ANSWERS

U6.7 Project: navigate on the estimate

State estimation · University · about 45 min

What this lesson is about

Reach a target the robot cannot see, using nothing but the filter you wrote.

- What the program contains
- The structure
- What comes next

Questions

6 marks in all

1. The robot has moved sideways to x = 30 cm at y = 60 cm, and tag 9 is at (0, 155), straight up the mat from the start. The code treats the tag's range as if it were the y distance. What does it print? [1 mark]

```
import math
x, y = 30.0, 60.0
TAG_Y = 155.0
rng = math.hypot(0 - x, TAG_Y - y)
y_from_fix = TAG_Y - rng
print(round(rng, 1), round(y_from_fix, 1), round(y - y_from_fix, 1))
```

Answer:

```
99.6 55.4 4.6
```

The range is sqrt(30 squared + 95 squared) = 99.6 cm, so the fix claims y = 55.4 and drags the estimate 4.6 cm backwards. Range equals y distance only on the tag's line.

2. How does the lesson suggest avoiding that trap? [1 mark]

- ☐ **A** Gate the fix to when the estimated x is small, or drive up the tag's line first
- ☐ **B** Raise R_TAG to a very large value
- ☐ **C** Use the tag range as the x coordinate instead
- ☐ **D** Take the fix only when the variance is small

Answer: A. The range and the y distance agree only on the tag's line, so either only accept fixes there or plan the route so that is where the fixes happen.

3. After a missed run, a plot shows the estimate tracked the truth closely, yet the robot still missed the target. Where is the fault? [1 mark]

- ☐ **A** The controller
- ☐ **B** The filter
- ☐ **C** The tag detector
- ☐ **D** The gyro calibration

Answer: A. If the robot knew where it was and still went to the wrong place, steering is at fault. If the estimate had drifted, the filter would be.

4. The controller uses speed = min(13.0, 0.6 * gap). What speed, in cm/s, does it ask for when the gap is 10 cm? [1 mark]

> **Answer:** 6. 0.6 x 10 = 6, which is below the 13 cm/s cap, so the robot slows proportionally as it closes on the target.

5. The prediction step uses the heading h + 0.5 * rate * DT to rotate the flow into the world frame. Why the half step? [1 mark]

- ☐ **A** It uses the average heading over the tick, since the robot turns during it
- ☐ **B** It halves the gyro noise
- ☐ **C** It corrects for gyro bias
- ☐ **D** It keeps the heading within half a turn

> **Answer:** A. The robot turns by rate x DT during the tick, so the midpoint heading is a better estimate of the direction it actually moved than the start heading.

6. Put the parts of the navigation program in the order the lesson lists them. [1 mark]

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

___ `Plot, to tell estimator faults from controller faults`
___ `Calibrate the gyro`
___ `Steer by the estimate through the inverse kinematics`
___ `Run a filter every tick, predicting with flow and correcting from tag 9`

> **Answer:**
>
> ```
> Calibrate the gyro
> Run a filter every tick, predicting with flow and correcting from tag 9
> Steer by the estimate through the inverse kinematics
> Plot, to tell estimator faults from controller faults
> ```
>
> Calibration comes first because the filter depends on it; the controller sits on top of the estimate; the plots let you assign blame.

The task: navigate on the estimate

Reach the green target, which is 105 across and 105 up from the start, using only your own estimate. Print `my x:` and `my y:` at the end, within 12 cm of the truth. `position()` is not allowed anywhere.

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

DT = 0.1
TX, TY = 105.0, 105.0
```

The hint students can ask for: The target's centre is 105 across and 105 up from the start, and nothing about it is visible to the robot. Tag 9 is straight up the mat from the start, and what it gives you is a range, so it only measures y while the robot is on the tag's line. Drive up that line first, take your fixes there, then turn for the target.

A solution

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

DT = 0.1
V_MAX, V_LAT = 20.0, 15.0
TX, TY = 105.0, 105.0
TAG_Y = 155.0
Q, R_TAG = 0.6, 9.0

rates = []
for i in range(40):
    rates.append(imu()[1])
    wait(DT)
bias = sum(rates) / len(rates)

x = y = h = 0.0
p = 4.0
set_cv("apriltag")

def step(n=1):
    global x, y, h, p
    for i in range(n):
        vx, vy = flow()
        rate = imu()[1] - bias
        a = math.radians(h + 0.5 * rate * DT)
        x += (vx * math.cos(a) + vy * math.sin(a)) * DT
        y += (-vx * math.sin(a) + vy * math.cos(a)) * DT
        h += rate * DT
        p += Q
        seen = [t for t in apriltags() if t[0] == 9]
        # the tag's range is only a measurement of y while we are on its line
        if seen and seen[0][3] < 130 and abs(x) < 12:
            k = p / (p + R_TAG)
            y += k * ((TAG_Y - seen[0][3]) - y)
            p = (1 - k) * p
        plot("x", x)
        plot("y", y)
        wait(DT)
```

```
def go_to(tx, ty, ticks=400):
    for tick in range(ticks):
        dx, dy = tx - x, ty - y
        gap = math.hypot(dx, dy)
        if gap < 5:
            break
        speed = min(13.0, 0.6 * gap)
        wx, wy = speed * dx / gap, speed * dy / gap
        a = math.radians(h)
        drive(100 * (wx * math.sin(a) + wy * math.cos(a)) / V_MAX,
              100 * (wx * math.cos(a) - wy * math.sin(a)) / V_LAT, 0)
        step()
    stop()
    step(3)

go_to(0.0, TY)        # up the tag's line first, taking fixes on the way
go_to(TX, TY)         # then across to the target

print("my x:", round(x, 1))
print("my y:", round(y, 1))
```

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