山东大学

崇新学堂

2024－2025 学年第一学期

实 验 报 告

课程名称：EECS

实验名称：DL14

专 业 班 级　崇新学堂

学 生 姓 名　于静明，程侃，张子诺

实 验 时 间　2024/12/22

Step1

Check Yourself 1

The variable testData indeed contains two elements. Each element is an instance of the SensorInput class, where the first part of each instance represents the distances measured by eight sonar sensors, and the second part represents the coordinates and angular posture of the vehicle (or robot).
Processing the Data:
For the first element of testData:
The sonar readings are all 0.2. This indicates that the distances measured by the eight sonar sensors are all the same, which suggests that the vehicle is likely facing a flat surface or is at a consistent distance from any obstacles.
The pose is given as util.Pose(1.0, 2.0, 0.0). This means the vehicle is located at coordinates (1.0, 2.0) with an orientation angle of 0.0 radians.
For the second element of testData:
The sonar readings are all 0.4. This indicates that the distances measured by the eight sonar sensors are greater than in the first element, suggesting that the vehicle is either further away from obstacles or facing a different environment.
The pose is given as util.Pose(4.0, 2.0, -math.pi). This means the vehicle is located at coordinates (4.0, 2.0) with an orientation angle of -π radians (i.e., facing in the opposite direction compared to the first pose).
Marking the Map:
Black Squares: Positions with obstacles are marked as black. Based on the sonar readings, positions where the sonar readings are relatively small (indicating proximity to an obstacle) should be marked as black. For example, in the first element of testData, if we assume a threshold distance below which an obstacle is considered present, say 0.3, then positions corresponding to the first element's sonar readings (0.2) would be marked as black.
Gray Squares: Positions slightly away from obstacles are marked as gray. Using the same threshold concept, positions where the sonar readings are greater than the threshold for an obstacle but still relatively close could be marked as gray. For example, if the threshold is 0.3, then positions corresponding to the second element's sonar readings (0.4) could be marked as gray.
Unlabeled Positions: Positions that have not been measured should remain unlabeled. This would be relevant if there are areas in the map that are outside the range of the sonar sensors for both elements of testData.

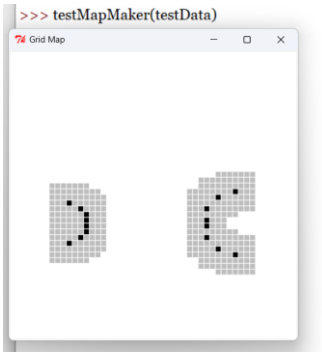
Step2

Implement the MapMaker class

```
class MapMaker(sm.SM):
  def __init__(self, xMin, xMax, yMin, yMax, gridSquareSize):
    self.startState = dynamicGridMap.DynamicGridMap(xMin, xMax, yMin, yMax, gridSquareSize)
  def getNextValues(self, state, inp):
    for i in range(8):
      if inp.sonars[i]<sonarDist.sonarMax:
        p=sonarDist.sonarHit(inp.sonars[i],sonarDist.sonarPoses[i],inp.odometry)
        g=state.pointToIndices(p)
        state.setCell(g)
    return state,state
```

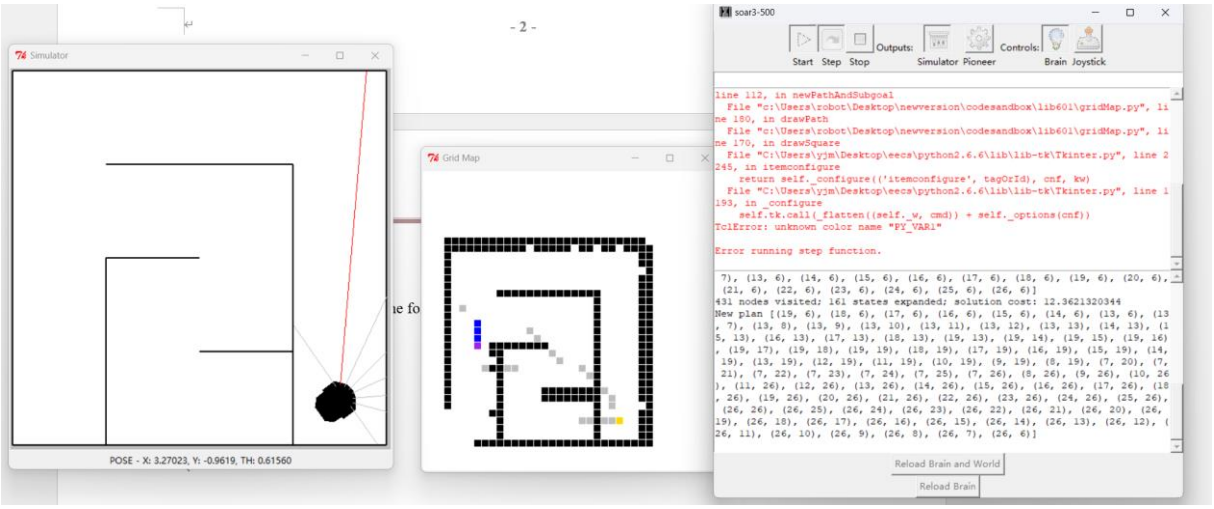
Step3

Result



Step4

The generated map:



Checkoff 1

The state machine "ReplannerWithDynamicMap" is initialized with a fixed goalPoint parameter, which is represented by a util.Point. This goalPoint specifies the desired location for the robot within the world.
During the operation of the state machine, it receives inputs from two primary sources:
Sensor Data: This data indicates the current location of the robot. It provides real - time information about where the robot is at any given moment.
DynamicGridMap: This map is generated by the MapMaker. It represents the environment that the robot is navigating through, taking into account any changes or obstacles that may be present.
In the initial step of the operation:
The replanner generates a new plan. This plan is formulated based on the goalPoint, the current location of the robot (from the sensor data), and the current state of the environment (from the DynamicGridMap).
The new plan is then incorporated into the map. This integration ensures that the map reflects the replanner's intentions and the path the robot is expected to take.
The first subgoal is outputted. This subgoal is the center of the grid square that the robot should move to next. It serves as the immediate objective for the robot's movement.
In subsequent steps, the replanner undertakes two main actions:
It continuously monitors and updates the plan based on any changes in the sensor data (robot's position) or the DynamicGridMap (changes in the environment). This dynamic updating is crucial for ensuring that the robot's path remains optimal and collision - free.
It also re - evaluates and potentially modifies the subgoals. As the robot moves and the environment changes, the replanner needs to ensure that the subsequent subgoals are still valid and appropriate for reaching the final goalPoint.

Step5

Check Yourself 2

Due to inaccurate observation values in the presence of noise, the car may misjudge and therefore may not function properly

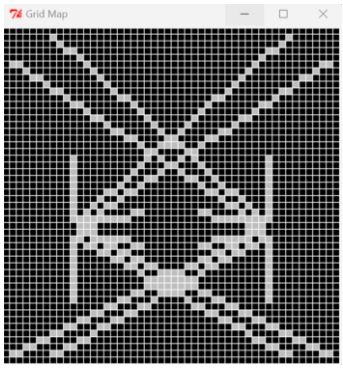
Step6

Code is as follows

```
class MapMaker(sm.SM):
  def __init__(self, xMin, xMax, yMin, yMax, gridSquareSize):
    self.startState = dynamicGridMap.DynamicGridMap(xMin, xMax, yMin, yMax, gridSquareSize)

  def getNextValues(self, state, inp):
    for i in range(8):
      #if inp.sonars[i]<sonarDist.sonarMax:
      #p=sonarDist.sonarHit(inp.sonars[i],sonarDist.sonarPoses[i],inp.odometry)
      #g=state.pointToIndices(p)
      #state.setCell(g)

      s=sonarDist.sonarHit(o,sonarDist.sonarPoses[i],inp.odometry)
      p=sonarDist.sonarHit(inp.sonars[i],sonarDist.sonarPoses[i],inp.odometry)
      g=state.pointToIndices(p)
      state.setCell(g)
      lst=util.lineIndices(state.pointToIndices(s),g)
      lst.remove(g)
      for j in lst:
        state.clearCell(j)
    return (state,state)
```

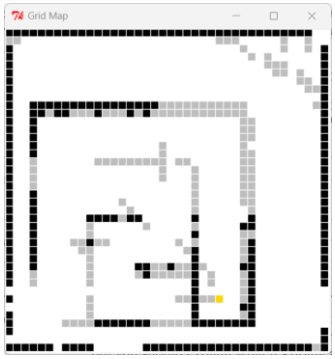
Step7

Result

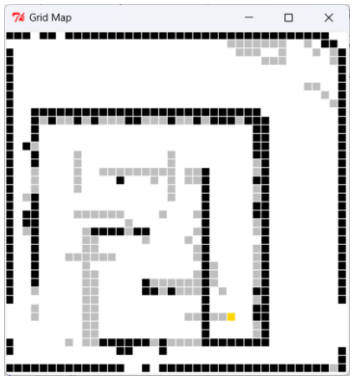


Step8

No noise



Medium noise



Step9

Code is as follows

```
def oGivenS(s):
  if s=='occupy':
    return dist.DDist({'hit':0.8,'free':0.2})
  elif s=='not':
    return dist.DDist({'hit': 0.2, 'free': 0.8})
# Transition model: P(newState | s | a)
def uGivenAS(a):
  def trans(prestate):
    if prestate=='occupy':
      return dist.DDist({'occupy':1.0,'not':0.0})
    elif prestate=='not':
      return dist.DDist({'occupy':0.0,'not':1.0})
  return trans

cellSSM = ssm.StochasticSM(dist.DDist({'occupy':0.5,'not':0.5}),uGivenAS,oGivenS)  # Your code here
```

Step10

Result:

```
>>>
[DDist(not: 0.200000, occupy: 0.800000), DDist(not: 0.058824, occupy: 0.941176), DDist(not: 0.015385, occ
upy: 0.984615), DDist(not: 0.058824, occupy: 0.941176)]
--------------------
[DDist(not: 0.800000, occupy: 0.200000), DDist(not: 0.941176, occupy: 0.058824), DDist(not: 0.984615, occ
upy: 0.015385), DDist(not: 0.941176, occupy: 0.058824)]
```

Step11

Just thinking something

Step12

Part 1:

1.'no'

2.'oh'

Part 2:

Code is as follows

```
def lotsOfClass(n,v):
    result=[]
    for i in range(n):
        result.append(MyClass(v[i]))
    return result
```

Part 2:

Code is as follows

```
def lotsOfClass(n,v):
    return util.makeVectorFill(n,lambda x:MyClass(v(x)))
```

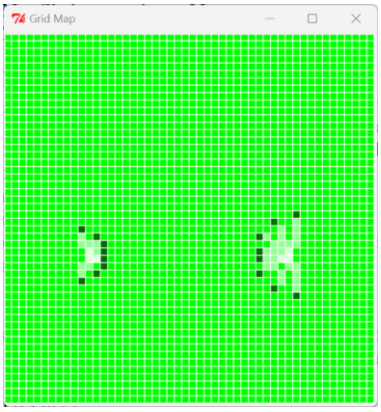
Step13

Code is as follows

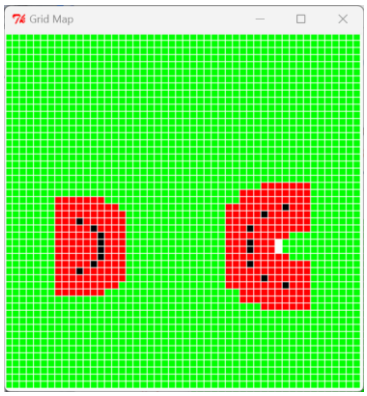
```
class BayesGridMap(dynamicGridMap.DynamicGridMap):

  def squareColor(self, (xIndex, yIndex)):
    p = self.occProb((xIndex, yIndex))
    if self.robotCanOccupy((xIndex,yIndex)):
      return colors.probToMapColor(p, colors.greenHue)
    elif self.occupied((xIndex, yIndex)):
      return 'black'
    else:
      return 'red'

  def occProb(self, (xIndex, yIndex)):
    return self.grid[xIndex][yIndex].state.prob('occupy')

  def makeStartingGrid(self):
    lst=util.make2DArrayFill(self.xN,self.yN,lambda x,y:seFast.StateEstimator(cellSSM))
    for i in lst:
      for j in i:
        j.start()
    return lst

  def setCell(self, (xIndex, yIndex)):
    self.grid[xIndex][yIndex].step(('hit',None))
    self.drawSquare((xIndex, yIndex))

  def clearCell(self, (xIndex, yIndex)):
    self.grid[xIndex][yIndex].step(('free',None))
    self.drawSquare((xIndex, yIndex))

  def occupied(self, (xIndex, yIndex)):
    return self.occProb((xIndex, yIndex))>0.99
```

Step14 and Step15

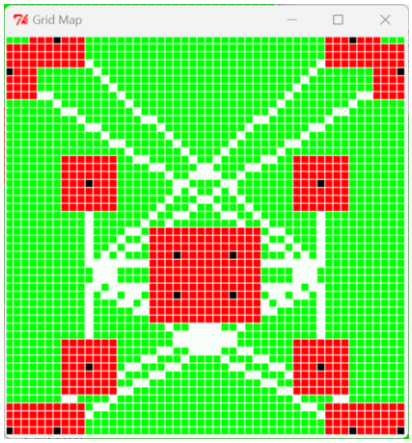
Result: testMapMakerN(1, testData)



Result: testMapMakerN(2, testData)



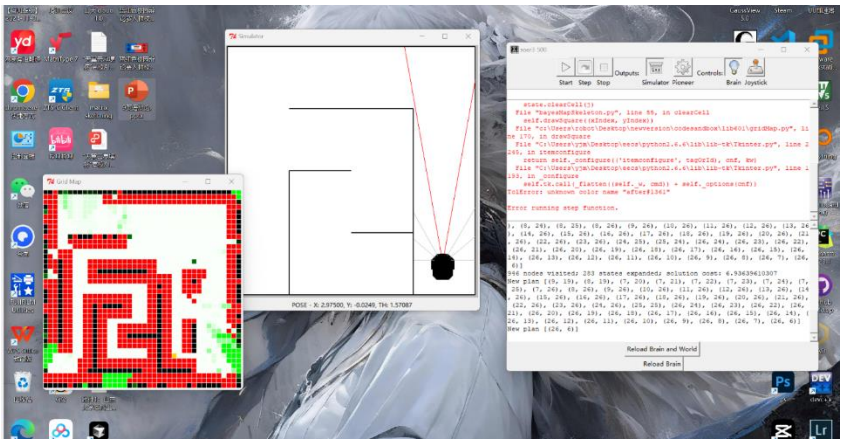
Result: testMapMakerN(2, testClearData)



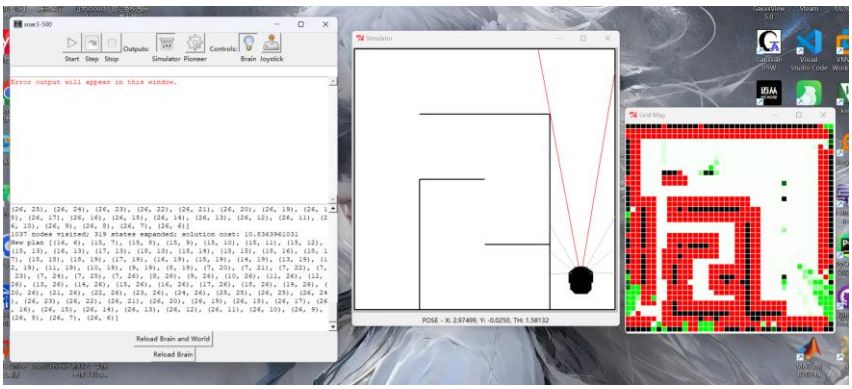
Step16

As the noise increases, the observed values become more unstable, and the accuracy of the map generated during dynamic planning is very low, requiring repeated observations to plan the correct route

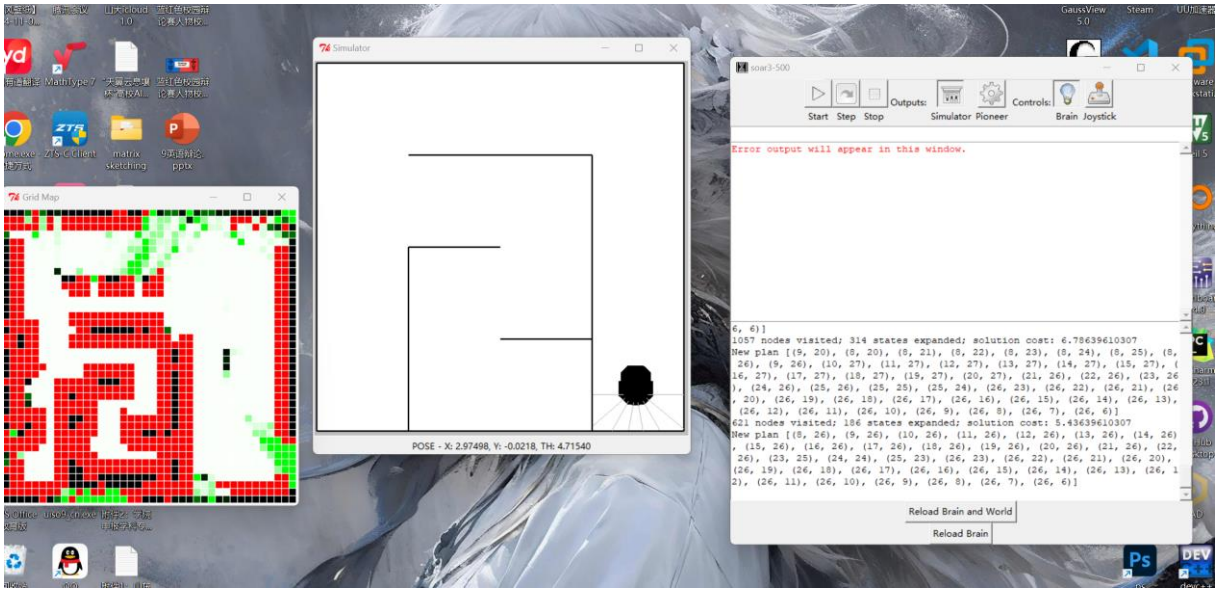
Result : No noise



Result : Medium noise

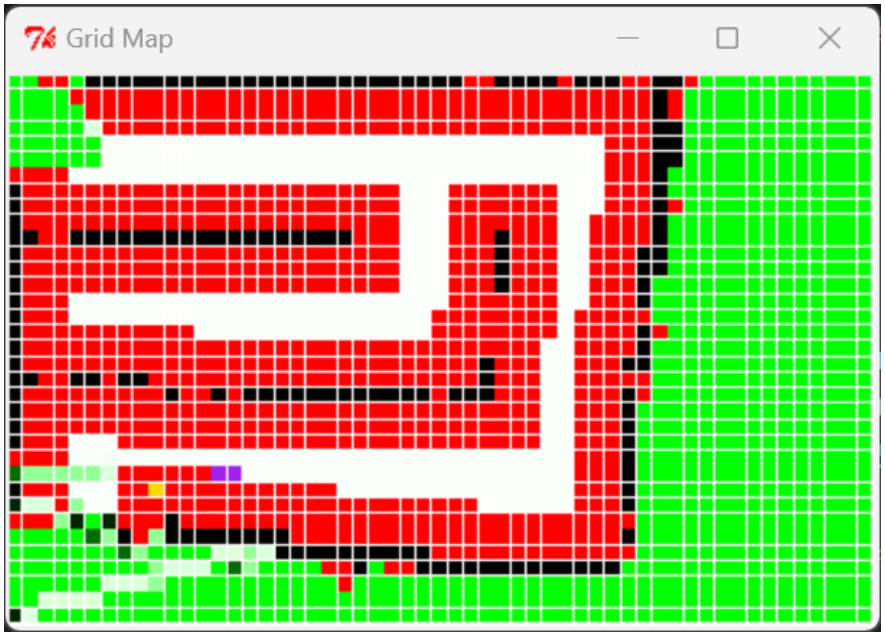


Result : High noise



Real Test

We successfully finished the map in the real world



Summary

We've learned a lot from this experiment. We built our code from scratch and truly applied what we had learned. Firstly, we encountered the problem that the code simulation got stuck. We chose to replace "start" with "step" and successfully solved the problem. Then, we entered the stage of the real vehicle test. At the very beginning, no matter how we adjusted the parameters, we just couldn't make the little car get out of the

maze. Later, we found that it was due to the mismatch of the map. So we changed the destination in the code and finally managed to make the car get out. We were very excited. Up to now, this course has also come to an end and we've really learned a great deal.