ANSWERS

U7.6 Monte Carlo localisation

Localisation · University · about 35 min

What this lesson is about

The three steps in a loop on a real mat, with tags as the measurement.

- Mat coordinates, not travel
- Only while it faces the wall
- Using tags as well
- When it goes wrong

Questions

7 marks in all

1. Put one tick of Monte Carlo localisation in order. [1 mark]

Number the lines 1 to 4 to put them in the right order.

___ `Weigh every particle against the sensor reading`
___ `Move every particle by the odometry, plus noise`
___ `Report the weighted mean as the estimate`
___ `If neff is low, resample`

Answer:

```
Move every particle by the odometry, plus noise
Weigh every particle against the sensor reading
If neff is low, resample
Report the weighted mean as the estimate
```

Predict, correct, resample if needed, then report. Resampling before weighting would copy particles on stale weights.

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

```
depth = [0.9, 0.3, 0.05]
tag = [0.05, 0.3, 0.9]
both = [a * b for a, b in zip(depth, tag)]
total = sum(both)
print([round(w / total, 2) for w in both])
added = [a + b for a, b in zip(depth, tag)]
print([round(w, 2) for w in added])
```

Answer:

```
[0.25, 0.5, 0.25]
[0.95, 0.6, 0.95]
```

Multiplying gives 0.045, 0.09, 0.045, so the middle particle, which neither sensor rules out, wins with half the weight. Adding scores the outer particles higher even though one sensor all but rules each of them out.

3. The filter has a weight from the depth sensor and a weight from a tag seen by the camera for each particle. How should they be combined? [1 mark]

- ☐ **A** Multiply them, which is Bayes' rule for independent measurements
- ☐ **B** Add them, so that each sensor contributes its share
- ☐ **C** Take whichever is larger
- ☐ **D** Average them, so neither sensor dominates

> **Answer:** A. Independent evidence multiplies. A particle one sensor rules out should stay ruled out, and multiplication is what does that.

4. The cloud collapses onto the wrong answer. Which of these could cause it? [1 mark]

Tick every answer that is true.

- ☐ **A** Not enough jitter when resampling
- ☐ **B** A measurement sigma that is too small
- ☐ **C** A motion model that is too confident
- ☐ **D** Readings taken while the robot turns, weighed with a model that assumes it faces the wall square on
- ☐ **E** A measurement that is not informative enough to rule places out

> **Answer:** A, B, C, D. The first three all make the filter over-confident, and readings its model cannot explain drag the cloud to the wrong place. An uninformative measurement causes the opposite failure: a cloud that never collapses.

5. The spread of the cloud stays large however long the robot stands still. What does the page recommend? [1 mark]

- ☐ **A** Move the robot, so one ambiguous reading becomes several that disagree about the wrong places
- ☐ **B** Inject more scattered particles each tick
- ☐ **C** Reduce the jitter to zero
- ☐ **D** Increase the number of particles until it collapses

> **Answer:** A. A cloud that never collapses means the measurement is not informative enough from here. More particles or less jitter cannot create information.

6. The filter tracks well for a minute and then loses the robot for good. What is the likely cause and fix? [1 mark]

- ☐ **A** Particle deprivation; inject a few scattered particles each tick
- ☐ **B** Too many particles; reduce N
- ☐ **C** Sigma too large; reduce it
- ☐ **D** The odometry noise is too small; set it to zero

> **Answer:** A. Over time the cloud loses diversity, and once it is wrong there is nothing elsewhere to rescue it. A few injected particles are the insurance.

7. What does MCL give the robot that dead reckoning does not? [1 mark]

- ☐ **A** A position in the mat's own frame, which other things are described in, rather than a distance from wherever it started
- ☐ **B** A smoother velocity estimate
- ☐ **C** An estimate with no noise in it
- ☐ **D** The map of the mat

> **Answer:** A. Localisation places the robot in the frame the goal is described in, which is what lets it go somewhere on purpose.

The task: Monte Carlo localisation

Run the full filter while driving at least 50 cm. Plot `spread`, and print `my y:`, the robot's position on the mat as your filter has it. No `position()`. Drive straight, and expect the tag card to pull your answer a few centimetres ahead once the robot is near it.

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

DT, N, SIGMA = 0.1, 300, 3.5
particles = [random.uniform(0, 200) for i in range(N)]
```

The hint students can ask for: Particles spread over the mat, moved by the flow sensor and weighted by the depth reading against each particle's predicted distance to the far wall. Resample when the effective sample size falls below half. Print where the cloud has settled, in mat coordinates, not from the start.

A solution

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

DT = 0.1
N = 300
SIGMA = 3.5
particles = [random.uniform(0, 200) for i in range(N)]
weights = [1.0 / N] * N

def weigh(measured):
    global weights
    w = []
    for y in particles:
        d = (200 - y) - measured
        w.append(math.exp(-d * d / (2 * SIGMA * SIGMA)) + 1e-12)
    total = sum(w)
    weights = [v / total for v in w]

def resample():
    global particles, weights
    step = 1.0 / N
    r = random.uniform(0, step)
    c = weights[0]
    i = 0
    fresh = []
    for m in range(N):
        u = r + m * step
        while u > c and i < N - 1:
            i += 1
            c += weights[i]
        fresh.append(particles[i] + random.gauss(0, 0.8))
    particles = fresh
    weights = [1.0 / N] * N

forward(60)
for tick in range(70):
```

```
    v = flow()[1]
    particles = [y + v * DT + random.gauss(0, 0.4) for y in particles]
    weigh(distance())
    neff = 1.0 / sum(w * w for w in weights)
    if neff < N / 2:
        resample()
    mean = sum(w * y for w, y in zip(weights, particles))
    spread = math.sqrt(sum(w * (y - mean) ** 2 for w, y in zip(weights, particles)))
    plot("spread", spread)
    if mean > 118:
        stop()
    wait(DT)
stop()
mean = sum(w * y for w, y in zip(weights, particles))
print("my y:", round(mean, 1))
```

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