Step1

Check Yourself 1.

What is the internal state of the machine?

The internal state of the machine is a tuple. If the current time is assumed to be t, then the first term of the tuple is the robot's pose at time (t-1) and the second term of the tuple is the sonar reading at time (t-1).

What is the starting state?

The starting state is tuple (None, None).

Step2

Test the preprocessor as follows.

We define numObservations=10 and stateWidth = 1.0.

```
class PreProcess(sm.SM):

    def __init__(self, numObservations, stateWidth):
        self.startState = (None, None)
        self.numObservations = numObservations
        self.stateWidth = stateWidth

    def getNextValues(self, state, inp):
        (lastUpdatePose, lastUpdateSonar) = state
        print state
        currentPose = inp.odometry
        currentSonar = idealReadings.discreteSonar(inp.sonars[0],
                                                   self.numObservations)
        # Handle the first step
        if lastUpdatePose == None:
            return ((currentPose, currentSonar), None)
        else:
            action = discreteAction(lastUpdatePose, currentPose,
                                    self.stateWidth)
            print (lastUpdateSonar, action)
            return ((currentPose, currentSonar), (lastUpdateSonar, action))

# Only works when headed to the right
def discreteAction(oldPose, newPose, stateWidth):
    return int(round(oldPose.distance(newPose) / stateWidth))

pp1=PreProcess(10, 1.0)
print pp1.transduce(preProcessTestData,verbose=True)
```

```
Start state: (None, None)
(None, None)
In: <__main__.SensorInput instance at 0x06A014B8> Out: None Next State: (pose:(1.000000, 0.500000, 0.000000), 5)
(pose:(1.000000, 0.500000, 0.000000), 5)
(5, 1)
In: <__main__.SensorInput instance at 0x06A014E0> Out: (5, 1) Next State: (pose:(2.400000, 0.500000, 0.000000), 1)
(pose:(2.400000, 0.500000, 0.000000), 1)
(1, 5)
In: <__main__.SensorInput instance at 0x06A01530> Out: (1, 5) Next State: (pose:(7.300000, 0.500000, 0.000000), 1)
[None, (5, 1), (1, 5)]
```

The Outputs are consistent with wk12.2.3.

`Preprocessor (at time 0):`

- Input: an instance of `io.SensorInput`:
  - `sonars = (0.8, 1.0, ...)`
  - `odometry = Pose(1.0, 0.5, 0.0)`
- Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
  - `obs =` None
  - `act =` None

`Preprocessor (at time 1):`

- Input: an instance of `io.SensorInput`:
  - `sonars = (0.25, 1.2, ...)`
  - `odometry = Pose(2.4, 0.5, 0.0)`
- Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
  - `obs =` 5
  - `act =` 1

`Preprocessor (at time 2):`

- Input: an instance of `io.SensorInput`:
  - `sonars = (0.16, 0.2, ...)`
  - `odometry = Pose(7.3, 0.5, 0.0)`
- Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
  - `obs =` 1
  - `act =` 5

Step3

Define startDistribution, which should be uniform over all possible discrete robot locations.

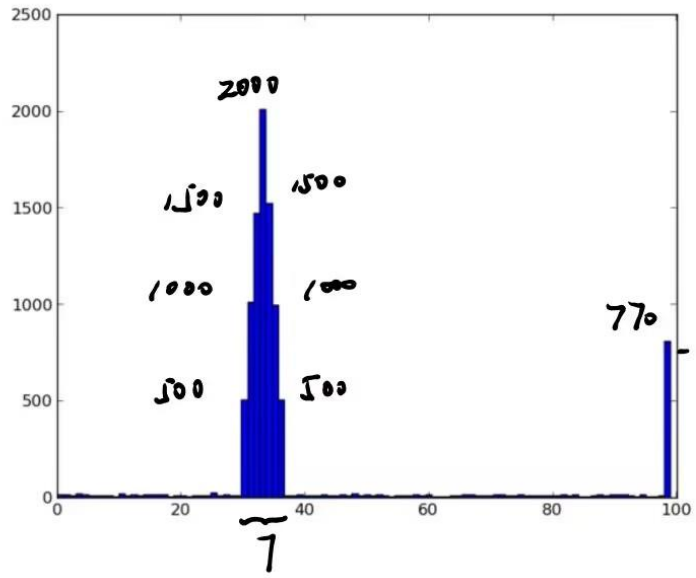
```
startDistribution = dist.squareDist(0,numStates)
```

The startDistribution is a squareDist from 0 to numStates-1. (dist.squareDist(lo, hi) returns an average probability distribution from lo to hi-1).

Check Yourself 2.

Sketch out your plan for the observation model. Be sure you understand the type of the model and the mixture distributions you want to create. Ask a staff member if you're unsure on any of these points.

Let's analyze the legend given.



The probability distribution in the figure is composed of three parts: sqareDist, triangleDist and deltaDist.

In x direction: The width of noise of triangleDist in this legend is 7 which means noise in my code is 3 when numStates is equal to 100. The deltaDist is in numStates-1. The other probabilities can be thought of as evenly distributed at each point.

In y direction: The sum of the number of measurement times in triangleDist distribution is 8000, accounting for 0.8 of the total number of measurement times. The

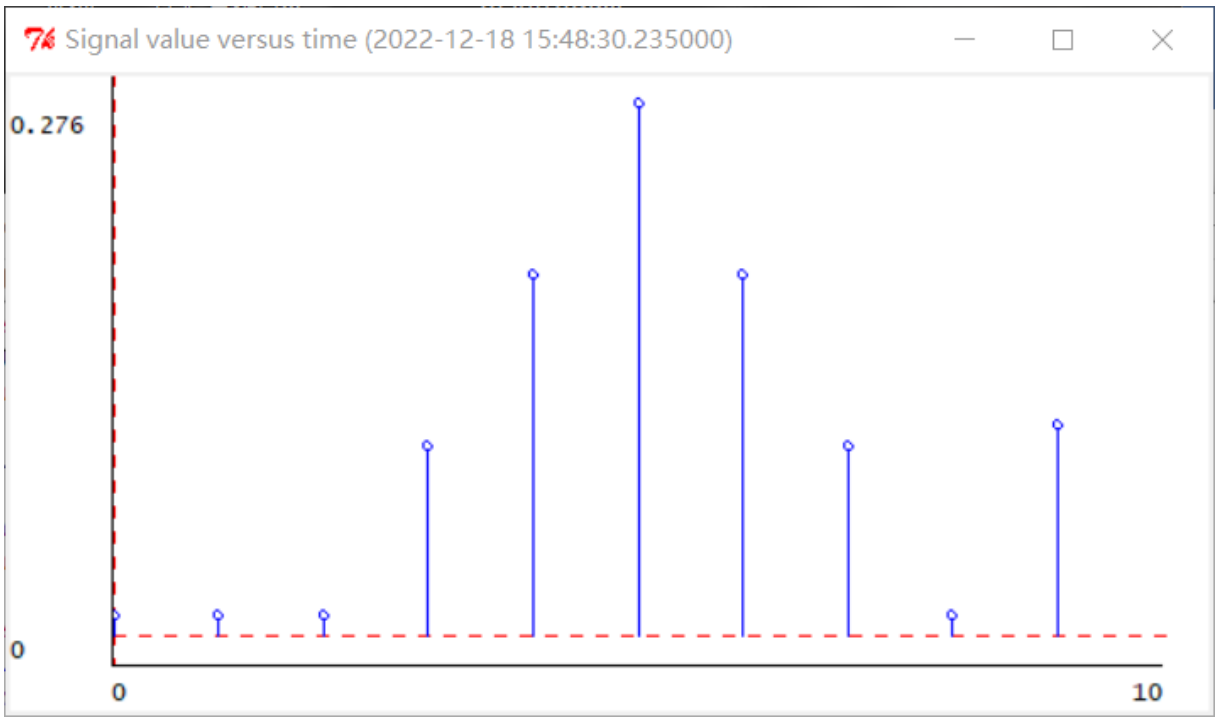
sum of the number of measurement times in the deltaDist distribution is 770, accounting for about 0.1 of the total number of measurements. Others accounts for about 0.1.

```
def observationModel(ix):
    noise = 3
    Fully_triangular_distribution = dist.triangleDist(ideal[ix],noise,0,numObservations-1)
    Fully_square_distribution = dist.squareDist(0,numObservations)

    Delta_distribution = dist.DeltaDist(numObservations-1)

    return dist.MixtureDist(Fully_triangular_distribution,
                            dist.MixtureDist(Fully_square_distribution
                                             ,Delta_distribution,0.6)
                            ,0.8)
```

Returns a Mixeturedist. The two distributions of the mixture are the Fully_triangular_distribution (with a probability of 0.8) and the other Mixeturedist, which is a mixture of the square distribution (with a probability of 0.1) and the delta distribution (with a probability of 0.1).

Step4

```
def makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations):

    startDistribution = dist.squareDist(0,numStates)

    def observationModel(ix):
        noise = 3
        Fully_triangular_distribution = dist.triangleDist(ideal[ix],noise,0,numObservations-1)
        Fully_square_distribution = dist.squareDist(0,numObservations)

        Delta_distribution = dist.DeltaDist(numObservations-1)

        return dist.MixtureDist(Fully_triangular_distribution,
                                dist.MixtureDist(Fully_square_distribution
                                                 ,Delta_distribution,0.5)
                                ,0.8)

    def transitionModel(a):
        pass

    distPlot.plot(observationModel(7))

    return ssm.StochasticSM(startDistribution, transitionModel,
                            observationModel)

model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
model.observationDistribution
```



The 4 highest-probability entries in observationModel(7) are sonar readings 4, 5, 6, 9. Sonar reading 5 is the highest-probability entry. The 7 stands for the eighth state (state 7) in discrete robot location.

Step5

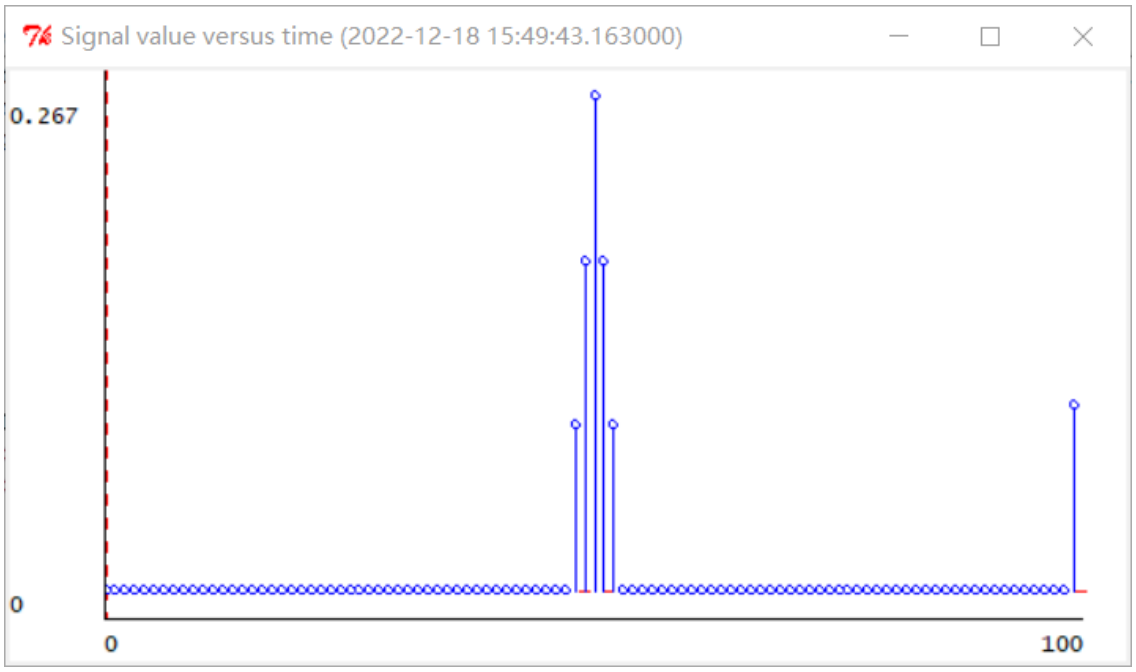
Now, make a model for the case with 100 observation bins, instead of 10.

```
def makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations):

    startDistribution = dist.squareDist(0,numStates)

    def observationModel(ix):
        noise = 3
        Fully_triangular_distribution = dist.triangleDist(ideal[ix],noise,0,numObservations-1)
        Fully_square_distribution = dist.squareDist(0,numObservations)

        Delta_distribution = dist.DeltaDist(numObservations-1)

        return dist.MixtureDist(Fully_triangular_distribution,
                                dist.MixtureDist(Fully_square_distribution
                                                 ,Delta_distribution,0.5)
                                ,0.8)

    def transitionModel(a):
        pass

    distPlot.plot(observationModel(7))

    return ssm.StochasticSM(startDistribution, transitionModel,
                            observationModel)

##model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
##model.observationDistribution
model100 = makeRobotNavModel(testIdealReadings100, 0.0, 10.0, 10, 100)
model100.observationDistribution
```

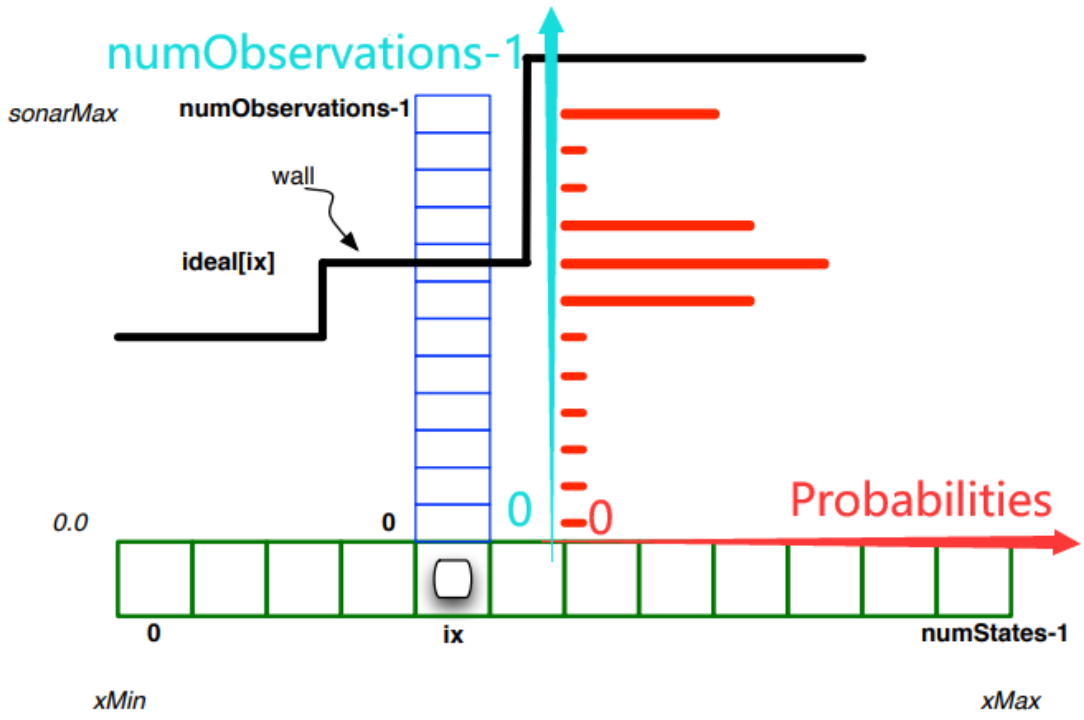
Checkoff 1.

Wk.13.2.1

Show your observation distribution plots to a staff member and explain what they mean.



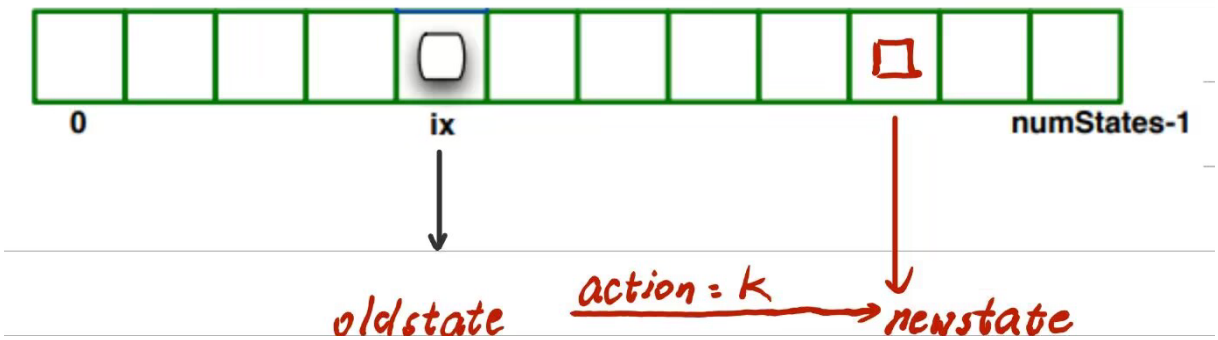
The distribution plots reflect as the following picture:



The X-axis of the probability distribution, which is the blue axis in the picture, is the discretized position of sonar readings from 0 to sonarMax under numObservations. The Y-axis of the probability distribution is the red axis in the picture, which is the probability distribution of sonar reading at position ix. The maximum value of the Y-axis should be given under the discretized position corresponding to ideal[ix]. When numObservations=100, the ideal reading for state7 = 50, so the maximum probability is taken at 50. The triangular distribution is caused by the error of sonar readings. The rest of the probability distribution is generated by other uncertainties such as someone walking by.

Check Yourself 3.

Sketch out your plan for the transition model. Be sure you understand the type of the models and the distributions you want to create. Ask a staff member if you're unsure on any of these points.



Ideal newstate should be a discretization being equal to discretized oldstate added to discretized action k. Newstate should be in range of 0 to numStates-1.

We use a triangle distribution to model the discretization error. That is to say, the maximum probability of newstate should be given under the discretized position

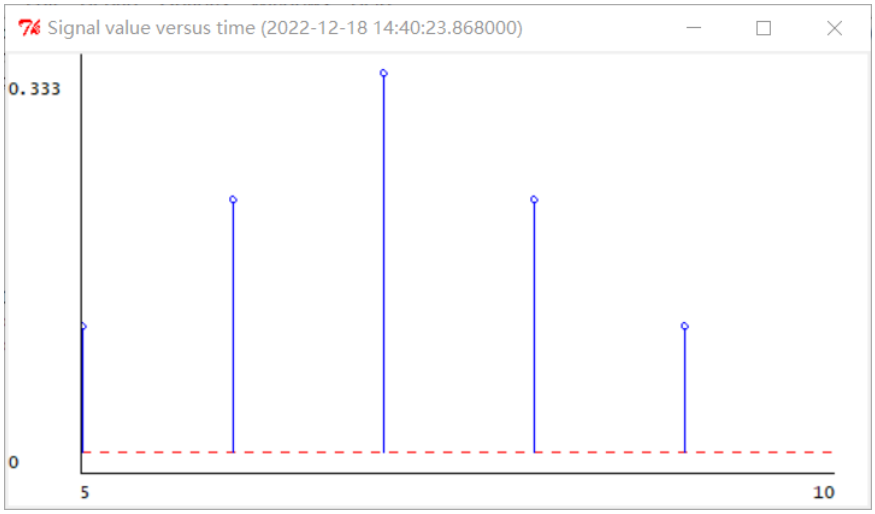
corresponding to ideal newstate and the noise we defined is similar to observationModel that is 7.

```
def transitionModel(a):
    noise = 3
    def transitionGivenI(oldState):
        return dist.triangleDist(util.clip(oldState+a, 0, numStates-1),
                                 noise, 0, numStates-1)
    return transitionGivenI
```

Step6

Let's add this code to test the program and run it.

```
def makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations):

    startDistribution = dist.squareDist(0,numStates)

    def observationModel(ix):
        noise = 3
        Fully_triangular_distribution = dist.triangleDist(ideal[ix],noise,0,numObservations-1)
        Fully_square_distribution = dist.squareDist(0,numObservations)

        Delta_distribution = dist.DeltaDist(numObservations-1)

        return dist.MixtureDist(Fully_triangular_distribution,
                                dist.MixtureDist(Fully_square_distribution
                                                 ,Delta_distribution,0.5)
                                ,0.8)

    def transitionModel(a):
        noise = 3
        def transitionGivenI(oldState):
            return dist.triangleDist(util.clip(oldState+a, 0, numStates-1),
                                     noise, 0, numStates-1)
        return transitionGivenI

    distPlot.plot(transitionModel(2)(5))
    distPlot.plot(observationModel(7))

    return ssm.StochasticSM(startDistribution, transitionModel,
                            observationModel)

model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
```



The result meets our design requirements in the previous section.

Step7

```
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
model.observationDistribution
model.transitionDistribution
pp1=PreProcess(10, 1.0)
pp2 = seGraphics.StateEstimator(model)
ppEst = sm.Cascade(pp1,pp2)
print ppEst.transduce(preProcessTestData,verbose = True)
```

Check Yourself 4.

Do ppEst.transduce(preProcessTestData). Compare the result to the belief states in Wk.12.2.3. Remember that you are now assuming noisy observations and noisy actions. Are your results consistent with the ones you found in the tutor?

Results:

```
>>>
Start state: ((None, None), DDist(0: 0.100000, 1: 0.100000, 2: 0.100000, 3: 0.10
0000, 4: 0.100000, 5: 0.100000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.1000
00))
In: <__main__.SensorInput instance at 0x08C5D260> Out: DDist(0: 0.100000, 1: 0.1
00000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100000, 6: 0.100000, 7: 0.100
000, 8: 0.100000, 9: 0.100000) Next State: ((pose:(1.000000, 0.500000, 0.000000)
, 5), DDist(0: 0.100000, 1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5:
0.100000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000))
(5, 1)
(5, 1)
In: <__main__.SensorInput instance at 0x08C5D288> Out: DDist(0: 0.080000, 1: 0.0
81905, 2: 0.083810, 3: 0.084762, 4: 0.084762, 5: 0.059365, 6: 0.059365, 7: 0.085
714, 8: 0.136508, 9: 0.243810) Next State: ((pose:(2.400000, 0.500000, 0.000000)
, 1), DDist(0: 0.080000, 1: 0.081905, 2: 0.083810, 3: 0.084762, 4: 0.084762, 5:
0.059365, 6: 0.059365, 7: 0.085714, 8: 0.136508, 9: 0.243810))
(1, 5)
(1, 5)
In: <__main__.SensorInput instance at 0x08C5D2D8> Out: DDist(3: 0.000614, 4: 0.0
18609, 5: 0.054389, 6: 0.089593, 7: 0.164714, 8: 0.204264, 9: 0.467818) Next Sta
te: ((pose:(7.300000, 0.500000, 0.000000), 1), DDist(3: 0.000614, 4: 0.018609, 5
: 0.054389, 6: 0.089593, 7: 0.164714, 8: 0.204264, 9: 0.467818))
[DDist(0: 0.100000, 1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100
000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000), DDist(0: 0.080000, 1:
0.081905, 2: 0.083810, 3: 0.084762, 4: 0.084762, 5: 0.059365, 6: 0.059365, 7: 0.
085714, 8: 0.136508, 9: 0.243810), DDist(3: 0.000614, 4: 0.018609, 5: 0.054389,
6: 0.089593, 7: 0.164714, 8: 0.204264, 9: 0.467818)]
>>>
```

Compared with wk12.2.3

Estimator (at time 0):

- Input: (obs, act) the output tuple from Preprocessor
- Output: probability distribution over robot states (x indices)

0.1	0.1	0.1	0.1	0.1	0.1
0.1	0.1	0.1	0.1		

. Estimator (at time 1):

- Input: (obs, act) the output tuple from Preprocessor
- Output: probability distribution over robot states (x indices)

0	0.25	0	0	0.25	0
0	0	0.25	0.25		

Estimator (at time 2):

- Input: (obs, act) the output tuple from Preprocessor
- Output: probability distribution over robot states (x indices)

0	0	0	0	0	0
$\frac{1}{3}$	0	0	$\frac{2}{3}$		

The results are different.

Checkoff 1.

Wk.13.2.2:

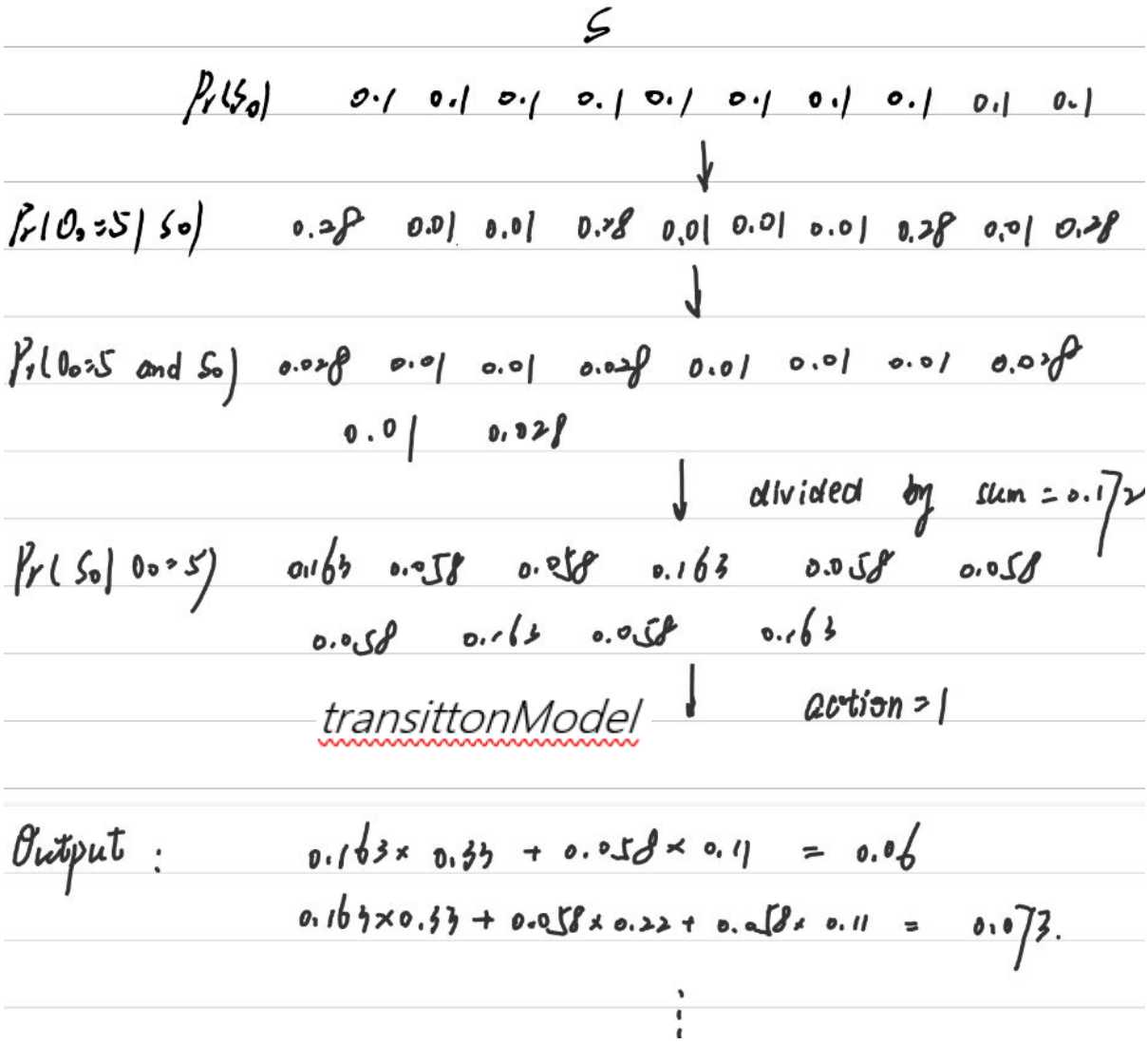
Show your answers to the questions above to a staff member. Explain what they mean.

In Wk12.2.3, We've only talked about the ideal case. In an ideal world, sonar would correctly detect the distance to the obstacle and the robot would move to the next step accurately. However, in this case, the distance measured by sonars and robot's moving position are all operated according to our previous definition, which means

noise. Let's use the following example to explain.

The transittonModel in time 1:

```
DDist(0: 0.333333, 1: 0.333333, 2: 0.222222, 3: 0.111111)
DDist(0: 0.111111, 1: 0.222222, 2: 0.333333, 3: 0.222222, 4: 0.111111)
DDist(1: 0.111111, 2: 0.222222, 3: 0.333333, 4: 0.222222, 5: 0.111111)
DDist(2: 0.111111, 3: 0.222222, 4: 0.333333, 5: 0.222222, 6: 0.111111)
DDist(3: 0.111111, 4: 0.222222, 5: 0.333333, 6: 0.222222, 7: 0.111111)
DDist(8: 0.111111, 4: 0.111111, 5: 0.222222, 6: 0.333333, 7: 0.222222)
DDist(8: 0.222222, 9: 0.111111, 5: 0.111111, 6: 0.222222, 7: 0.333333)
DDist(8: 0.333333, 9: 0.333333, 6: 0.111111, 7: 0.222222)
DDist(8: 0.222222, 9: 0.666667, 7: 0.111111)
DDist(8: 0.222222, 9: 0.666667, 7: 0.111111)
```



Compared with the result in time1:

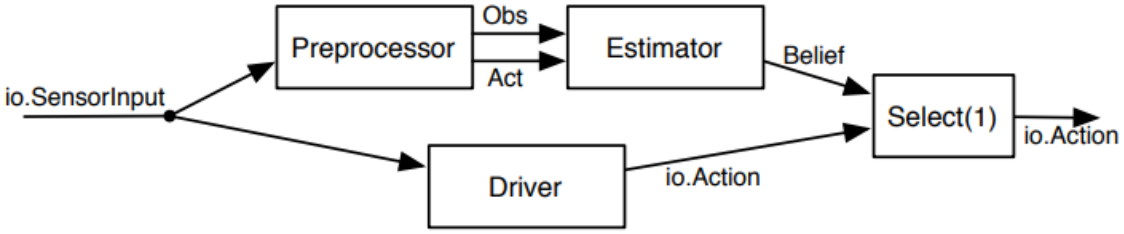
```
 1), DDist(0: 0.080000, 1: 0.081905, 2: 0.083810, 3: 0.084762, 4: 0.084762, 5:
.059365, 6: 0.059365, 7: 0.085714, 8: 0.136508, 9: 0.243810))
```

Because we are approximating the decimal part here, the final output is somewhat different from the actual output of the computer, but the error is acceptable.

Step8

```
def makeLineLocalizer(numObservations, numStates, ideal, xMin, xMax, robotY):
    stateWidth = (xMax-xMin)/float(numStates)
    pp1=PreProcess(numObservations,stateWidth)
    model = makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations)
    pp2 = seGraphics.StateEstimator(model)
    ppEst = sm.Cascade(pp1,pp2)
    driver = move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0), maxVel = 0.5)
    return sm.Cascade(sm.Parallel(ppEst, driver), sm.Select(1))
```

The code embodies the following structure.

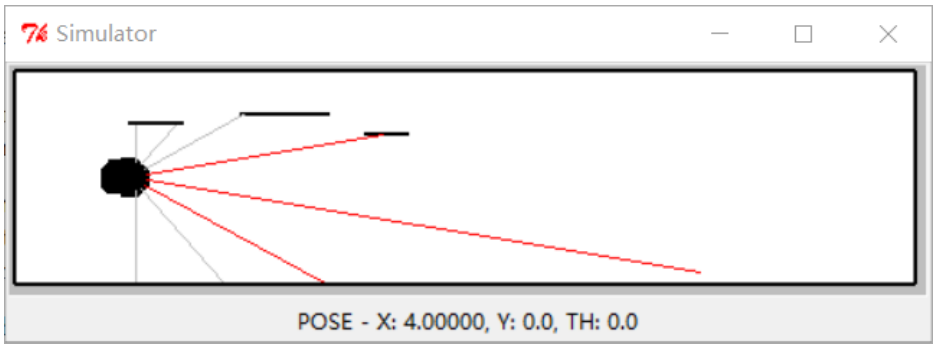


Step9

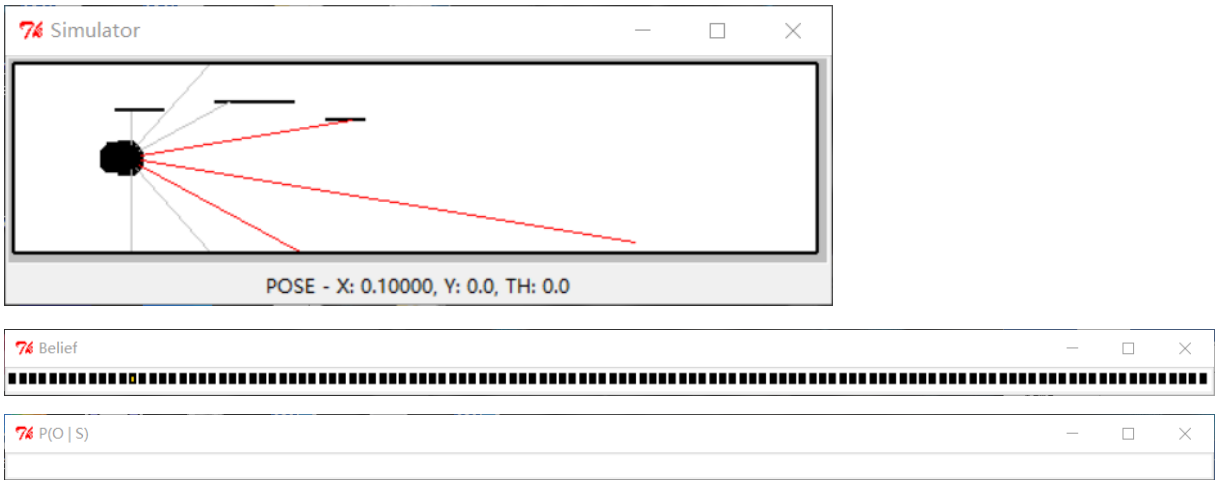
The robot's current observation is one that is likely to be observed when it is in any of the locations that is colored blue, and unlikely to be observed in the locations colored red.

The robot's actual position is shown in the belief stats in the way that Black is the uniform probability, brighter blue is more likely, brighter red is less likely.

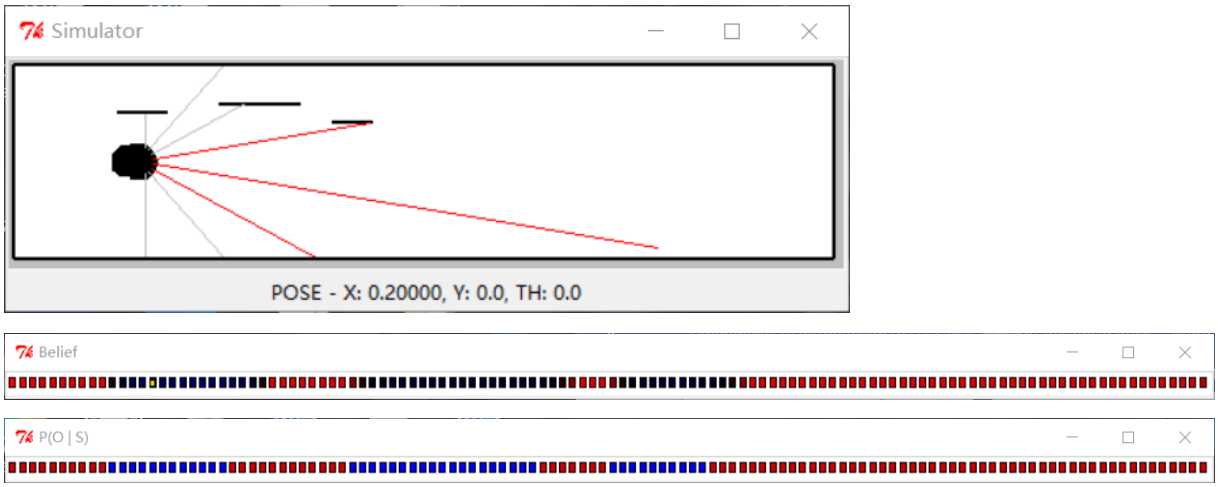
In oneDdiff:



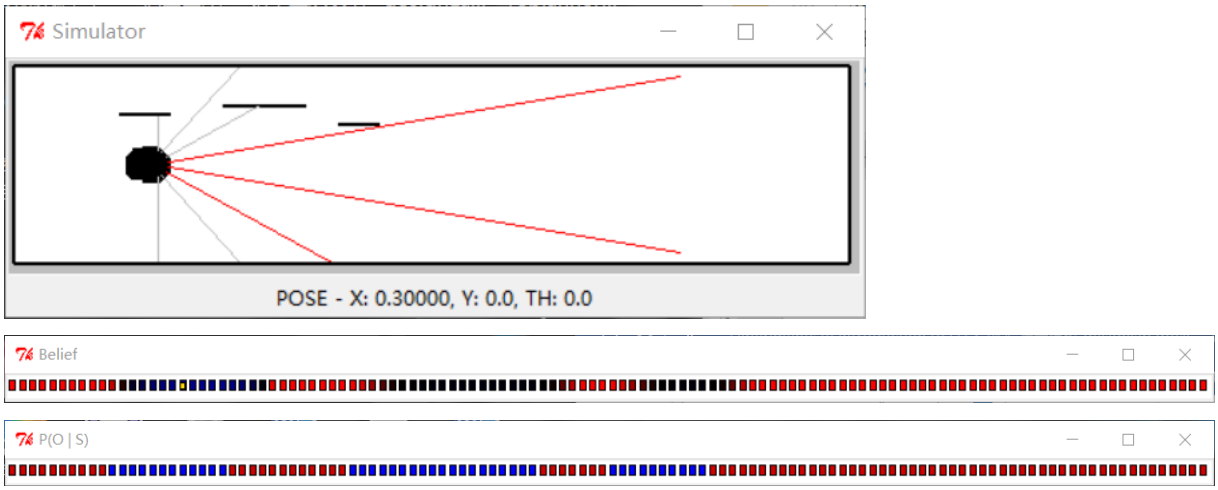
Step0:



Step1:



Step2:



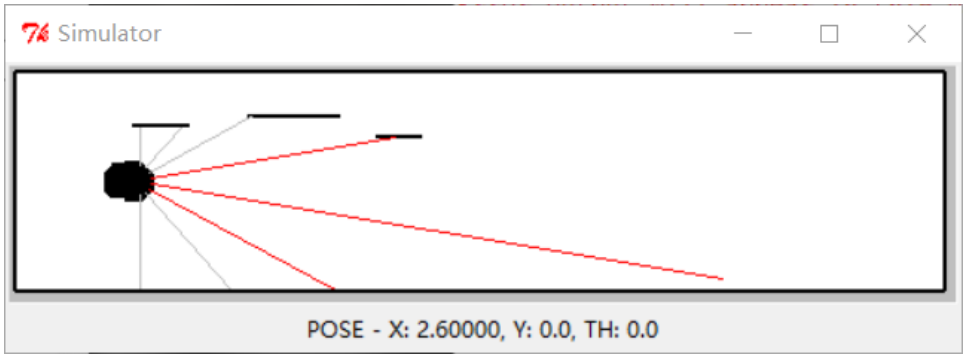
.....

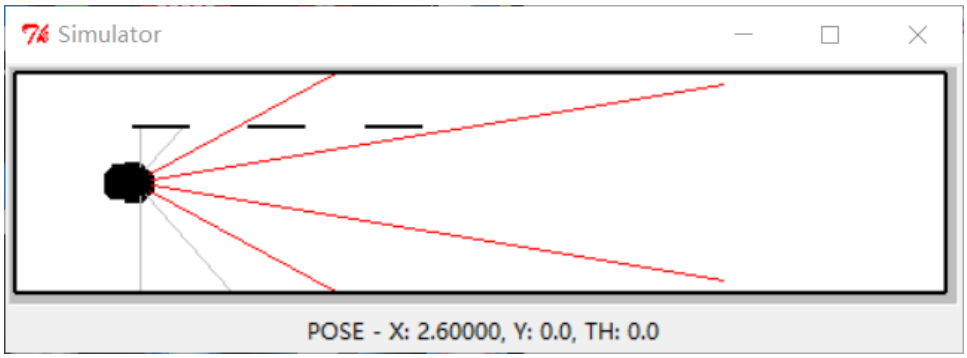
Checkoff 3.

Wk.13.2.3

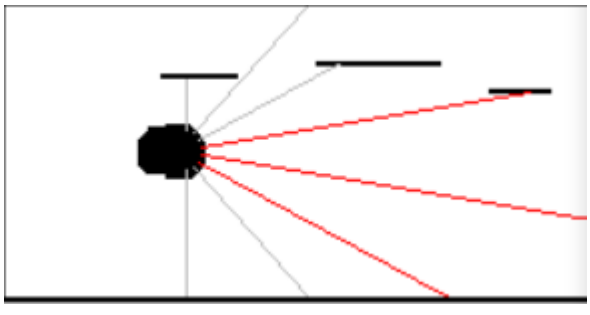
Step10

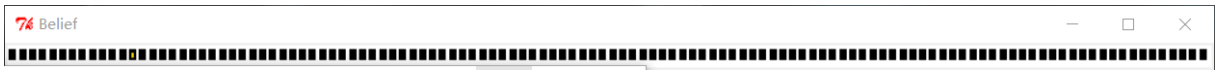
Compare the results of the two worlds. The first is in oneDdiff, the second is in oneDreal.

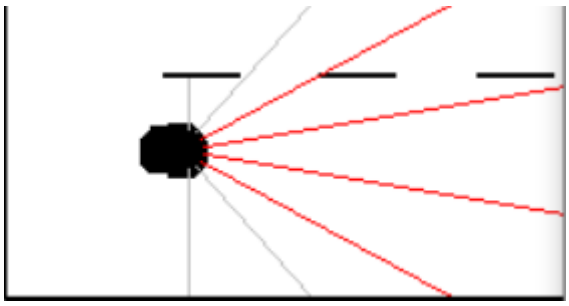


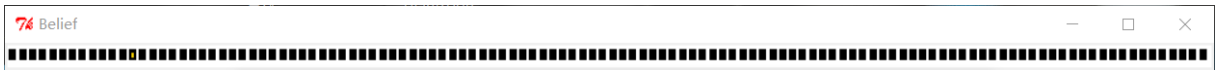


Run1:



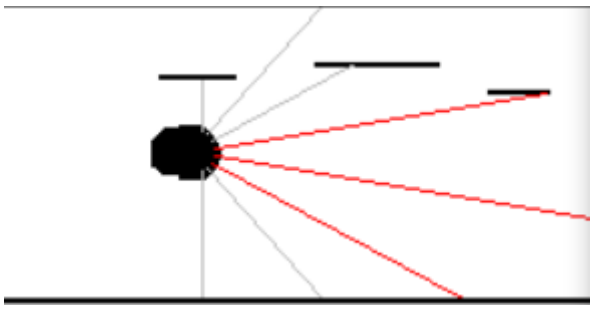


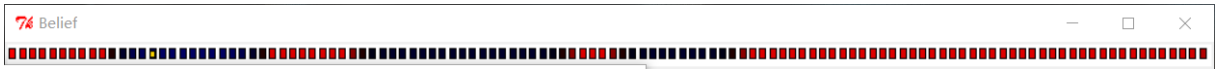


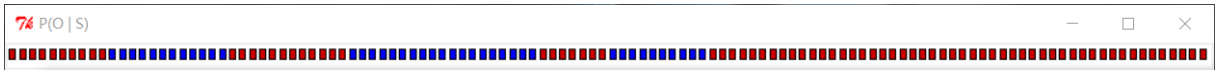


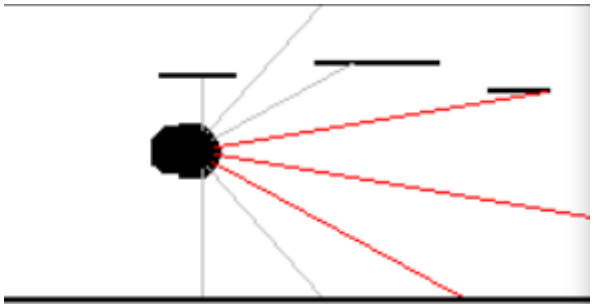
There is no sensor state input at the beginning, so the belief states are evenly distributed.

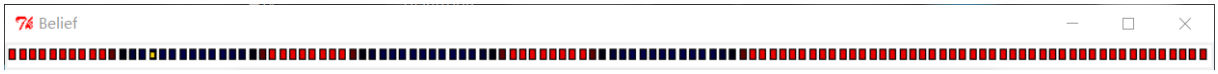
Run2:

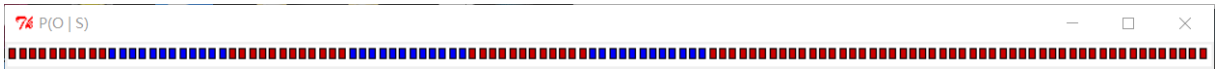




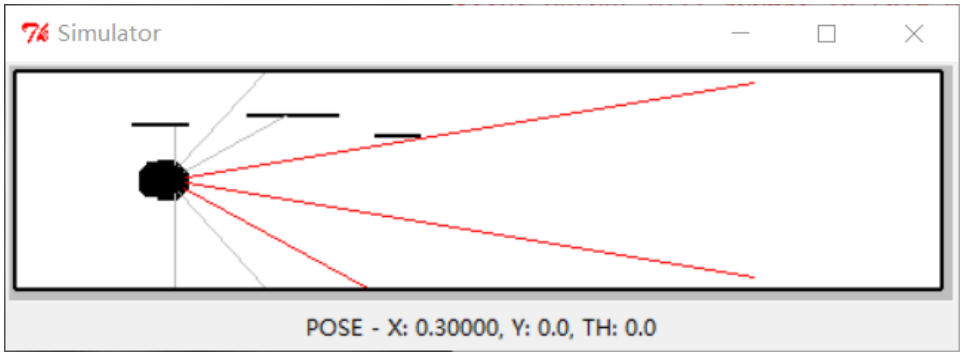


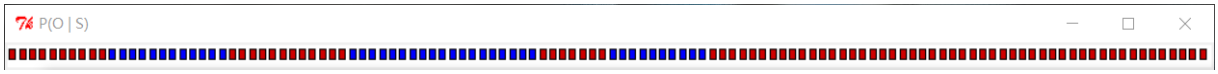


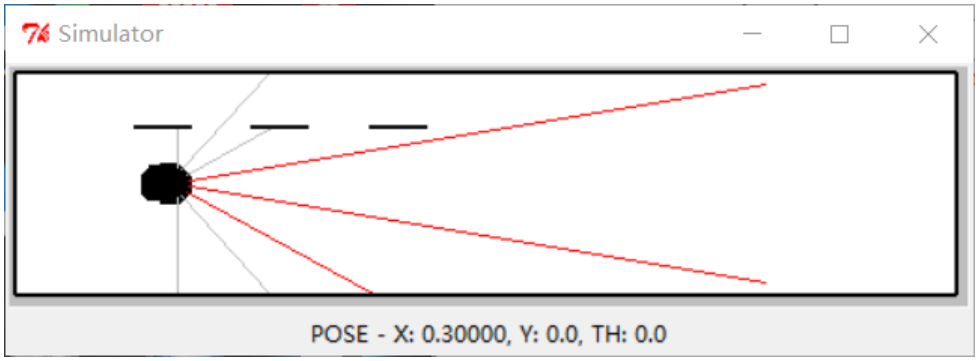


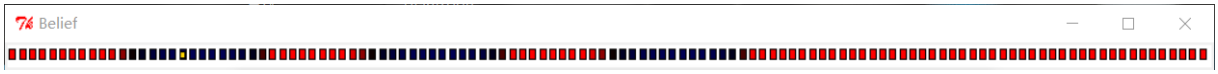


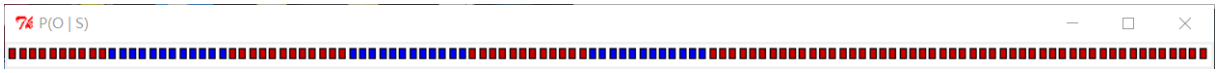
Run3:





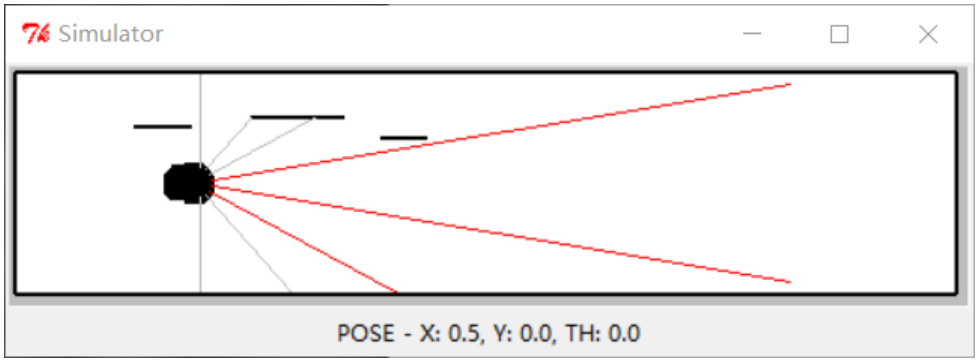


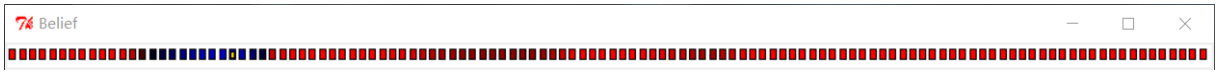


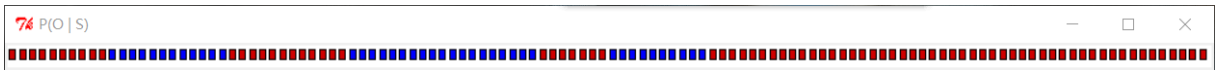


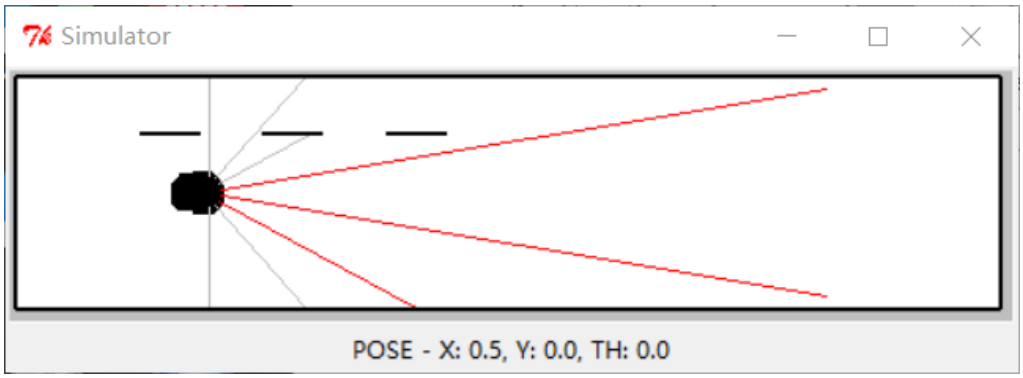
The belief state will shift forward gradually along with the robot's progress until the input of observation value changes in run5.

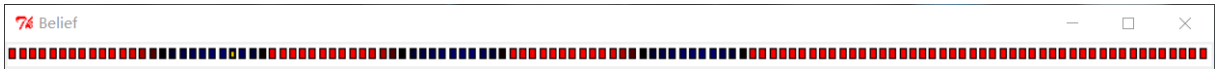
Run5:

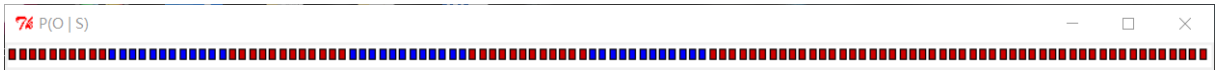






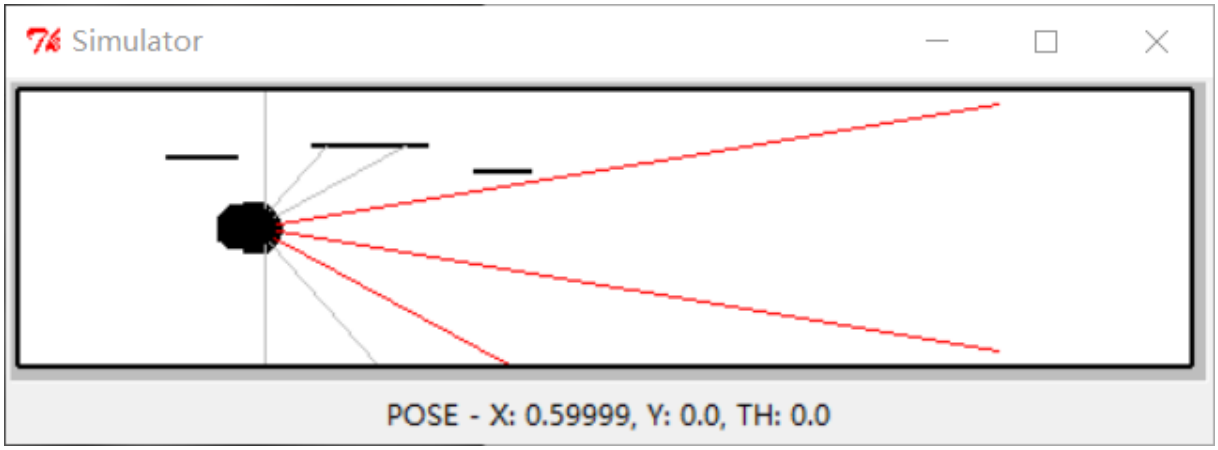


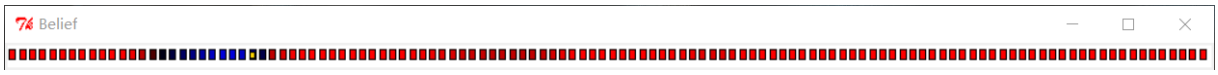


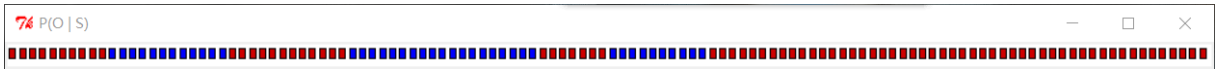


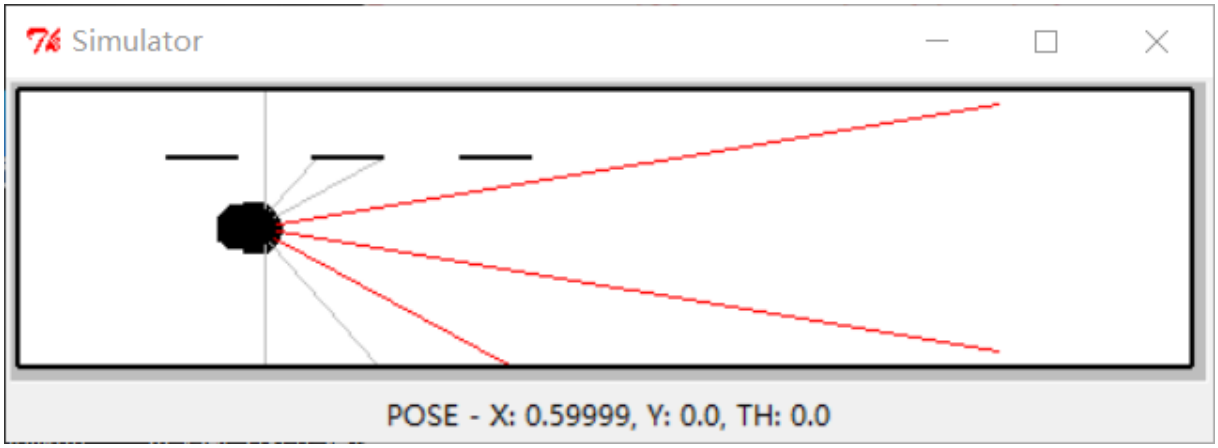
The belief state changes obviously in run6.

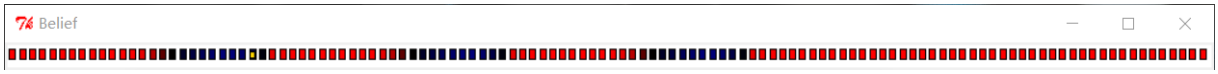
Run6:

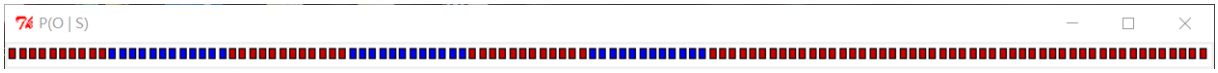






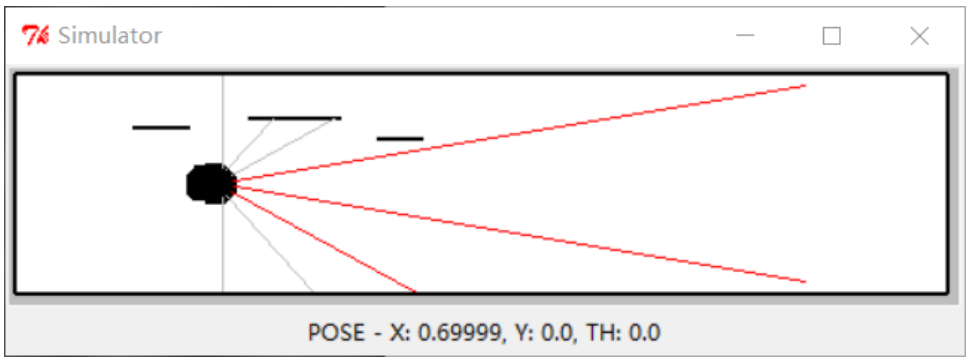


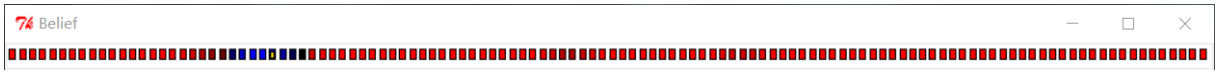


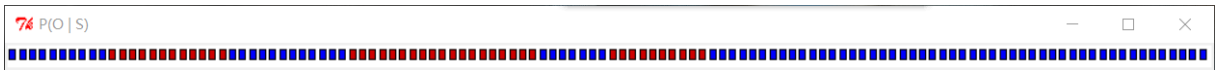


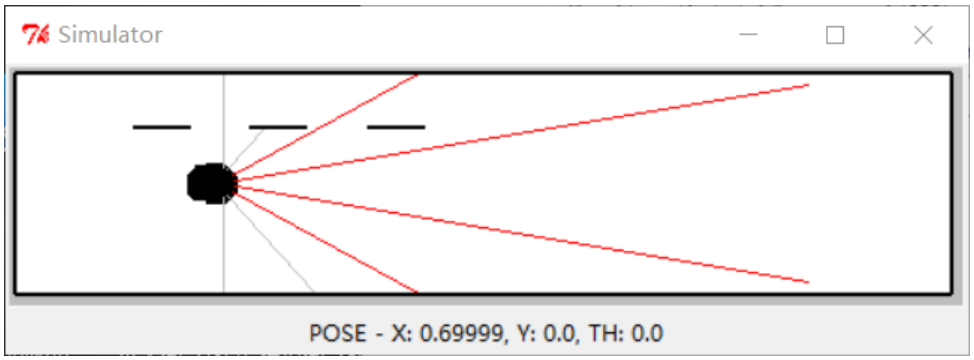
The observed probability distribution changes in run7.

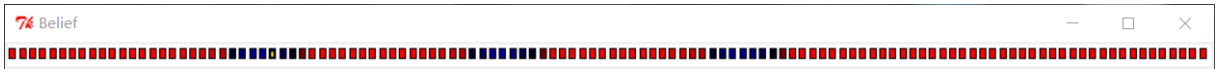
Run7:

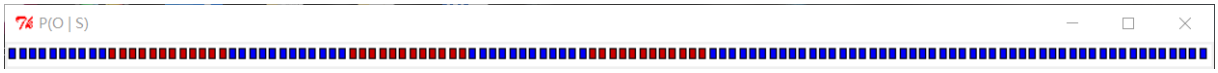






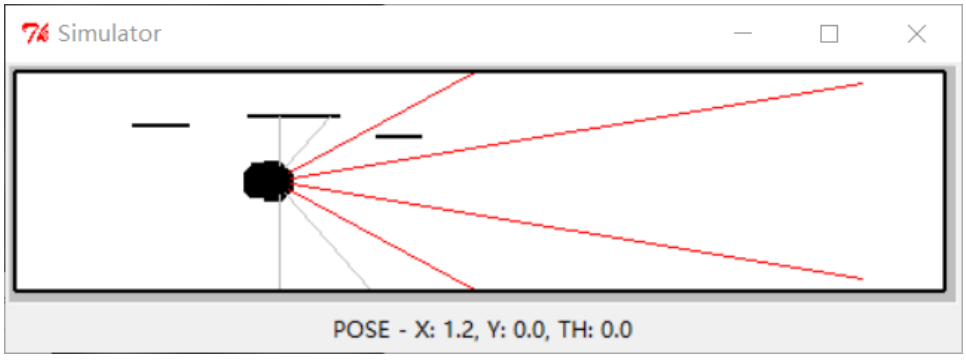


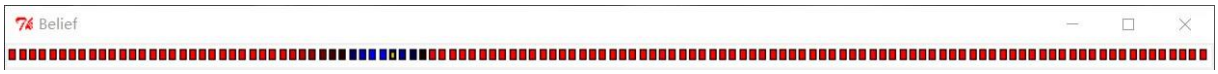


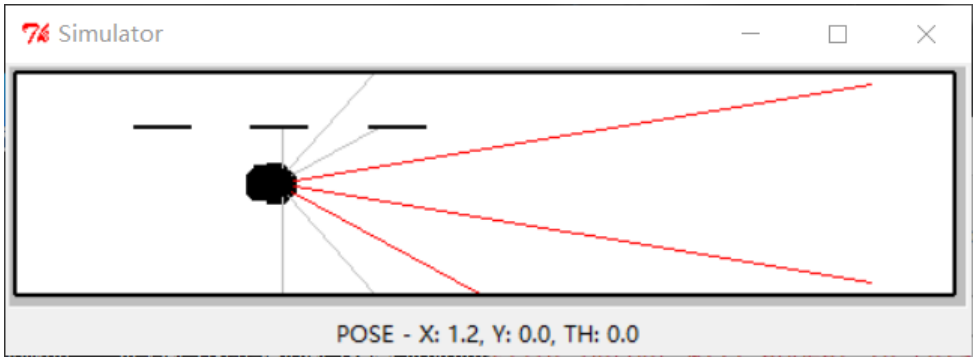


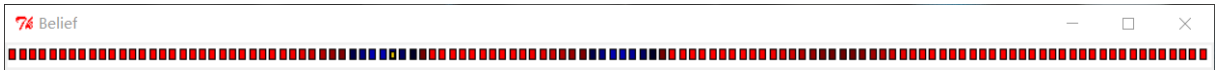
The observed probability distribution is constant util run11.

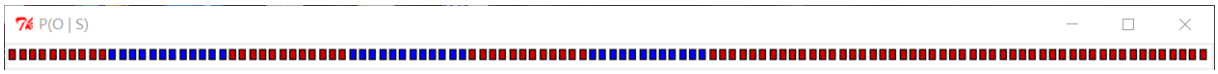
Run 11:





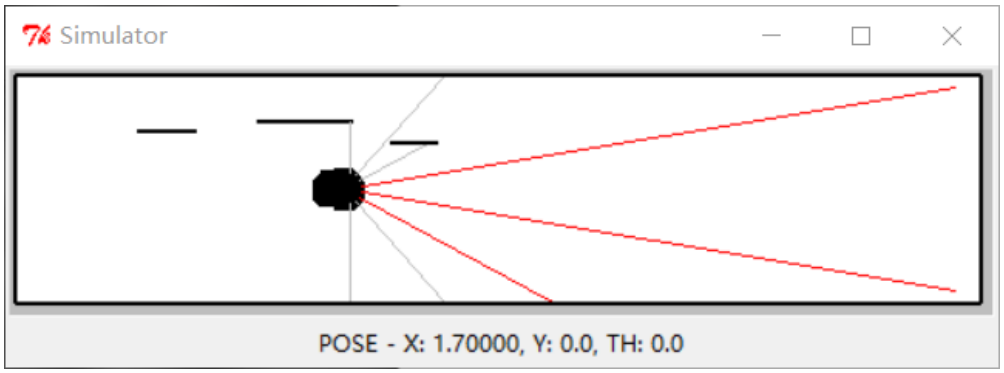


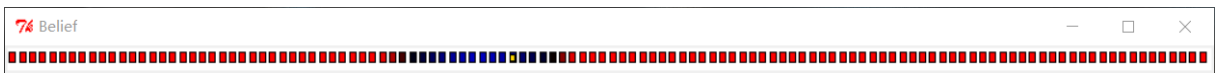


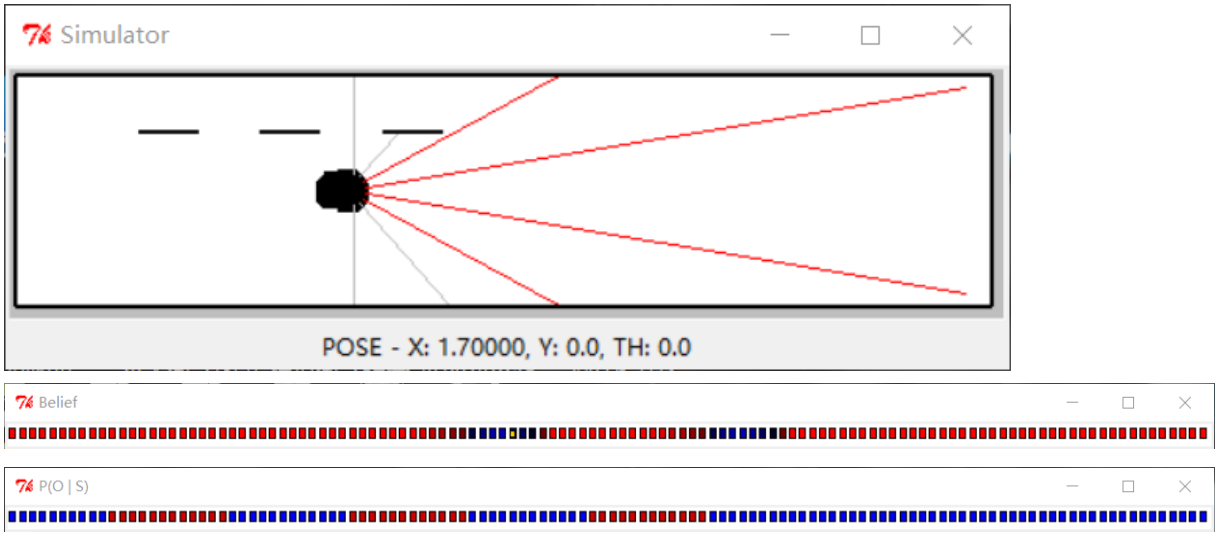


In run17, the observed probability distribution in second changes.

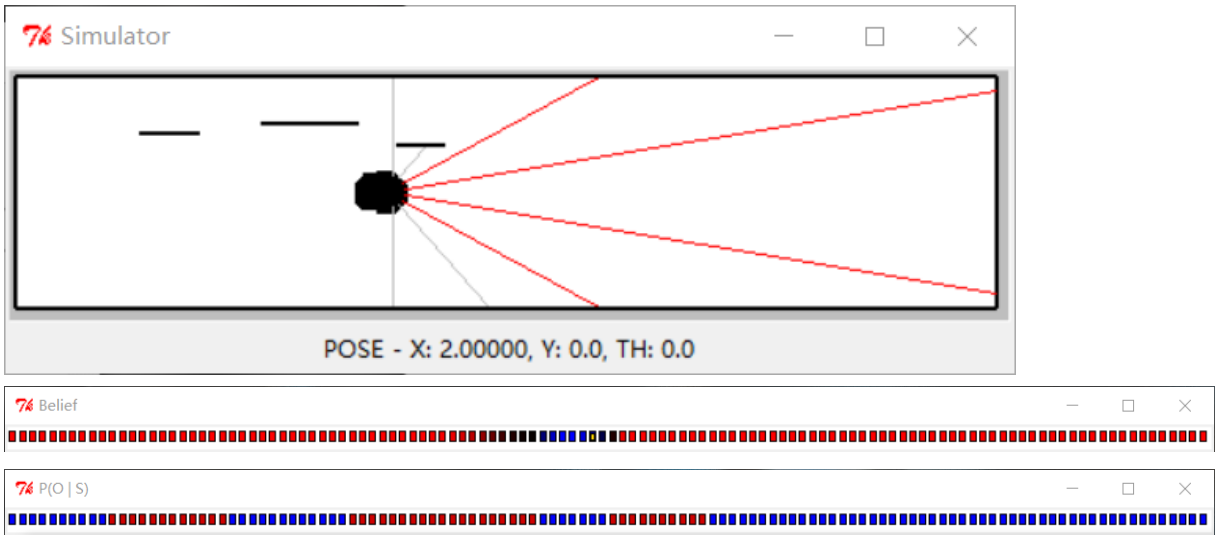
Run17:

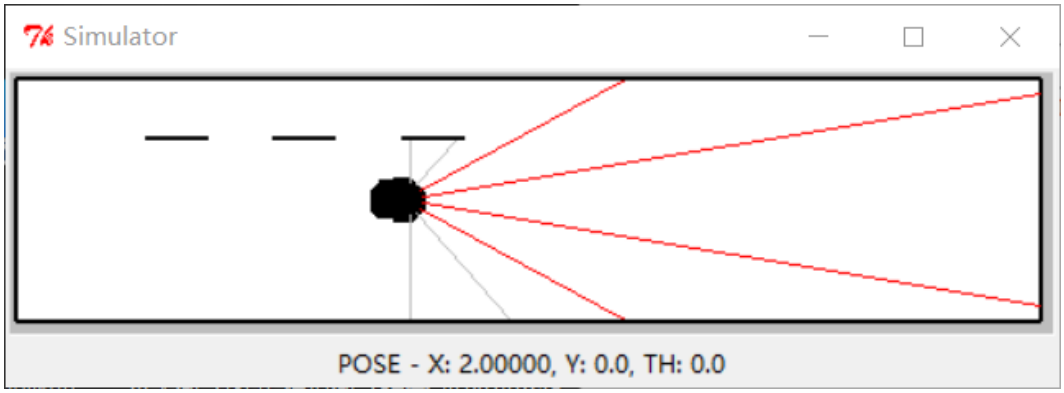


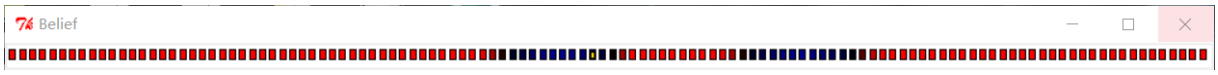


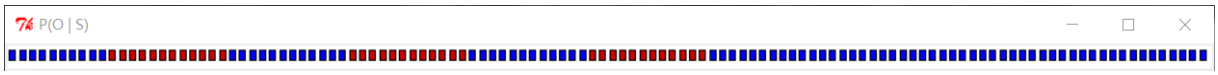


In run20, the observed probability distribution changes again.

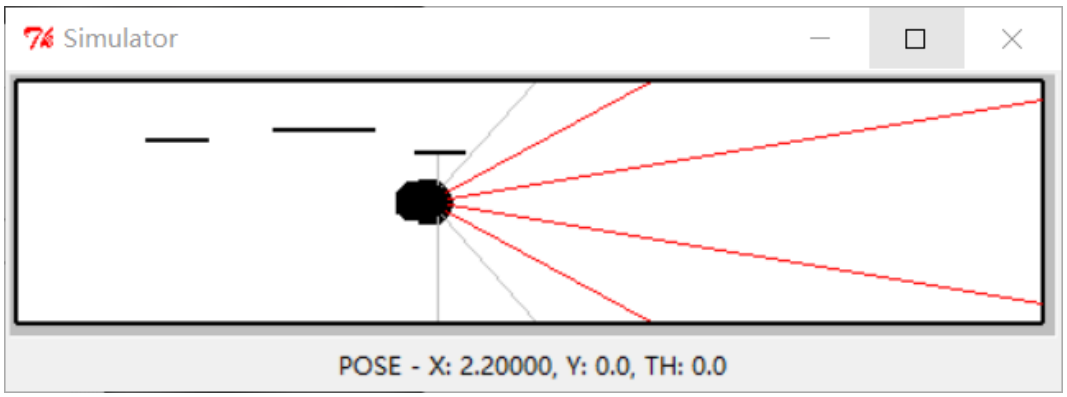


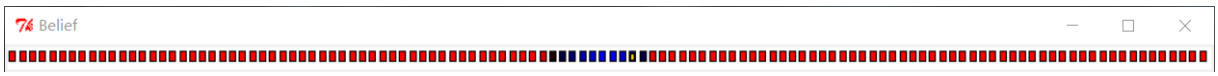


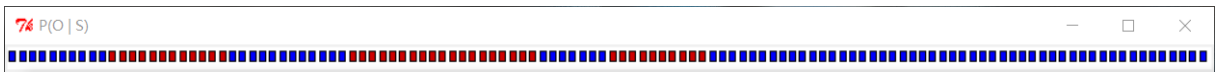


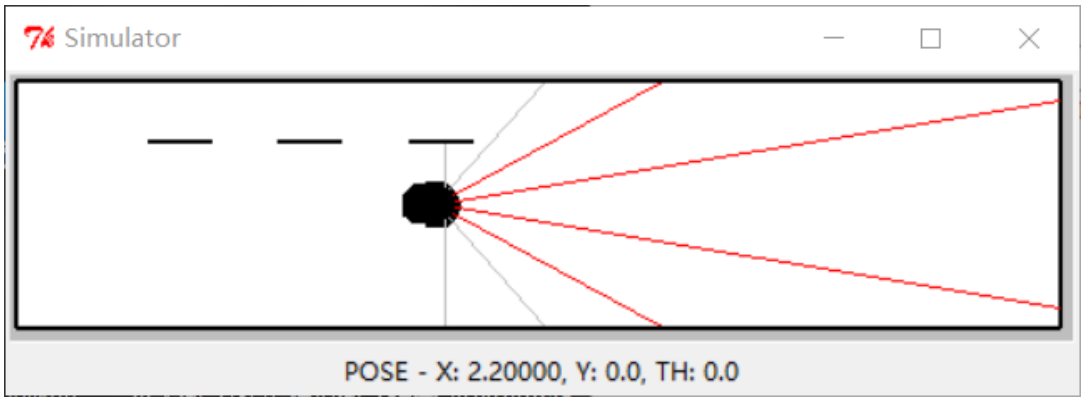


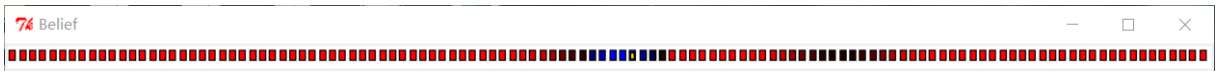
In run22, the second observed probability distribution in second changes.

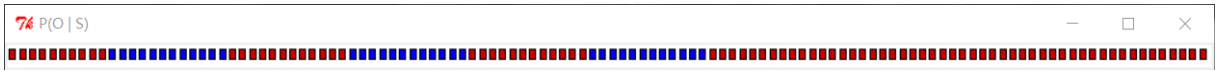




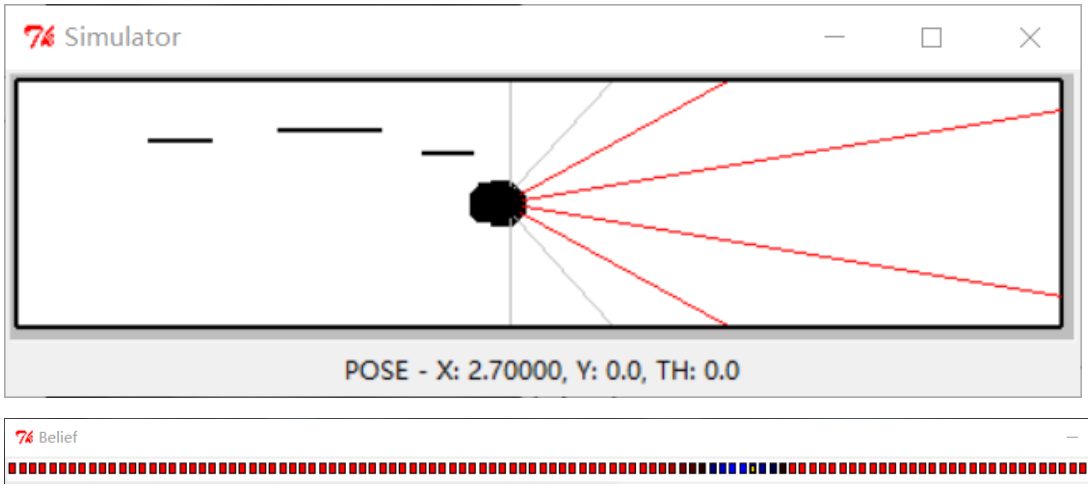


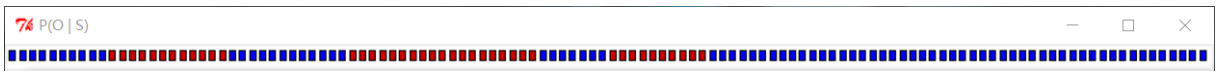


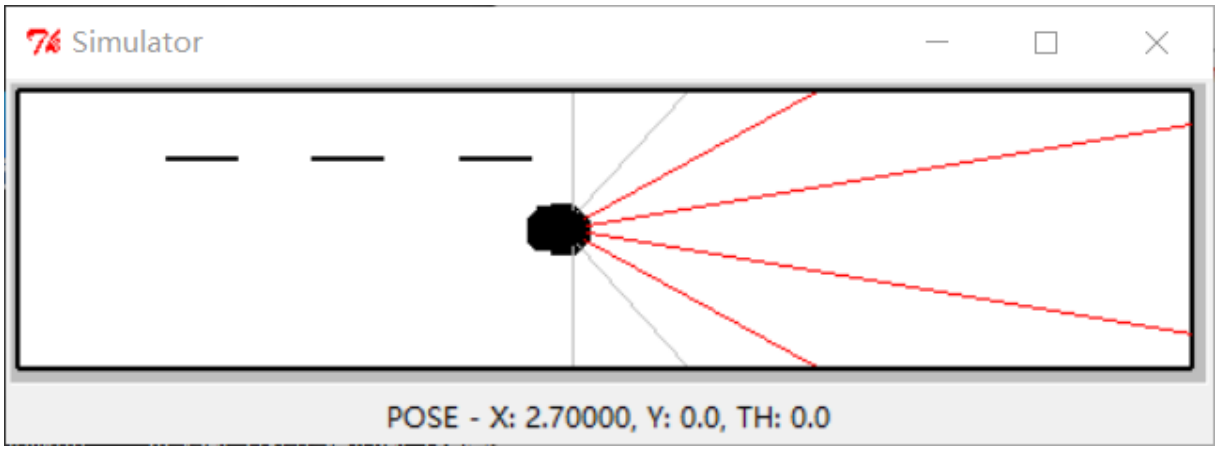


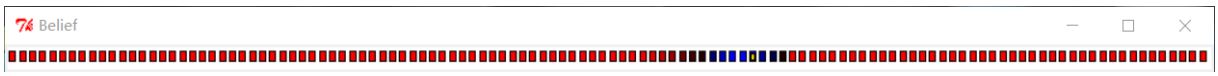


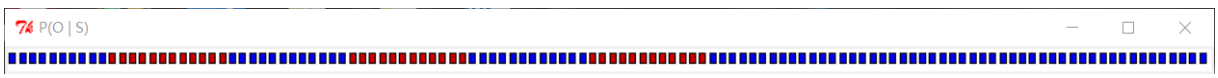
In run27, the observed probability distribution changes again.











And then it will continue until the end. Compare their output in distance of sonar reading [0].

The first world from run2:	*The second world from run2*
(7, 2)	(7, 2)
(7, 2)	(7, 2)
(7, 2)	(7, 2)
(7, 2)	(7, 2)
(7, 2)	(7, 2)
(17, 2)	(17, 2)
(17, 2)	(17, 2)
(17, 2)	(17, 2)
(17, 2)	(17, 2)
(17, 2)	(17, 2)
(9, 2)	(7, 2)
(9, 2)	(7, 2)
(9, 2)	(7, 2)
(9, 2)	(7, 2)
(9, 2)	(7, 2)
(9, 2)	(17, 2)
(9, 2)	(17, 2)
(9, 2)	(17, 2)
(17, 2)	(17, 2)
(17, 2)	(17, 2)
(17, 2)	(7, 2)
(5, 2)	(7, 2)
(5, 2)	(7, 2)
(5, 2)	(7, 2)
(5, 2)	(7, 2)
(17, 2)	(17, 2)

So, in the second world, the probability distribution repeats itself in ten-step

increments. In the first world, a new probability distribution appears every time the robot reaches a new wall. In the second world, at every distance from the wall, the state of confidence tended to slip with motion, and it was impossible to accurately judge the robot's position. That's because in this world, the robot's distance from the wall is repeated. But in the first world, we can see that the robot's judgment of the confidence state is extremely accurate as the number of inputs increases, within the margin of error.

Step11

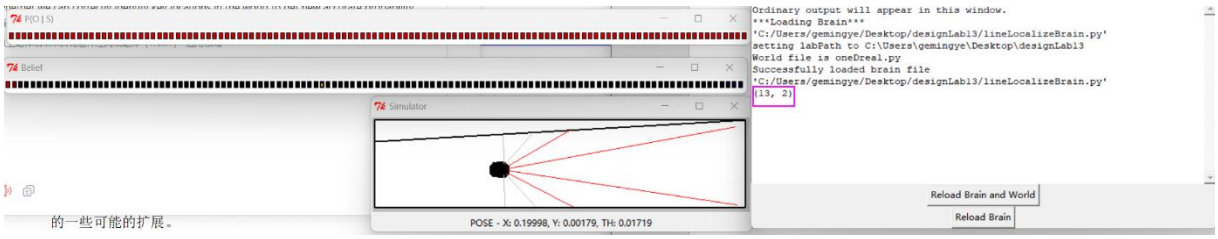
Let's use the oneDreal world's brain to run in the oneDslope world.

Our observational model is extremely dependent on the selection of the world, which means whether we can correctly identify key locations in the world to get new accurate probability distributions

In run1, It presents a probability distribution as in 'oneDreal.py' :

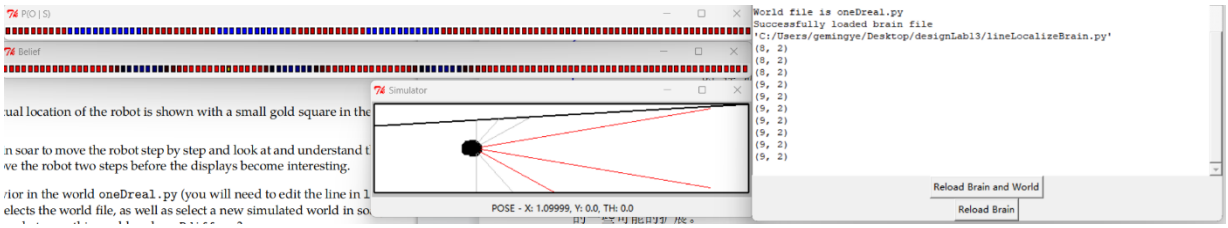
This is because the sonar reading is still in the same range as the first reading of 'oneDreal.py'

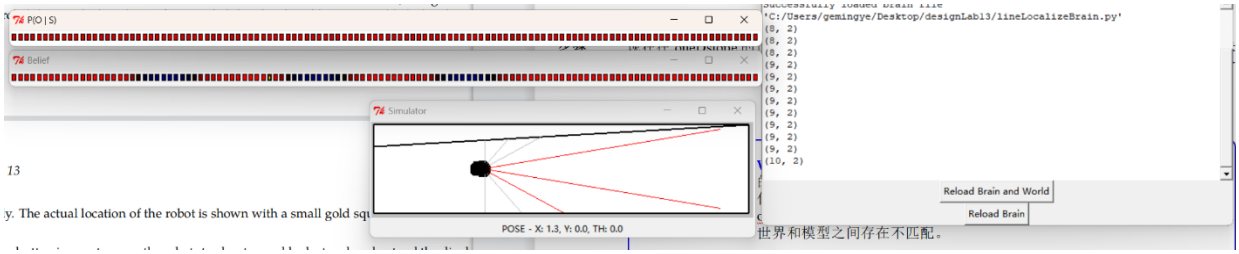
(If we artificially change the initial spacing:



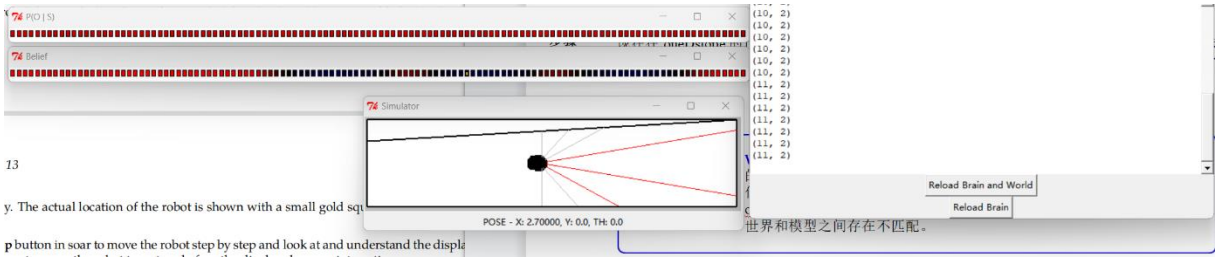
We find that both the state distribution and the belief distribution of the robot are unavailable, which can be expected)

There is a point of transition where there is no special match for the 'anomaly' in the sonar readings, and the probability model changes to an equal probability distribution:





If we keep walking, it's to be expected that we don't have any accurate probability distribution for the robot's position, which is going to be an equal probability that appears black, and the only thing we know is how much the robot has moved forward, which is the only belief distribution we can reflect:



13

y. The actual location of the robot is shown with a small gold sq

p button in soar to move the robot step by step and look at and understand the displa

世界和模型之间存在不匹配。