ANSWERS

U8.7 Project: map the mat

Mapping · University · about 50 min

What this lesson is about

An unknown layout, a grid built from scratch, and a measurement taken off the map.

- What the program has to do
- Reading a doorway out of a grid
- The structure
- What to look at when it is wrong
- What comes next

Questions

6 marks in all

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

```
CELL = 5.0
T, F = True, False
band = [T, F, F, T, T, T, F, F, F, F, T, T, F, F]
best = None
c = 0
while c < len(band):
    if not band[c]:
        start = c
        while c < len(band) and not band[c]:
            c += 1
        bounded = start > 0 and c < len(band)
        if bounded and (best is None or c - start > best[1] - best[0]):
            best = (start, c)
    else:
        c += 1
print(best, (best[0] * CELL + best[1] * CELL) / 2)
```

Answer:

```
(6, 10) 40.0
```

The runs of clear columns are 1 to 2, 6 to 9 and 12 to 13, but the last is not bounded by wall on the right. The longest bounded run spans columns 6 to 9, from x = 30 to x = 50, so the gap is centred at 40 cm.

2. When searching the band for the doorway, why must a clear run have wall on both sides? [1 mark]

- ☐ **A** Beyond the ends of the wall there is also clear space, and it must not be mistaken for a door
- ☐ **B** Doorways are always narrower than the wall is thick
- ☐ **C** It removes the half-cell bias from the answer
- ☐ **D** Runs at the edge of the grid are always unknown

Answer: A. Bounding the run makes it robust, and taking the longest bounded run also survives a stray column that was never observed.

3. A program reports the wall at y = 0 on every run. What has it most likely found? [1 mark]

- ☐ **A** The mat's own edge behind the robot, which is mapped as an obstacle
- ☐ **B** A timeout marked as a hit
- ☐ **C** The half-cell bias
- ☐ **D** A shadow behind the wall

> **Answer:** A. The lowest occupied row is the edge behind the robot, so search above the robot's start and leave the border out.

4. The reported doorway is always off by exactly one cell. What should you check first? [1 mark]

- ☐ **A** Rounding: whether you are reporting the edge of a cell or its centre
- ☐ **B** The heading error on each scan
- ☐ **C** Whether the far side of the wall is unknown
- ☐ **D** The value of L_OCC

> **Answer:** A. An error of exactly one cell is almost always an indexing or rounding question, not a sensing one.

5. Why does the project sweep from a second viewpoint as well as the first? [1 mark]

- ☐ **A** It fills in the first viewpoint's shadows and confirms the walls that both saw
- ☐ **B** One rotation cannot cover 360 degrees of bearings
- ☐ **C** The first sweep's cells are reset when the robot moves
- ☐ **D** The clamp stops a single viewpoint from marking any cell

> **Answer:** A. Confirmation matters as much as coverage: a cell two viewpoints agree on is worth far more than one a single beam clipped.

6. After a full turn on the spot, the robot calls forward() and drives sideways across the mat. Why? [1 mark]

- ☐ **A** It is left facing an arbitrary direction, and forward() is relative to the robot, not the mat
- ☐ **B** The map has rotated the robot's coordinate frame
- ☐ **C** The wheels slip after turning
- ☐ **D** The inverse sensor model has changed the heading

> **Answer:** A. Turn back to a known heading before driving, or steer in the world frame with the inverse kinematics from U2.

The task: map the mat

The robot starts at (100, 25) facing up the mat. Map it, and print `explored:`, `wall y:` and `gap x:`. Do not touch anything.

```
from bugbot import *
import math
connect()

CELL, W = 5.0, 40
L_OCC, L_FREE, L_MAX, FAR = 0.85, -0.4, 8.0, 170.0
START_X, START_Y = 100.0, 25.0
grid = [0.0] * (W * W)
```

The hint students can ask for: The robot starts at (100, 25) facing up the mat and there is a wall across it with one way through. Turn on the spot to fill in the grid, move to a second viewpoint and turn again, then read the answers off the map: the lowest occupied row, and the run of cells in that row that is not wall.

A solution

```
from bugbot import *
import math
connect()

CELL, W = 5.0, 40
L_OCC, L_FREE, L_MAX = 0.85, -0.4, 8.0
FAR = 170.0
START_X, START_Y = 100.0, 25.0
grid = [0.0] * (W * W)

def bump(x, y, amount):
    c, r = int(x / CELL), int(y / CELL)
    if 0 <= c < W and 0 <= r < W:
        i = r * W + c
        grid[i] = max(-L_MAX, min(L_MAX, grid[i] + amount))

def integrate():
    px, py = position()
    x, y = START_X + px, START_Y + py
    h = heading()
    for a, d in scan():
        th = math.radians(h + a)
        sx, sy = math.sin(th), math.cos(th)
        free = set()                        # each cell the beam crosses, once
        r = 0.0
        while r < min(d, FAR) - CELL:
            free.add((int((x + r * sx) / CELL), int((y + r * sy) / CELL)))
            r += CELL / 2
        for c, row in free:
            bump(c * CELL, row * CELL, L_FREE)
        if d < FAR:
            bump(x + d * sx, y + d * sy, L_OCC)

def explored():
    return sum(1 for v in grid if abs(v) > 1.0) / float(W * W)

def sweep(ticks):
```

```
    drive(0, 0, 30)
    for tick in range(ticks):
        integrate()
        plot("known", explored())
        wait(0.1)
    stop()
    for tick in range(5):
        integrate()
        plot("known", explored())
        wait(0.1)

def face_up():
    """Point back along +y, so that forward() goes where it is meant to."""
    for tick in range(200):
        e = (heading() + 180) % 360 - 180
        if abs(e) < 4:
            break
        drive(0, 0, -40 if e > 0 else 40)
        wait(0.05)
    stop()
    wait(0.3)

sweep(110)                      # a turn on the spot from where it woke up
face_up()
forward(55, distance=45)        # a second viewpoint, further up the mat
sweep(110)

print("explored:", round(explored(), 3))

# the wall is the lowest row with a proper run of occupied cells in it. The mat's own edge
is a wall
# too, so start above the robot and leave the border columns out of the count.
face = None
for r in range(7, W - 1):
    if sum(1 for c in range(1, W - 1) if grid[r * W + c] > 1.0) >= 8:
        face = r
        break
print("wall y:", round(face * CELL + CELL / 2, 1))

# a column of that band is wall if any of its cells there is occupied
band = []
for c in range(W):
    band.append(any(grid[r * W + c] > 1.0 for r in (face, face + 1, face + 2)))

# the doorway is the longest run of clear columns with wall on both sides
best, run = (0, -1, -1), None
for c in range(W):
    if not band[c]:
        run = c if run is None else run
    else:
        if run is not None and run > 0 and band[run - 1]:
            if c - run > best[0]:
                best = (c - run, run, c - 1)
        run = None
print("gap x:", round((best[1] + best[2] + 1) / 2.0 * CELL, 1))
```

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