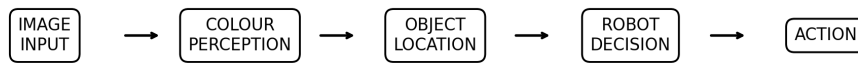


Getting Started with Vision-Enabled Robotics

What This Chapter Is For

This chapter follows the Week 1 workshop. The workshop asks you to build a small vision-enabled robotics pipeline: image input, coloured-object detection, object location and a simple robot action. You will not control a real robot or train a deep-learning model in this workshop.



Keep this pipeline visible while you work 😊

The key question

How can a robot turn pixels into information that is useful for deciding what to do?

What you should be able to do by the end

- Load and inspect a colour image with OpenCV.
- Explain $H \times W \times C$ and the BGR/RGB difference.
- Create a binary mask using HSV colour thresholding.
- Clean the mask using morphology.
- Find candidate regions with contours and select a target.
- Compute a bounding box and image-space centre.
- Turn the centre location into a simple action.
- Test failure cases and explain why perception errors can become action errors.

How to use the practice sections

Icon / section	What you do
READ	Understand the engineering idea before touching the code.
COPY & RUN	Use the complete code block as a working baseline.
PLAY	Change one parameter or line and observe what changes.
THINK	Answer the reasoning question before checking the answer.
DEBUG	Use the suggested diagnostic order instead of changing random lines.
EXTEND	Make a small improvement without rewriting the whole pipeline.

The supplied workshop solutions are reference implementations. They are useful because they show a clean modular structure; your job is to understand the responsibility of each function.

Part 1: Setup and Preparation

Workshop instruction: prepare a working Python environment and confirm that the required packages are available.

Recommended environment

Item	Workshop recommendation
Python	Python $\geq$ 3.9; the workshop setup example uses Python 3.11.
Environment	Miniconda / Anaconda
Packages	opencv-python / OpenCV, NumPy, Matplotlib
Tools	PyCharm, VS Code, Jupyter Notebook, Google Colab, or your familiar environment

Copy and run

```
conda create -n 3006ict python=3.11 -y
conda activate 3006ict
conda install opencv numpy matplotlib -c conda-forge
```

Quick environment check

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

print("OpenCV:", cv2.__version__)
print("NumPy:", np.__version__)
```

Engineering habit

If the program cannot import a package, do not start changing the vision algorithm. First isolate the environment problem.

Practice

Q1. Why is environment checking part of engineering rather than just administration?

Q2. What would you check first if cv2 cannot be imported?

Q3. Why is it useful to record the Python/package versions used for an experiment?

Part 2: Load and Display an Image

Workshop task: load 01_clear_red_cup_centre.png, display it, and print height, width and number of channels.

One colour image

$$I \in \mathbb{R}^{(H \times W \times C)}$$

H = height W = width C = channels

Example pixel

$$(B, G, R) = (0, 0, 255)$$

OpenCV stores colour as BGR.

A colour image is numerical data. Each pixel contains channel values.

Reference solution

```
import cv2
import matplotlib.pyplot as plt

from vision_utils import load_image

def load_and_display(image_path, figure_size=(10, 6)):
    image = load_image(image_path)

    height, width, channels = image.shape

    plt.figure(figsize=figure_size)
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()

    return {
        "height": height,
        "width": width,
        "channels": channels,
        "image": image,
    }

def main():
    result = load_and_display(
        "images/01_clear_red_cup_centre.png"
    )

    print("Height:", result["height"])
    print("Width:", result["width"])
    print("Channels:", result["channels"])

if __name__ == "__main__":
    main()
```

Key lines to understand

Line / function	Meaning	Why it matters
-----------------	---------	----------------

<code>cv2.imread()</code>	Reads image into memory.	The image becomes an array that later stages can process.
<code>image.shape</code>	Returns dimensions.	You need width/height for geometry and decisions.
<code>cv2.COLOR_BGR2RGB</code>	Converts OpenCV BGR for Matplotlib.	Without conversion, displayed colours may look wrong.
<code>plt.imshow()</code>	Displays the array.	Visual inspection is one of the fastest debugging tools.

Do not confuse these

The image's array representation is BGR when loaded by OpenCV. Matplotlib normally expects RGB for correct colour display. This is a representation issue, not a change to the physical image.

Try it

```
# Inspect one pixel.
pixel = image[100, 100]
print("BGR pixel:", pixel)

# Change the display conversion deliberately.
plt.imshow(image)          # Try this first.
plt.axis("off")
plt.show()

# Now display correctly.
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis("off")
plt.show()
```

Discussion questions — think first

Q1. What does each pixel represent?

Q2. Why is image classification alone insufficient for robot action?

Reference answers

- A pixel stores numerical intensity values for the image channels at one image location.
- Classification tells a robot what category may be present, but not where the object is. A robot needs location information to decide how to move, target or interact.

Part 3: Detect a Coloured Object

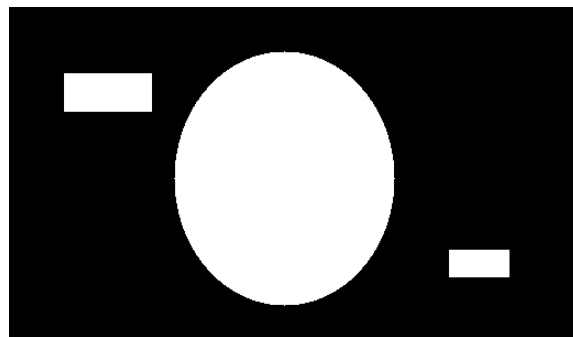
Workshop task: choose a target colour, define a colour range, create a binary mask, and display the mask.

Why HSV?

The workshop deliberately introduces HSV because direct RGB/BGR thresholding is brittle. For example, bright red, dark red and pink can look like the same colour to a person while having very different channel values.

Channel	Student-friendly interpretation
H — Hue	What colour family is this?
S — Saturation	How vivid/pure is the colour?
V — Value	How bright is it?

The mask



Illustrative binary mask: white = selected target pixels; black = background.

The workshop defines a binary mask as white for detected target pixels and black for background pixels.

Reference implementation

```
from pathlib import Path

import cv2
import numpy as np

def load_image(image_path):
    image = cv2.imread(str(Path(image_path)))

    if image is None:
        raise FileNotFoundError(image_path)

    return image

def create_red_mask(
    image,
    lower_1=(0, 80, 50),
    upper_1=(10, 255, 255),
    lower_2=(170, 80, 50),
    upper_2=(179, 255, 255),
    kernel_size=5,
):
    hsv = cv2.cvtColor(
```

```

        image,
        cv2.COLOR_BGR2HSV
    )

    mask_1 = cv2.inRange(
        hsv,
        np.array(lower_1),
        np.array(upper_1)
    )

    mask_2 = cv2.inRange(
        hsv,
        np.array(lower_2),
        np.array(upper_2)
    )

    mask = cv2.bitwise_or(
        mask_1,
        mask_2
    )

    kernel = np.ones(
        (kernel_size, kernel_size),
        np.uint8
    )

    mask = cv2.morphologyEx(
        mask,
        cv2.MORPH_OPEN,
        kernel
    )

    return cv2.morphologyEx(
        mask,
        cv2.MORPH_CLOSE,
        kernel
    )

```

Key code

1. ``cv2.cvtColor(..., COLOR_BGR2HSV)`` changes the representation used for colour selection.
2. ``cv2.inRange()`` asks whether every pixel is inside a chosen HSV interval.
3. Two masks are created because red lies near both ends of the OpenCV hue interval.
4. ``cv2.bitwise_or()`` combines the two red masks.
5. A kernel defines the local neighbourhood used by morphology.
6. `OPEN` removes small isolated white regions.
7. `CLOSE` fills small holes inside detected regions.

Copy, run, and play

```

image = load_image(
    "images/01_clear_red_cup_centre.png"
)

mask = create_red_mask(image)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))

```

```
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(mask, cmap="gray")
plt.axis("off")

plt.show()
```

Experiments

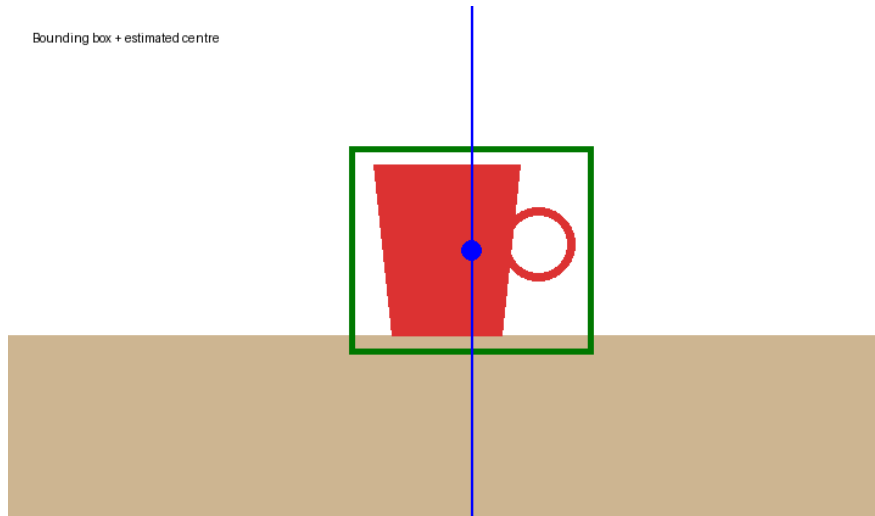
- Change kernel_size from 5 to 3. What changes?
- Change kernel_size from 5 to 9. What changes?
- Temporarily comment out MORPH_OPEN. Does noise increase?
- Temporarily comment out MORPH_CLOSE. Do holes appear?
- Make the saturation lower. Does more background enter the mask?
- Try one red range only. What happens to the mask?

Discussion questions — with useful answers

Question	What a strong answer should mention
What happens if lighting changes?	Pixel values can shift; a fixed threshold may miss the object or include unwanted pixels.
What happens if the background contains similar colours?	The mask may contain false regions; later contour selection may choose the wrong object.
Why is exact pixel equality weak?	Real objects contain shading, highlights and colour variation.

Part 4: Extract the Bounding Box and Object Centre

Workshop task: convert the binary mask into object-location information by finding regions, removing small regions, selecting the target, drawing a box, and computing the centre.



A bounding box provides image-space location and size; the centre is still an image coordinate.

Reference solution

```
def largest_object(mask, min_area=500):
    contours, _ = cv2.findContours(
        mask,
        cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE
    )

    contours = [
        c for c in contours
        if cv2.contourArea(c) >= min_area
    ]

    if not contours:
        return None

    contour = max(
        contours,
        key=cv2.contourArea
    )

    x, y, w, h = cv2.boundingRect(contour)

    return {
        "area": cv2.contourArea(contour),
        "bbox": (x, y, w, h),
        "centre": (
            x + w / 2,
            y + h / 2
        ),
    }
```

Understand the data flow

Stage

Output

findContours	A list of candidate boundaries.
contourArea	A size estimate for each candidate.
min_area filter	Removes small candidate regions.
max(..., key=contourArea)	Selects the largest remaining region.
boundingRect	x, y, w, h.
centre	$cx = x + w/2$, $cy = y + h/2$.

The selected centre is an image-space coordinate. It is not automatically a physical position in metres and does not tell the robot the object's depth.

Copy, run, and inspect the dictionary

```
result = largest_object(mask)

print("Detection result:")
print(result)

if result is not None:
    print("Area:", result["area"])
    print("Bounding box:", result["bbox"])
    print("Centre:", result["centre"])
else:
    print("No target detected.")
```

Discussion questions

Q1. Why is the object centre useful for a robot?

Q2. What happens if there are multiple detected objects?

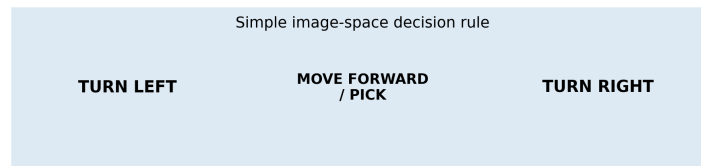
Q3. Why is 'choose the largest red region' only an assumption rather than true object recognition?

Reference answers

- The horizontal centre can support steering or target alignment; it provides a compact location signal for a decision rule.
- The system may detect several regions. The supplied solution explicitly chooses the largest remaining region, which may be wrong if the target is not the largest.
- Colour segmentation does not understand object identity. A large red background region could be selected as if it were the target.

Part 5: Generate a Simple Robot Action

Workshop task: compare the object centre with the image centre, define a tolerance, generate an action, and display it.



The simple decision rule is based only on horizontal image position.

Reference action function

```
def choose_action(
    cx,
    width,
    tolerance_ratio=0.1
):
    tolerance = width * tolerance_ratio
    centre = width / 2

    if cx < centre - tolerance:
        return "TURN LEFT"

    if cx > centre + tolerance:
        return "TURN RIGHT"

    return "MOVE FORWARD / PICK"
```

Why use a tolerance?

Without a tolerance region, a tiny centre error around the middle could switch the decision from left to right. The tolerance creates an acceptable centre region. It is a simple way to make the decision less sensitive.

Full Part 5 reference function

```
def generate_robot_action(
    image_path,
    min_area=500,
    tolerance_ratio=0.1,
    lower_1=(0, 80, 50),
    upper_1=(10, 255, 255),
    lower_2=(170, 80, 50),
    upper_2=(179, 255, 255),
    kernel_size=5,
):
    image = load_image(image_path)

    mask = create_red_mask(
        image,
        lower_1,
        upper_1,
        lower_2,
        upper_2,
        kernel_size,
    )
```

```

result = largest_object(
    mask,
    min_area
)

action = (
    "NO TARGET / STOP"
    if result is None
    else choose_action(
        result["centre"][0],
        image.shape[1],
        tolerance_ratio,
    )
)

output = draw_result(
    image,
    result,
    action,
    tolerance_ratio,
)

return action, result, output, mask

```

A complete visualisation helper

```

def draw_result(
    image,
    result,
    action=None,
    tolerance_ratio=0.1
):
    output = image.copy()
    height, width = output.shape[:2]

    if result is None:
        cv2.putText(
            output,
            "NO TARGET / STOP",
            (40, 70),
            cv2.FONT_HERSHEY_SIMPLEX,
            1.2,
            (0, 0, 255),
            3,
        )
        return output

    x, y, w, h = result["bbox"]
    cx, cy = result["centre"]

    cv2.rectangle(
        output,
        (x, y),
        (x + w, y + h),
        (0, 255, 0),
        3,
    )

    cv2.circle(
        output,
        (round(cx), round(cy)),
        8,
        (255, 0, 0),
        -1,
    )

```

```

)

if action is not None:
    centre = width / 2
    tolerance = width * tolerance_ratio

    for line_x in (
        centre,
        centre - tolerance,
        centre + tolerance,
    ):
        cv2.line(
            output,
            (round(line_x), 0),
            (round(line_x), height),
            (0, 255, 255),
            2,
        )

    cv2.putText(
        output,
        action,
        (40, 70),
        cv2.FONT_HERSHEY_SIMPLEX,
        1.2,
        (255, 255, 255),
        3,
    )

return output

```

Discussion questions — strong answers

Question	Answer
Why is the action rule too simple for a real robot?	It uses only horizontal image position. It ignores depth, object identity confidence, obstacles, robot state, dynamics, grasp pose, uncertainty and safety.
What additional information is needed to pick up the object?	At minimum, a reliable object location; in a realistic manipulation system, depth/3D position, object pose/size, camera calibration and a grasp/planning strategy would also be important.
What if no target is detected?	The supplied design explicitly uses a safe non-action response: NO TARGET / STOP.

The most important transition is not 'drawing a box'. It is converting perception into a decision. Once perception controls an action, an incorrect visual result can become an incorrect physical behaviour.

Part 6: Analyse Failure Cases

This is the independent part of the workshop. Apply your red-object pipeline to the remaining case images and record the detected region, estimated centre and generated action.

What to record

Image	Detected?	Area	Bounding box	Centre	Action	Failure / observation
case-1.png	☐ Yes ☐ No					
case-2.png	☐ Yes ☐ No					
case-3.png	☐ Yes ☐ No					
case-4.png	☐ Yes ☐ No					
case-5.png	☐ Yes ☐ No					
case-6.png	☐ Yes ☐ No					
case-7.png	☐ Yes ☐ No					
case-8.png	☐ Yes ☐ No					
case-9.png	☐ Yes ☐ No					

Reference implementation for batch analysis

```
from pathlib import Path

import cv2

from vision_utils import (
    choose_action,
    create_red_mask,
    draw_result,
    largest_object,
    load_image,
)

def analyse_failure_cases(
    image_dir,
    output_dir=None,
    pattern="*.png",
    min_area=500,
    tolerance_ratio=0.1,
):
    image_dir = Path(image_dir)
    output_dir = (
        Path(output_dir)
        if output_dir
        else None
    )

    if output_dir:
        output_dir.mkdir(
            parents=True,
            exist_ok=True
        )

    results = []

    for image_path in sorted(
        image_dir.glob(pattern)
```

```

):
    if "preview" in image_path.name.lower():
        continue

    image = load_image(image_path)

    mask = create_red_mask(image)

    result = largest_object(
        mask,
        min_area
    )

    action = (
        "NO TARGET / STOP"
        if result is None
        else choose_action(
            result["centre"][0],
            image.shape[1],
            tolerance_ratio,
        )
    )

    output = draw_result(
        image,
        result,
        action,
        tolerance_ratio,
    )

    if output_dir:
        cv2.imwrite(
            str(output_dir / image_path.name),
            output
        )

    results.append({
        "image": image_path.name,
        "detected": result is not None,
        "action": action,
        "area": (
            None if result is None
            else result["area"]
        ),
        "bbox": (
            None if result is None
            else result["bbox"]
        ),
        "centre": (
            None if result is None
            else result["centre"]
        ),
    })

return results

results = analyse_failure_cases(
    "images",
    output_dir="outputs"
)

```

```
for result in results:  
    print(result)
```

Failure analysis questions

Q1. In which cases did the method work well?

Q2. In which cases did the method fail?

Q3. Why did the method fail?

Q4. What problems might occur if this perception output controlled a real robot?

Core lesson

In robotics, perception errors can become action errors. A visual mistake is not merely a bad image-processing result if it changes the robot's behaviour.

Appendix

Complete Copy-and-Run Reference Code

The following files match the modular structure in the supplied solutions. Copy them into the same folder structure, then run the parts in order.

Suggested folder structure

```
week1_lab/
├── images/
│   ├── 01_clear_red_cup_centre.png
│   ├── case-1.png
│   ├── ...
│   └── case-9.png
├── vision_utils.py
├── part2_load_display.py
├── part3_colour_detection.py
├── part4_bbox_centre.py
├── part5_action_decision.py
└── part6_failure_analysis.py
```

vision_utils.py

```
from pathlib import Path

import cv2
import numpy as np

def load_image(image_path):
    image = cv2.imread(str(Path(image_path)))

    if image is None:
        raise FileNotFoundError(image_path)

    return image

def create_red_mask(
    image,
    lower_1=(0, 80, 50),
    upper_1=(10, 255, 255),
    lower_2=(170, 80, 50),
    upper_2=(179, 255, 255),
    kernel_size=5,
):
    hsv = cv2.cvtColor(
        image,
        cv2.COLOR_BGR2HSV
    )

    mask_1 = cv2.inRange(
        hsv,
        np.array(lower_1),
        np.array(upper_1)
    )
```

```

mask_2 = cv2.inRange(
    hsv,
    np.array(lower_2),
    np.array(upper_2)
)

mask = cv2.bitwise_or(
    mask_1,
    mask_2
)

kernel = np.ones(
    (kernel_size, kernel_size),
    np.uint8
)

mask = cv2.morphologyEx(
    mask,
    cv2.MORPH_OPEN,
    kernel
)

return cv2.morphologyEx(
    mask,
    cv2.MORPH_CLOSE,
    kernel
)

def largest_object(mask, min_area=500):
    contours, _ = cv2.findContours(
        mask,
        cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE
    )

    contours = [
        c for c in contours
        if cv2.contourArea(c) >= min_area
    ]

    if not contours:
        return None

    contour = max(
        contours,
        key=cv2.contourArea
    )

    x, y, w, h = cv2.boundingRect(contour)

    return {
        "area": cv2.contourArea(contour),
        "bbox": (x, y, w, h),
        "centre": (
            x + w / 2,
            y + h / 2
        ),
    }

```

```

def choose_action(
    cx,
    width,
    tolerance_ratio=0.1
):
    tolerance = width * tolerance_ratio
    centre = width / 2

    if cx < centre - tolerance:
        return "TURN LEFT"

    if cx > centre + tolerance:
        return "TURN RIGHT"

    return "MOVE FORWARD / PICK"

def draw_result(
    image,
    result,
    action=None,
    tolerance_ratio=0.1
):
    output = image.copy()
    height, width = output.shape[:2]

    if result is None:
        cv2.putText(
            output,
            "NO TARGET / STOP",
            (40, 70),
            cv2.FONT_HERSHEY_SIMPLEX,
            1.2,
            (0, 0, 255),
            3,
        )
        return output

    x, y, w, h = result["bbox"]
    cx, cy = result["centre"]

    cv2.rectangle(
        output,
        (x, y),
        (x + w, y + h),
        (0, 255, 0),
        3,
    )

    cv2.circle(
        output,
        (round(cx), round(cy)),
        8,
        (255, 0, 0),
        -1,
    )

    if action is not None:
        centre = width / 2
        tolerance = width * tolerance_ratio

        for line_x in (

```

```

        centre,
        centre - tolerance,
        centre + tolerance,
    ):
        cv2.line(
            output,
            (round(line_x), 0),
            (round(line_x), height),
            (0, 255, 255),
            2,
        )

    cv2.putText(
        output,
        action,
        (40, 70),
        cv2.FONT_HERSHEY_SIMPLEX,
        1.2,
        (255, 255, 255),
        3,
    )

    return output

```

Part 2–5: recommended order

- Run Part 2 and confirm the image loads and displays correctly.
- Run Part 3 and inspect the mask before doing any contour processing.
- Run Part 4 and inspect area, bounding box and centre.
- Run Part 5 and verify the action against the visible centre position.
- Only then run Part 6 across multiple images.

One-Notebook Practice Version

If you prefer Jupyter Notebook, the following compact version lets you experiment without creating multiple Python files. Paste the functions into one cell, then run the experiment cells one by one.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

def load_image(image_path):
    image = cv2.imread(str(Path(image_path)))
    if image is None:
        raise FileNotFoundError(image_path)
    return image

def create_red_mask(image, kernel_size=5):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    lower_1 = np.array([0, 80, 50])
    upper_1 = np.array([10, 255, 255])

    lower_2 = np.array([170, 80, 50])
    upper_2 = np.array([179, 255, 255])

    mask_1 = cv2.inRange(hsv, lower_1, upper_1)
    mask_2 = cv2.inRange(hsv, lower_2, upper_2)

    mask = cv2.bitwise_or(mask_1, mask_2)

    kernel = np.ones(
        (kernel_size, kernel_size),
        np.uint8
    )

    mask = cv2.morphologyEx(
        mask,
        cv2.MORPH_OPEN,
        kernel
    )

    mask = cv2.morphologyEx(
        mask,
        cv2.MORPH_CLOSE,
        kernel
    )

    return mask

def largest_object(mask, min_area=500):
    contours, _ = cv2.findContours(
        mask,
        cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE
    )
```

```

contours = [
    c for c in contours
    if cv2.contourArea(c) >= min_area
]

if not contours:
    return None

contour = max(
    contours,
    key=cv2.contourArea
)

x, y, w, h = cv2.boundingRect(contour)

return {
    "area": cv2.contourArea(contour),
    "bbox": (x, y, w, h),
    "centre": (x + w / 2, y + h / 2),
}

def choose_action(cx, width, tolerance_ratio=0.1):
    tolerance = width * tolerance_ratio
    centre = width / 2

    if cx < centre - tolerance:
        return "TURN LEFT"

    if cx > centre + tolerance:
        return "TURN RIGHT"

    return "MOVE FORWARD / PICK"

image = load_image(
    "images/01_clear_red_cup_centre.png"
)

mask = create_red_mask(image)

result = largest_object(mask)

if result is None:
    action = "NO TARGET / STOP"
else:
    action = choose_action(
        result["centre"][0],
        image.shape[1]
    )

print("Result:", result)
print("Action:", action)

```

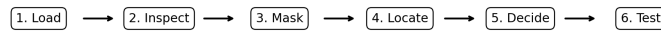
Play with these three variables

Variable	Try	What you are learning
kernel_size	3, 5, 9	How morphological neighbourhood size affects the mask.

min_area	100, 500, 2000	How filtering changes which regions survive.
tolerance_ratio	0.05, 0.10, 0.20	How decision sensitivity changes.

Change one variable at a time. Otherwise you cannot tell which change caused the result.

Visual Debugging: The Fastest Way to Understand Failure



A practical debugging order for this workshop.

When the final action is wrong, do not immediately edit `choose_action()`. The action may be correct for an incorrect perception result.

Debug from left to right

If this is wrong...	Inspect this first
Image cannot load	Path, filename, working directory.
Colours look wrong	BGR/RGB conversion.
Mask is empty	HSV thresholds, saturation/value thresholds, image lighting.
Mask has too much background	HSV range too broad or similar-colour background.
Many blobs appear	Mask quality and morphology.
Wrong object selected	min_area and target-selection policy.
Centre is wrong	bounding box and centre calculation.
Action is wrong	image width, centre comparison and tolerance.
No target	Explicit fallback branch.
Key engineering principle	
Debug the earliest incorrect stage. Later stages can only work with the information they receive.	

Engineering Insights to Carry Forward

Insight	Why it matters later
Separate perception from decision-making.	You can change the detector without rewriting the action policy.
Use small functions with clear inputs/outputs.	Each stage can be tested independently.
Visualise intermediate results.	A mask or bounding box can reveal errors immediately.
Make assumptions explicit.	'Largest red region is the target' is testable and replaceable.
Handle no-target cases explicitly.	Uncertainty should influence behaviour.
Image-space is not physical-space.	A pixel coordinate is not automatically a position in metres.
Test failure cases, not only easy examples.	A robot must behave acceptably outside the ideal image.
Change one parameter at a time.	Controlled experiments make debugging interpretable.

A reusable perception skeleton

```
image
↓
preprocess
↓
perceive
↓
filter candidates
↓
select target
↓
estimate image-space location
↓
decide
↓
```

```
act / stop / search
↓
evaluate
```

This structure will remain useful when the perception method changes from hand-designed colour thresholding to a learned detector or segmentation model.

Using AI as an Engineering Assistant

The workshop's teaching philosophy is not 'AI writes the code, so coding knowledge is unnecessary'. The important skills are understanding, evaluating, adapting and improving solutions.

Task	Good prompt
Explain	Explain why HSV is used here instead of RGB. Do not rewrite my code.
Inspect	Explain what this function returns and what each field means.
Debug	My mask is empty. Give three likely causes and one test for each.
Modify	Change only the target colour from red to blue and explain the required threshold changes.
Test	Suggest five images or conditions that could break this segmentation method.
Review	Review my target-selection rule. Identify assumptions and failure modes.

Rule of thumb

If AI generates a line, you should still be able to explain what it does, why it belongs in the pipeline, and how you would test it.

Chapter 1 Practice Checklist

Skill	I can...	Done
Environment	Create/activate the environment and import OpenCV, NumPy and Matplotlib.	☐
Image data	Explain $H \times W \times C$ and inspect <code>image.shape</code> .	☐
Colour	Explain BGR versus RGB.	☐
HSV	Explain H, S and V in practical terms.	☐
Mask	Create and display a binary mask.	☐
Morphology	Explain OPEN and CLOSE.	☐
Contours	Find candidate regions.	☐
Filtering	Explain why <code>min_area</code> is used.	☐
Bounding box	Interpret <code>x</code> , <code>y</code> , <code>w</code> , <code>h</code> .	☐
Centre	Compute <code>cx</code> and <code>cy</code> .	☐
Decision	Explain the left / middle / right rule.	☐
Failure	Handle no target explicitly.	☐
Testing	Run more than one image and analyse failure.	☐
Engineering	Explain assumptions and limitations.	☐

Final reflection

Which stage of the pipeline is most likely to fail in a new environment, and why?

Which assumption in this workshop would you remove first for a real robot?

What did you learn from a failed visualisation that you could not learn from the final action alone?

If you replaced colour thresholding with a deep-learning detector next week, which parts of this pipeline would remain?

Takeaway

A robot does not act on pixels directly. It extracts useful information, estimates where the relevant object is, makes a decision, and must account for what happens when perception is wrong.

Discussion Answer Key

Workshop question	Concise reference answer
What does each pixel represent?	Numerical channel values at one image location.
Why is classification alone insufficient?	A class label does not provide object location needed for targeted action.
What happens if lighting changes?	Fixed colour thresholds may miss the target or produce false positives.
What if the background has similar colours?	The mask may contain unwanted regions.
Why is HSV useful?	It separates hue from saturation and brightness, making colour-based thresholding more interpretable.
What does a binary mask mean?	White pixels satisfy the selected target criterion; black pixels do not.
Why use morphology?	OPEN removes small isolated noise; CLOSE fills small holes.
Why use contours?	They turn connected mask regions into candidate objects that can be measured.
Why use min_area?	To reject very small regions likely to be noise.
Why is the centre useful?	It provides a compact image-space location for steering/targeting.
What if multiple objects are detected?	The supplied solution chooses the largest surviving region; this is a simple target-selection policy.
Why is the action rule too simple?	It ignores depth, uncertainty, object identity, obstacles, robot state and physical constraints.
What is needed for picking?	Reliable localisation plus additional geometric/physical information such as depth/3D position, calibration, pose and grasp planning.
What if no target is detected?	Use an explicit fallback such as NO TARGET / STOP.
What is the main robotics lesson?	Perception errors can become action errors.

Function Reference

Function	Purpose
cv2.imread()	Load an image.
image.shape	Inspect image dimensions.
cv2.cvtColor()	Convert BGR to RGB or HSV.
cv2.inRange()	Create a binary threshold mask.
cv2.bitwise_or()	Combine binary masks.
cv2.morphologyEx()	Clean a binary mask.
cv2.findContours()	Find candidate connected regions.
cv2.contourArea()	Measure region area.
cv2.boundingRect()	Return x, y, w, h.
cv2.rectangle()	Draw a bounding box.
cv2.circle()	Draw a centre marker.
cv2.putText()	Draw an annotation.
plt.imshow()	Display an image.

Submission reminder from the workshop

The workshop specifies Python implementations for Parts 2–6 and a results document for the failure-analysis part containing screenshots and discussion. Follow the current course/assessment instructions provided by your teaching team when preparing any submission.