3006ICT Robotics and Computer Vision

Week 8 Practical Book

Part 2: Build the Webots Navigation Pipeline

This chapter takes you through Workshop 8 Parts 2-6.

The aim is to understand how a robot turns sensor measurements into decisions, wheel commands, and eventually a planned route.

Several of these ideas are directly useful when you build the 60% group project.

Workshop	Main idea	What you should be able to explain
Part 2	Differential-drive motion	How two wheel speeds create translation and rotation.
Part 3	Camera + red target	How pixels become a target location using HSV segmentation and moments.
Part 4	Closed-loop following	How image error becomes steering and why feedback is necessary.
Part 5	Obstacle avoidance	How safety overrides the navigation objective through behaviour priority.
Part 6	A* planning	How an occupancy grid becomes a collision-free sequence of cells/waypoints.

Before You Start

Workshop 8 is deliberately structured as a progression. You first learn to move the robot, then extract information from its camera, then use that information to steer, then add a safety layer, and finally move from purely reactive behaviour to global path planning.

The official workshop describes this progression as camera → target detection → steering → obstacle avoidance → robot motion, together with occupancy grid → A* → waypoints.

> Part 4 reuses Parts 2-3.
>
> Part 5 reuses Parts 2-4.
>
> Part 6 introduces a different level of reasoning: a planner produces a route, but another controller is still needed to execute that route.
>
> This layered structure is exactly the kind of integration thinking you will need in the group project.

Submission

- Python code for Parts 2-6.
- A separate Result.pdf or Result.docx containing screenshots and results from Parts 2–6.
- Answers to all discussion questions. These are optional for marking, but they are useful for checking your understanding.

> **Important project connection**
>
> The group project is open-ended: it does not prescribe one navigation or vision method. However, it requires the RGB camera to determine which observation station contains the requested target, followed by safe autonomous navigation to the associated observation position.
>
> The concepts in this chapter are therefore building blocks.

Part 2: Control the Differential-Drive Robot

The e-puck is a differential-drive robot: the left and right wheels are independently controlled. This is one of the most important robotics ideas in the workshop because every higher-level behaviour eventually has to become a pair of wheel commands.

Wheel command	Typical motion	Why
left = right > 0	Forward	Both wheels move the robot in the same direction.
left < right	Turn left	The right side moves faster, causing the robot to curve left.
left > right	Turn right	The left side moves faster, causing the robot to curve right.
left < 0, right > 0	Rotate approximately in place	The wheels drive in opposite directions.
left = right = 0	Stop	No commanded wheel motion.

Why velocity control is used

First put both wheel motors into velocity-control mode by setting their target positions to infinity. In Webots, this means that the motor is not trying to move to a finite angular position; instead, we control its rotational velocity.

Core motor command:

```
def set_speed(left_motor, right_motor, left, right):
    left = max(-MAX_SPEED, min(MAX_SPEED, left))
    right = max(-MAX_SPEED, min(MAX_SPEED, right))
    left_motor.setVelocity(left)
    right_motor.setVelocity(right)
```

The important idea is the separation between a high-level decision and the low-level actuator interface. Your navigation algorithm should be able to say 'turn left moderately' without repeating the motor API details everywhere. That is why you will need to wrap the two motor commands inside set_speed().

Safe wheel-speed limits

Sample set_speed() function

```
MAX_SPEED = 6.28

def set_speed(left_motor, right_motor, left, right):
    left = max(-MAX_SPEED, min(MAX_SPEED, left))
    right = max(-MAX_SPEED, min(MAX_SPEED, right))
    left_motor.setVelocity(left)
    right_motor.setVelocity(right)
```

The clamp is important. The requested speed is first restricted to the range -6.28 to +6.28 rad/s, then sent to the two motors. This gives the rest of the controller a safe interface: higher-level code can request a speed without accidentally exceeding the chosen motor limit.

> **Engineering pattern: protect the actuator boundary.**
>
> A good controller has one place where wheel commands are limited. This prevents different behaviours from using inconsistent limits and makes later tuning much easier.

Run a motion for a fixed simulation time

Sample run_for() helper

```
def run_for(robot, timestep, left_motor, right_motor, left, right, seconds=1.0):
    set_speed(left_motor, right_motor, left, right)
    end = robot.getTime() + seconds
    while robot.step(timestep) != -1 and robot.getTime() < end:
        pass
```

run_for() demonstrates a useful Webots pattern: issue a command, then repeatedly call robot.step(timestep) until the requested simulation time has elapsed. Webots advances the simulation only when step() is called. Therefore, a controller should not simply call time.sleep() and expect the simulated robot to progress.

Follow the workshop test sequence

Sample test sequence

```
def main():
    robot, timestep, left_motor, right_motor, _, _ = setup()

    for left, right in [(2, 2), (1, 3), (3, 1), (-2, 2), (0, 0)]:
        run_for(robot, timestep, left_motor, right_motor, left, right)
```

1. Run the controller and observe the forward command (2, 2).
2. Observe (1, 3): the wheel speeds are unequal, so the robot follows a curved path.
3. Observe (3, 1): the curvature reverses.
4. Observe (-2, 2): the wheels rotate in opposite directions, producing an approximately in-place turn.
5. Observe (0, 0): the robot stops.
6. Reset and repeat if necessary. Do not move on until you can predict the motion before pressing Run.

What changes the turning behaviour?

For this workshop, the simplest model is: the larger the difference between left and right wheel speeds, the stronger/faster the turn. Equal wheel speeds produce approximately straight motion because the two sides are driven symmetrically. The exact trajectory also depends on the robot geometry and wheel dynamics, so think of these commands as motion primitives rather than exact geometric guarantees.

> **STOP is a behaviour, not an afterthought.** A robot should have a reliable stop state for normal completion, uncertain perception, safety events, and debugging. In later parts, stopping is also important when the robot has reached its goal.

Part 3: Read the Camera and Detect the Red Target

Part 3 introduces the perception side of the robot. Instead of commanding the robot blindly, we obtain an image, extract pixels likely to belong to the red target, and reduce those pixels to a compact measurement: the target centroid (u, v).

The camera data path

The e-puck camera returns an image buffer. You will need to convert that buffer into a NumPy array with four channels and then converts BGRA to the BGR representation expected by the OpenCV operations.

Sample camera conversion

```
def camera_bgr(camera):
    h, w = camera.getHeight(), camera.getWidth()
    image = np.frombuffer(camera.getImage(), np.uint8).reshape(h, w, 4)
    return cv2.cvtColor(image, cv2.COLOR_BGRA2BGR)
```

> **Why reshape?** The raw camera buffer is a one-dimensional byte sequence. reshape(h, w, 4) tells NumPy how to interpret those bytes as an image with height h, width w, and four channels.

Why HSV instead of direct RGB thresholding?

You need to convert the image from BGR to HSV. HSV separates colour information (hue) from saturation and brightness/value. For this simple workshop target, that makes it convenient to describe a 'red-looking' region using ranges in HSV space.

Sample red segmentation

```
def detect_red_target(image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    m1 = cv2.inRange(hsv, (0, 100, 80), (10, 255, 255))
    m2 = cv2.inRange(hsv, (170, 100, 80), (180, 255, 255))
    mask = m1 | m2
```

Red is represented near both ends of the OpenCV hue scale. Therefore you will use two masks: one near hue 0 and one near hue 180, then combines them with a bitwise OR. Saturation and value lower bounds reject many weakly coloured or dark pixels.

Parameter	Meaning	Role
H	Hue / colour type	Select the red hue regions.
S	Saturation	Reject pixels that are insufficiently coloured.
V	Value / brightness	Reject very dark pixels.
100 / 80	Lower S/V thresholds	Starting thresholds; not universal constants.
0-10 and 170-180	Two red hue intervals	Covers both ends of the hue scale.

From a binary mask to a target centroid

Sample centroid calculation

```
    m = cv2.moments(mask)
    if m["m00"] < 100:
        return False, None, None, mask

    u = int(m["m10"] / m["m00"])
    v = int(m["m01"] / m["m00"])
    return True, u, v, mask
```

cv2.moments() calculates image moments from the binary mask. The quantity m00 is related to the area of the detected region, while m10 and m01 encode the first-order spatial moments. Dividing m10 and m01 by m00 gives the centroid coordinates.

> **The key reduction is important.** Thousands of pixels are reduced to two numbers: u tells you where the target is

horizontally; v tells you where it is vertically. Later, Part 4 uses u to control steering.

Why reject tiny detections?

A tiny red region could be noise, a reflection, or an accidental red pixel. The area threshold therefore acts as a simple confidence filter (requires m00 to be at least 100). It is a starting point, not a guarantee that the detection is correct.

Convert the centroid into a steering-relevant label

Sample LEFT / CENTRE / RIGHT classification

```
def target_side(detected, u, width):
    if not detected:
        return "NOT FOUND"
    if u < width / 3:
        return "LEFT"
    if u > 2 * width / 3:
        return "RIGHT"
    return "CENTRE"
```

The image width is divided into three horizontal regions. This is deliberately simple: the robot does not yet need a precise target distance. It only needs to know whether the target is to the left, near the centre, or to the right.

Limitation: this is a workshop detector

Colour thresholding can fail under changed illumination, shadows, reflections, background colours, or objects with similar colours. Note that a trained detector, segmentation model, or visual tracker could replace the colour thresholding approach.

Group-project connection: The group project is more difficult than this red-target exercise. The project requires the camera to determine which observation station displays the requested target, in the presence of distractor images, and the target-to-station assignment can change between missions. Therefore, do not assume that the Part 3 red-mask method is sufficient for the project.

Part 4: Closed-Loop Visual Target Following

Part 4 is the key integration step: the camera measurement now changes the robot's motion continuously. This is a feedback controller. The robot observes, computes an error, acts, observes again, and corrects again.

The feedback loop

Stage	Input	Output
1. Sense	Current camera frame	Target detected? Centroid u, v
2. Measure error	u and image width	Normalised horizontal error
3. Compute control	Error and kp	Turn command
4. Act	Base speed + turn	Left/right wheel velocities
5. Repeat	New camera frame after motion	New correction

Sample follow_target() controller

```
def follow_target(left_motor, right_motor, u, width, base_speed=2.0, kp=1.5):
    error = (u - width / 2) / (width / 2)
    turn = kp * error
    set_speed(left_motor, right_motor, base_speed + turn, base_speed - turn)
```

The normalised error is useful because it scales with image width. When the target is at the image centre, u = width/2 and the error is approximately zero. A target to the left produces a negative error; a target to the right produces a positive error.

Read the sign before memorising the formula. Ask: 'If the target moves to the right side of the image, what should the robot do?' In this workshop's controller, positive error increases the left-wheel command and decreases the right-wheel command, producing the corresponding turn.

What is proportional control?

The controller uses turn = kp × error. This is proportional control: the steering correction grows in proportion to the current image error. If the target is almost centred, the correction is small. If the target is far from centre, the correction is larger.

kp choice	Likely behaviour	Typical problem
Too small	Gentle response	Robot reacts slowly and may take a long time to centre the target.
Reasonable	Smooth correction	Target is approached without excessive oscillation.
Too large	Aggressive response	Overshoot, oscillation, unstable-looking steering.

Why is closed-loop control necessary?

A single image only tells you what the target looked like at one instant. After the robot moves, the target's image position changes. The controller therefore has to re-measure the error repeatedly. This is the difference between an open-loop command and feedback control.

Sample SEARCH behaviour

```
def search(left_motor, right_motor):
    set_speed(left_motor, right_motor, -1.0, 1.0)
```

If the target disappears, the controller rotates slowly rather than continuing to drive forward blindly. This is a simple search behaviour. It is useful because perception is intermittent: the target can leave the camera field of view during a turn.

The complete Part 4 loop

Sample Part 4 main loop

```
def main():
    robot, timestep, left_motor, right_motor, camera, _ = setup()

    while robot.step(timestep) != -1:
        image = camera_bgr(camera)
        detected, u, _, _ = detect_red_target(image)

        if detected:
            follow_target(left_motor, right_motor, u, camera.getWidth())
        else:
```

1. Read the current camera image.
2. Detect the target.
3. If detected, calculate the steering command from the target's horizontal location.
4. If not detected, switch to SEARCH.
5. Repeat on the next Webots timestep.

A subtle issue: 'target centred' is not the same as 'target reached'

Part 4 only controls horizontal image alignment. A target can be visually centred while still being far away. For a real navigation task, you need additional information such as target size, depth/distance estimation, robot pose, a known observation position, or another stopping condition. This distinction becomes important in the group project.

Part 5: Obstacle Avoidance and Behaviour Priority

Part 5 introduces a second source of information: proximity sensors. The robot now has two competing objectives: follow the target and avoid collisions. The important lesson is not only how to read the sensors, but how to decide which behaviour gets priority.

Read the e-puck proximity sensors

Sample obstacle_state()

```
def obstacle_state(ps, threshold=80.0):
    values = [sensor.getValue() for sensor in ps]
    right = any(values[i] > threshold for i in (0, 1, 2))
    left = any(values[i] > threshold for i in (5, 6, 7))
    return left, right, values
```

In this workshop, we group ps0-ps2 as the right/front-right side and ps5-ps7 as the left/front-left side. A side is considered 'blocked' if any sensor in that group exceeds the chosen threshold.

> **Thresholds should be measured, not guessed.** The workshop recommends starting around 80 and printing the actual readings near obstacles. Sensor values depend on the simulated geometry and setup. Treat 80 as an initial tuning value, not a universal physical constant.

From raw readings to a behaviour state

Notice the abstraction: the controller does not make every later function reason about eight independent sensor numbers. obstacle_state() converts raw measurements into two higher-level facts: left blocked and right blocked. This is a useful software-engineering pattern for robotics.

Raw information	Intermediate representation	Decision
ps0-ps7 values	left blocked? right blocked?	Avoid / follow / search
Camera pixels	target detected + centroid	Follow / search
Wheel limits	clamped left/right speeds	Safe actuator command

Implement the avoidance action

Sample avoid_obstacle()

```
def avoid_obstacle(left_motor, right_motor, left, right):
    if left and right:
        set_speed(left_motor, right_motor, -2.0, 2.0)
    elif left:
        set_speed(left_motor, right_motor, 2.0, -2.0)
    elif right:
        set_speed(left_motor, right_motor, -2.0, 2.0)
```

In this workshop, we use simple rotations. If both sides are blocked, it rotates one direction. If only the left side is blocked, it rotates away from that side; similarly for the right side. This is intentionally a simple reactive policy for the workshop.

Sample Part 5 decision structure

```
def main():
    robot, timestep, left_motor, right_motor, camera, ps = setup()

    while robot.step(timestep) != -1:
        image = camera_bgr(camera)
        detected, u, _, _ = detect_red_target(image)
        left, right, _ = obstacle_state(ps)
```

```
        if left or right:                          # highest priority
            avoid_obstacle(left_motor, right_motor, left, right)
        elif detected:
            follow_target(left_motor, right_motor, u, camera.getWidth())
        else:
            search(left_motor, right_motor)
```

This if/elif/else structure is one of the most important pieces of code in the workshop. It defines a priority hierarchy:

1. SAFETY: If an obstacle is detected, avoid it immediately.
2. TARGET FOLLOWING: Only if the path is currently safe and the target is visible, steer toward the target.
3. SEARCH: Only if the path is safe and the target is not visible, search for it.

Why priority matters? A robot that continues to chase a visually attractive target while an obstacle is directly ahead is not autonomous in a useful sense. It is optimising the wrong objective. Safety should be able to override the navigation objective.

Parts 4-5 are already becoming a small behaviour-based controller. Each behaviour has a condition and an action:

Behaviour	Condition	Action
Avoid obstacle	left or right blocked	Turn away / rotate
Follow target	target detected and path clear	Proportional steering
Search	target not detected and path clear	Slow rotation
Stop	goal reached / failure / completion	Set both wheel speeds to zero

Why the workshop controller is not yet a complete project controller

The controller demonstrates local reactive navigation in this workshop. It does not solve the group-project problem of identifying one of eight possible visual targets among stations and then reaching the corresponding designated observation position. A project solution may need additional stages such as station recognition, target-reference matching, localisation, global route selection, waypoint following, goal detection, and robust recovery.

Part 6: A* Path Planning on an Occupancy Grid

Part 6 changes level. Parts 3–5 react to what the robot sees now. A* reasons about a known map and computes a route from a start cell to a goal cell. This is global planning.

Understand the occupancy grid

Grid value	Meaning	Can the planned path use it?
0	Free space	Yes
1	Obstacle	No

The workshop uses a supplied grid with a provided start cell (10, 1) and goal cell (1, 10). In the group project, the supplied occupancy grid is 40 × 40 with 0.10 m cells. The map and pose resources may be used for navigation, but they do not reveal which station contains the requested visual target.

6.2 The four-connected grid

A 4-connected grid allows movement from a cell to its immediate neighbour above, below, left, or right. Diagonal moves are not included in this workshop.

Sample A* neighbour expansion and cost update

```
        r, c = current
        for dr, dc in ((-1, 0), (1, 0), (0, -1), (0, 1)):
            nxt = (r + dr, c + dc)

            if not (0 <= nxt[0] < grid.shape[0] and 0 <= nxt[1] < grid.shape[1]):
                continue
            if grid[nxt] == 1:
```

```
                continue

            new_g = g[current] + 1
            if new_g < g.get(nxt, float("inf")):
                g[nxt] = new_g
                parent[nxt] = current
                heapq.heappush(queue, (new_g + heuristic(nxt, goal), nxt))
```

What are g(n), h(n), and f(n)?

Quantity	Meaning	In this implementation
g(n)	Cost accumulated from start to n	Number of 4-connected steps taken.
h(n)	Estimated remaining cost from n to goal	Manhattan distance.
f(n)	Total priority estimate	g(n) + h(n).

Sample Manhattan heuristic

```
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

The Manhattan heuristic is appropriate for a 4-connected grid because it counts the horizontal and vertical steps still required if there were no obstacles. It does not need to be an exact future path length; its role is to guide the search toward promising nodes.

Understand the priority queue

Sample A* initialisation and main search loop

```
def astar(grid, start, goal):
    queue = [(0, start)]
    parent = {}
    g = {start: 0}

    while queue:
        _, current = heapq.heappop(queue)
        if current == goal:
            break
```

heapq provides a priority queue. The algorithm repeatedly removes the currently most promising node. The tuple pushed into the queue is (f, node), so nodes with smaller f(n) are considered first.

Why record parent nodes?

Finding the goal is not enough. The robot needs the complete route. The parent dictionary records which previous cell produced the best-known route to each new cell.

Sample path reconstruction

```
    if goal not in g:
        return []

    path = [goal]
    while path[-1] != start:
        path.append(parent[path[-1]])
    return path[::-1]
```

The path is reconstructed backwards from goal to start and then reversed. The result is a sequence of grid cells.

Visualise the result

Sample visualisation code

```
def main():
    grid = np.load(GRID_FILE)
    path = astar(grid, START, GOAL)

    print("Path length:", len(path))
    print("Path:", path)
```

```
    plt.imshow(grid, cmap="gray_r", vmin=0, vmax=1)
    rows, cols = zip(*path)
    plt.plot(cols, rows, "-o", markersize=3)
    plt.scatter([START[1]], [START[0]], marker="s", s=100, label="Start")
    plt.scatter([GOAL[1]], [GOAL[0]], marker="*", s=140, label="Goal")
    plt.legend()
    plt.title("A* Path")
    plt.grid(True, linewidth=0.4)
    plt.tight_layout()
    plt.savefig("week8_astar_path.png", dpi=150)
```

The visualisation is an important debugging tool. Do not judge A* only from the printed path length. Plot the path and inspect whether it travels through free cells and whether the start and goal are where you expect.

A* produces waypoints — not wheel commands

> A* answers: 'Which cells should the robot pass through?'
>
> It does not answer: 'What left/right wheel velocities should I send right now?'
>
> A separate waypoint-following controller is required to convert route geometry into motion.

Planning layer	Control layer
Input: map + start + goal	Input: current robot pose + next waypoint
Output: cell/waypoint sequence	Output: wheel velocities
Usually updated less frequently	Updated continuously
Global reasoning	Local feedback

Why A* does not remove the need for sensors

The workshop explicitly highlights that local perception is still required after planning. A map can be imperfect, the robot can deviate from the planned path, and obstacles or simulation conditions may require immediate correction. Therefore, global planning and local control should be viewed as complementary.

Converting grid cells into physical waypoints

Conceptually, if a grid has a known cell size, a cell index can be mapped into a world coordinate. The exact origin and coordinate convention should come from the supplied Webots project rather than being guessed. Once a physical waypoint is known, the controller can use robot pose information to calculate a heading/distance error and generate wheel commands.

> **Project connection:** The group project supplies an occupancy grid, station coordinates, observation-position coordinates, GPS and orientation information. A group may use these resources for navigation. However, the requested target station should still be determined from RGB camera observations; map or station-coordinate information cannot be used as a substitute for visual target identification.

Integration

This workshop guideline is intentionally modular. Each part imports and reuses functions from earlier parts. This is a useful model for your own project software structure.

Why modular functions matter

Suppose you change the wheel-speed limit. With a single set_speed() function, the change is made in one place. Suppose you replace red segmentation with a better detector. If the detector still returns a simple interface such as detected, u, v, the higher-level controller can remain largely unchanged.

This is abstraction: each module hides implementation details behind a small, understandable interface.

The integrated reactive controller

Sample integration in Part 5

```
def main():
```

```
    robot, timestep, left_motor, right_motor, camera, ps = setup()

    while robot.step(timestep) != -1:
        image = camera_bgr(camera)
        detected, u, _, _ = detect_red_target(image)
        left, right, _ = obstacle_state(ps)

        if left or right:                          # highest priority
            avoid_obstacle(left_motor, right_motor, left, right)
        elif detected:
            follow_target(left_motor, right_motor, u, camera.getWidth())
        else:
            search(left_motor, right_motor)
```

Read the logic from top to bottom. The controller does not 'blend everything together' equally. Instead, it makes one decision according to a priority order. This makes the system easier to reason about and debug.

A useful architecture for your group project

Layer	Question it answers	Possible course concept
Mission input	What target identity should I find?	Provided mission configuration
Visual search	Which station actually displays it?	Camera + vision
Station confirmation	Do I have enough evidence to commit?	Multi-frame / confidence reasoning
Global planning	How should I reach its observation position?	Occupancy grid + A* or another planner
Waypoint control	How do I move toward the next waypoint?	Pose + differential-drive control
Local safety	Is it safe to continue right now?	ps0–ps7
Goal condition	Have I reached the designated observation position?	Pose + distance threshold
Stop/recovery	What should happen if perception or navigation fails?	STOP / search / recovery

This is a design framework, not a project solution. The project is explicitly open-ended. You may use different vision, planning, control, and software approaches. The important requirement is that the final system is autonomous, robust, reproducible, and obtains the target-station identity visually.

Appendix

Step-by-Step Workshop Procedure

Use this section during the workshop.

Step 1 — Prepare your controller folder

- Start from the provided Week 8 materials and keep the supplied folder structure unchanged.
- Confirm that Webots is using the intended Conda Python environment.
- Open the Week 8 navigation world and confirm the e-puck and devices are visible.
- Use a working copy for experimentation if you are worried about breaking your starter code.

Step 2 — Complete Part 2 before touching vision

- Configure both motors for velocity control.
- Write set_speed(left, right).
- Clamp both values to the chosen safe range.
- Test forward, left turn, right turn, in-place rotation, and stop.
- Record one screenshot of a meaningful motion result for your Result file.

Step 3 — Build Part 3 as a perception-only test

- Enable the camera.
- Convert the camera buffer into a BGR image.
- Create the HSV representation.
- Create the two red masks and combine them.
- Calculate image moments.
- Reject detections below the minimum area.
- Print the centroid and LEFT/CENTRE/RIGHT classification.
- Inspect failures before moving on.

Step 4 — Add closed-loop control

- Reuse camera_bgr() and detect_red_target().
- Calculate the normalised horizontal error.
- Start with kp = 1.5.
- Test a smaller kp and a larger kp.
- Compare smoothness, responsiveness, overshoot, and oscillation.
- Add SEARCH for target loss.
- Make sure every wheel command passes through the safe set_speed() function.

Step 5 — Add safety as a higher-priority layer

- Enable ps0-ps7.
- Print readings while approaching obstacles.
- Choose an initial threshold around 80.
- Tune it using actual observations.
- Create left/right blocked states.
- Implement avoidance.
- Place avoidance before target following in the decision logic.
- Test at least two obstacle configurations.

Step 6 — Complete A* separately

- Load the occupancy grid.
- Display the grid so you understand its geometry.
- Set the provided start and goal cells.
- Implement Manhattan distance.
- Expand four-connected neighbours.
- Reject out-of-bounds and obstacle cells.
- Update g-values and parent pointers.
- Reconstruct the path.
- Plot the path and inspect it.
- Explain how cells could become physical waypoints.

Debugging Guide

When the robot behaves badly, avoid immediately changing many parameters.

Identify which layer is wrong. A navigation system can fail because perception is wrong, control is wrong, safety is wrong, or planning is wrong.

Symptom	Check first	Likely layer
Robot does not move	Motor names, position mode, step loop	Actuation / Webots setup
Robot turns opposite to expectation	left/right mapping and sign convention	Differential-drive control

Symptom	Check first	Likely layer
No red target detected	camera image, HSV ranges, m00 threshold	Vision
Target detected but robot oscillates	kp, image error sign, wheel clamping	Feedback control
Robot loses target repeatedly	SEARCH speed and camera field of view	Perception + behaviour
Robot collides while following target	Sensor threshold and priority order	Safety
Avoidance turns the wrong way	left/right sensor grouping and wheel signs	Safety + actuation
A* path goes through obstacle	grid values, neighbour checks, visualisation	Planning
A* path looks correct but robot cannot follow it	cell-to-world conversion and waypoint controller	Planning → control integration

Use diagnostic printouts strategically

- Print camera resolution once, not every timestep.
- Print target centroid periodically rather than flooding the console.
- Print proximity values when tuning thresholds.
- Print controller state changes such as SEARCH → FOLLOW or FOLLOW → AVOID.
- Print A* path length and the final path during offline testing.

> **Good debugging question:** Do not ask only 'Why did the robot crash?' Ask 'What did the robot believe at the moment before the crash, and which command did that belief produce?'

Using AI Tools Responsibly

AI can be useful for explaining unfamiliar Python/Webots APIs, suggesting debugging hypotheses, or helping you compare alternative implementations.

However, the course project requires you to understand and be able to explain your submitted system, and the course materials state that any use of generative AI must comply with course instructions and Griffith University guidance.

Good use	Poor use
Ask why setPosition(float('inf')) is used.	Paste an entire controller into AI and submit it without understanding it.
Ask AI to explain an OpenCV moment calculation.	Ask AI to invent a result or screenshot you did not obtain.
Ask for debugging hypotheses after observing a failure.	Claim AI-generated code as independently developed if disclosure is required.
Ask AI to compare two control strategies, then test them yourself.	Use AI to bypass the requirement to understand the complete system.

> **Rule for your project:** If you cannot explain what a piece of code does, why you need it, what assumptions it makes, and how you tested it, it is not yet part of a defensible engineering solution.

Discussion questions

Use these as learning explanations.
You should be able to express the same ideas in your own words.

Part 2 — Q1: Why do equal left and right wheel speeds produce approximately straight motion?

Answer: Both sides of the differential-drive robot are commanded symmetrically, so there is approximately no turning bias and the robot moves forward along a near-straight path.

Part 2 — Q2: How does increasing the wheel-speed difference affect turning?

Answer: A larger difference between left and right wheel speeds produces stronger curvature and therefore a faster/stronger turn.

Part 2 — Q3: Why should a controller always have STOP behaviour?

Answer: STOP provides a safe state for normal completion, uncertainty, failures, and emergency situations.

Part 3 — Q1: Why is the image centre useful for steering?

Answer: The controller can treat the image centre as the desired horizontal target location. The difference between the target centroid and that centre becomes a steering error.

Part 3 — Q2: Why can colour segmentation fail?

Answer: Lighting, shadows, reflections, camera effects, backgrounds, and similar colours can change the pixel values and produce missed or false detections.

Part 3 — Q3: What could replace colour segmentation?

Answer: A trained object detector, segmentation model, or visual tracker could provide a more general perception method.

Part 4 — Q1: Why must target following operate as a closed loop?

Answer: Robot motion changes the target's image position. Repeated sensing lets the controller correct its action using the latest measurement.

Part 4 — Q2: What happens when kp is too large?

Answer: The steering correction becomes too aggressive, which can cause overshoot and oscillation.

Part 4 — Q3: Why might the robot oscillate around the image centre?

Answer: The controller can repeatedly over-correct because of high gain, delayed/noisy measurements, or the robot's motion dynamics.

Part 5 — Q1: Why should obstacle avoidance have higher priority?

Answer: Avoiding collision is safety-critical. The robot must be able to override its navigation objective when continuing would be unsafe.

Part 5 — Q2: Why should the threshold be checked experimentally?

Answer: Sensor readings depend on distance, geometry, surface properties, and the simulation setup, so a threshold that works in one condition may not work in another.

Part 5 — Q3: Why combine multiple behaviours?

Answer: A robot needs different responses for different situations. Behaviour priority lets it pursue a goal while still reacting to safety-critical changes.

Part 6 — Q1: What does g(n) represent?

Answer: The accumulated path cost from the start to node n.

Part 6 — Q2: What does h(n) represent?

Answer: An estimate of the remaining cost from node n to the goal.

Part 6 — Q3: Why can A* search toward the goal more efficiently than uninformed search?

Answer: The heuristic gives the search a directional estimate of remaining cost, prioritising nodes that look promising for reaching the goal.

Part 6 — Q4: Why is an A* path not directly a motor command?

Answer: The path is a sequence of map cells or positions. A separate controller must convert those positions and the robot's current pose into wheel velocities.

Part 6 — Q5: Why does a robot still need local perception after A*?

Answer: The robot can deviate from the planned route, the map may not perfectly represent the current situation, and immediate safety decisions still require current sensor information.

From Workshop 8 to Group Project

The project is not simply 'repeat Workshop 8 with a different target'.

The workshop gives you reusable engineering concepts, while the project adds a harder mission-level requirement: the robot should visually determine which station contains the requested target, then reach that station's designated observation position, safely and autonomously, using the same submitted controller across missions.

Workshop concept	Project relevance	What should become more robust
Camera processing	Find station target	Move beyond a single red colour mask.
Centroid / image error	Orient toward visual evidence	Handle different target appearances and viewpoints.
Closed-loop steering	Move while observing	Add pose/goal reasoning and recovery.
Proximity safety	Avoid barriers	Tune and integrate with global navigation.
A* path	Navigate to a known destination	Convert route cells into physical waypoints and execute them.
Behaviour priority	Resolve competing objectives	Define search, confirmation, navigation, avoidance and stop states.
Modular code	Integrate a complete system	Keep perception, planning, control and safety separable and testable.

Project constraint to remember: The project allows the provided map, GPS, orientation, station information and occupancy grid for navigation. But the station containing the requested target must be determined from RGB camera observations. A fixed target-to-station mapping, simulator ground-truth recognition, .wbt parsing, or metadata-based shortcut is not allowed.

A sensible development order

1. Make the robot move reliably and stop reliably.
2. Make camera acquisition reliable and inspect actual images.
3. Build and validate target/station visual recognition independently of navigation.
4. Build waypoint following independently of target recognition.
5. Add proximity-based safety.
6. Integrate visual search with navigation.
7. Add target confirmation so one noisy frame does not immediately commit the mission.
8. Add recovery behaviours for target loss or route failure.
9. Test from multiple starts and with different target identities/arrangements.
10. Measure completion time, success/failure, and meaningful failure cases rather than reporting only one successful run.

Suggested Result File Structure

The workshop asks for screenshots and results from Parts 2-6.

A concise but useful Result document can use the following structure.

Section	Include
Part 2	One or two motion screenshots; wheel-speed settings; short observation.
Part 3	Camera/target screenshot; mask or detection result; centroid;

Section	Include
	short failure observation if useful.
Part 4	Following screenshot; kp values tested; comparison of behaviour.
Part 5	Obstacle configuration screenshots; sensor threshold; explanation of priority behaviour.
Part 6	Occupancy grid; A* path plot; path length; short explanation of g/h/f.
Discussion	Answers to all workshop questions.

Quick Reference

Concept	Core expression / idea
Motor velocity control	setPosition(float('inf')) + setVelocity(...)
Safe wheel command	clamp each wheel speed to the chosen limit
Camera image	camera.getImage() → NumPy → BGR
Red mask	HSV + two red hue ranges + bitwise OR
Centroid	u = m10/m00, v = m01/m00
Horizontal error	error = (u - width/2)/(width/2)
Proportional turn	turn = kp × error
Wheel steering	left = base + turn; right = base - turn
Search	slow rotation when target is not detected
Obstacle state	sensor reading > threshold → blocked
Priority	SAFETY > TARGET FOLLOWING > SEARCH
A* score	f(n) = g(n) + h(n)
Manhattan heuristic	|r1-r2| + |c1-c2|
A* output	sequence of grid cells / waypoints, not motor commands

Workshop Checklist

☐ Part 2: I can control and stop the e-puck using differential-drive wheel commands.
☐ Part 3: I can obtain a camera image and detect the workshop red target.
☐ Part 3: I can explain the centroid calculation rather than treating it as a black box.
☐ Part 4: I can explain image error, proportional control, SEARCH, and closed-loop feedback.
☐ Part 5: I can explain proximity thresholds and why safety overrides target following.
☐ Part 6: I can explain A*, g(n), h(n), f(n), parent reconstruction, and waypoints.
☐ I understand the difference between global planning and local control.
☐ I know which workshop components are directly reusable in the group project and which are only demonstrations.
☐ I have captured the required screenshots/results for Parts 2–6.
☐ I can explain every important piece of code I submit.