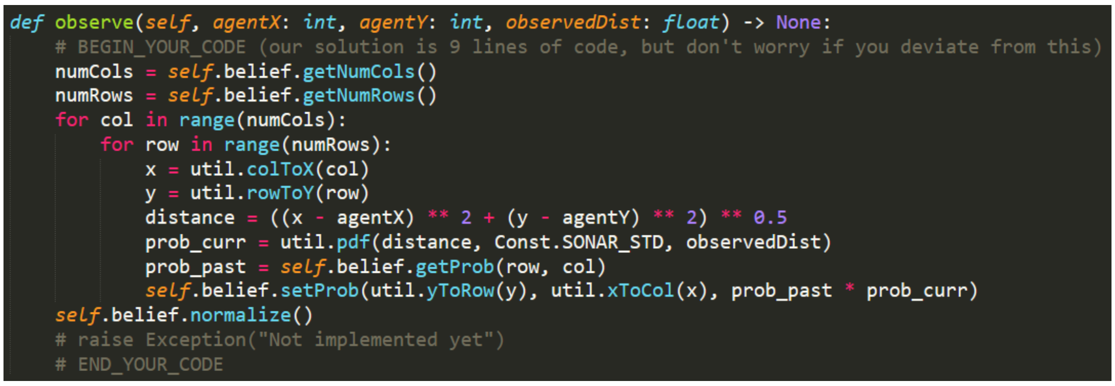
HW4 Car Tracking

0616098 黃秉茂

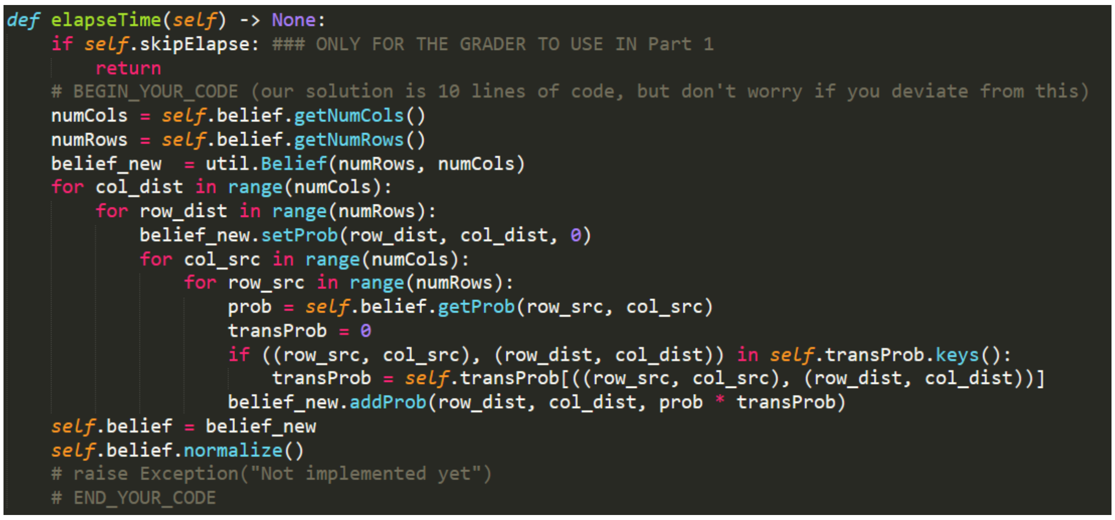
Part 1



```python
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE (our solution is 9 lines of code, but don't worry if you deviate from this)
    numCols = self.belief.getNumCols()
    numRows = self.belief.getNumRows()
    for col in range(numCols):
        for row in range(numRows):
            x = util.colToX(col)
            y = util.rowToY(row)
            distance = ((x - agentX) ** 2 + (y - agentY) ** 2) ** 0.5
            prob_curr = util.pdf(distance, Const.SONAR_STD, observedDist)
            prob_past = self.belief.getProb(row, col)
            self.belief.setProb(util.yToRow(y), util.xToCol(x), prob_past * prob_curr)
    self.belief.normalize()
    # raise Exception("Not implemented yet")
    # END_YOUR_CODE
```

Update the probabilities based on an observation.
To iterate all the tiles to calculate the probabillity of each tile, I choose to use the function in class belief to get the number of columns and rows and then iterate it.
Since all I get are the index, I need to convert the tile with column and row indices to the true location in x and y coordinate with the function in util. Such that I can calculate the distance.
Calculate the distance between the observed car and my car with L2-norm.
To get the emission probabilities which governs the car's movement, use util.pdf to compute the probability density function of a Gaussian with given mean and standard deviation, evaluated at value
To get the posterior distribution, the probability representing our belief that there's a car on that tile, Accessing the value with self.belief.getProb.
Multiply the emission probabilities and the posterior distribution to get the current posterior probability, and updates it into the class Belief.
After each tile has been updated, normalize the probabilities in the class Belief.

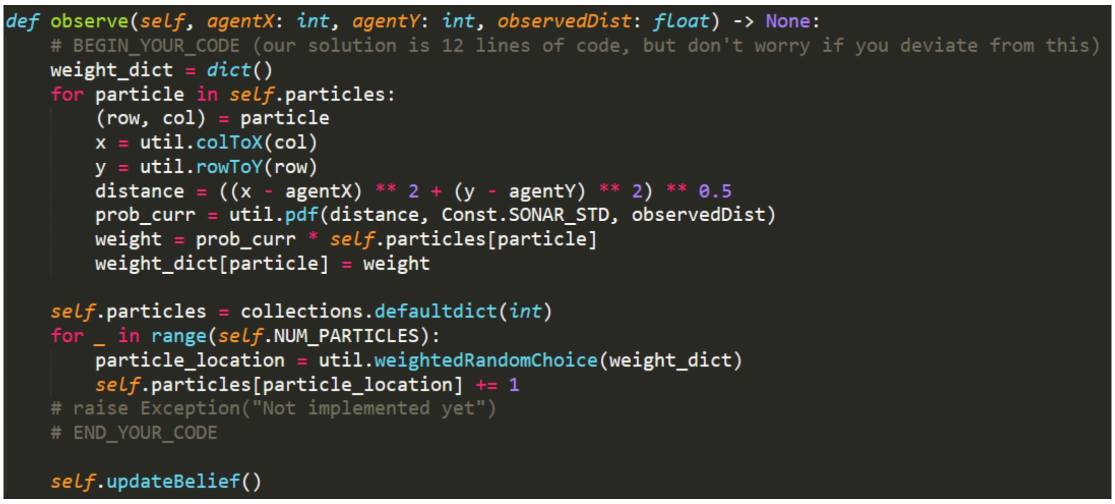
Part 2



```python
def elapseTime(self) -> None:
    if self.skipElapse: ### ONLY FOR THE GRADER TO USE IN Part 1
        return
    # BEGIN_YOUR_CODE (our solution is 10 lines of code, but don't worry if you deviate from this)
    numCols = self.belief.getNumCols()
    numRows = self.belief.getNumRows()
    belief_new  = util.Belief(numRows, numCols)
    for col_dist in range(numCols):
        for row_dist in range(numRows):
            belief_new.setProb(row_dist, col_dist, 0)
            for col_src in range(numCols):
                for row_src in range(numRows):
                    prob = self.belief.getProb(row_src, col_src)
                    transProb = 0
                    if ((row_src, col_src), (row_dist, col_dist)) in self.transProb.keys():
                        transProb = self.transProb[((row_src, col_src), (row_dist, col_dist))]
                    belief_new.addProb(row_dist, col_dist, prob * transProb)
    self.belief = belief_new
    self.belief.normalize()
    # raise Exception("Not implemented yet")
    # END_YOUR_CODE
```

Propose a new belief distribution based on a learned transition model.
To iterate all the tiles to calculate the probabillity of each tile, I choose to use the function in class belief to get the number of columns and rows and then iterate it.
For each tile, I have to calculate the
To be sure that I am using only the CURRENT self.belief distribution to compute updated beliefs, I create a new Belief to store the new value. Use new dictionary to avoid changing the value in old one which used for calculating.
For each tile, the elapse time is the sum of the probabilities which are the posterior probability of each tile multiply the transition probabilities start from each tile and end to the tile.
_src and _dist represent the starting point and ending point of the transition probabilities.
Set the probability of each tile 0 first, and then increase it with the probabilities which are the posterior probability of each tile multiply the transition probabilities start from each tile and end to the tile and util.addProb iteratively.
The transition probability is 0 when the starting point and ending point are not in the keys of the dictionary. Otherwise, get it from the dictionary.
After calculating all the elapse time of each tile, assign it to the self.belief.
Finally, normalize the probabilities in the class Belief.

Part 3-1



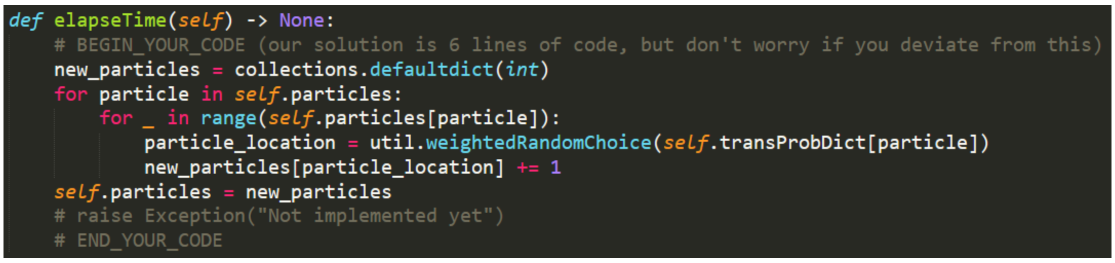
```python
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE (our solution is 12 lines of code, but don't worry if you deviate from this)
    weight_dict = dict()
    for particle in self.particles:
        (row, col) = particle
        x = util.colToX(col)
        y = util.rowToY(row)
        distance = ((x - agentX) ** 2 + (y - agentY) ** 2) ** 0.5
        prob_curr = util.pdf(distance, Const.SONAR_STD, observedDist)
        weight = prob_curr * self.particles[particle]
        weight_dict[particle] = weight

    self.particles = collections.defaultdict(int)
    for _ in range(self.NUM_PARTICLES):
        particle_location = util.weightedRandomChoice(weight_dict)
        self.particles[particle_location] += 1
    # raise Exception("Not implemented yet")
    # END_YOUR_CODE

    self.updateBelief()
```

Update the probabilities based on an observation.
weight_dict is the dictionary stores the partical location with its new weight. Use new dictionary to avoid changing the value in old one which used for calculating.
Re-weight:
For each particle, since all I get is the index, I need to convert the tile with column and row indices to the true location in x and y coordinate with the function in util. Such that I can calculate the distance.
Calculate the distance between the observed car and my car with L2-norm.
To get the emission probabilities which governs the car's movement, use util.pdf to compute the probability density function of a Gaussian with given mean and standard deviation, evaluated at value
Multiply the emission probabilities and the origin weight of the particle to get the new weight of the indices of the tile of the particle, and keep it into the dictionary weight_dict.

Re-sample:
clear and initialize the self.particles to the clean dictionary.
Sample the numbers of the particles times.
Sample with the function util.weightedRandomChoice and the dictionary weight_dict. I'll get the sampled indices and increase the self.particles with the indices by 1.

Finally, update self.belief with self.updateBelief().

Part 3-2



```python
def elapseTime(self) -> None:
    # BEGIN_YOUR_CODE (our solution is 6 lines of code, but don't worry if you deviate from this)
    new_particles = collections.defaultdict(int)
    for particle in self.particles:
        for _ in range(self.particles[particle]):
            particle_location = util.weightedRandomChoice(self.transProbDict[particle])
            new_particles[particle_location] += 1
    self.particles = new_particles
    # raise Exception("Not implemented yet")
    # END_YOUR_CODE
```

Propose a new belief distribution based on a learned transition model.
new_particles is the dictionary stores the partical location with its new numbers of the particles, and it is new distribution of the sampled particles. Use new dictionary to avoid changing the value in old one which used for calculating.
For each particle location, get the particle numbers of the particle location,and then sample the numbers of time. If there are multiple particles at a particular location, call util.weightedRandomChoice() once for each of them.
For each iteration, sample with the function util.weightedRandomChoice and the transition probability dictionary, which can view as weights of particles will move on. Get the sampled indices and increase the new_particles with the indices by 1, which keep the partical location with its new numbers of the particles, and it is new distribution of the sampled particles.
Finally, assign new_particles to self.particles

Describe problems you meet and how you solve them.

- How to represent the distribution?
  - The probability stores in each column and each row in 2D-list.
- How to calculate the current posterior probability in Part 1?
  - Multiply the emission probabilities and the posterior distribution.
- How to ensure not to change the value which I need to used for calculate?
  - Create another likely data structure and store new value into it.
- How to calculate elapse time in Part 2?
  - Use 4-layer for-loop to iterate the starting point and ending point of the transition probabilities, and then calculate the sum of the probabilities which are the posterior probability of each tile multiply the transition probabilities start from each tile and end to the tile.
- How to calculate weight in Part 3-1?
  - Multiply the emission probabilities and the origin weight of the particle instead of probability in self.belief to get the new weight of the indices of the tile of the particle.
- How to sample in Part 3-1?
  - Create a new dictionary, call util.weightedRandomChoice() with weight dictionary and increased by 1 with the index we get.
- How to sample in Part 3-2?
  - Sample with calling util.weightedRandomChoice() with the transition probability dictionary, which can view as weights of particles will move on, and increased by 1 with the index we get.