山东大学

崇新学堂

2025－2026 学年第一学期

实　验　报　告

课程名称：	电子信息工程导论
实验名称：	I Walk the Line

专　业　班　级	崇新学堂
学　生　姓　名	高子轩，钱竹玉，吕思洁，徐亚骐
实　验　时　间	2025 年 12 月 11 日

Overview

In this lab, we will implement, and ultimately test in soar, a robot localizer, as outlined in tutor problems Wk.11.1.7 and Wk.12.2.3.

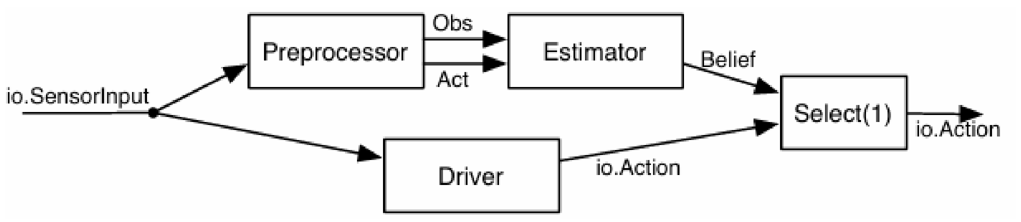


Figure 1 The architecture of the system

Preprocessor

The PreProcess class in the lineLocalizeSkeleton.py file is responsible for converting the sonar and odometer data of the robot into the observation and action information required by the state estimator.

Step 1. Be sure you understand the implementation of the preprocessor machine.

Code logic:

The initial state's pose and sonar are both "None". Obtain the pose and sonar readings from the sonar. If it is the initial state, then return the newly observed state. From the second time onwards, you can directly start discretization to obtain data

Here is our code:

```
    def getNextValues(self, state, inp):
        (lastUpdatePose, lastUpdateSonar) = 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))
```

Check Yourself 1.

What is the internal state of the machine?

Our result:

The internal state is a tuple. The first element is the position (Pose) at the time of the last update, and the second element is the Sonar reading from the last update.

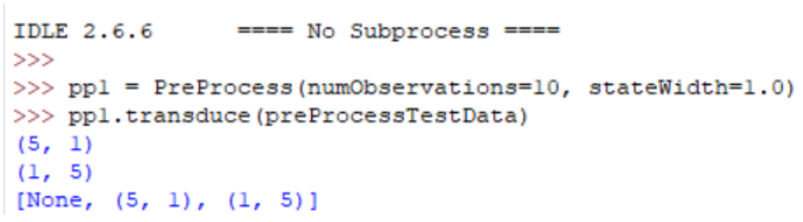
What is the starting state?

Our result:

(None, None)

Step2. test the preprocessor on the example from tutor problem Wk.12.2.3

Our result:



```
IDLE 2.6.6      ==== No Subprocess ====
>>> 
>>> pp1 = PreProcess(numObservations=10, stateWidth=1.0)
>>> pp1.transduce(preProcessTestData)
(5, 1)
(1, 5)
[None, (5, 1), (1, 5)]
```

Figure 2 the result of wk.12.2.3

There is no problem with this program

State Estimator

We need to construct a ssm.StochasticSM containing the transition, observation, and initial belief models to enable the StateEstimator to track the robot's discretized x-coordinate

Step 3. Define startDistribution, which should be uniform over all possible discrete robot locations.

Code logic:

The requirement of even distribution can be fulfilled by using dist.squareDist

Here is our code:

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

observation model.

Check Yourself 2. Sketch out your plan for the

Our Plan:

The observation model is composed of three parts: the main reading, the maximum error, and the background noise. They can be simulated using dist.triangleDist, dist.DeltaDist, and dist.squareDist respectively. Finally, dist.MixtureDist is used according to 0.9. Just mix in a ratio of 0.02 to 0.08

Step 4. Implement the observation model and test the data, and answer related questions

Here is our code:

```
def observationModel(ix):
       idealObservation = ideal[ix]

       triangle_dist = dist.triangleDist(idealObservation, 4)
       uniform_dist = dist.squareDist(0, numObservations)
       max_dist = dist.DeltaDist(numObservations - 1)

       tail_dist = dist.MixtureDist(uniform_dist, max_dist, 0.8)
       combined_dist = dist.MixtureDist(triangle_dist, tail_dist, 0.9)

       return combined_dist
```

Based on the test of testIdealReadings, we have the following picture:

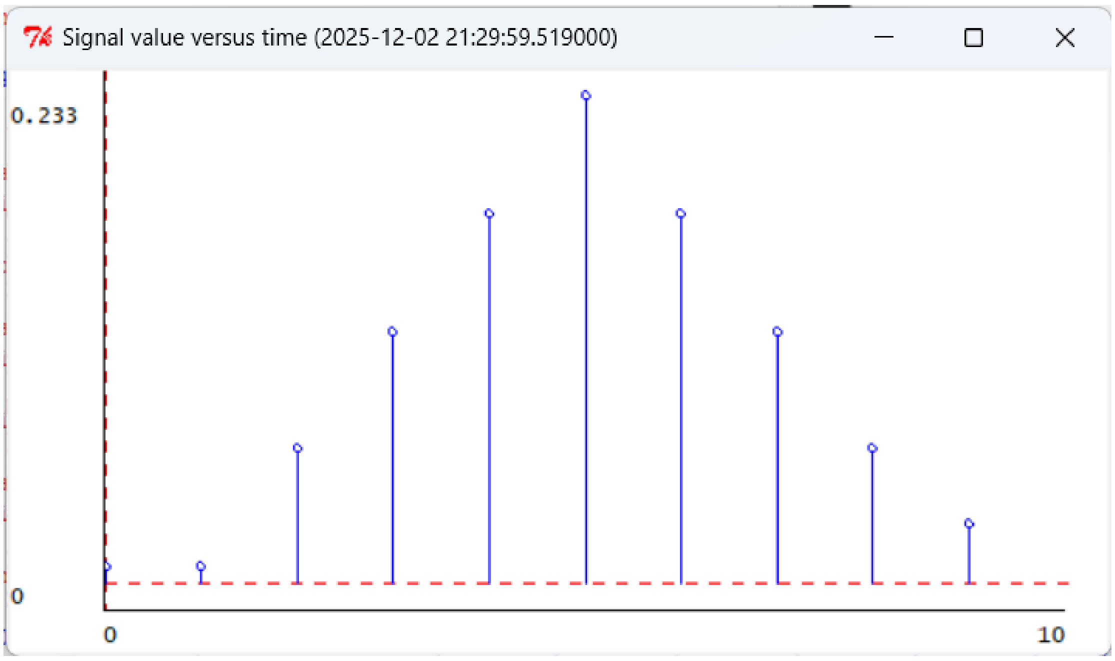


Figure 3 The probability distribution of 10 observation bins

The indexes of the four items with the highest

probabilities are 3, 4, 5, 6, and 7 respectively

The number 7 in observationModel(7) indicates that the discrete position index is 7

Step 5. Run the observation data containing 100 pieces of data

Here is our plot:

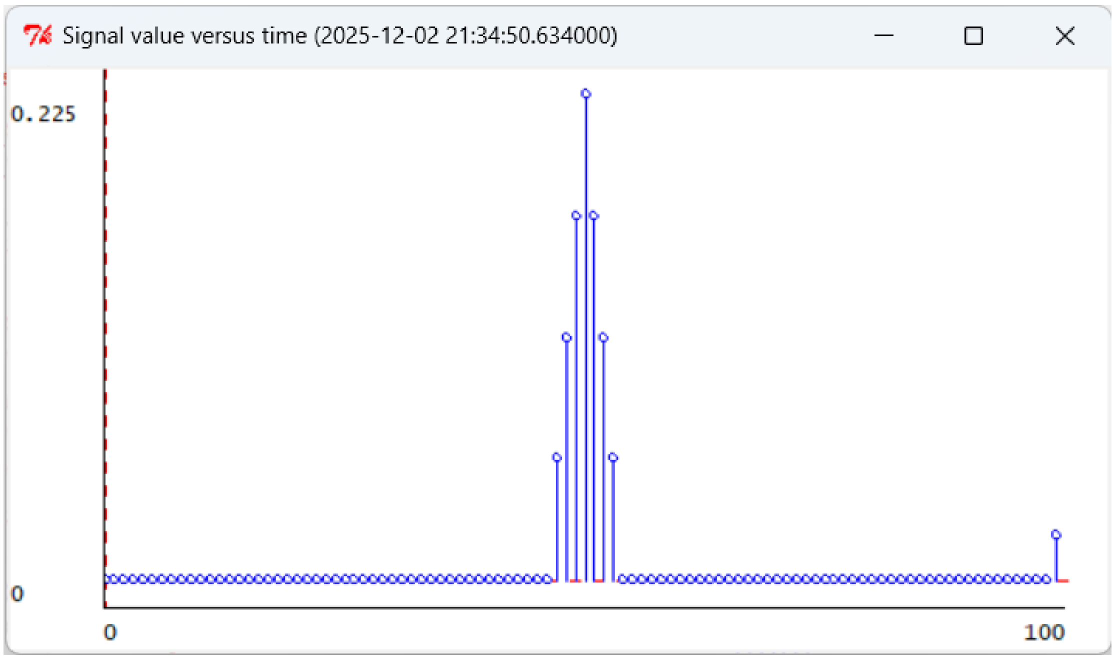


Figure 4 The probability distribution of 100 observation bins

Checkoff 1: Explain the content of the picture and the reason for it

Although these two pictures have different resolutions, their physical shapes are consistent. The tall part in the middle of the triangle indicates the measurement error when the sensor is working properly. The maximum peak has a small protrusion on the far right, which represents the probability of sonar failure.

The background floor noise has a very low non-zero probability layer at the entire bottom, representing random noise.

Transition model

The transition model should return a distribution over the next discrete state by applying a triangle distribution to the calculated ideal destination to account for discretization errors in the reported action.

Check Yourself 3. Sketch out your plan for the transition model.

Our Plan:

Pass in two parameters, action and oldPos, which can be in the form of a closure, with the outer layer being action and the inner layer being oldPos. Calculate the final position as action plus oldPos. Due to the existence of errors, we can construct a triangular distribution at this point.

Step 6. Implement the transition model and test it to be sure it’s reasonable.

Here is our code:

```
def transitionModel(a):
    def transition(oldPos):
        idealPos = oldPos + a

        if idealPos > numStates - 1:
            idealPos = numStates - 1
        elif idealPos < 0:
            idealPos = 0

        width = 1
        return dist.triangleDist(idealPos, width, 0, numStates)
```

Based on the test of testIdealReadings, we have the following picture:

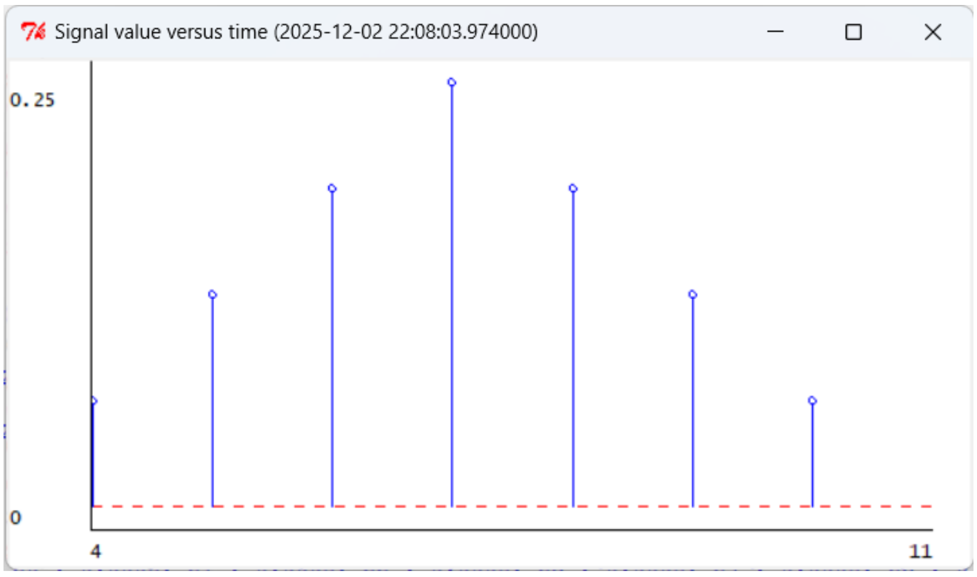


Figure 5

Here, the 5 represents OldPos and the 2 represents action

Step 7. Combined preprocessing and estimation

Code logic:

Firstly, we use a preprocessor to convert continuous sonar readings and odometry poses into discrete observation-action pairs. Then, the state estimator updates the belief distribution of the robot's position based on the noisy observation model and transition model.

Meanwhile, the driver generates control commands for moving towards the target position based on the current pose. Finally, the selector outputs the control commands of the driver, enabling the robot to achieve real-time localization and autonomous navigation in the presence of sensor noise and motion errors.

Here is our code:

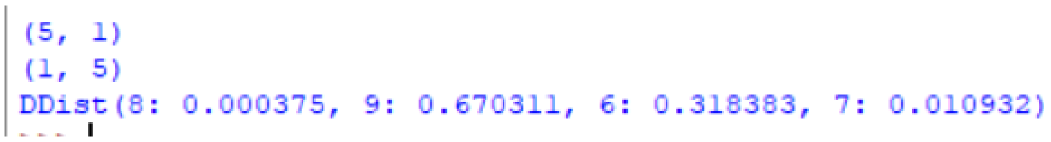
```
def makeLineLocalizer(numObservations, numStates, ideal, xMin, xMax, robotY):

    stateWidth = (xMax - xMin) / float(numStates)

    preproc = PreProcess(numObservations, stateWidth)
    navModel = makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations)
    estimator = seGraphics.StateEstimator(navModel)
    driver = move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0), maxVel=0.5)
    est_path = sm.Cascade(preproc, estimator)
    combined = sm.Parallel(est_path, driver)

    return sm.Cascade(combined, sm.Select(1))
        return transition
```

The running result is as follows:



```
(5, 1)
(1, 5)
DDist(8: 0.000375, 9: 0.670311, 6: 0.318383, 7: 0.010932)
```

Figure 6 The running result

Checkoff 2. Check the results and explain whether they are consistent with those in wk12.2.3

The results of the noise model are basically consistent with the ideal model in core features, but there are reasonable differences in the details of probability distribution.

Similarities

• The overall pattern of the probability distribution is consistent, both output uniform distribution, and the main probability mass is concentrated on the same subset of states (e.g., both models are concentrated on state 6 and state 9);

・The most likely state identified is the same, and the relative size relationship of the probability values is completely consistent (e.g., the most likely state of both models is state 9)；

・The logic of Bayesian update is consistent, both achieving a reasonable evolution from the initial uniform distribution to a gradually concentrated distribution.

Differences

・There are differences in the accuracy and concentration of the probability distribution. The noise model has a small probability in states that should not occur (e.g., state 7 and state 8), indicating uncertainty;

・There are differences in the smoothness of the probability distribution. The ideal model has a sharp probability distribution, while the noise model has a smooth distribution.

Putting it All Together

Step 9. Start soar for oneDdiff.py

The interface visualizes the localization process in two windows—one for observation likelihoods and one for the belief state—using a color scale where blue indicates high probability and red indicates low probability, with the robot's actual position marked in gold. We advance the robot at least two steps to meaningfully observe how these distributions evolve.

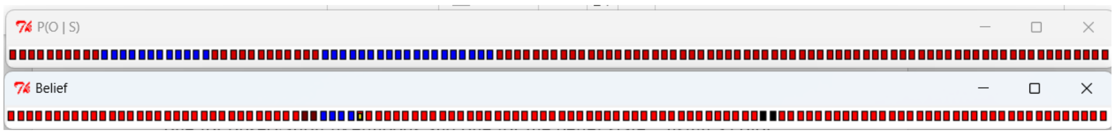
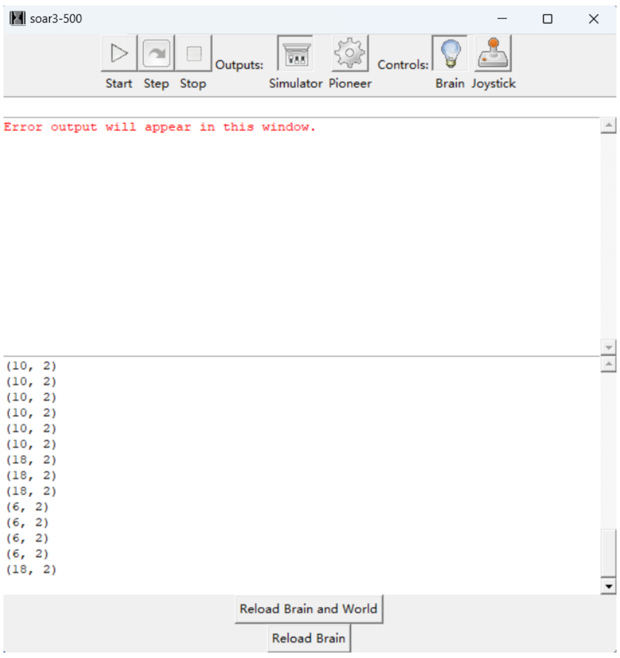


Figure 7 The plot for oneDdiff



```
Error output will appear in this window.
```

```
(10, 2)
(10, 2)
(10, 2)
(10, 2)
(10, 2)
(10, 2)
(18, 2)
(18, 2)
(18, 2)
(6, 2)
(6, 2)
(6, 2)
(6, 2)
(18, 2)
```

Figure 8 The process of oneDdiff

Step 10. Start soar for oneDreal.py and differentiate the two plots.

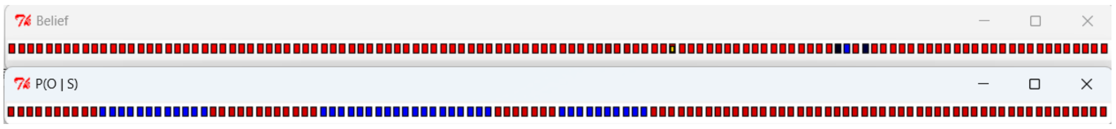
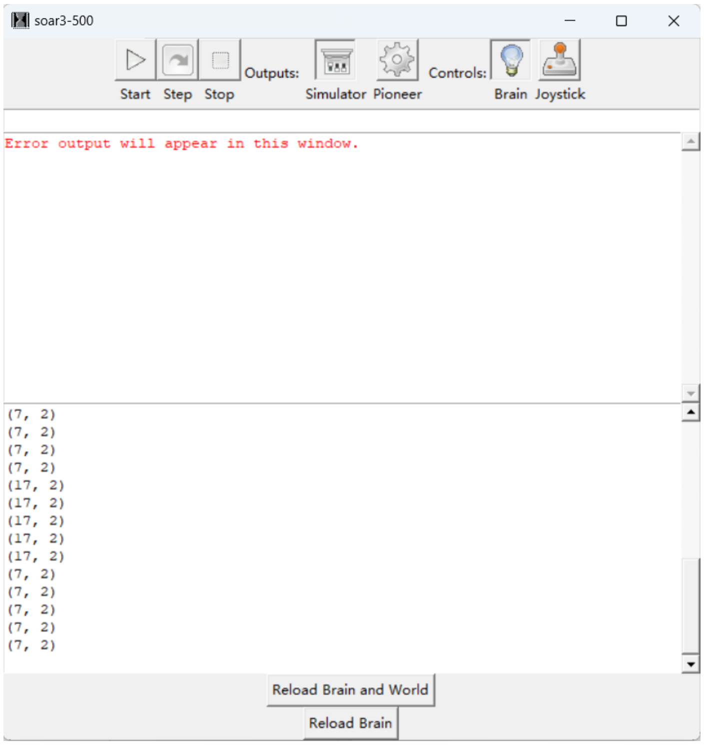


Figure 9 The plot for oneDreal



```
Error output will appear in this window.
```

```
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(17, 2)
(7, 2)
(7, 2)
(7, 2)
(7, 2)
(7, 2)
```

Figure 10 The process of oneDreal

In oneDdiff.py, you will find that the positioning is usually more accurate and stable.

In oneDreal.py, positioning might be a bit difficult, as the blue areas of belief distribution tend to spread out more.

Step 11. Start soar for oneDslope.py

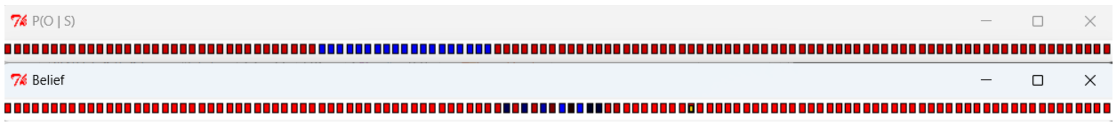


Figure 11 The plot for oneDslope

Here is our answer:

In this way, it will be found that the distance between the true position of the belief distribution and the blue position is quite different. The distance seen by the robot does not match what it expected in the map, causing it to rule out the correct true position

Checkoff 3.

Explain the meanings of the colors in the display windows

Here is our answer:

The P(O/S) window indicates the probability of seeing the current sonar reading at each position on this map.

The Belief window is the final belief distribution of the robot regarding its current position.

Explain why the behavior differs between oneDreal and oneDdiff.

Here is our answer：Because oneDdiff represents a relatively ideal or standard corridor environment, with distinct wall features and sensor noise within the expected range

The oneDreal map feature shows that the distance between the walls in two places is the same, which makes the sonar readings more prone to ambiguity

Explain what happens when there is a mismatch between the world and the model.

Here is our answer:

The observation model of the robot assumes that the wall is straight, but in the real world, the wall is slanted

The distance the robot saw did not match what it expected on its map, causing it to rule out the correct real position

Appendix: The Description of AI Usage in the Report

In this section, we primarily utilize AI to comprehend key concepts and perform English translation.