Autonomous Systems Lab

Image Processing

Lab Overview
In this lab, you will:

1. Capture real-time video on a Raspberry Pi 5 using a Pi Camera v3.
2. Implement fundamental image processing techniques (grayscale, thresholding, masking).
3. Detect features (lines, circles, contours) for robotic vision tasks.
4. Stream live video to a web-based dashboard.

Deliverables
By the end, you should be able to:

- Demonstrate real-time object or shape detection (circles or lines) on the Pi Camera feed.
- Provide brief documentation or screenshots showing each major step: grayscale conversion, thresholding, morphological filtering, and Hough line/circle detection.
- (Optional) Showcase a working web stream with real-time updates.

1. Introduction

Image processing underpins many autonomous robotic tasks, from line following to object detection. Here, you will learn how to process live camera data and extract meaningful features in real time.

Key Concepts

- **Resolution & Field of View**: Understand how higher resolution can reduce FPS but provide more detail.
- **Color Spaces**: Use RGB vs. HSV for more intuitive color segmentation.
- **Thresholding & Masking**: Segment key regions (e.g., color-coded markers) by converting to binary images.
- **Feature Extraction**: Apply Hough transforms for lines/circles and use contour detection to outline objects.
- **Coordinate Estimation (Advanced)**: Relate pixel positions to real-world geometry, enabling navigation.

2. Short Theoretical Background

Digital Image Processing is the manipulation of image data—often in matrix (2D array) form—to enhance, analyze, or extract information. A standard 2D digital image is composed of pixels (picture elements), each with an associated intensity or color value. In robotics, image processing allows us to detect objects or features in real-time scenes, enabling robots to make decisions based on visual cues.

Some key points in this lab:

- **Camera Models:**
  - A camera's field of view (FOV) describes how much of the scene is captured.
  - Resolution (e.g., 640×480 or 1280×720) affects image detail and processing speed.
- **Color Spaces:**
  - Images are commonly captured in RGB (Red, Green, Blue). Computer vision tasks often use alternative color representations (e.g., HSV) for easier color-based segmentation.
- **Thresholding and Masking:**
  - Converting images to grayscale or HSV and applying thresholds allows segmenting regions of interest. This is crucial when detecting brightly colored objects (balls, cones, lines).
- **Feature Extraction** (Hough Transforms, Contours, etc.):
  - Hough Line/Circle transforms detect geometric shapes (lines, circles).
  - Contour extraction finds object boundaries in a binary (thresholded) image.
  - Generalized Hough transforms can detect arbitrary shapes.
- **Coordinate Estimation:**
  - By combining camera geometry (tilt, height) and lens parameters (FOV), it is possible to approximate the distance or position of objects relative to the camera. This is essential for robot navigation.

These foundational concepts are applied in robotics for tasks like line following, object detection, and environment mapping.

3. Pre-Lab Activity

Goal: Familiarize yourself with camera capturing, basic processing, and simple feature detection on your laptop/desktop before the in-lab session. Review **Appendix B** for a python based example and matlab adaptation.

1. Environment Setup

- **Install/Verify Software**
  - **Python** users: Confirm `opencv-python` is installed, or install it via `pip install opencv-python`.
  - **MATLAB** users: Have the Computer Vision Toolbox.
- **Check Camera Access**
  - Verify that your laptop's camera (or USB webcam) is recognized by your OS.
  - In Python, run a simple script to open a `cv2.VideoCapture(0)` and verify it captures frames.
  - In MATLAB, use `webcam`

2. Basic Camera Capture

- **Capture and Display Frames**
  - Python: create a loop with `video.read()` to grab frames and use `cv2.imshow()` to display.
  - MATLAB: use `snapshot(cam)` in a loop or the `preview(cam)` function.
- **Record FPS** (Optional)
  - Time how long it takes to process and display each frame. Print or plot the resulting FPS in your console or figure.

3. Simple Image Processing

- **Grayscale Conversion**
  - Python: `cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)`
  - MATLAB: `rgb2gray(frame)`
- **Color Filtering**
  Demonstrate color range masking (e.g., isolate a specific color object).
  - Python: use `cv2.inRange()` in HSV color space.
  - MATLAB: convert to HSV via `rgb2hsv()` and apply logical operations.
- **Thresholding / Binarization**
  - Use simple binary thresholding or adaptive thresholding to see how it segments the image.

4. Basic Feature Detection

- **Edge Detection**
  - Apply **Canny** edge detection to see outlines of objects and your background.
- **Contour Detection**
  - Python: use `cv2.findContours()` on a binary image.
  - MATLAB: `bwboundaries()` or region-based functions.
- **Circle / Shape Detection**
  - Try **Hough Circle Transform** if available, or a basic shape detection approach.
  - Mark detected shapes on the displayed feed.

5. Annotate and Overwrite on Live Feed

- **Draw Shapes or Text**
  - Python: `cv2.putText()`, `cv2.line()`, `cv2.circle()`
  - MATLAB: `insertText()`, `insertShape()`
- **Display Basic Metrics**
  - For example, overlay the current FPS, resolution, or a color label on the feed.

4. In-Lab Activity

Overview & Objectives

This in-lab activity will allow you to:

- **Capture Images** from Pi Camera v3 on a Raspberry Pi 5.
- **Process Images** in real-time using Python and OpenCV (color conversion, masking, thresholding).
- **Overlay Graphics and Compute FPS** to understand performance metrics.
- **Stream Video** to a web-based GUI for remote monitoring.
- **Detect Features** (e.g., lines, circles, contours, connected components) to identify specific targets.
- **Optional**: Explore **Generalized Hough Transforms** to detect arbitrary shapes.

1. Hardware & Environment Setup

- **Start the raspberry pi:** Power on the zumoPiV02 platform by pressing On/Off button.
- Open VScode and connect to the platform.
- Copy the **ImageProcessing** folder to your students folder. Located under: WS_Zumo/Python/Examples/ImageProcessing

2. Initialize and Capture Images

This section covers initializing the Pi Camera, capturing an image, displaying it in real-time, and saving it to disk.

Practice:

- Open Example **PiCameraLive.py** located in your student folder. Review the code.
- Since the example contains live local streaming we will run the script using the terminal through VNC or by using the terminal in MobaXterm (as it contains the X11 extension which is a remote display forwarding through SSH)

Explanation:

- **Live Display**: The script continuously shows the camera feed in an OpenCV window.
- **Saving an Image**: Pressing **'s'** captures and saves an image with a timestamp-based filename.
- **Exiting**: Press **'q'** to close the window and terminate the program.

3. Measure Frames per Second (FPS)

Frames per second (FPS) reflects how quickly your system can capture and render images in real time. Several factors can influence your observed FPS:

- **Resolution**:
    - Higher resolutions (e.g., 1280×720) produce larger images that require more computation and memory, typically resulting in **lower FPS**.
    - Lower resolutions (e.g., 640×480) generally allow for **faster processing** and thus **higher FPS**.
- **Display/Rendering Method**:
    - **Local Execution (Direct Display on Pi)**:
        - The most accurate measure of performance.
        - The images are rendered directly on the Pi's GPU/display with minimal latency.
    - **VNC (Virtual Network Computing)**:
        - Shares the **entire desktop** of the Pi over the network.
        - Some overhead due to compression and network transmission.
        - Often more efficient and user-friendly than raw X11 forwarding, but still not as smooth as local display.
    - **X11 Forwarding**:
        - Sends window-rendering commands from the Pi to a remote X server (your local machine).
        - Can introduce **significant network latency**, making FPS measurements lower than the Pi's actual capture rate.
        - Useful for quick remote testing but not ideal for real-time performance analysis.

Practice:

- Open Example **PiCameraLiveFPS.py** located in your student folder. Review change in the code that prints frame rates to the console
- Compare the measured FPS in VNC mode and X11 Forwarding.
- Change to higher resolution (e.g., 1280×720) and notice the impact on FPS.

Key Takeaways:

- **Resolution Trade-offs**: Higher resolution = lower FPS; choose a balance based on your application's needs.
- **Local vs. VNC vs. X11**: For **true performance** metrics, run **locally** on the Pi or via VNC if a monitor is not an option. X11 forwarding can be slower, primarily for debugging or remote control rather than real-time observations.

4. Streaming to a Web-Based GUI (Dash)

Web-based dashboards allow for flexible remote monitoring and control of your camera feed. **JPEG compression** is crucial for efficient streaming because it **reduces file size**, making the images quicker to transmit over the network at the cost of some image quality. When multiple users connect simultaneously, **network load** can increase significantly, further reducing overall streaming performance.

Practice:

- Open Example **PiCameraLiveHTML.py** located in your student folder.
- Since this example doesn`t run a local GUI it can be launched directly from VScode.
- Start the application and navigate in the chrome or explorer to the Pi IP address at port: 8500. You will see a live camera feed and an FPS display updating in real time.

  http://<RaspberryPiIP>:8500

- Experiment with compression ratio, image resolution and update interval.

Key Points:

- **JPEG Compression (`IMWRITE_JPEG_QUALITY, 50`)**: Reduces data size for faster streaming; adjust as needed to balance quality vs. performance.
- **Update Interval (`interval=100`)**: Requests up to ~10 frames per second in the browser. Actual performance depends on CPU load and network speed.
- **FPS Monitoring**: Tracks how many frames are served per second, helping to confirm whether the system can keep up with your chosen interval.
- **Multiple Users**: Be aware that **each additional user on the network** can significantly **increase network load**, potentially reducing available bandwidth and lowering the overall streaming performance.

By integrating **JPEG compression** with a **Dash** app, you can **monitor** and **control** your camera feed remotely, enabling a lightweight, user-friendly setup for robotics projects where real-time visibility is essential.

5. Image Processing Basics

Before attempting advanced feature extraction—such as line detection, object tracking, or machine learning—you must often perform **basic image processing** to simplify or highlight the relevant parts of an image. Techniques like **grayscale conversion**, **color masking**, and **thresholding** help you visually inspect and analyze the data, guiding you toward the appropriate algorithms for your application.

Why These Operations Matter

- **Grayscale**: Reduces information to intensity values, making edge detection and certain morphological tasks more efficient.
- **Color Masking**: Isolates parts of the image with specific colors—particularly useful for color-coded markers (e.g., orange cones).
- **Thresholding (Binary Images)**: Distinguishes foreground from background, simplifying contour finding or blob detection.
- **Morphological Operations**: Cleans up noise in binary images, improving shape recognition.

By **experimenting** with these techniques and **visually inspecting** the results, you can make informed decisions about which processing pipeline to pursue.

Practice:

- Open Example **PiCameraBasics.py** located in your student folder. Review the code.
- Since the example contains live local streaming we will run the script using the terminal through VNC or by using the terminal in MobaXterm.
- Run the example in different settings e.g:
    - python PiCameraBasics.py threshold 150
    - python PiCameraBasics.py morph_open 150

Customizing Morphological Operations

Morphological transformations are powerful tools for cleaning up binary images before **contour detection** or **object segmentation**. The `cv2.morphologyEx()` function applies a **kernel** to remove small noise while preserving larger objects.

Settings to Experiment With:

1. **Kernel Size** (`(3,3)`, `(5,5)`, `(7,7)`, etc.)
    - **Smaller kernels** (e.g., `(3,3)`) remove only very small noise.
    - **Larger kernels** (e.g., `(7,7)`) remove larger unwanted details but may distort smaller objects.
2. **Different Morphological Operations:**
    - **`cv2.MORPH_OPEN`**: Removes small noise.
    - **`cv2.MORPH_CLOSE`**: Fills small holes inside objects.
    - **`cv2.MORPH_GRADIENT`**: Enhances edges.

6. Feature Extraction

Feature extraction is the process of identifying important shapes or objects—like lines, circles, contours, or blobs—within an image. It is typically performed **after** some preliminary operations (e.g., thresholding, morphological filtering) that simplify the image and help isolate the regions of interest. Each feature extraction technique (e.g., Hough transforms, contour detection, connected components) comes with **multiple adjustable parameters** that affect accuracy and robustness. These parameters often control thresholds, distance metrics, or kernel sizes and should be tuned based on your specific application—whether it's detecting thin lane lines, large circular objects, or intricate shapes against a noisy background.

Practice:

- Open Example **PiCameraFeatures.py** located in your student folder. Review the code.
- Since the example contains live local streaming we will run the script using the terminal through VNC or by using the terminal in MobaXterm.
- Run the example in different settings e.g:
    - python PiCameraFeatures.py circles
    - python PiCameraFeatures.py lines
- Explore the parameters of the different functions for feature extraction.

Explanation

- **lines**:
    - `cv2.Canny` finds edges, then `cv2.HoughLinesP` estimates straight lines.
    - The resulting lines are drawn in **green**.
- **circles**:
    - A **median blur** helps reduce noise for more stable circle detection.
    - `cv2.HoughCircles` finds circular objects (like balls).
    - Each circle is drawn in **green**, with a small red dot at the center.
- **contours**:
    - Simple **binary thresholding** (value = 127) separates objects from background.
    - `cv2.findContours` locates shape boundaries, which are drawn in green.
- **components**:
    - `cv2.connectedComponentsWithStats` identifies distinct blobs in a binary image.
    - Each blob is surrounded by a **green bounding box** and a **red centroid** marker.

This modular approach lets you quickly **visualize and experiment** with different feature extraction methods, guiding decisions for **robotic navigation**, **object detection**, or **further vision** applications.

7. Coordinate Estimation:

Camera coordinates provide a way to measure where objects appear relative to the camera's viewpoint. By interpreting pixel positions in terms of angles and distances, developers can estimate real-world coordinates—like how far away an object is or whether it lies to the left or right of the camera. In practice, such coordinate mapping is crucial for tasks like robotic navigation (e.g., guiding a robot around obstacles), augmented reality (e.g., overlaying virtual elements accurately in a scene), and industrial inspection (e.g., measuring part dimensions or alignment). This framework allows systems to make sense of visual data in real time and perform meaningful actions based on a camera's perspective.

Additional reading: Camera Coordinates

Practice:

- Open Example **DiscPosition.py** located in your student folder. Review the code.
- Since the example contains live local streaming we will run the script using the terminal through VNC or by using the terminal in MobaXterm.
- Move the small disc around and see how the coordinates are updated.
- Explore the code to understand it`s flow.

Key Points of the Disc Position code:

Camera Setup and Configuration
- Initialize PiCamera2 and configure to capture frames at 640×480, BGR888 format.
- Maintain a loop that grabs each new frame, limiting the refresh rate to around 10 FPS.

Preprocessing
- Convert the frame to grayscale.
- Apply a Gaussian blur to reduce noise.

Thresholding
- Create a **black mask** by thresholding the grayscale image (inverted) to highlight dark areas.
- Create a **white mask** by thresholding the same image at a higher value, highlighting bright regions.

Contour Detection & Filtering
- Find contours in the black mask.
- Skip contours outside the specified area range (too small or too large).
- For each candidate contour, compute how much of it overlaps with the white mask.
- If the fraction of white is within a target range, treat this as the detected disk.

Centroid and Metrics
- Compute the contour's centroid (using image moments).
- Calculate perimeter and area for displaying debug information.

Geometry & Real-World Coordinates
- Convert the centroid's pixel coordinates from the image center to angles, accounting for camera field of view.
- Add the camera's tilt angle.
- Use the pinhole geometry to estimate distance from the camera on the ground plane.

- Multiply and add any empirical factors/offsets to align with real-world measurements.

Debug Overlay

- If debug mode is active, draw all candidate contours in one color, the best contour in another, plus the centroid mark.
- Overlay text for position (in mm), contour perimeter, fraction of white, pixel coordinates, and FPS.

Closing & Cleanup

- Press ‘q’ to exit.
- Destroy all OpenCV windows and stop the PiCamera2 when finished.

8. Exploring Further

For applications that involve locating irregular shapes or specific patterns, you can explore **Generalized Hough Transforms**, such as **GeneralizedHoughGuil** in OpenCV. Unlike traditional Hough methods that target lines or circles, GeneralizedHoughGuil is designed to detect **arbitrary shapes** by matching a template in the grayscale image. This approach can handle objects of varying sizes and minor pose changes, making it a powerful option for more complex tasks—such as detecting unique markers, logos, or custom contours—where standard Hough transforms may not suffice.

9. Assignment

Using an existing capture script (e.g., **PiCameraLive.py**), acquire several images of a chosen lab object (examples: a competitor’s vehicle, cones, discs, balls, perimeter lines). Then, implement a **classic image processing approach**—in MATLAB *or* Python—to detect the object in those images. Demonstrate the detection through clear results or visualizations.

5. Summary

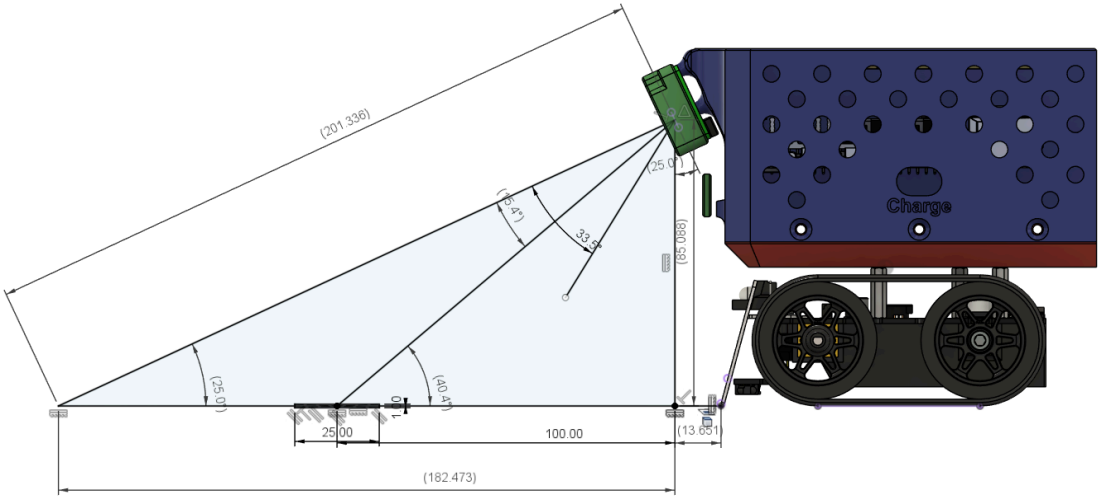
This lab introduced fundamental techniques to capture, process, and analyze live video from a Pi Camera v3 on a Raspberry Pi 5 for autonomous robotics applications. Students learned how to:

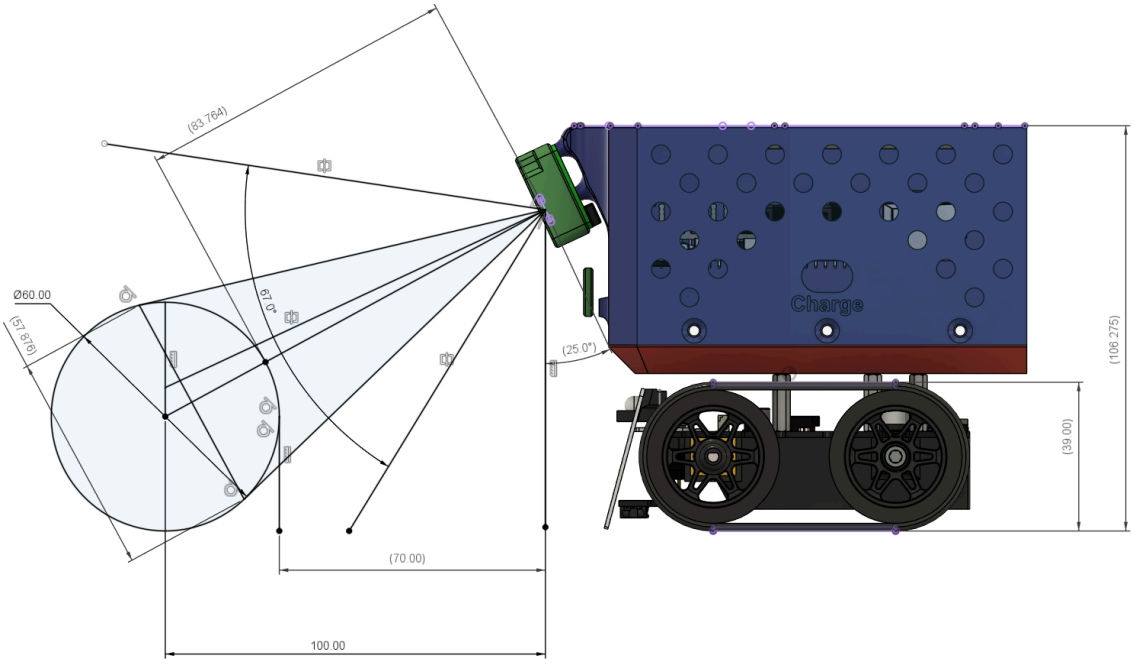
- **Configure and Capture** video streams at different resolutions, evaluating the tradeoff between image detail and frame rate.
- **Perform Basic Image Processing** (grayscale conversion, thresholding, morphological filtering) to isolate objects or features of interest (e.g., cones, discs, lines).
- **Extract Features** using methods such as Hough transforms (for lines/circles) or contour detection, enabling real-time object detection and shape recognition.
- **Estimate Coordinates** by mapping pixel positions to real-world geometry. This involved factoring in camera height, tilt, and field of view, providing a path to more advanced navigation tasks.
- **Stream Video** locally or to a web-based GUI, demonstrating various approaches to viewing the live camera feed and measuring system performance.

By completing the lab, students gained practical experience with OpenCV-based workflows for live image capture, thresholding and masking, shape detection, and basic camera geometry—key competencies for autonomous robotics and vision-based control.

Appendix A:

```
# --- Camera and Object Parameters ---
CAMERA_HEIGHT = 85.0        # mm above the ground
CAMERA_TILT_DEG = 25.0      # Camera tilted downward by 25 degrees
HORIZONTAL_FOV_DEG = 102.0 # Horizontal field of view (degrees)
VERTICAL_FOV_DEG = 67.0     # Vertical field of view (degrees)
```





Appendix B:

Below is a single Python script that demonstrates multiple image-processing modes, similar to the PiCamera lab examples used in the lab. It uses a webcam on a PC.

```
import sys
import cv2

def print_usage():
    print("Usage: python webcam_demo.py <mode>")
    print("Available modes:")
    print("  rgb2gray   - Convert frames to grayscale")
    print("  contour    - Simple contour detection on binary threshold")

if len(sys.argv) < 2:
    print_usage()
    sys.exit(1)

mode = sys.argv[1].lower()

# Open default webcam (index 0)
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Could not open webcam.")
    sys.exit(1)

print(f"Running mode: {mode}")

while True:
    ret, frame = cap.read()
    if not ret:
        print("Failed to capture frame from webcam.")
        break

    # Check which mode the user selected
    if mode == "rgb2gray":
        display_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    elif mode == "contour":
        # Convert to grayscale first
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        # Threshold to get a binary image
        _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
        # Find and draw contours
        contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        display_frame = frame.copy()
        cv2.drawContours(display_frame, contours, -1, (0, 255, 0), 2)

    else:
        print("Unknown mode:", mode)
        print_usage()
        break

    # Display the result
    cv2.imshow("Webcam Demo", display_frame)

    # Press 'q' to exit
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

How to Use

- **Save** the script as `webcam_demo.py` (or any preferred filename).
- **Install** OpenCV if you haven't already:
    - pip install opencv-python
- **Run** the script from the command line:
    - python webcam_demo.py rgb2gray
- **Switch** to the **contour** mode:
    - python webcam_demo.py contour
- **Press** `q` in the display window to quit

What Each Mode Does

- **none**: Shows the unmodified webcam feed.
- **grayscale**: Converts frames to grayscale.
- **hsv_mask**: Performs a color mask in **HSV** space (default is **blue**).
- **contour**: Thresholds the grayscale image and draws **green** contours around blobs.
- **lines**: Uses **Canny edge detection** plus **probabilistic Hough transforms** to draw **green lines**.
- **circles**: Applies **median blur** then detects circles with **HoughCircles**, drawing **green** outlines and **red** centers.

Adapting for MATLAB

In **MATLAB**, you can replicate the same logic with:

- `cam = webcam;` to open the default webcam,
- `snapshot(cam);` to capture images,
- `rgb2gray(image);` for grayscale,
- `imbinarize(image, threshold);` for thresholding,
- `bwboundaries` or `bwconncomp` for contour or component detection,
- `insertShape()` or `plot()` for annotation and displaying results in a figure.

This structure mirrors the **in-lab** approach for **Raspberry Pi + PiCamera2**, but here you can **test** everything on a **laptop webcam**, verifying you can capture frames and switch between different **image processing** modes.