ANSWERS

U10.1 A path and a trajectory

Following a trajectory · University · about 25 min

What this lesson is about

The same geometry with and without a clock attached, and why the clock changes what you can check.

- Arc length is the natural parameter
- The time law
- Why a moving target beats a fixed one
- Feasibility

Questions

6 marks in all

1. What is the practical reason for keeping the path and the time law separate? [1 mark]

- ☐ **A** The route can be re-timed for a low battery or a delicate payload without planning the geometry again
- ☐ **B** A trajectory cannot be checked for collisions, so the path must be kept for that
- ☐ **C** Arc length only exists for a path, not for a trajectory
- ☐ **D** The planner needs the time law before it can search the map

> **Answer:** A. Planning the geometry is expensive and needs the map; putting a clock on it is cheap and needs only the machine. Keeping them apart means the cheap part can be redone on its own.

2. What does this print? It is the lesson's route with a constant speed time law. [1 mark]

```
import math
WAYPOINTS = [(40, 40), (40, 140), (140, 140), (140, 60)]
CRUISE = 12.0
length = 0.0
for (ax, ay), (bx, by) in zip(WAYPOINTS, WAYPOINTS[1:]):
    length += math.hypot(bx - ax, by - ay)
print(round(length, 1), round(length / CRUISE, 1))
```

> **Answer:**
>
> ```
> 280.0 23.3
> ```
>
> The legs are 100, 100 and 80 cm, so the path is 280 cm long, and at 12 cm/s that takes 280 / 12 = 23.3 s.

3. Where does the reference say the robot should be 10 s into the route? [1 mark]

```
import math
WAYPOINTS = [(40, 40), (40, 140), (140, 140), (140, 60)]
CRUISE = 12.0
legs, length = [], 0.0
for (ax, ay), (bx, by) in zip(WAYPOINTS, WAYPOINTS[1:]):
    d = math.hypot(bx - ax, by - ay)
    legs.append((ax, ay, bx, by, d))
    length += d

def at(t):
    s = max(0.0, min(length, CRUISE * t))
    for ax, ay, bx, by, d in legs:
        if s <= d:
            u = s / d
            return ax + (bx - ax) * u, ay + (by - ay) * u
        s -= d
    return WAYPOINTS[-1]

print(at(10.0))
```

Answer:

```
(60.0, 140.0)
```

At 10 s the arc length is 120 cm. The first leg uses 100 of it, so the point is 20 cm along the second leg, from (40, 140) towards (140, 140): (60, 140).

4. A robot drives at waypoint 1 until it is close, then at waypoint 2, and so on. Why does it stop and start at every waypoint? [1 mark]

- ☐ **A** Its demand is proportional to an error that collapses to nothing as it arrives at each waypoint
- ☐ **B** The planner inserted a pause at every waypoint
- ☐ **C** The drive cannot turn a corner while moving
- ☐ **D** The arc length is reset to zero at each waypoint

Answer: A. Chasing a fixed point drives the error, and so the command, to zero at every waypoint. A reference that moves along the path keeps a demand in front of the robot the whole way.

5. Which of these can be checked on a trajectory but not on the path alone? [1 mark]

Tick every answer that is true.

- ☐ **A** Whether any part demands more speed than the robot has
- ☐ **B** Whether it demands more acceleration than the drive can produce
- ☐ **C** How long the whole route will take
- ☐ **D** Whether the route passes through an obstacle
- ☐ **E** How long the route is in centimetres

Answer: A, B, C. Speed, acceleration and duration all need a clock. Collisions and length are properties of the geometry, which the path already answers.

6. Why does the lesson call the constant speed time law s(t) = v t a lie? [1 mark]

- ☐ **A** It jumps from rest to cruise speed instantly, which would need infinite acceleration
- ☐ **B** It gives the wrong total duration for the route
- ☐ **C** Arc length cannot be proportional to time
- ☐ **D** It only works on straight legs

> **Answer:** A. The duration and positions are fine as a plan, but no machine reaches cruise speed in zero time. U10.2 replaces it with a profile the machine can follow.

The task: put a clock on a path

Print `length:`, the total length of the path through the four waypoints, and `duration:`, how long it takes at 12 cm/s. Then sample the trajectory every 0.1 s and plot `ref x` and `ref y`, with a `wait(0.1)` between samples so the chart spreads them out in time. The robot does not move in this task.

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

WAYPOINTS = [(40, 40), (40, 140), (140, 140), (140, 60)]
CRUISE = 12.0
```

The hint students can ask for: The path is the four waypoints joined by straight lines, and its length is the sum of the legs. The trajectory is the same geometry with a rule for where you are at time t: travel at the cruise speed, work out the arc length so far, and walk the legs until you find the one it lands on.

A solution

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

WAYPOINTS = [(40, 40), (40, 140), (140, 140), (140, 60)]
CRUISE = 12.0            # cm/s

# the path: geometry, and nothing about when
legs = []
length = 0.0
for (ax, ay), (bx, by) in zip(WAYPOINTS, WAYPOINTS[1:]):
    d = math.hypot(bx - ax, by - ay)
    legs.append((ax, ay, bx, by, d))
    length += d
print("length:", round(length, 1))

# the trajectory: the same geometry with a clock on it
duration = length / CRUISE
print("duration:", round(duration, 1))

def at(t):
    """where the reference is at time t"""
    s = max(0.0, min(length, CRUISE * t))
    for ax, ay, bx, by, d in legs:
        if s <= d:
            u = s / d if d else 0.0
            return (ax + (bx - ax) * u, ay + (by - ay) * u)
        s -= d
    return (WAYPOINTS[-1][0], WAYPOINTS[-1][1])

for i in range(int(duration / 0.1) + 1):
    x, y = at(i * 0.1)
    plot("ref x", x)
    plot("ref y", y)
    wait(0.1)
```

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