ANSWERS

U8.2 The inverse sensor model

Mapping · University · about 30 min

What this lesson is about

One reading is two statements: free all the way along the ray, occupied at the end of it.

- Why it is called inverse
- The model, in full
- Where the beam actually points
- One ray, drawn
- The beam is not a line

Questions

7 marks in all

1. A mapper marks only the cell at the end of each beam as occupied. What is wrong with the map it builds? [1 mark]

- ☐ **A** It outlines the walls but knows nothing about the floor, which is what a planner needs
- ☐ **B** It puts a ring of imaginary wall around the robot
- ☐ **C** The walls never become confident
- ☐ **D** It comes out mirrored about the diagonal

> **Answer:** A. A reading also says every cell the beam passed through is free, about twenty cells for one, and that is what makes a grid fill in.

2. Which of these is the inverse sensor model? [1 mark]

- ☐ **A** Given a reading and a pose, what the world is like along the beam
- ☐ **B** Given the world and a pose, what the sensor will read
- ☐ **C** Given two readings, how the robot moved between them
- ☐ **D** Given the map, which cells are frontiers

> **Answer:** A. The forward model predicts a reading from the world, as in U7. The inverse model goes from a reading to the world, as here.

3. The sensor finds nothing within range and returns its maximum. How should the mapper treat that beam? [1 mark]

- ☐ **A** Mark the cells along it free, and mark nothing occupied
- ☐ **B** Mark the cell at maximum range occupied, as for any reading
- ☐ **C** Ignore the beam entirely
- ☐ **D** Mark every cell along it occupied

> **Answer:** A. A timeout is a statement about free space only. Marking a hit at maximum range draws a ring of imaginary wall around the robot.

4. Why does the free marking stop one cell short of the hit? [1 mark]

☐ **A** Otherwise the free update fights the occupied update in the same cell and the wall never becomes confident

☐ **B** The last cell is always outside the mat

☐ **C** It corrects for the half-cell bias of a ray caster

☐ **D** Free cells next to a wall are unsafe for a planner

> **Answer:** A. Each update to the hit cell would be partly cancelled by a free update from the same beam.

5. What does this program print? [1 mark]

```
import math
CELL = 5.0
x, y, h = 100.0, 50.0, 90.0
a, r = 0.0, 40.0
th = math.radians(h + a)
hit_x = x + r * math.sin(th)
hit_y = y + r * math.cos(th)
print(round(hit_x, 1), round(hit_y, 1))
print(int(hit_x / CELL), int(hit_y / CELL))
```

> **Answer:**
>
> ```
> 140.0 50.0
> 28 10
> ```
>
> Heading is clockwise from +y, so 90 degrees faces along +x: sin is 1 and cos is 0. The hit is at (140, 50), which is cell (28, 10).

6. What does this program print? [1 mark]

```
CELL = 5.0
X0, Y0 = 100.0, 50.0
d = 100.0
free = set()
r = 0.0
while r < d - CELL:
    free.add((int(X0 / CELL), int((Y0 + r) / CELL)))
    r += CELL / 2
print(len(free), (int(X0 / CELL), int((Y0 + d) / CELL)))
```

> **Answer:**
>
> ```
> 19 (20, 30)
> ```
>
> The steps run from y = 50 to 142.5, covering rows 10 to 28, which is 19 distinct cells. Half-cell steps visit each row twice, and the set removes the repeats. The hit at y = 150 is row 30.

7. A mapper's walls come out mirrored about the diagonal of the mat, and otherwise look plausible. What is the most likely bug? [1 mark]

- ☐ **A** The sine and cosine are the wrong way round in the hit position
- ☐ **B** The heading is in degrees instead of radians
- ☐ **C** The free marking does not stop one cell short
- ☐ **D** A timeout is being marked as a hit

> **Answer:** A. With heading measured clockwise from +y, x uses sin and y uses cos. Swapping them exchanges x and y, which is a reflection about the diagonal.

The task: one ray, two statements

Take a reading, run the inverse sensor model along it, and print `free:`, the number of different cells the ray passes through, and `wall y:`, the mat y of the cell it ended in.

```
from bugbot import *
connect()

CELL = 5.0
X0, Y0 = 100.0, 50.0
```

The hint students can ask for: Step out along the ray in steps smaller than a cell, keeping the cells you land in, and stop one cell short of the reading. The cell at the reading itself is the occupied one, and its centre is half a cell above the bottom of the row.

A solution

```
from bugbot import *
connect()

CELL = 5.0
X0, Y0 = 100.0, 50.0

readings = []
for i in range(12):
    readings.append(distance())
    wait(0.1)
d = sum(readings) / len(readings)

# free all the way along the ray, stopping one cell short of the end
free = set()
r = 0.0
while r < d - CELL:
    free.add((int(X0 / CELL), int((Y0 + r) / CELL)))
    r += CELL / 2
print("free:", len(free))

# and occupied at the end of it
row = int((Y0 + d) / CELL)
print("wall y:", round(row * CELL + CELL / 2, 1))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.