ANSWERS

U11.6 Visual servoing

Vision · University · about 35 min

What this lesson is about

Closing the loop on pixels instead of converting to metres first, and why that is more robust.

- Two families
- The two loops on this robot
- The interaction matrix
- Gains, dead bands and losing sight

Questions

7 marks in all

1. What distinguishes image based visual servoing (IBVS) from position based (PBVS)? [1 mark]

☐ **A** IBVS defines the error in pixels and never converts to metres; PBVS estimates the target's pose in the world and controls to it

☐ **B** IBVS needs an accurate calibration; PBVS does not

☐ **C** IBVS uses a depth sensor; PBVS uses a camera

☐ **D** IBVS controls rotation only; PBVS controls translation only

> **Answer:** A. PBVS inherits every error in the calibration and pose estimate. IBVS closes the loop on the image quantity you actually care about.

2. The forward loop should stop when a 6 cm ball is 25 cm away. What apparent width in pixels is the setpoint, to 1 decimal place, with f = 92.4? [1 mark]

> **Answer:** 22.2 (accept within 0.1). w = f S / d = 92.4 x 6 / 25 = 22.2 pixels.

3. The loop from the last question, with its 22.2 pixel setpoint, is pointed at a ball that is really 8 cm across. At what range does the robot stop, in cm to 1 decimal place? [1 mark]

> **Answer:** 33.3 (accept within 0.2). It stops when the ball looks 22.18 pixels wide: d = 92.4 x 8 / 22.18 = 33.3 cm. The setpoint is wrong but the loop is still stable, which is the robustness IBVS buys.

4. This is the lesson's rotation law for several column errors. What does it print? [1 mark]

```
for err in (40, 10, 4, -30, 200):
    rot = max(-55, min(55, 0.35 * err))
    if abs(err) < 6:
        rot = 0
    elif abs(rot) < 17:
        rot = 17 * (1 if rot > 0 else -1)
    print(err, round(rot))
```

Answer:

```
40 17
10 17
4 0
-30 -17
200 55
```

Small errors are pushed up to the 17 percent dead band, errors under 6 pixels give nothing, and 0.35 x 200 = 70 is clamped to 55.

5. Why is rotating to centre a target the most reliable visual control there is? [1 mark]

- ☐ **A** The rotational terms of the interaction matrix have no depth in them, so no range estimate is needed
- ☐ **B** Rotation is faster than translation on the BugBot
- ☐ **C** Turning has no dead band
- ☐ **D** The principal point is always exactly at column 160

Answer: A. Every translational term carries 1 / Z, so forward control needs at least a rough depth for its gains. Rotation does not.

6. A real camera pipeline has 100 ms of latency and the robot drives at 20 cm/s. How far does it move before it acts on a frame, in cm? [1 mark]

Answer: 2 (accept within 0.01). 0.1 s x 20 cm/s = 2 cm. Latency is phase lag, so a vision loop tolerates less gain than one closed on an encoder.

7. The blob list comes back empty in the middle of an approach. What is the best behaviour? [1 mark]

- ☐ **A** Keep turning the way the target was last seen to go
- ☐ **B** Keep driving with the last command
- ☐ **C** Stop and wait for the ball to reappear
- ☐ **D** Spin in a fixed direction until something is seen

Answer: A. A robot that remembers which side it lost the target on recovers in a fraction of a second. Driving on blindly or spinning the wrong way does not.

The task: servo on the pixels

A red ball, 6 cm across, is off to the robot's right and the robot is not facing it. Drive up and stop about 25 cm away, controlling rotation from the blob's column and forward speed from its apparent width. Plot `cx` and `width`, and print `width:`, the apparent width you stopped at. Neither `distance()` nor `position()` is allowed.

```
from bugbot import *
connect()

F, R = 92.4, 3.0
STANDOFF = 25.0
set_cv("blob", "red")
wait(0.3)
```

The hint students can ask for: Two errors, both in pixels, and neither converted to metres. The horizontal one is the blob's centre against 160 and it drives rotation. The other is the blob's width against the width it would have at the standoff you want, and it drives forward speed. The ball is 6 cm across, so work out what width 25 cm of range looks like. Remember the dead band: a command under about 15 does nothing at all, so a proportional law alone will stall short of the target.

A solution

```
from bugbot import *
connect()

F = 92.4
R = 3.0
STANDOFF = 25.0
WANT = 2 * R * F / STANDOFF          # the width, in pixels, of a 6 cm ball 25 cm away

set_cv("blob", "red")
wait(0.3)

last, w = 1.0, 0.0
for tick in range(400):
    seen = blobs()
    if not seen:
        drive(0, 0, 25 * last)           # lost it: sweep back the way it went
        wait(0.1)
        continue
    cx, cy, area, x0, y0, x1, y1, aspect = seen[0]
    w = x1 - x0
    err = cx - 160

    last = 1.0 if err > 0 else -1.0
    rot = max(-55, min(55, 0.35 * err))
    if abs(err) < 6:
        rot = 0
    elif abs(rot) < 17:
        rot = 17 * (1 if rot > 0 else -1)

    gap = WANT - w
    fwd = max(-40, min(45, 4.0 * gap))
    if abs(gap) < 1.5:
        fwd = 0
    elif abs(fwd) < 18:
        fwd = 18 * (1 if fwd > 0 else -1)
```

```
    if abs(err) > 60:
        fwd = 0                              # turn towards it before closing in

    drive(fwd, 0, rot)
    plot("cx", cx)
    plot("width", w)
    wait(0.1)
    if fwd == 0 and rot == 0:          # both errors inside their bands: arrived
        break
stop()
wait(0.5)
seen = blobs()
if seen:
    w = seen[0][5] - seen[0][3]
print("width:", w)
```

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