



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'm the Map!

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 18 日

Introduction

In this lab, we will connect the planner from Software Lab 14 with a state machine (named Map Maker) that dynamically builds a map as the robot moves through the world.

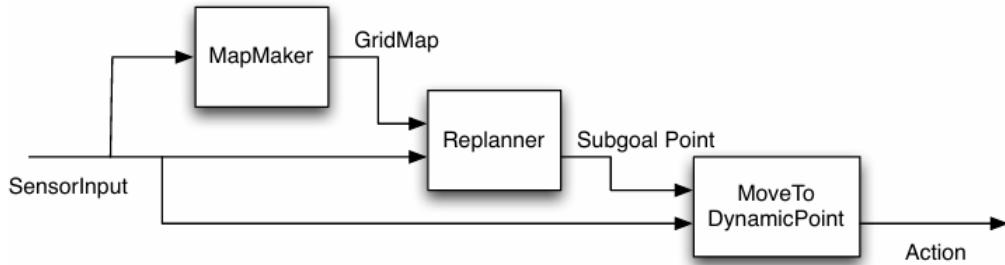


Figure 1 diagram of the architecture of the system

Mapmaker, mapmaker, make me a map

The MapMaker state machine dynamically builds a world representation by transforming sonar distance readings into global coordinates and marking the corresponding cells in a DynamicGridMap as occupied obstacles.

Step 1.

Check Yourself 1.

We have defined a `SensorInput` class for testing in idle that simulates the `io.SensorInput` class in soar.

First set of inputs: `testData = [SensorInput([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2], util.Pose(1.0, 2.0, 0.0))]`

The robot is at (1,2), facing 0 radians, with 8 sonar readings of 0.2. Since all readings are less than the maximum range detectable by the

sonar, this indicates that there is an obstacle at the end of some sonar sensors. Second set of inputs: testData = [SensorInput([0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4], util.Pose(4.0, 2.0, -math.pi))]

The robot is at (4,2), facing $-\pi$, and all sonar readings are 0.4, which is less than the maximum range detectable by the sonar, indicating that there is an obstacle at the end of some sonar sensors.

Step 2. Implement the MapMaker class.

The core functionality of this code is to process the robot's odometry and sensor data while dynamically updating the occupancy state of a Bayesian map. First, a function is defined to accept the map's coordinate range and grid size, while initializing the state of the state machine to store the occupancy probabilities of the Bayesian map.

The function getNextValues receives the current map state and inputs. It extracts the robot's odometry data from the input inp to determine the robot's current pose, while calling pointToIndices in the Bayesian map to convert the starting coordinates to map grid indices and determine the starting point of the sonar rays. It iterates over all sonar sensor data, filtering out invalid data.

It retrieves all cells traversed by the sonar rays. When the sonar reading is less than the maximum range detectable by the sonar, cells along the ray are marked as free, and the terminal cell of the ray is marked as occupied, thereby updating the map accordingly.

```

1. class MapMaker(sm.SM):
2.     def __init__(self, xMin, xMax, yMin, yMax, gridSquareSize):
3.         self.startState = bayesMap.BayesGridMap(xMin, xMax, yMin, yMax,
gridSquareSize)
4.     def getNextValues(self, state, inp):
5.         robotPose = inp.odometry
6.         startPoint = util.Point(robotPose.x, robotPose.y)
7.         startIndices = state.pointToIndices(startPoint)
8.
9.         for i in range(len(inp.sonars)):
10.             reading = inp.sonars[i]
11.             if reading < 0.1:
12.                 continue
13.             sonarPose = sonarDist.sonarPoses[i]
14.
15.             dist = reading
16.             if dist > sonarDist.sonarMax:
17.                 dist = sonarDist.sonarMax
18.
19.             hitPoint = sonarDist.sonarHit(dist, sonarPose, robotPose)
20.             endIndices = state.pointToIndices(hitPoint)
21.
22.
23.             if startIndices is None or endIndices is None:
24.                 continue
25.
26.             cellIndices = util.lineIndices(startIndices, endIndices)
27.
28.             if reading < sonarDist.sonarMax:
29.                 for cell in cellIndices[:-1]:
30.                     state.clearCell(cell)
31.
32.                     state.setCell(cellIndices[-1])
33.
34.
35.             else:
36.                 pass
37.
38.         return (state, state)

```

Step 3. Test our map maker inside idle

When we call the mapMakerSkeleton function, the result we get is:

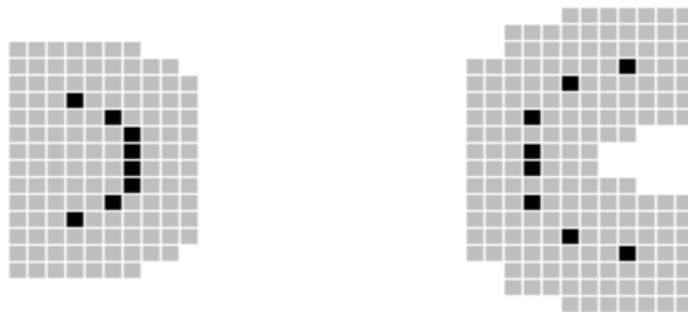


Figure 2 The plot of step3

The final result is consistent with the image given in Step 1.

Step 4. Now, test our code in soar.

Testing and visualization are performed by selecting corresponding simulated world in Soar and using the useWorld function, which generates a pop-up window displaying occupied cells in black and non-occupiable safety zones in gray.

The figure below shows running result of the mapAndReplanBrain file.

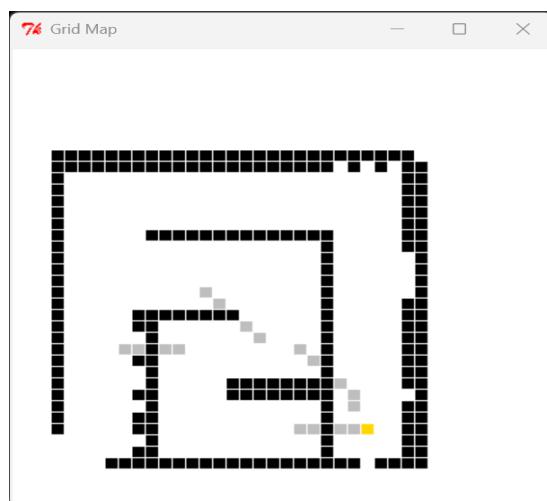


Figure 3 The result in soar

Checkoff 1. Wk.14.2.1: How does the dynamically updated map interact with the planning and replanning process?

In mapAndReplanBrain, the goal is to achieve dynamic interaction between map updates and route planning.

Dynamic Map Generation:

First, the map needs to be initialized by inputting the map range and grid size, and a Bayesian grid map should be initialized as the initial state. To achieve dynamic map generation, real-time updates are necessary. The step function calls robot.behavior.step(io.SensorInput(cheat=True)) 10 times per second, reading odometry and sonar sensor data each time, and updating the status of obstacles and free cells in the grid map, which is then transmitted to the planning module in real time through sm.Parallel.

Route Planning:

First, the initial output map is read, and using the preset target point as the midpoint, a heuristic search is used to generate the optimal path from the current location to the endpoint. Subsequently, every time MapMaker updates the map, the update is transmitted to ReplannerWithDynamicMap through sm.Parallel. During the re-planning process, the validity of the current path is continuously checked.

Execution Process:

In the MoveToDynamicPoint function, the latest path output by ReplannerWithDynamicMap is transmitted to MoveToDynamicPoint. This part splits the path into a series of consecutive sub-goals, allowing the robot to move. At the same time, the odometry and sensor data obtained during the robot's movement are used as input for MapMaker, driving map updates and forming a closed loop.

```

io.setDiscreteStepLength(0.5)

# Noise variances
noNoise = 0
smallNoise = 0.025
mediumNoise = 0.05
bigNoise = 0.1

# Change noise here
# soar.outputs.simulator.SONAR_VARIANCE = lambda mean: meidumNoise
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise
#soar.outputs.simulator.SONAR_VARIANCE = lambda mean: smallNoise
#soar.outputs.simulator.SONAR_VARIANCE = lambda mean: bigNoise

mapTestWorld = [0.18, util.Point(2.0, 5.5), (-0.5, 5.5, -0.5, 8.5)]
mazeWorld = [0.15, util.Point(2.0, 0.5), (-0.5, 5.5, -0.5, 5.5)]
dl14World = [0.15, util.Point(3.5, 0.5), (-0.5, 5.5, -0.5, 5.5)]
bigPlanWorld = [0.25, util.Point(3.0, 1.0), (-0.5, 10.5, -0.5, 10.5)]
lizWorld = [0.25, util.Point(9.0, 1.0), (-0.5, 10.5, -0.5, 10.5)]

sduWorld = [0.15, util.Point(2.5, 0.5), (-0.5, 3.5, -0.5, 4.5)]

def useWorld(data):
    global gridSquareSize, goalPoint, xMin, xMax, yMin, yMax
    (gridSquareSize, goalPoint, (xMin, xMax, yMin, yMax)) = data

#useWorld(dl14World)
useWorld(dl14World)

# this function is called when the brain is (re)loaded
def setup():
    mapper = mapMaker.MapMaker(xMin, xMax, yMin, yMax, gridSquareSize)

```

```

replannerSM = replanner.ReplannerWithDynamicMap(goalPoint)
robot.behavior = \\
    sm.Cascade(sm.Parallel(sm.Cascade(sm.Parallel(mapper, sm.Wire()),
                                         replannerSM),
                             sm.Wire()),
                move.MoveToDynamicPoint())

# this function is called when the start button is pushed
def brainStart():
    robot.behavior.start()

# this function is called 10 times per second
def step():
    robot.behavior.step(io.SensorInput(cheat = True)).execute()
    io.done(robot.behavior.isDone())

# called when the stop button is pushed
def brainStop():
    pass

# called when brain or world is reloaded (before setup)
def shutdown():
    for w in windows.windowList:
        w.destroy()

```

A noisy noise annoys an oyster

Step 5. Investigation under conditions where the sonar readings are not perfect.

We can obtain the map under no noise sonar readings in Soar:

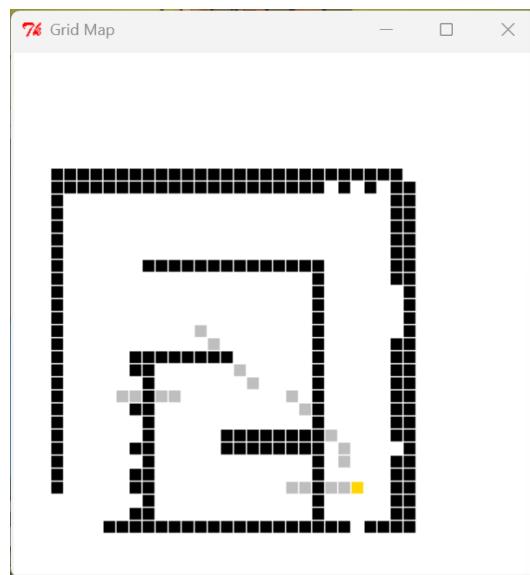


Figure 4 The map under no noise sonar readings

We can obtain the map under medium noise sonar readings in Soar:

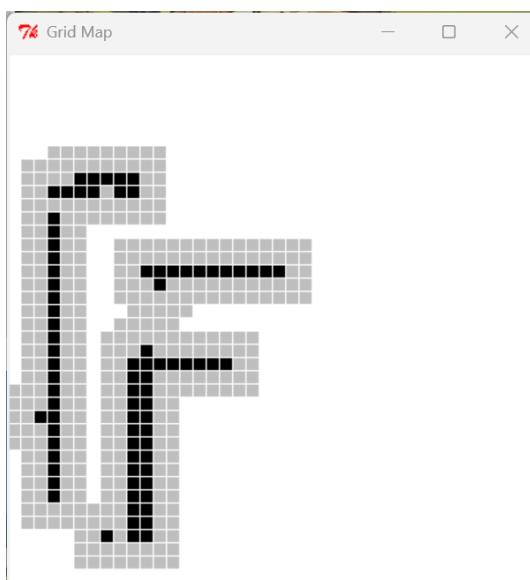


Figure 5 The map under medium noise sonar readings

Check Yourself 2. Investigating the Impact of Sensor

Reading Noise on Brain Function

Here is our answer:

The system lacks an uncertainty handling mechanism and does not consider the issue of sensor uncertainty. Noise can lead to incorrect map construction, which in turn causes planning failure and unstable control.

If the noise causes the readings to be too small, the free space is easily wrongly marked as an obstacle, as shown in the figure. In the case of small noise, the search failure is caused by the too-small sonar readings; if the noise causes the readings to be too large, the real obstacles are ignored, resulting in collision risks.

There may also be multiple observations at the same location yielding different results. The map cannot converge stably or the position of the obstacles "jumps" between grids, making a reliable environmental representation impossible. And the code completely relies on the accuracy of the map, which may lead to unstable control, frequent re-planning of paths, reduced resource efficiency, and a comprehensive deterioration of navigation performance.

Step 6. Improve the MapMaker class by using the util.lineIndices function

Code logic:

We construct the environmental map in real time using 8 sonar sensors. For each sensor, first, calculate its own position (starting point) and the position of the detected obstacle (ending point) in the grid map. If the sonar reading is valid (i.e., an obstacle is detected), mark the grid where the ending point is located as an occupied state. Then, obtain all the grids along the path from the sonar starting point to the obstacle

ending point, and mark all these path grids (except the grid where the obstacle is located) as idle areas. Finally, return the updated grid map, thereby gradually building a passable map that reflects the distribution of environmental obstacles.

Here is our code:

```
def getNextValues(self, state, inp):
    for i in range(8):
        # Calculate the starting point of the sonar (sensor position)
        sensor_start = sonarDist.sonarHit(0, sonarDist.sonarPoses[i], inp.odometry)
        start_indices = state.pointToIndices(sensor_start)

        # Calculate and mark the end point (position of the obstacle) of the sonar
        if inp.sonars[i] < sonarDist.sonarMax:
            obstacle_point = sonarDist.sonarHit(inp.sonars[i],
sonarDist.sonarPoses[i], inp.odometry)
            end_indices = state.pointToIndices(obstacle_point)
            state.setCell(end_indices)

        # Obtain all the grid indices on the path
        path_indices = util.lineIndices(start_indices, end_indices)

        # Clear the grids on the path
        for cell in path_indices[:-1]:
            state.clearCell(cell)
    return (state, state)
```

Step 7. Test new MapMaker in Idle

Check Yourself 3. Predict what the resulting map should look like.

Predicted map pattern:

- **Starting from position (1.0, 2.0):** 4 short rays (reading 1.0) form black dots at the end point, and the path turns white; 4 long rays (reading 5.0) form a white long passage.

- **Starting from position (4.0, 2.0):** Similar ray pattern but in the opposite direction (towards the west)
- **Overall effect:** Two symmetrical "star-shaped" clearing areas, with black obstacle points at the end of the short rays, and a white cross passage formed by the long rays.

Here is the basis of our prediction:

- **Sonar reading pattern analysis:** The sonar reading patterns in the test data are [1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0]. This alternating pattern of short distances (1.0) and long distances (5.0) determines the length and distribution of the rays.
- **Influence of robot position and orientation:** The robot is positioned at (1.0, 2.0) facing east (0 radians), while at (4.0, 2.0) facing west (- π radians). The positions are symmetrical and the orientations are opposite, resulting in a mirror-symmetric ray pattern.
- **Map construction algorithm principle:** According to the util.lineIndices algorithm, the grid on the sonar ray path is marked as idle (white), while the end points of the rays are marked as obstacles (black).
- **Geometric transformation rules:** The fixed layout of the sonar sensor combined with the robot's orientation determines the emission direction of the rays in space, forming a specific star-shaped distribution pattern.

Here is the resulting map in Idle by doing testMapMakerClear:

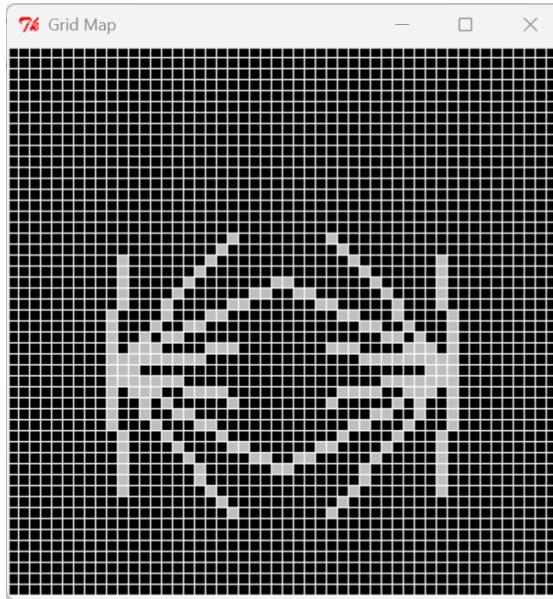


Figure 6 The resulting map by doing `testMapMakerClear(testClearData)`

Step 8. Run `mapAndReplanBrain` in soar again to obtain the new results under both noise-free and moderate noise conditions

We can obtain the new map under no noise sonar readings:

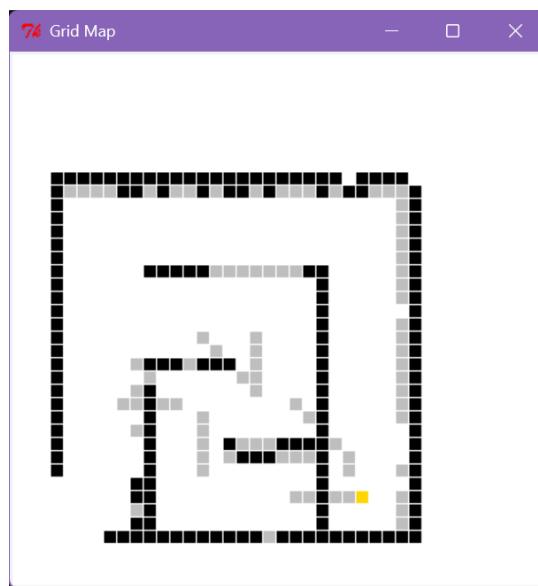


Figure 7 The new map under no noise sonar readings

We can obtain the new map under medium noise sonar reading:

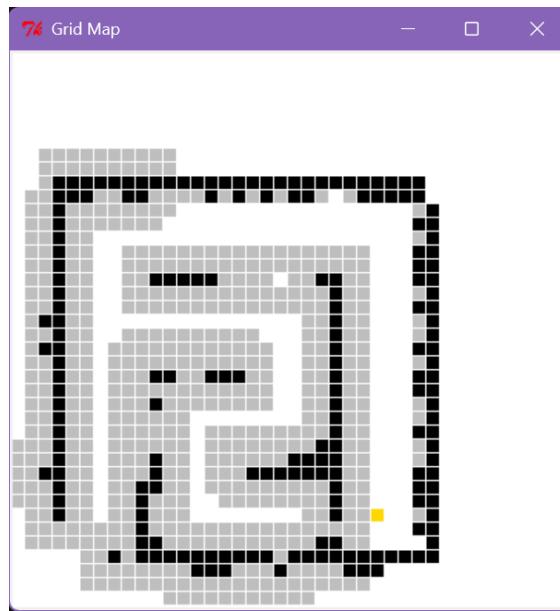


Figure 8 The new map under medium noise sonar readings

Checkoff 2. Wk. 14.2.2: Explain the principle of our new map maker under no noise and medium noise sonar conditions.

Here is our answer:

(1) No noise test result and cause:

- **Result:** The system was able to navigate smoothly, and the robot was able to accurately create a map and plan a path to reach the destination.
- **Reason:** In an environment without noise, the sonar readings are accurate, and the map builder can correctly mark obstacles and clear the empty areas on the path. This ensures the authenticity of the map. The path planner generates feasible paths based on the accurate map,

thus enabling smooth navigation.

(2) Medium noise test result and cause:

- **Result:** The system performance has improved. It can complete path exploration, but there are still occasional failures in path search, and it is not very stable.
- **Reason:** The improved map builder reduces the accumulation of incorrect obstacle markers by clearing the grids along the sonar ray paths. It also partially compensates for the obstacle position calculation errors caused by noise through dynamic re-planning. However, the randomness of noise makes it difficult to achieve completely reliable navigation.

(3) The function and improvement principle of our new map maker

- **The map maker function:** The map maker dynamically updates the grid map, marking obstacles and idle areas.
- **Improvement principle:** The original system only marked obstacles and was sensitive to noise. The improved system utilizes the "path information" (the area traversed by the sound wave is idle) and "endpoint information" (the end of the sound wave is an obstacle) of the sonar rays, reducing the impact of a single noise reading . However, when the noise is too large, incorrect readings may still exceed the system's correction capability.

Bayes Map

To handle sensor noise and improve map reliability, the BayesGridMap class decomposes the global mapping problem into independent state estimation tasks for each grid cell, using individual StateEstimator instances to maintain a probability distribution over the binary states of "occupied" or "not occupied".

Step 9. Finish code in bayesMapSkeleton.py

Code logic:

We must first establish the observation model and state transition model. During the integration phase, since the initial state does not specify each cell's condition, we assign a 0.5 probability to it, thereby forming the Stochastic State Model .

Here is our code:

```
def oGivenS(s):
    if s == True:
        return dist.DDist({'hit': 0.9, 'free': 0.1})
    else:
        return dist.DDist({'hit': 0.1, 'free': 0.9})
def uGivenAS(a):
    s = return lambda s: dist.DDist({s: 1.0})
cellSSM = ssm.StochasticSM(dist.DDist({True: 0.5, False: 0.5}), uGivenAS, oGivenS)
```

Step 10. Test the grid cell model

In the mostlyHits dataset, the probability of sonar colliding with the wall increases progressively as it repeatedly strikes the wall.

For mostlyFree data, the probability of collision decreases progressively as the sonar fails to detect obstacles multiple times consecutively.

```
IDLE 2.6.6
>>> ===== RESTART =====
>>>
>>> print testCellDynamics(cellSSM, mostlyHits)
[DDist(False: 0.100000, True: 0.900000), DDist(False: 0.012195, True: 0.987805), DDist(False: 0.001370, True: 0.998630), DDist(False: 0.012195, True: 0.987805)]
>>> print testCellDynamics(cellSSM, mostlyFree)
[DDist(False: 0.900000, True: 0.100000), DDist(False: 0.987805, True: 0.012195), DDist(False: 0.998630, True: 0.001370), DDist(False: 0.987805, True: 0.012195)]
>>> |
```

Figure 9 The test result of these data

Step 12. Explain what we learn from Wk. 14.2.3

As shown in the comparison of issues in Wk 14.2.3, we should pay attention to the coupling of instance creation. The first method results in identical changes across all instances, whereas the second method produces independent instances.

Step 14. Implement the BayesGridMap class in bayesMapSkeleton.py.

Code logic:

Here is our code:

The system first initializes each cell independently. The occProb function evaluates the probability of an obstacle being present in the cell. If an obstacle is detected, the probability is increased by Setcell; otherwise, it is decreased by clearcell. When the probability exceeds 0.5, the cell is marked as having an obstacle.

```
def occProb(self, (xIndex, yIndex)):
```

```

    return self.grid[xIndex][yIndex].state.prob(True)

def makeStartingGrid(self):
    grid = util.make2DArrayFill(self.xN, self.yN,
                                lambda x, y: seFast.StateEstimator(cellSSM))
    for x in range(self.xN):
        for y in range(self.yN):
            grid[x][y].start()
    return grid

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)) > 0.5

```

Step 15. Test the code in Idle

Here are our results:

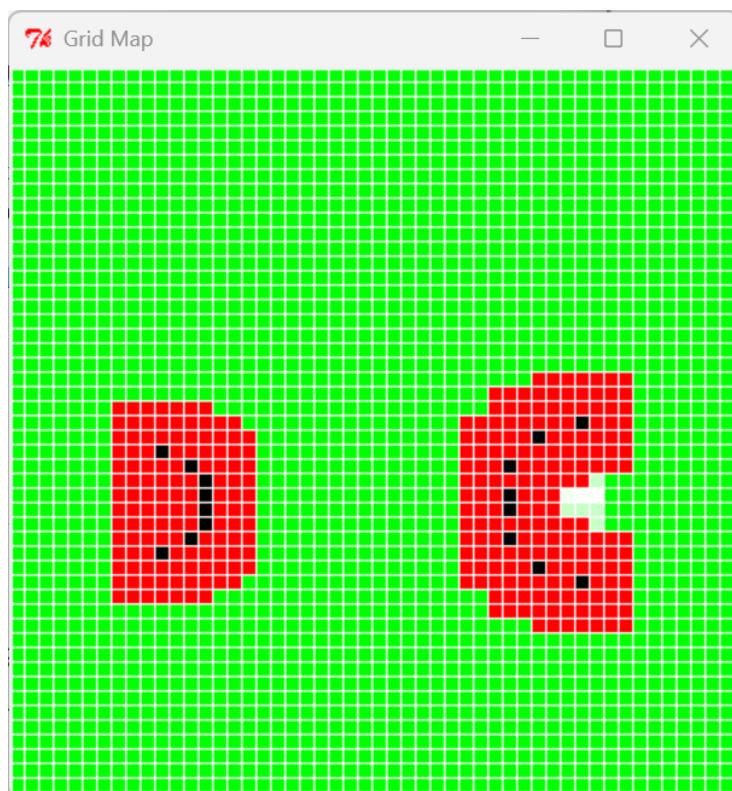


Figure 10 The plot of testMapMakerN(1, testData)

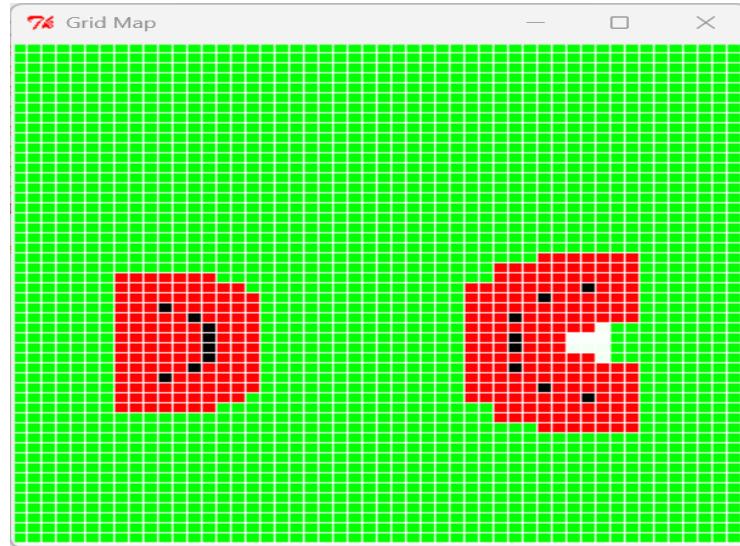


Figure 11 The plot of testMapMakerN(2, testData)

Step 16. Test the mapper in soar

Our Explanation:

Initially, the car was programmed to move straight to the right toward the destination.

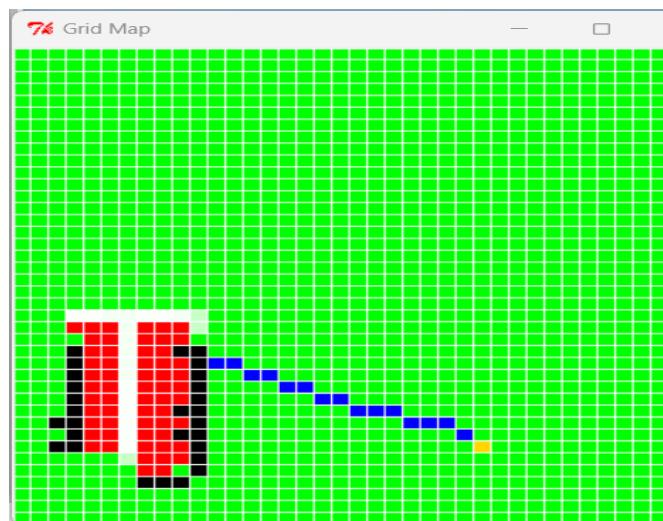


Figure 12 The initial plan in map

When it reaches the fork in the road, we notice it seems closer and leads directly to the destination.

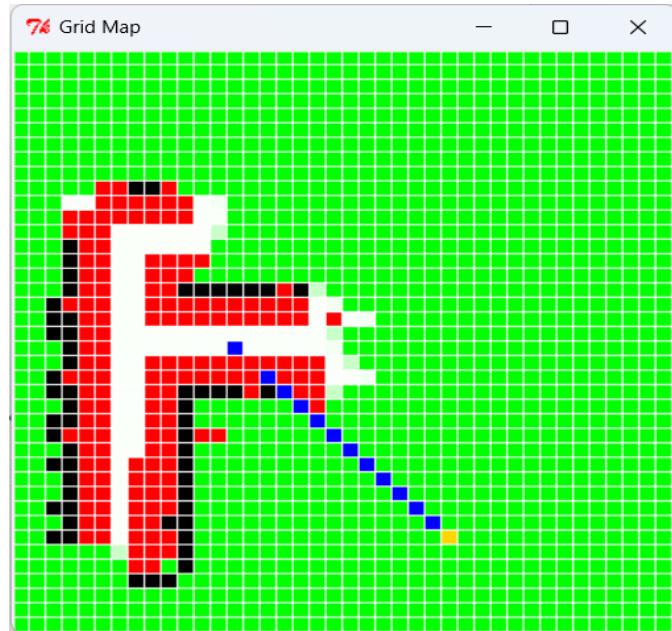


Figure 13 replanning after finding a fork

The route is blocked. Re-plan using the known route.



Figure 14 Re-plan after the fork

Finally, we successfully reached our destination.

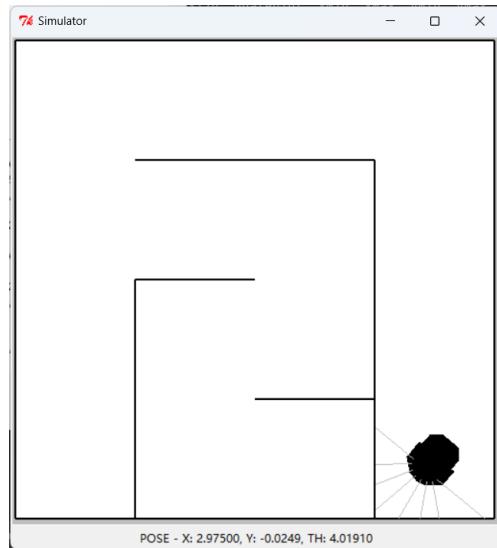


Figure 15 Finally arriving at our destination.

Checkoff 3. Using the BayesMap module with medium noise and high noise.

This is the operational result under moderate noise conditions.

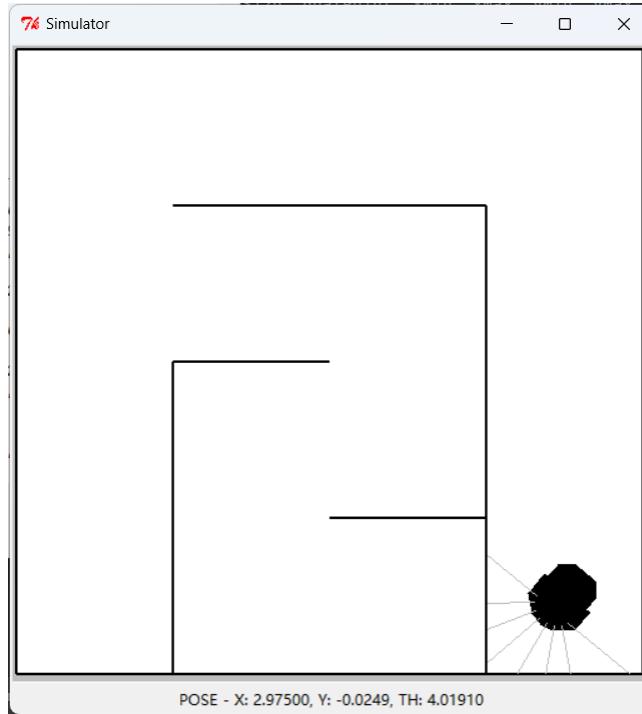


Figure 16 Finally arriving at our destination with medium noise

In high-noise environments, the established red walls may

occasionally vanish spontaneously. The vehicle requires more route planning iterations and takes additional detours, ultimately achieving successful destination.



Figure 17 Finally arriving at our destination with big noise

Real robot

Checkoff 4. Our Experience and Insight in Real Vehicle

Debugging

We encountered three issues while debugging the actual vehicle.

First, the car makes an excessively early turn at the fork, which could result in a collision with the wall.

The Bayes occupy threshold was set too high, resulting in the probability of hitting the right wall being below 0.9. We lowered the occupy threshold to prevent wall collisions.

Secondly, the car tends to hit the wall at the end of the second intersection and often fails to detect the wall in the corner.

solve:

There's an issue with handling the excessively close distance. I used continue to skip it, but the minimum distance should actually be set.

Eventually, we made the robot arrive at the goal point!

Go, speed racer, go!

Step 17. Check Yourself 6. Run robot through raceWorld and see score

```
Total steps: 296  
Elapsed time in seconds: 35.3990001678
```

Figure 18 base score

By optimizing the post-processing logic of the path, enhancing the forward and rotational gains of the proportional controller, and widening the Angle tolerance, the robot will be able to significantly reduce pauses and oscillations between grid points, thereby achieving lower step count and time scores in official competitions.

Step 18. Implement some improvements to make the robot go faster.

By implementing "line sub-target merging" at the path planning level and "gain parameter optimization" at the control level, We significantly reduced the number of decelerations of the robot in each grid cell and the fine-tuning time for in-place rotation, thereby minimizing the total competition duration to the greatest extent without violating the maximum speed limit.

Check Yourself 7. Post best scores in simulation on raceWorld and lizWorld on the board.

Through multiple logical optimizations and parameter adjustments, the ultimate speed under this algorithm was finally achieved as follows:

```
Total steps: 83  
Elapsed time in seconds: 9.2389998436
```

Figure 19 Improved result

Appendix: The Description of AI Usage in the Report

In this report, we primarily utilize AI for English translation. Additionally, AI assists in troubleshooting issues encountered during actual vehicle testing, providing us with actionable solutions.