in Figure 2.

When running the particle filter (`p` key), the simulator (`simulator.py`) will construct a `ParticleFilter` object from `particle_filter.py`, which initializes its particles throughout the map by calling

`initialize_particles`, a method you will implement. After updating the position and orientation of the car on each loop of the simulator, the loop will call `filtering_and_estimation` on its `ParticleFilter` object, to perform a single filtering step and compute a new estimate of position and orientation. This method invokes `filtering`, a method you will implement to perform the particle filtering algorithm given in the PARTICLE-FILTERING pseudocode found in the AIMA textbook. In your implementation of `filtering`, you should call `transition_sample` and `compute_prenorm_weight`, both of which you will also implement. This amounts to completing four methods listed below in Task 1.1:

Task 1.1 Complete the implementation of the particle filter in `particle_filter.py` to track the location and orientation of the car in the map. Methods to edit:

`initialize_particles`
Initializes the particles uniformly randomly within the bounds of the rectangular region.

`transition_sample`
Samples a next pose for a given particle according to the car's transition model.

`compute_prenorm_weight`
Computes the pre-normalization weight of a particle given evidence, such that normalizing the pre-normalization weights for all particles gives your $P(\text{evidence}|\text{particle pose})$ for each particle.

`filtering`
Performs one step of particle filtering according to particle-filtering pseudocode in AIMA. This should closely resemble their pseudocode.

The file `particle_filter.py` is heavily commented so you can know what exactly these methods need to do. There are some hints inside these methods, as well. The comments should be helpful, so read through them. You can complete this task in just 20 lines of code.

Task 1.2 Fix maximum sensor range to 400 and standard deviation of sensor noise to 2.0. Estimate the lowest particle count that reliably localizes the car by manually tuning this parameter (you probably will want to use `plots.py` for visualization instead of the GUI). Plot the estimated path and error using `plots.py`, and submit the plots along with the particle count you used.

Task 1.3 Characterize the dependence of accuracy and efficiency on sensor noise, number of particles, and max sensor range. Support your assertions using quantitative or anecdotal evidence (the more rigorous, the better). Relationships to characterize: (accuracy vs noise), (accuracy vs number), (accuracy vs range), (efficiency vs noise), (efficiency vs number), (efficiency vs range).

Task 1.4 What probabilistic assumptions are we making about the state transition model? How well do these assumptions match the true state transition model? In what way and to what extent is performance affected by making these assumptions?

Task 1.5 Describe some cases in which the particle filter fails to localize the car in this map, and explain why this happens.

Task 1.6 Describe one modification you could make to the system to improve accuracy.

Extra credit: Implement this modification and demonstrate the improvement in accuracy, using plots from `plots.py` or other appropriate figures.

2 Bayesian Networks

Suppose we are racing an opponent car up the diagonal road in the middle of our racetrack, approaching the second-to-last turn in the race before the finish line. We are behind our opponent, but we are moving faster toward the turn, and we are going to attempt to overtake them to win the race. In order to win the race, we must overtake the opponent car, ideally without colliding

with the tires, though a small collision does not rule out the possibility of victory. Our ability to overtake the other car depends (in part) on how much faster we are traveling than the other car, and how close we are to the turn (e.g., an early overtake attempt at much higher speed is more likely to succeed than a late overtake attempt when we're not moving much faster than the opponent). Similarly, our ability to avoid collision depends on how much faster we are traveling and how close we are to the turn (e.g., if we're traveling too fast and attempting the overtake too late, then we might miss the apex and crash hard into the tires).

Figure 3: Two cars racing through a turn.

Task 2.1 Let $MuchFaster \in \{T, F\}$ represent whether we are moving much faster than the opponent (as opposed to just a little faster), $Early \in \{T, F\}$ represent whether the attempt is early, $Overtake \in \{T, F\}$ represent whether we overtook the opponent, $Crash \in \{T, F\}$ represent whether we crashed, and $Win \in \{T, F\}$ represent whether we win the race. Draw the structure of the Bayesian network that represents the conditional dependencies described in this situation (and no other dependencies), using only the given variables. Assume $MuchFaster$ and $Early$ are independent.

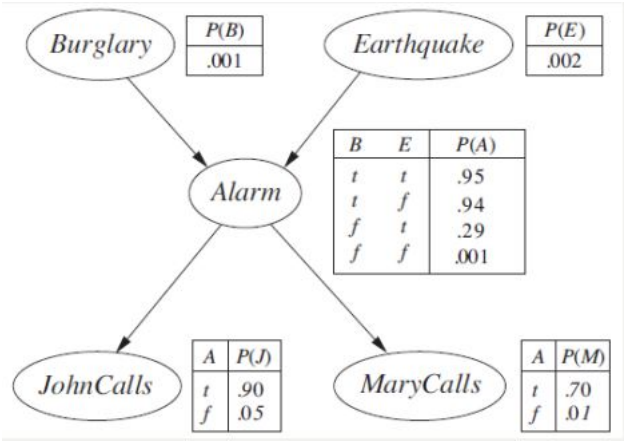


Figure 4: Bayesian network from AIMA.

Task 2.2 Suppose we have recorded 1000 different overtake attempts on this racetrack, and we want to model the joint probability distribution $P(MuchFaster, Early, Overtake, Crash, Win)$. For each instance, we have recorded the values for $MuchFaster, Early, Overtake, Crash$, and Win as $True$ or $False$, and stored the data points inside of a dataset. Instantiate your Bayesian network from Task 2.1 as a `BayesNet` from the AIMA code repository, complete with conditional probability tables for each node. Write your code inside the method `generate_bayesnet` in `bayesian_network.py`.

Hint: To estimate probabilities, use frequencies of occurrences in the dataset (e.g., estimate $P(Alarm = T, Burglary = F)$ as $\frac{\text{number of data points with } Alarm=T, Burgulary=F}{\text{total number of data points}}$).

Hint: If you were to construct a BayesNet for the network in Figure 4, you would write the following code:

```
T, F = True, False

burglary = BayesNet([
    ('Burglary', '', 0.001),
    ('Earthquake', '', 0.002),
    ('Alarm', 'Burglary Earthquake',
     {(T, T): 0.95, (T, F): 0.94, (F, T): 0.29, (F, F): 0.001}),
    ('JohnCalls', 'Alarm', {T: 0.90, F: 0.05}),
    ('MaryCalls', 'Alarm', {T: 0.70, F: 0.01})
])
```

The BayesNet constructor takes a list of node specifications (variable,parents,CPT), where variable is the name of the variable, parents is a string listing the names of parent nodes joined by spaces (empty string if there are no parents), and CPT is the conditional probability table represented as a dictionary, or just probability if there are no parents. The list must be ordered so that parent nodes come before child nodes.

Task 2.3 Using the network you generated, determine the overtaking condition (values for $MuchFaster$ and $Early$) that yields the highest probability of winning the race without crashing. Write your code inside `find_best_overtake_condition` in `bayesian_network.py`. Your code should display the correct answer when you run `python3 bayesian_network.py` from the command line. Write the mathematical formula that represents this optimal overtaking condition in terms of conditional probabilities found in the Bayesian network.

Hint: Use any one of the following methods from `probability.py` to compute a conditional probability using your BayesNet: `enumeration_ask, elimination_ask, rejection_sampling, likelihood_weighting, gibbs_ask`. We've already imported them for you. Example usage: to use variable elimination to compute the distribution $P(Burglary|JohnCalls = T, MaryCalls = T)$ on the burglary network, you would call `elimination_ask('Burglary', dict(JohnCalls=True, MaryCalls=True), burglary)`.

3 Kalman Filtering (Extra Credit)

Task 3.1 Implement a Kalman filter to track the location and velocity of the car in the map. Write your code in `kalman_filter.py`

Task 3.2 Explain how you set your parameters.

Task 3.3 Run the standard plot code in `plots.py` to generate plots, and submit them. Characterize the dependence of accuracy on sensor noise and sensor distribution (generate plots for Gaussian and uniform noise distributions, with different variance and width). Support your assertions using quantitative and/or anecdotal evidence (the more rigorous, the better).

Task 3.4 What assumptions are we making about the sensor readings? How well do these assumptions match the process that generates the data? In what way and to what extent is performance affected by these making these assumptions?