ANSWERS

U8.4 Building a grid

BugBotLab

Mapping · University · about 35 min

What this lesson is about

Eight beams, a known pose, and a map that fills in while the robot drives.

- Storing the grid
- The pose goes in first
- Watching it fill
- The known fraction
- Reading an answer off the map

Questions

6 marks in all

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

```
W = 40
row, col = 3, 7
i = row * W + col
print(i)
print(divmod(1234, W))
```

Answer:

```
127
(30, 34)
```

Cell (row 3, col 7) is at 3 x 40 + 7 = 127 in the flat list, and index 1234 is 1234 = 30 x 40 + 34, so row 30, col 34.

2. The robot drives at 30 cm/s and the mapper uses the pose from 150 ms after the scan was taken. How far, in cm, is every wall smeared? [1 mark]

Answer: 4.5 (accept within 0.01). 30 cm/s x 0.15 s = 4.5 cm. The pose must be the one from when the scan was taken, not the pose now.

3. Every wall in a map comes out shifted and some land off the edge of the grid, but their shapes are right. What is the most likely bug? [1 mark]

- ☐ **A** position() is relative to where the robot started, and the start position was not added
- ☐ **B** heading() was passed to math.sin in degrees
- ☐ **C** The free marking does not stop one cell short
- ☐ **D** L_MAX is too small

Answer: A. A missing offset shifts everything by the same amount. Degrees passed as radians scrambles directions instead, which is unmistakable.

4. Put the steps of integrating one tick of scans into the grid in order. [1 mark]

Number the lines 1 to 5 to put them in the right order.

___ `If the range is a real return, add L_OCC at the hit cell`

___ `Add L_FREE once to each cell in the set`

___ `Step along the beam in half cells to one cell short of the range, collecting the cells in a set`

___ `For each beam, work out th = radians(heading + beam angle)`

___ `Fetch the pose and add the start position`

> **Answer:**
>
> ```
> Fetch the pose and add the start position
> For each beam, work out th = radians(heading + beam angle)
> Step along the beam in half cells to one cell short of the range, collecting the cells
> in a set
> Add L_FREE once to each cell in the set
> If the range is a real return, add L_OCC at the hit cell
> ```
>
> The pose comes first, since every cell depends on it. Then each beam is two statements: free along it, occupied at the end if something was hit. The set is there because half-cell steps land in most cells twice, and one beam is one piece of evidence.

5. The plot of the known fraction has gone flat while the robot keeps driving in the same direction. What does that tell you? [1 mark]

- ☐ **A** Driving further this way is buying almost no new information, so it is time to go somewhere else
- ☐ **B** The map has become wrong and needs resetting
- ☐ **C** The clamp has been reached in every cell
- ☐ **D** The robot has stopped moving

> **Answer:** A. The first scan from a new place is mostly new information; re-observing known cells adds none. That observation is the start of U8.6.

6. Why is it normal to require several occupied cells in a row before believing a wall is there? [1 mark]

- ☐ **A** A single occupied cell can be a stray reflection
- ☐ **B** Walls are always wider than one cell
- ☐ **C** The ray caster cannot see a single cell
- ☐ **D** It removes the half-cell bias

> **Answer:** A. It is the simplest form of the filtering a real system does with morphological operations.

The task: build a grid while you drive

The robot starts at (100, 40). Drive at least 40 cm up the mat, integrating every scan into a log odds grid. Plot `known`, then print `known:`, the fraction of the grid your map has an opinion about, and `wall y:`, where your map puts the near face of the wall.

```
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, 40.0
grid = [0.0] * (W * W)
```

The hint students can ask for: The robot starts at (100, 40) on the mat. Each tick, take the pose, turn every scan() pair into a world direction, and run the inverse sensor model along it. Plot how much of the grid is known as you go, then read the wall's position off the finished map rather than off a reading.

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                  # past this the beam has run out of mat, not found a wall
START_X, START_Y = 100.0, 40.0
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        # position() is from the start, so put it on
the mat
    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 known_fraction():
    return sum(1 for v in grid if abs(v) > 1.0) / float(W * W)
```

```
forward(55)
for tick in range(85):
    integrate()
    plot("known", known_fraction())
    if position()[1] > 60:
        stop()
    wait(0.1)
stop()
integrate()

print("known:", round(known_fraction(), 3))

# the near face of the wall is the lowest row that has occupied cells in it
face = None
for r in range(W):
    if any(grid[r * W + c] > 1.0 for c in range(8, 32)):
        face = r
        break
print("wall y:", round(face * 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.