SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Computer Vision in Smart Robotics (PGI20D17J)

Lab Manual

Lab 1: Installing OpenCV and Displaying Images

Aim To successfully install the OpenCV library and write a basic Python program to load an image from a file and display it on the screen.

Procedure

1. **Install Python:** Ensure Python 3.x is installed on your system.
2. **Install OpenCV:** Open your terminal or command prompt and run the following command to install OpenCV:
3. `pip install opencv-python`

4. **Prepare an Image:** Have an image file (e.g., `image.jpg`, `image.png`) ready in the same directory as your Python script, or provide its full path.
5. **Write the Source Code:** Create a Python file (e.g., `lab1.py`) and write the provided source code.
6. **Run the Program:** Execute the Python script from your terminal:
7. `python lab1.py`

8. **Observe Output:** A window should appear displaying the image. Press any key to close the window.

Source Code

```
import cv2

def display_image(image_path):
    """
    Loads an image from the specified path and displays it.
    Waits for a key press to close the window.
    """
    # Read the image from the specified path
    # cv2.imread() returns a NumPy array representing the image
    img = cv2.imread(image_path)

    # Check if the image was loaded successfully
    if img is None:
        print(f"Error: Could not load image from {image_path}")
        return

    # Display the image in a window named 'Image Display'
    # cv2.imshow() takes the window name and the image array as arguments
```

```
    cv2.imshow('Image Display', img)

    # Wait indefinitely until a key is pressed
    # 0 means wait forever, any positive number means wait for that many
milliseconds
    cv2.waitKey(0)

    # Destroy all OpenCV windows
    # This closes all windows created by cv2.imshow()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    # Specify the path to your image file
    # Make sure this image file exists in the same directory as your script,
    # or provide the full path to the image.
    input_image_file = 'example_image.jpg' # Replace with your image file name

    print(f"Attempting to display image: {input_image_file}")
    display_image(input_image_file)
    print("Program finished.")
```

Input A valid image file (e.g., `example_image.jpg`) located in the same directory as the Python script.

Expected Output A new window titled "Image Display" will appear, showing the content of `example_image.jpg`. The window will close when any key is pressed.

Lab 2: Reading and Writing Images with OpenCV

Aim To read an image from a specified file path and then write (save) a copy of that image to a new file path using OpenCV functions.

Procedure

1. **Prepare an Image:** Have an image file (e.g., `input.png`) ready.
2. **Write the Source Code:** Create a Python file (e.g., `lab2.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab2.py`

5. **Verify Output:** Check the directory where your script is located for the newly created image file.

Source Code

```
import cv2

def read_and_write_image(input_path, output_path):
    """
    Reads an image from input_path and writes it to output_path.
    Displays the original image before saving.
    """
    # Read the image
    img = cv2.imread(input_path)

    # Check if image loading was successful
    if img is None:
        print(f"Error: Could not read image from {input_path}")
        return

    print(f"Successfully read image from: {input_path}")

    # Optionally, display the original image before saving
    cv2.imshow('Original Image', img)
    cv2.waitKey(0) # Wait for a key press
    cv2.destroyAllWindows()

    # Write the image to a new file
    # The file extension in output_path determines the format (e.g., .jpg, .png)
    success = cv2.imwrite(output_path, img)

    if success:
        print(f"Successfully wrote image to: {output_path}")
    else:
        print(f"Error: Could not write image to {output_path}")

if __name__ == "__main__":
    # Define input and output file paths
    input_image_file = 'input_image.jpg'  # Replace with your input image
    output_image_file = 'output_image_copy.png' # New file name and format

    print(f"Attempting to read '{input_image_file}' and write to
'{output_image_file}'")
    read_and_write_image(input_image_file, output_image_file)
    print("Program finished.")
```

Input An image file named `input_image.jpg` (or any valid image format) in the same directory as the script.

Expected Output A message indicating successful reading and writing of the image. A new file named `output_image_copy.png` will be created in the same directory, which will be a copy of the input image. The original image will also be briefly displayed.

Lab 3: Color Space Conversion and Thresholding with OpenCV

Aim To convert an image from one color space to another (e.g., BGR to Grayscale, BGR to HSV) and apply different thresholding techniques (e.g., binary, Otsu's) to segment image regions.

Procedure

1. **Prepare an Image:** Use an image file (e.g., `color_image.jpg`).
2. **Write the Source Code:** Create a Python file (e.g., `lab3.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab3.py`

5. **Observe Output:** Multiple windows will appear, showing the original image, its grayscale and HSV conversions, and images after different thresholding operations. Close each window to proceed.

Source Code

```
import cv2
import numpy as np

def color_space_and_thresholding(image_path):
    """
    Loads an image, converts it to different color spaces,
    and applies various thresholding techniques.
    """
    # Read the image
    img = cv2.imread(image_path)

    if img is None:
        print(f"Error: Could not load image from {image_path}")
        return

    print(f"Processing image: {image_path}")

    # --- Color Space Conversions ---

    # 1. Convert BGR to Grayscale
    # Grayscale images have only one channel, representing intensity
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    cv2.imshow('Original Image (BGR)', img)
    cv2.imshow('Grayscale Image', gray_img)
    print("Converted to Grayscale.")

    # 2. Convert BGR to HSV (Hue, Saturation, Value)
    # HSV is often used for color-based segmentation
    hsv_img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    cv2.imshow('HSV Image', hsv_img)
    print("Converted to HSV.")

    # --- Thresholding ---
    # Thresholding requires a single-channel image, so we use the grayscale
image.

    # Define a threshold value
```

```
    threshold_value = 127 # A common mid-point for 0-255 pixel values
    max_value = 255     # The value assigned to pixels above the threshold

    # 1. Binary Thresholding
    # Pixels above threshold_value become max_value, others become 0
    ret, binary_threshold = cv2.threshold(gray_img, threshold_value, max_value,
cv2.THRESH_BINARY)
    cv2.imshow('Binary Threshold (Threshold=127)', binary_threshold)
    print("Applied Binary Thresholding.")

    # 2. Inverse Binary Thresholding
    # Pixels above threshold_value become 0, others become max_value
    ret, binary_inv_threshold = cv2.threshold(gray_img, threshold_value,
max_value, cv2.THRESH_BINARY_INV)
    cv2.imshow('Binary Inverse Threshold (Threshold=127)', binary_inv_threshold)
    print("Applied Inverse Binary Thresholding.")

    # 3. Truncate Thresholding
    # Pixels above threshold_value become threshold_value, others remain
unchanged
    ret, trunc_threshold = cv2.threshold(gray_img, threshold_value, max_value,
cv2.THRESH_TRUNC)
    cv2.imshow('Truncate Threshold (Threshold=127)', trunc_threshold)
    print("Applied Truncate Thresholding.")

    # 4. To Zero Thresholding
    # Pixels below threshold_value become 0, others remain unchanged
    ret, to_zero_threshold = cv2.threshold(gray_img, threshold_value, max_value,
cv2.THRESH_TOZERO)
    cv2.imshow('To Zero Threshold (Threshold=127)', to_zero_threshold)
    print("Applied To Zero Thresholding.")

    # 5. To Zero Inverse Thresholding
    # Pixels above threshold_value become 0, others remain unchanged
    ret, to_zero_inv_threshold = cv2.threshold(gray_img, threshold_value,
max_value, cv2.THRESH_TOZERO_INV)
    cv2.imshow('To Zero Inverse Threshold (Threshold=127)',
to_zero_inv_threshold)
    print("Applied To Zero Inverse Thresholding.")

    # 6. Otsu's Binarization
    # Automatically finds the optimal threshold value based on image histogram
    # Use THRESH_OTSU flag along with THRESH_BINARY
    ret_otsu, otsu_threshold = cv2.threshold(gray_img, 0, max_value,
cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    cv2.imshow(f'Otsu Threshold (Optimal Threshold={ret_otsu:.2f})',
otsu_threshold)
    print(f"Applied Otsu's Thresholding. Optimal threshold found:
{ret_otsu:.2f}")

    # Wait for a key press to close all windows
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    print("All windows closed.")

if __name__ == "__main__":
    input_image_file = 'color_test_image.jpg' # Replace with your image file
    print(f"Starting Lab 3 for image: {input_image_file}")
    color_space_and_thresholding(input_image_file)
    print("Lab 3 completed.")
```

Input An image file named `color_test_image.jpg` (preferably a color image with varying intensities) in the same directory as the script.

Expected Output Six separate windows will appear sequentially:

1. "Original Image (BGR)" showing the input image.
2. "Grayscale Image" showing the grayscale version.
3. "HSV Image" showing the image in HSV color space.
4. "Binary Threshold (Threshold=127)" showing the binary thresholded image.
5. "Binary Inverse Threshold (Threshold=127)" showing the inverse binary thresholded image.
6. "Truncate Threshold (Threshold=127)" showing the truncate thresholded image.
7. "To Zero Threshold (Threshold=127)" showing the to-zero thresholded image.
8. "To Zero Inverse Threshold (Threshold=127)" showing the to-zero inverse thresholded image.
9. "Otsu Threshold (Optimal Threshold=...)" showing the image binarized using Otsu's method. Each window will close upon a key press.

Lab 4: Morphological Operations (Opening and Closing) with OpenCV

Aim To understand and apply fundamental morphological operations, specifically "Opening" and "Closing," on binary images using OpenCV to remove noise and fill small holes.

Procedure

1. **Prepare an Image:** Use a binary image or an image that can be easily binarized (e.g., `binary_noise.png`).
2. **Write the Source Code:** Create a Python file (e.g., `lab4.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab4.py`

5. **Observe Output:** Three windows will appear, displaying the original binary image, the image after applying opening, and the image after applying closing.

Source Code

```
import cv2
import numpy as np

def morphological_operations(image_path):
    """
    Loads a grayscale image, applies binary thresholding,
    and then performs morphological opening and closing operations.
    """
    # Read the image in grayscale
    # It's good practice to work with grayscale for morphological operations
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    if img is None:
        print(f"Error: Could not load image from {image_path}")
        return

    print(f"Processing image for morphological operations: {image_path}")

    # Apply binary thresholding to ensure we have a binary image
    # Otsu's method is often good for automatically finding a threshold
    ret, binary_img = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
    cv2.imshow('Original Binary Image', binary_img)
    print("Image binarized.")

    # Define a kernel for morphological operations
    # A 5x5 rectangular kernel is common. You can experiment with different
shapes/sizes.
    kernel = np.ones((5, 5), np.uint8)
    print(f"Using a {kernel.shape[0]}x{kernel.shape[1]} kernel.")

    # --- Morphological Opening ---
    # Opening = Erosion followed by Dilation
    # Useful for removing small objects (noise) from the foreground
    # and smoothing contours of objects.
    opening = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel)
    cv2.imshow('Image After Opening', opening)
```

```python
    print("Applied Morphological Opening.")

    # --- Morphological Closing ---
    # Closing = Dilation followed by Erosion
    # Useful for filling small holes inside the foreground objects
    # and connecting nearby objects.
    closing = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
    cv2.imshow('Image After Closing', closing)
    print("Applied Morphological Closing.")

    # Wait for a key press to close all windows
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    print("All windows closed.")

if __name__ == "__main__":
    input_image_file = 'morph_test_image.png' # Replace with your image file
    print(f"Starting Lab 4 for image: {input_image_file}")
    morphological_operations(input_image_file)
    print("Lab 4 completed.")
```

Input A grayscale image file named `morph_test_image.png` (or any valid image format) in the same directory as the script. Ideally, this image should have some small noise dots or tiny holes to observe the effect of the operations.

Expected Output Three separate windows will appear:

1. "Original Binary Image" showing the binarized input image.
2. "Image After Opening" showing the image after the opening operation (small foreground noise removed).
3. "Image After Closing" showing the image after the closing operation (small holes filled, gaps connected). Each window will close upon a key press.

Lab 5: Image Acquisition and Display from Camera

Aim To capture live video frames from a connected camera (webcam) and display them in real-time on the screen using OpenCV.

Procedure

1. **Connect Camera:** Ensure a webcam is connected to your computer and is recognized by the system.
2. **Write the Source Code:** Create a Python file (e.g., `lab5.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab5.py`

5. **Observe Output:** A window will appear displaying the live feed from your camera. Press the 'q' key to stop the capture and close the window.

Source Code

```
import cv2

def capture_and_display_camera_feed():
    """
    Captures video frames from the default camera and displays them in real-
time.
    Press 'q' to quit.
    """
    # Initialize video capture object
    # 0 indicates the default camera. If you have multiple cameras,
    # you might try 1, 2, etc.
    cap = cv2.VideoCapture(0)

    # Check if camera opened successfully
    if not cap.isOpened():
        print("Error: Could not open video stream. Make sure camera is connected
and not in use.")
        return

    print("Camera feed started. Press 'q' to quit.")

    while True:
        # Read a frame from the camera
        # ret (boolean): True if frame was read successfully, False otherwise
        # frame (numpy array): The captured frame
        ret, frame = cap.read()

        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        # Display the captured frame
        cv2.imshow('Live Camera Feed', frame)

        # Wait for a key press for 1 millisecond
        # If 'q' is pressed, break the loop
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
```

```
    # Release the video capture object
    # This is important to free up camera resources
    cap.release()

    # Destroy all OpenCV windows
    cv2.destroyAllWindows()
    print("Camera feed stopped and windows closed.")

if __name__ == "__main__":
    print("Starting Lab 5: Camera Acquisition and Display.")
    capture_and_display_camera_feed()
    print("Lab 5 completed.")
```

Input A connected and functional webcam.

Expected Output A window titled "Live Camera Feed" will appear, displaying the real-time video stream from your webcam. The program will continue to display the feed until the 'q' key is pressed, at which point the window will close.

Lab 6: Simple Color Detection and Tracking

Aim To implement a basic color detection algorithm to identify and track objects of a specific color within a live video stream.

Procedure

1. **Connect Camera:** Ensure a webcam is connected.
2. **Choose a Color:** Decide on a color to detect (e.g., blue, green, red). You'll need to find its HSV range.
3. **Write the Source Code:** Create a Python file (e.g., `lab6.py`) and write the provided source code. Adjust the `lower_color` and `upper_color` HSV bounds based on the color you want to detect.
4. **Run the Program:** Execute the Python script:
5. `python lab6.py`

6. **Observe Output:** A window will display the live camera feed. Move an object of the chosen color in front of the camera. The detected color regions should be highlighted or appear as a mask. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np

def color_detection_and_tracking():
    """
    Detects and tracks objects of a specific color (e.g., blue) in a live video
stream.
    """
    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        print("Error: Could not open video stream. Make sure camera is
connected.")
        return

    print("Color detection started. Place an object of the target color in front
of the camera.")
    print("Press 'q' to quit.")

    # Define the HSV range for the color you want to detect
    # These are example values for a shade of BLUE.
    # You might need to adjust these values based on your lighting conditions
    # and the exact shade of the object you are tracking.
    # Use online HSV color pickers or experiment to find accurate ranges.
    lower_color = np.array([100, 50, 50])  # Lower bound for blue (Hue,
Saturation, Value)
    upper_color = np.array([130, 255, 255]) # Upper bound for blue

    # For green:
    # lower_color = np.array([40, 40, 40])
    # upper_color = np.array([80, 255, 255])

    # For red (note: red wraps around 0 in HSV, so often needs two ranges):
    # lower_red1 = np.array([0, 50, 50])
    # upper_red1 = np.array([10, 255, 255])
    # lower_red2 = np.array([170, 50, 50])
```

```
    # upper_red2 = np.array([180, 255, 255])

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        # Convert the frame from BGR to HSV color space
        hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

        # Create a mask for the specified color range
        # Pixels within the range will be white (255), others black (0)
        mask = cv2.inRange(hsv_frame, lower_color, upper_color)

        # If detecting red, you might need to combine two masks:
        # mask1 = cv2.inRange(hsv_frame, lower_red1, upper_red1)
        # mask2 = cv2.inRange(hsv_frame, lower_red2, upper_red2)
        # mask = cv2.add(mask1, mask2)

        # Apply a series of morphological operations to clean up the mask
        # This helps to remove small noise and fill small gaps
        kernel = np.ones((5, 5), np.uint8)
        mask = cv2.erode(mask, kernel, iterations=1)
        mask = cv2.dilate(mask, kernel, iterations=1)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)


        # Find contours in the mask
        # Contours are curves joining all continuous points (along the boundary)
        # having same color or intensity.
        contours, _ = cv2.findContours(mask, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        # Iterate through detected contours
        for contour in contours:
            # Calculate the area of the contour
            area = cv2.contourArea(contour)

            # Filter out small contours (noise)
            if area > 500: # Adjust this value based on object size
                # Get the bounding rectangle for the contour
                x, y, w, h = cv2.boundingRect(contour)

                # Draw a rectangle around the detected object
                cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2) #
Green rectangle
                cv2.putText(frame, 'Object Detected', (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

        # Display the original frame with detections and the mask
        cv2.imshow('Original Frame with Detections', frame)
        cv2.imshow('Color Mask', mask)

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

    cap.release()
    cv2.destroyAllWindows()
    print("Color detection stopped.")

if __name__ == "__main__":
    print("Starting Lab 6: Simple Color Detection and Tracking.")
    color_detection_and_tracking()
    print("Lab 6 completed.")
```

Input A live video stream from a webcam. An object of a distinct color (e.g., a blue ball, a green toy) to be placed in front of the camera.

Expected Output Two windows will appear:

1. "Original Frame with Detections" showing the live camera feed with a green bounding box drawn around regions detected as the target color.
2. "Color Mask" showing the binary mask, where white pixels represent the detected color. The bounding box will move and resize as the colored object moves, effectively tracking it. The program will terminate upon pressing 'q'.

Lab 7: Object Detection with Pre-trained Models (YOLO/SSD)

Aim To utilize a pre-trained deep learning model (such as YOLO or SSD) to detect common objects in a video stream and draw bounding boxes and labels around the detected instances.

Procedure

1. **Download Model Files:** This lab requires pre-trained model weights and configuration files. For example, for YOLOv3, you'd need `yolov3.weights` and `yolov3.cfg`, along with `coco.names` (for class labels). These files are large and need to be downloaded separately. You can find them on the official OpenCV GitHub or other deep learning model repositories. Place them in the same directory as your script.
   - Example download links (check for latest versions):
     - YOLOv3 weights: `https://pjreddie.com/media/files/yolov3.weights`
     - YOLOv3 config: `https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg`
     - COCO names: `https://github.com/pjreddie/darknet/blob/master/data/coco.names`
2. **Write the Source Code:** Create a Python file (e.g., `lab7.py`) and write the provided source code. Update the `model_weights`, `model_config`, and `labels_file` variables with the correct paths.
3. **Run the Program:** Execute the Python script:
4. `python lab7.py`

5. **Observe Output:** A window will display the live camera feed (or a video file). Bounding boxes and class labels will appear around detected objects. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np

def object_detection_with_yolo():
    """
    Performs object detection on a live camera feed using a pre-trained YOLO
model.
    """
    # --- Configuration for YOLO ---
    # Path to the YOLO weights file
    model_weights = 'yolov3.weights'
    # Path to the YOLO configuration file
    model_config = 'yolov3.cfg'
    # Path to the file containing class labels (e.g., COCO dataset classes)
    labels_file = 'coco.names'

    # Confidence threshold for detections (minimum confidence to consider a
detection)
    conf_threshold = 0.5
    # Non-maximum suppression threshold (to remove overlapping bounding boxes)
    nms_threshold = 0.4

    # Load class names from file
    try:
```

```
        with open(labels_file, 'rt') as f:
            classes = f.read().rstrip('\n').split('\n')
        print(f"Loaded {len(classes)} classes from {labels_file}")
    except FileNotFoundError:
        print(f"Error: Labels file '{labels_file}' not found. Please download
it.")
        return

    # Load the pre-trained YOLO model
    # cv2.dnn.readNet() loads a network from files
    net = cv2.dnn.readNet(model_weights, model_config)
    net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
    net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU) # Can change to
DNN_TARGET_CUDA if GPU is available

    # Get the names of the output layers
    # YOLO has three output layers for different scales
    output_layers = [net.getLayerNames()[i[0] - 1] for i in
net.getUnconnectedOutLayers()]
    print(f"YOLO output layers: {output_layers}")

    # Initialize video capture (0 for default camera)
    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        print("Error: Could not open video stream. Ensure camera is connected
and model files are correct.")
        return

    print("Object detection started. Press 'q' to quit.")

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        # Get frame dimensions
        height, width, channels = frame.shape

        # Create a blob from the image
        # The blob is the input to the neural network
        # Scale factor (1/255.0 to normalize pixel values to 0-1)
        # Size (416x416 is common for YOLOv3)
        # Mean subtraction (0,0,0 as YOLO was trained on images without mean
subtraction)
        # Swap RB (True because OpenCV uses BGR, but models often expect RGB)
        # Crop (False)
        blob = cv2.dnn.blobFromImage(frame, 1/255.0, (416, 416), swapRB=True,
crop=False)

        # Set the blob as input to the network
        net.setInput(blob)

        # Run forward pass to get output from the output layers
        # This performs the actual detection
        outs = net.forward(output_layers)

        # Initialize lists for detected bounding boxes, confidences, and class
IDs
        class_ids = []
        confidences = []
        boxes = []

        # Process each output layer
        for out in outs:
            # Process each detection in the output layer
```

```
            for detection in out:
                # Get class probabilities
                scores = detection[5:]
                # Find the class with the highest probability
                class_id = np.argmax(scores)
                confidence = scores[class_id]

                # Filter out weak detections below the confidence threshold
                if confidence > conf_threshold:
                    # Scale bounding box coordinates back to original image size
                    center_x = int(detection[0] * width)
                    center_y = int(detection[1] * height)
                    w = int(detection[2] * width)
                    h = int(detection[3] * height)

                    # Calculate top-left corner coordinates
                    x = int(center_x - w / 2)
                    y = int(center_y - h / 2)

                    boxes.append([x, y, w, h])
                    confidences.append(float(confidence))
                    class_ids.append(class_id)

        # Apply Non-Maximum Suppression (NMS) to remove redundant overlapping
boxes
        # This helps to keep only the best bounding box for each detected object
        indexes = cv2.dnn.NMSBoxes(boxes, confidences, conf_threshold,
nms_threshold)

        # Draw bounding boxes and labels on the frame
        if len(indexes) > 0:
            for i in indexes.flatten(): # flatten() is used because indexes can
be a 2D array
                x, y, w, h = boxes[i]
                label = str(classes[class_ids[i]])
                confidence = str(round(confidences[i], 2))
                color = (0, 255, 0) # Green color for bounding box

                cv2.rectangle(frame, (x, y), (x + w, y + h), color, 2)
                cv2.putText(frame, f"{label} {confidence}", (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

        # Display the frame with detections
        cv2.imshow('Object Detection (YOLO)', frame)

        # Exit if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
    print("Object detection stopped.")

if __name__ == "__main__":
    print("Starting Lab 7: Object Detection with Pre-trained Models.")
    print("Ensure 'yolov3.weights', 'yolov3.cfg', and 'coco.names' are in the
same directory.")
    object_detection_with_yolo()
    print("Lab 7 completed.")
```

Input A live video stream from a webcam. The necessary YOLO model files (`yolov3.weights`, `yolov3.cfg`, `coco.names`) must be present in the script's directory.

Expected Output A window titled "Object Detection (YOLO)" will display the live camera feed. As common objects (e.g., person, car, dog, chair) appear in the frame, green bounding boxes will be drawn around them, along with their predicted class label and confidence score. The program will terminate upon pressing 'q'.

Lab 8: Basic Object Tracking (KLT Tracker)

Aim To implement a basic object tracking algorithm, specifically the Kanade-Lucas-Tomasi (KLT) sparse optical flow tracker, to follow a selected moving object in a video stream.

Procedure

1. **Connect Camera:** Ensure a webcam is connected.
2. **Write the Source Code:** Create a Python file (e.g., `lab8.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab8.py`

5. **Select Object:** When the camera feed appears, use your mouse to draw a bounding box around the object you wish to track. Press 'Enter' or 'Space' to confirm the selection.
6. **Observe Output:** The selected object will be tracked, and a bounding box will follow its movement. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np

def klt_object_tracking():
    """
    Implements KLT (Kanade-Lucas-Tomasi) sparse optical flow for object 
tracking.
    Allows user to select an initial object to track.
    """
    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        print("Error: Could not open video stream. Make sure camera is 
connected.")
        return

    # Parameters for ShiTomasi corner detection (for feature points)
    feature_params = dict(maxCorners = 100,      # Maximum number of corners to 
return
                          qualityLevel = 0.3,    # Minimum accepted quality of 
image corners
                          minDistance = 7,       # Minimum Euclidean distance 
between corners
                          blockSize = 7)         # Size of an average block for 
computing a derivative covariation matrix

    # Parameters for KLT optical flow
    lk_params = dict(winSize = (15, 15),         # Size of the search window
                     maxLevel = 2,               # Maximum number of pyramid 
levels
                     criteria = (cv2.TERM_CRITERIA_EPS | 
cv2.TERM_CRITERIA_COUNT, 10, 0.03))
                                                 # Termination criteria for 
iterative search

    # Take first frame and find corners in it
    ret, old_frame = cap.read()
    if not ret:
```

```
        print("Failed to grab initial frame.")
        return

    # Convert first frame to grayscale
    old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)

    # --- User selects ROI for tracking ---
    print("Draw a bounding box around the object to track and press ENTER or
SPACE.")
    # cv2.selectROI allows the user to select a region of interest with the
mouse
    bbox = cv2.selectROI("Select Object to Track", old_frame, fromCenter=False,
showCrosshair=True)
    cv2.destroyWindow("Select Object to Track") # Close the selection window

    if bbox == (0, 0, 0, 0): # If no selection was made
        print("No object selected. Exiting.")
        cap.release()
        return

    x, y, w, h = [int(v) for v in bbox]
    # Crop the selected region from the grayscale frame
    roi_gray = old_gray[y:y+h, x:x+w]

    # Find good features to track within the selected ROI
    # These are the points that KLT will try to track
    p0 = cv2.goodFeaturesToTrack(roi_gray, mask = None, **feature_params)

    if p0 is None or len(p0) == 0:
        print("No good features found in the selected region. Exiting.")
        cap.release()
        return

    # Adjust feature points to be relative to the full frame
    p0[:, :, 0] += x
    p0[:, :, 1] += y

    # Create a mask image for drawing purposes
    mask = np.zeros_like(old_frame)

    print("Tracking started. Press 'q' to quit.")

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Calculate optical flow
        # p1: new positions of input feature points
        # st: status array (1 if feature is found, 0 otherwise)
        # err: error array (error for each feature)
        p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)

        # Select only the good points (those that were successfully tracked)
        if p1 is not None and st is not None:
            good_new = p1[st==1]
            good_old = p0[st==1]

            # If enough good points are tracked, update the bounding box
            if len(good_new) > 0:
                # Calculate the new bounding box based on the tracked points
                # This is a simple way to update the bounding box
                min_x = int(np.min(good_new[:, 0]))
```

```
                max_x = int(np.max(good_new[:, 0]))
                min_y = int(np.min(good_new[:, 1]))
                max_y = int(np.max(good_new[:, 1]))

                # Ensure the bounding box remains within frame boundaries
                min_x = max(0, min_x)
                min_y = max(0, min_y)
                max_x = min(width - 1, max_x)
                max_y = min(height - 1, max_y)

                # Draw the bounding box
                cv2.rectangle(frame, (min_x, min_y), (max_x, max_y), (0, 255,
0), 2)
                cv2.putText(frame, "Tracking", (min_x, min_y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

                # Draw the tracked feature points
                for i, (new, old) in enumerate(zip(good_new, good_old)):
                    a, b = new.ravel()
                    c, d = old.ravel()
                    # Draw a line between old and new position (optional, for
visualization)
                    mask = cv2.line(mask, (int(a), int(b)), (int(c), int(d)),
(0, 0, 255), 2)
                    # Draw a circle at the new position
                    frame = cv2.circle(frame, (int(a), int(b)), 5, (0, 255, 0),
-1)

                # Update the old frame and old points for the next iteration
                old_gray = frame_gray.copy()
                p0 = good_new.reshape(-1, 1, 2)
            else:
                print("Lost track of object. Re-select if needed.")
                # Optionally, you could re-initialize tracking here or prompt
user to re-select
                # For simplicity, we just stop drawing the box if points are
lost.
        else:
            print("No points to track. Exiting.")
            break


        # Combine the frame with the mask (for visualizing flow lines)
        # This makes the lines visible on top of the video feed
        img = cv2.add(frame, mask)

        cv2.imshow('Object Tracking (KLT)', img)

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

    cap.release()
    cv2.destroyAllWindows()
    print("Object tracking stopped.")

if __name__ == "__main__":
    print("Starting Lab 8: Basic Object Tracking (KLT Tracker).")
    klt_object_tracking()
    print("Lab 8 completed.")
```

Input A live video stream from a webcam. The user will interactively select an object in the first frame.

Expected Output A window titled "Object Tracking (KLT)" will display the live camera feed. After the user selects an object by drawing a bounding box, the system will attempt to track that object. A green bounding box will follow the object's movement, and small green circles (feature points) might be visible on the tracked object. Red lines might briefly appear showing the movement of these feature points. The program will terminate upon pressing 'q'.

Lab 9: Camera Calibration using Chessboard Pattern

Aim To calibrate a camera using a chessboard pattern to determine its intrinsic parameters (focal length, principal point, distortion coefficients) and correct for lens distortion.

Procedure

1. **Prepare Chessboard:** Obtain a printed chessboard pattern. Note the size of each square (e.g., 20mm) and the number of inner corners (e.g., 7x7).
2. **Capture Images:** Capture multiple images (at least 10-15) of the chessboard from different angles and distances, ensuring the entire chessboard is visible and well-lit in each image. Save them in a designated folder (e.g., `calibration_images`).
3. **Write the Source Code:** Create a Python file (e.g., `lab9.py`) and write the provided source code. Update `chessboard_size` and `square_size_mm` and the path to your calibration images.
4. **Run the Program:** Execute the Python script:
5. `python lab9.py`

6. **Observe Output:** The program will process the images, display detected corners, and then print the camera matrix, distortion coefficients, and reprojection error. It will also show an example of an undistorted image.

Source Code

```
import cv2
import numpy as np
import glob # For reading multiple image files

def camera_calibration():
    """
    Performs camera calibration using a set of chessboard images.
    Calculates camera matrix, distortion coefficients, and undistorts an example
image.
    """
    # --- Configuration ---
    # Define the size of the chessboard (number of inner corners per row and
column)
    # E.g., for an 8x8 chessboard, if you count inner corners, it's 7x7
    chessboard_size = (7, 7) # (cols, rows) of inner corners
    # Define the actual size of each square on the chessboard in your chosen
unit (e.g., millimeters)
    square_size_mm = 20.0 # Example: 20 mm per square

    # Path to your calibration images
    images_path = 'calibration_images/*.jpg' # Assumes images are in a folder
named 'calibration_images'

    # --- Prepare object points and image points ---
    # Object points are the (x, y, z) coordinates of the chessboard corners in
the real world.
    # We assume the chessboard is on the Z=0 plane.
    objp = np.zeros((chessboard_size[0] * chessboard_size[1], 3), np.float32)
    # Create a grid of 3D points for the chessboard corners
    objp[:, :2] = np.mgrid[0:chessboard_size[0],
0:chessboard_size[1]].T.reshape(-1, 2) * square_size_mm

    # Arrays to store object points and image points from all the images.
```

```
    objpoints = [] # 3D points in real world space
    imgpoints = [] # 2D points in image plane

    # Get list of calibration images
    images = glob.glob(images_path)

    if not images:
        print(f"Error: No images found at '{images_path}'. Please check the path
and file types.")
        print("Make sure you have images of a chessboard pattern in the
specified folder.")
        return

    print(f"Found {len(images)} calibration images.")

    # Iterate through each image
    for fname in images:
        img = cv2.imread(fname)
        if img is None:
            print(f"Warning: Could not load image {fname}. Skipping.")
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Find the chessboard corners
        # ret (boolean): True if corners are found, False otherwise
        # corners (numpy array): Array of detected corners
        ret, corners = cv2.findChessboardCorners(gray, chessboard_size, None)

        # If corners are found, add object points and image points
        if ret == True:
            objpoints.append(objp)
            imgpoints.append(corners)

            # Draw and display the corners (optional, for visualization)
            cv2.drawChessboardCorners(img, chessboard_size, corners, ret)
            cv2.imshow('Chessboard Corners', img)
            cv2.waitKey(500) # Display for 500ms
        else:
            print(f"Could not find chessboard corners in {fname}. Skipping.")

    cv2.destroyAllWindows()
    print("Finished processing images for corner detection.")

    # --- Perform Camera Calibration ---
    # If enough points are collected, perform calibration
    if len(objpoints) > 0 and len(imgpoints) > 0:
        print(f"Calibrating camera with {len(objpoints)} successful corner
detections...")
        # cv2.calibrateCamera returns:
        # ret: Reprojection error
        # mtx: Camera matrix (intrinsic parameters)
        # dist: Distortion coefficients
        # rvecs: Rotation vectors for each image
        # tvecs: Translation vectors for each image
        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1], None, None)

        print("\n--- Calibration Results ---")
        print(f"Reprojection Error: {ret}")
        print("\nCamera Matrix (Intrinsic Parameters):")
        print(mtx)
        print("\nDistortion Coefficients (k1, k2, p1, p2, k3):")
        print(dist)

        # --- Undistort an example image ---
        if images: # Use the first successfully loaded image for demonstration
```

```
            example_img_path = images[0]
            example_img = cv2.imread(example_img_path)
            if example_img is not None:
                h, w = example_img.shape[:2]

                # Get the optimal new camera matrix and ROI
                # This helps to remove black borders after undistortion
                new_camera_mtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist,
(w,h), 1, (w,h))

                # Undistort the image
                undistorted_img = cv2.undistort(example_img, mtx, dist, None,
new_camera_mtx)

                # Crop the image to the ROI (optional, to remove black edges)
                x, y, w, h = roi
                undistorted_img_cropped = undistorted_img[y:y+h, x:x+w]

                cv2.imshow('Original Image', example_img)
                cv2.imshow('Undistorted Image', undistorted_img)
                cv2.imshow('Undistorted Image (Cropped)',
undistorted_img_cropped)
                print("\nDisplaying original and undistorted example images.
Press any key to close.")
                cv2.waitKey(0)
                cv2.destroyAllWindows()
            else:
                print(f"Could not load example image for undistortion:
{example_img_path}")
        else:
            print("No images were successfully loaded to demonstrate
undistortion.")

    else:
        print("Calibration failed: Not enough successful chessboard corner
detections.")
        print("Ensure your images are clear, well-lit, and contain the full
chessboard.")

if __name__ == "__main__":
    print("Starting Lab 9: Camera Calibration using Chessboard Pattern.")
    print("Make sure you have a 'calibration_images' folder with chessboard
photos.")
    camera_calibration()
    print("Lab 9 completed.")
```

Input A folder named `calibration_images` containing at least 10-15 images of a chessboard pattern taken from various angles and distances. The `chessboard_size` and `square_size_mm` in the code must match your physical chessboard.

Expected Output During execution, windows might briefly appear showing the detected chessboard corners in each image. After processing, the console will print:

- The overall reprojection error (a lower value indicates better calibration).
- The `camera_matrix` (intrinsic parameters: focal lengths fx,fy and principal point cx,cy).
- The `distortion_coefficients` (radial k1,k2,k3 and tangential p1,p2). Finally, three windows will appear: "Original Image", "Undistorted Image", and "Undistorted Image (Cropped)", demonstrating the effect of distortion correction on one of the input images.

Lab 10: Motion Estimation using Optical Flow

Aim To implement a basic optical flow algorithm (e.g., Farneback dense optical flow) to estimate and visualize the motion of objects or pixels in a video stream.

Procedure

1. **Prepare Video Source:** Use a live webcam or a video file (e.g., `motion_video.mp4`) that contains some motion.
2. **Write the Source Code:** Create a Python file (e.g., `lab10.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab10.py`

5. **Observe Output:** A window will display the video stream. Areas with motion will show colored lines or a color map indicating the direction and magnitude of the optical flow. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np

def motion_estimation_optical_flow():
    """
    Estimates and visualizes dense optical flow using the Farneback algorithm.
    """
    # Initialize video capture (0 for default camera, or provide video file
path)
    cap = cv2.VideoCapture(0) # Use 0 for webcam, or 'path/to/your/video.mp4'
for a file

    if not cap.isOpened():
        print("Error: Could not open video stream. Make sure camera is connected
or video path is correct.")
        return

    print("Motion estimation started. Press 'q' to quit.")

    # Read the first frame
    ret, frame1 = cap.read()
    if not ret:
        print("Failed to grab initial frame.")
        cap.release()
        return

    # Convert the first frame to grayscale
    prvs = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)

    # Create a blank mask for drawing the optical flow vectors
    # This mask will be used to overlay the flow visualization on the original
frame
    hsv = np.zeros_like(frame1)
    # Set saturation to max (255) for vibrant colors
    hsv[..., 1] = 255

    while True:
        ret, frame2 = cap.read()
```

```
        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        next = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)

        # Calculate dense optical flow using Farneback algorithm
        # flow: A 2-channel array (dx, dy) that contains the optical flow
vectors for each pixel
        # parameters:
        #   prvs: first 8-bit single-channel input image
        #   next: second 8-bit single-channel input image
        #   flow: computed flow image that has the same size as input images and
type CV_32FC2
        #   pyr_scale: parameter, specifying the image scale (less than 1) to
build pyramids for each image
        #   levels: number of pyramid layers including the original image
        #   winsize: averaging window size; larger values give more robust, but
blurred motion field
        #   iterations: number of iterations the algorithm does at each pyramid
level
        #   poly_n: size of the pixel neighborhood used to find polynomial
expansion in each pixel
        #   poly_sigma: standard deviation of the Gaussian that is used to
smooth derivatives used as a basis for the polynomial expansion
        #   flags: operation flags (e.g., OPTFLOW_FARNEBACK_GAUSSIAN for a
Gaussian window)
        flow = cv2.calcOpticalFlowFarneback(prvs, next, None, 0.5, 3, 15, 3, 5,
1.2, 0)

        # Convert the flow vectors to magnitude and angle
        # mag: magnitude of the flow vectors
        # ang: angle (direction) of the flow vectors
        mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])

        # Set the hue (color) of the HSV image based on the angle (direction of
motion)
        # Angles are in radians, so convert to degrees and map to 0-180 for
OpenCV's HSV hue range
        hsv[..., 0] = ang * 180 / np.pi / 2
        # Set the value (brightness) of the HSV image based on the magnitude
(speed of motion)
        # Normalize magnitude to 0-255
        hsv[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM_MINMAX)

        # Convert HSV image back to BGR for display
        bgr_flow = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

        # Display the original frame and the optical flow visualization
        cv2.imshow('Original Frame', frame2)
        cv2.imshow('Optical Flow', bgr_flow)

        # Update the previous frame for the next iteration
        prvs = next

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

    cap.release()
    cv2.destroyAllWindows()
    print("Motion estimation stopped.")

if __name__ == "__main__":
    print("Starting Lab 10: Motion Estimation using Optical Flow.")
    motion_estimation_optical_flow()
    print("Lab 10 completed.")
```

Input A live video stream from a webcam or a video file (e.g., `motion_video.mp4`). The video should contain some moving objects or camera motion to observe the optical flow.

Expected Output Two windows will appear:

1. "Original Frame" showing the live video feed.
2. "Optical Flow" showing a visualization of the motion. Different colors will represent different directions of motion, and the intensity/brightness of the color will indicate the speed of motion. For example, a red hue might indicate motion to the right, while a blue hue indicates motion to the left. The program will terminate upon pressing 'q'.

Lab 11: Scene Classification using Machine Learning

Aim To build a machine learning model capable of classifying images into predefined scene categories (e.g., indoor, outdoor, kitchen, bedroom) using a given dataset.

Procedure

1. **Obtain Dataset:** Acquire a dataset of images categorized into various scenes (e.g., MIT Places, ImageNet subsets, or a custom dataset). Organize it into folders, where each folder name is a scene category (e.g., `dataset/indoor/img1.jpg`, `dataset/outdoor/img2.jpg`).
2. **Install Libraries:** Install necessary libraries:
3. `pip install scikit-learn tensorflow # or pytorch, depending on your choice`
4. `pip install matplotlib`

5. **Write the Source Code:** Create a Python file (e.g., `lab11.py`) and write the provided source code. This code will be conceptual, focusing on the workflow. Actual training might take significant time and resources.
6. **Run the Program:** Execute the Python script:
7. `python lab11.py`

8. **Observe Output:** The program will train a model and then print evaluation metrics (e.g., accuracy, classification report).

Source Code

```
import cv2
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC # Support Vector Machine for classification
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.applications import VGG16 # Using a pre-trained CNN as a
feature extractor
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import matplotlib.pyplot as plt

def scene_classification():
    """
    Builds a scene classifier using a pre-trained CNN for feature extraction
    and an SVM for classification.
    """
    # --- Configuration ---
    dataset_path = 'scene_dataset' # Path to your dataset folder (e.g.,
scene_dataset/indoor, scene_dataset/outdoor)
    image_size = (224, 224) # VGG16 expects 224x224 input images

    # --- Load and Preprocess Data ---
    print(f"Loading images from: {dataset_path}")
    data = []
    labels = []
    for scene_folder in os.listdir(dataset_path):
        scene_path = os.path.join(dataset_path, scene_folder)
        if os.path.isdir(scene_path):
            print(f"Processing scene: {scene_folder}")
```

```
            for img_name in os.listdir(scene_path):
                img_path = os.path.join(scene_path, img_name)
                try:
                    # Load image and resize it
                    img = load_img(img_path, target_size=image_size)
                    img_array = img_to_array(img)
                    data.append(img_array)
                    labels.append(scene_folder)
                except Exception as e:
                    print(f"Could not load image {img_path}: {e}")

    if not data:
        print("No images loaded. Please check your dataset path and structure.")
        print("Expected structure: scene_dataset/category1/img.jpg,
scene_dataset/category2/img.png")
        return

    data = np.array(data)
    labels = np.array(labels)

    print(f"Loaded {len(data)} images with {len(np.unique(labels))} unique
scenes.")

    # Encode labels to numerical format
    le = LabelEncoder()
    encoded_labels = le.fit_transform(labels)
    class_names = le.classes_
    print(f"Scene categories: {class_names}")

    # Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(data, encoded_labels,
test_size=0.2, random_state=42, stratify=encoded_labels)
    print(f"Training samples: {len(X_train)}, Testing samples: {len(X_test)}")

    # --- Feature Extraction using Pre-trained CNN (VGG16) ---
    # Load VGG16 model without the top (classification) layer
    # This allows us to use it as a fixed feature extractor
    print("Loading VGG16 model for feature extraction...")
    model = VGG16(weights='imagenet', include_top=False,
input_shape=(image_size[0], image_size[1], 3))
    model.trainable = False # Freeze the VGG16 layers

    # Preprocess images for VGG16
    X_train_processed = preprocess_input(X_train)
    X_test_processed = preprocess_input(X_test)

    print("Extracting features from training images...")
    train_features = model.predict(X_train_processed)
    # Flatten the features for SVM input
    train_features_flat = train_features.reshape(train_features.shape[0], -1)

    print("Extracting features from testing images...")
    test_features = model.predict(X_test_processed)
    test_features_flat = test_features.reshape(test_features.shape[0], -1)

    print(f"Extracted features shape (training): {train_features_flat.shape}")
    print(f"Extracted features shape (testing): {test_features_flat.shape}")

    # --- Train a Classifier (SVM) ---
    print("Training SVM classifier...")
    svm_classifier = SVC(kernel='linear', random_state=42) # Linear kernel is
often good for high-dimensional features
    svm_classifier.fit(train_features_flat, y_train)
    print("SVM classifier trained.")

    # --- Evaluate the Model ---
    print("\n--- Model Evaluation ---")
```

```
    y_pred = svm_classifier.predict(test_features_flat)

    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.4f}")

    # Print detailed classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, target_names=class_names))

    # --- Optional: Visualize some predictions ---
    print("\nDisplaying some example predictions. Close plot to finish.")
    plt.figure(figsize=(10, 8))
    for i in range(5): # Display 5 random test images
        idx = np.random.randint(0, len(X_test))
        img_to_show = X_test[idx].astype(np.uint8) # Convert back to uint8 for
display
        true_label = le.inverse_transform([y_test[idx]])[0]
        predicted_label = le.inverse_transform([y_pred[idx]])[0]

        plt.subplot(1, 5, i + 1)
        plt.imshow(img_to_show)
        plt.title(f"True: {true_label}\nPred: {predicted_label}",
                  color='green' if true_label == predicted_label else 'red')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    print("Starting Lab 11: Scene Classification using Machine Learning.")
    print("This lab requires a dataset of images organized by scene category.")
    print("Example dataset structure: 'scene_dataset/kitchen/img1.jpg',
'scene_dataset/bedroom/img2.jpg'")
    scene_classification()
    print("Lab 11 completed.")
```

Input A dataset of images organized into subfolders, where each subfolder represents a scene category (e.g., `scene_dataset/indoor`, `scene_dataset/outdoor`, `scene_dataset/kitchen`, `scene_dataset/bedroom`).

Expected Output The program will print:

- Messages indicating data loading, feature extraction, and model training progress.
- The overall accuracy of the classifier on the test set.
- A detailed classification report showing precision, recall, f1-score, and support for each scene category.
- Optionally, a plot showing a few test images with their true and predicted labels.

Lab 12: Semantic Segmentation

Aim To implement a semantic segmentation algorithm that labels each pixel in an image with its corresponding object class, providing a detailed, pixel-level understanding of the scene's content.

Procedure

1. **Obtain Dataset:** Acquire a semantic segmentation dataset (e.g., Pascal VOC, Cityscapes, or a custom dataset). These datasets typically contain images and corresponding pixel-level annotation masks.
2. **Install Libraries:** Install necessary deep learning libraries (e.g., TensorFlow or PyTorch) and other utilities:
3. `pip install tensorflow # or pytorch`
4. `pip install numpy matplotlib scikit-image`

5. **Write the Source Code:** Create a Python file (e.g., `lab12.py`) and write the provided source code. This will be a conceptual outline, as training a full segmentation model is complex and resource-intensive. It will focus on demonstrating inference with a pre-trained model or a simplified approach.
6. **Run the Program:** Execute the Python script:
7. `python lab12.py`

8. **Observe Output:** The program will load an image, perform segmentation, and display the original image alongside the predicted segmentation mask.

Source Code

```
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, 
concatenate, Activation
import matplotlib.pyplot as plt

def build_simple_unet(input_shape, num_classes):
    """
    Builds a simplified U-Net like architecture for semantic segmentation.
    This is a basic example and not a full-fledged U-Net.
    """
    inputs = Input(input_shape)

    # Encoder
    conv1 = Conv2D(32, 3, activation='relu', padding='same')(inputs)
    conv1 = Conv2D(32, 3, activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(64, 3, activation='relu', padding='same')(pool1)
    conv2 = Conv2D(64, 3, activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    # Bottleneck
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(pool2)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(conv3)
```

```
    # Decoder
    up4 = concatenate([UpSampling2D(size=(2, 2))(conv3), conv2], axis=-1)
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(up4)
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(conv4)

    up5 = concatenate([UpSampling2D(size=(2, 2))(conv4), conv1], axis=-1)
    conv5 = Conv2D(32, 3, activation='relu', padding='same')(up5)
    conv5 = Conv2D(32, 3, activation='relu', padding='same')(conv5)

    # Output layer (pixel-wise classification)
    outputs = Conv2D(num_classes, 1, activation='softmax')(conv5) # Softmax for
multi-class

    model = Model(inputs=inputs, outputs=outputs)
    return model

def semantic_segmentation():
    """
    Demonstrates semantic segmentation using a simplified U-Net model.
    This example uses dummy data for demonstration. In a real scenario,
    you would load a dataset with image-mask pairs and train the model.
    """
    # --- Configuration ---
    image_path = 'segmentation_test_image.jpg' # Example image for inference
    input_shape = (128, 128, 3) # Example input size for the model
    num_classes = 3 # Example: background, object1, object2 (adjust based on
your dataset)

    # --- Build and (Conceptually) Load Model ---
    print("Building a simplified U-Net model...")
    model = build_simple_unet(input_shape, num_classes)
    # In a real scenario, you would load pre-trained weights here:
    # model.load_weights('path_to_your_trained_weights.h5')
    print("Model built. (Note: This is an untrained model for demonstration.)")

    # --- Prepare a dummy image for inference ---
    try:
        original_img = load_img(image_path, target_size=input_shape[:2])
        img_array = img_to_array(original_img)
        # Add batch dimension
        input_img = np.expand_dims(img_array, axis=0)
        # Preprocess input for the model (e.g., normalize to -1 to 1 for
MobileNetV2 based models)
        input_img = preprocess_input(input_img)
        print(f"Loaded image {image_path} for inference.")
    except Exception as e:
        print(f"Error loading image {image_path}: {e}")
        print("Creating a dummy image for demonstration.")
        original_img = np.random.randint(0, 255, input_shape, dtype=np.uint8)
        input_img = np.expand_dims(original_img, axis=0)
        input_img = preprocess_input(input_img.astype(np.float32))


    # --- Perform Inference ---
    print("Performing conceptual inference...")
    # This will produce random output if the model is not trained
    predictions = model.predict(input_img)[0] # Get predictions for the single
image
    # The output is a probability map for each class at each pixel.
    # Take the argmax along the channel axis to get the most probable class for
each pixel.
    predicted_mask = np.argmax(predictions, axis=-1)

    print("Inference completed.")

    # --- Visualize Results ---
```

```
    # Create a colormap for visualization
    # Each class will have a distinct color
    cmap = plt.get_cmap('viridis', num_classes)
    colored_mask = cmap(predicted_mask / (num_classes - 1))[:, :, :3] #
Normalize for colormap

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.imshow(original_img.astype(np.uint8)) # Display original image
    plt.title('Original Image')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(colored_mask) # Display colored segmentation mask
    plt.title('Predicted Segmentation Mask')
    plt.axis('off')

    plt.suptitle('Semantic Segmentation (Conceptual Demo)')
    plt.show()
    print("Displaying original and predicted segmentation mask. Close plot to
finish.")

if __name__ == "__main__":
    print("Starting Lab 12: Semantic Segmentation.")
    print("This is a conceptual demonstration. A real implementation requires a
dataset, training, and potentially pre-trained models.")
    semantic_segmentation()
    print("Lab 12 completed.")
```

Input A `segmentation_test_image.jpg` (or any image) in the same directory. For a real implementation, a dataset with image-mask pairs would be required for training.

Expected Output A plot will appear showing two images:

1. "Original Image": The input image.
2. "Predicted Segmentation Mask": A colored mask where each color represents a different predicted object class for each pixel. Since this example uses an untrained model, the mask will likely appear random. In a real scenario with a trained model, it would accurately delineate objects. The plot will close when you close the window.

Lab 13: Moving Object Detection and Tracking in Dynamic Environments

Aim To develop a vision-based system that can detect and track moving objects in a dynamic environment by combining techniques like background subtraction, contour detection, and basic tracking logic.

Procedure

1. **Prepare Video Source:** Use a live webcam or a video file (e.g., `dynamic_scene.mp4`) where objects are moving against a relatively static or slowly changing background.
2. **Write the Source Code:** Create a Python file (e.g., `lab13.py`) and write the provided source code.
3. **Run the Program:** Execute the Python script:
4. `python lab13.py`

5. **Observe Output:** A window will display the video feed. Moving objects will be detected, and bounding boxes will be drawn around them, indicating they are being tracked. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np

def moving_object_detection_and_tracking():
    """
    Detects and tracks moving objects in a dynamic environment using background
subtraction.
    """
    cap = cv2.VideoCapture(0) # Use 0 for webcam, or 'path/to/your/video.mp4'
for a file

    if not cap.isOpened():
        print("Error: Could not open video stream. Make sure camera is connected
or video path is correct.")
        return

    # Initialize background subtractor
    # MOG2 (Mixture of Gaussians) is a common and robust background subtraction
algorithm.
    # history: Number of previous frames to consider for background modeling
    # varThreshold: Threshold on the squared Mahalanobis distance to decide if a
pixel is foreground
    # detectShadows: Whether to detect and mark shadows
    fgbg = cv2.createBackgroundSubtractorMOG2(history=500, varThreshold=16,
detectShadows=True)

    print("Moving object detection and tracking started. Press 'q' to quit.")

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame. Exiting...")
            break

        # Apply background subtractor to get the foreground mask
```

```
        # The foreground mask will be white for foreground pixels and black for
background
        fgmask = fgbg.apply(frame)

        # Apply morphological operations to clean up the foreground mask
        # Erosion removes small white noise (speckles)
        # Dilation fills small black holes and connects nearby foreground
regions
        kernel = np.ones((5, 5), np.uint8)
        fgmask = cv2.erode(fgmask, kernel, iterations=1)
        fgmask = cv2.dilate(fgmask, kernel, iterations=2)

        # Find contours in the foreground mask
        # Contours represent the boundaries of detected foreground objects
        contours, _ = cv2.findContours(fgmask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        # Iterate through detected contours
        for contour in contours:
            # Filter out small contours (noise) or very large contours (e.g.,
entire frame change)
            if cv2.contourArea(contour) < 500: # Minimum area for a valid object
(adjust as needed)
                continue

            # Get the bounding rectangle for the contour
            x, y, w, h = cv2.boundingRect(contour)

            # Draw a rectangle around the detected moving object on the original
frame
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2) # Green
bounding box
            cv2.putText(frame, 'Moving Object', (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

        # Display the original frame with detections and the foreground mask
        cv2.imshow('Original Frame with Moving Objects', frame)
        cv2.imshow('Foreground Mask', fgmask)

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

    cap.release()
    cv2.destroyAllWindows()
    print("Moving object detection and tracking stopped.")

if __name__ == "__main__":
    print("Starting Lab 13: Moving Object Detection and Tracking in Dynamic
Environments.")
    moving_object_detection_and_tracking()
    print("Lab 13 completed.")
```

Input A live video stream from a webcam or a video file (`dynamic_scene.mp4`) that contains moving objects.

Expected Output Two windows will appear:

1. "Original Frame with Moving Objects": Displays the live video feed with green bounding boxes drawn around detected moving objects.
2. "Foreground Mask": Shows the binary mask generated by background subtraction, where white pixels represent moving foreground objects. The bounding boxes will follow the detected moving objects. The program will terminate upon pressing 'q'.

Lab 14: Event Detection in Surveillance Videos

Aim To develop a system capable of detecting specific events of interest (e.g., abnormal behavior, object entry/exit, loitering) in surveillance video footage. This lab will focus on a simple event, like "object entering/exiting a defined region."

Procedure

1. **Prepare Video Source:** Use a surveillance video file (e.g., `surveillance_event.mp4`) where a simple event (like a person walking into or out of a specific area) can be observed.
2. **Write the Source Code:** Create a Python file (e.g., `lab14.py`) and write the provided source code. You will need to define the Region of Interest (ROI) coordinates in the code.
3. **Run the Program:** Execute the Python script:
4. `python lab14.py`

5. **Observe Output:** A window will display the video. A defined ROI will be visible. When a moving object crosses the boundary of this ROI, an "Event Detected!" message will appear on the screen. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np
import time

def event_detection_surveillance():
    """
    Detects a simple event: an object entering or exiting a predefined Region of
Interest (ROI)
    in a surveillance video using background subtraction and contour analysis.
    """
    # Initialize video capture (use a video file for consistent testing)
    video_path = 'surveillance_event.mp4' # Replace with your surveillance video
file
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print(f"Error: Could not open video file {video_path}. Please check the
path.")
        print("Alternatively, you can use a webcam by changing 'video_path' to
0.")
        return

    # Define the Region of Interest (ROI) coordinates (x, y, width, height)
    # This is the area where we want to detect events. Adjust these values for
your video.
    roi_x, roi_y, roi_w, roi_h = 200, 150, 400, 300 # Example ROI: top-left
(200,150), width 400, height 300

    # Initialize background subtractor
    fgbg = cv2.createBackgroundSubtractorMOG2(history=500, varThreshold=16,
detectShadows=True)

    # Variables for event tracking
    event_active = False
    event_start_time = None
    event_display_duration = 2 # seconds to display event message
```

```
    print(f"Event detection started for video: {video_path}. Press 'q' to
quit.")
    print(f"Monitoring ROI: ({roi_x},{roi_y}) to ({roi_x+roi_w},{roi_y+roi_h})")

    while True:
        ret, frame = cap.read()
        if not ret:
            print("End of video stream or failed to grab frame. Exiting...")
            break

        # Resize frame for consistent processing (optional, but good for
different video sizes)
        # frame = cv2.resize(frame, (640, 480)) # Uncomment and adjust if needed

        # Apply background subtractor
        fgmask = fgbg.apply(frame)

        # Apply morphological operations to clean up the mask
        kernel = np.ones((5, 5), np.uint8)
        fgmask = cv2.erode(fgmask, kernel, iterations=1)
        fgmask = cv2.dilate(fgmask, kernel, iterations=2)

        # Find contours in the foreground mask
        contours, _ = cv2.findContours(fgmask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        # Draw the ROI on the frame
        cv2.rectangle(frame, (roi_x, roi_y), (roi_x + roi_w, roi_y + roi_h),
(255, 0, 0), 2) # Blue ROI

        # Check for objects within or crossing the ROI
        object_in_roi = False
        for contour in contours:
            if cv2.contourArea(contour) < 1000: # Filter out small noise
                continue

            # Get bounding box of the contour
            x, y, w, h = cv2.boundingRect(contour)
            center_x, center_y = x + w // 2, y + h // 2

            # Check if the object's bounding box overlaps with the ROI
            # A more robust check might involve checking if the center is in
ROI,
            # or if a significant portion of the object is in ROI.
            if (center_x > roi_x and center_x < roi_x + roi_w and
                center_y > roi_y and center_y < roi_y + roi_h):
                object_in_roi = True
                cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2) #
Green for detected object
                cv2.putText(frame, 'Object', (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
                break # Only need to find one object in ROI for this simple
event

        # --- Event Logic ---
        if object_in_roi and not event_active:
            # Event started (object entered ROI)
            event_active = True
            event_start_time = time.time()
            print(f"Event Detected: Object entered ROI at {time.ctime()}")

        elif not object_in_roi and event_active:
            # Event ended (object exited ROI)
            event_active = False
            print(f"Event Ended: Object exited ROI at {time.ctime()}")

        # Display event message if active
```

```
        if event_active:
            cv2.putText(frame, "EVENT DETECTED!", (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 3)
            # You could also log this event to a file or trigger an alert

        # Display the frame
        cv2.imshow('Surveillance Event Detection', frame)
        cv2.imshow('Foreground Mask', fgmask)

        if cv2.waitKey(30) & 0xFF == ord('q'): # Wait 30ms between frames
            break

    cap.release()
    cv2.destroyAllWindows()
    print("Event detection system stopped.")

if __name__ == "__main__":
    print("Starting Lab 14: Event Detection in Surveillance Videos.")
    print("Ensure 'surveillance_event.mp4' is available or use a webcam (change
video_path to 0).")
    event_detection_surveillance()
    print("Lab 14 completed.")
```

Input A video file named `surveillance_event.mp4` (or a live webcam feed) that shows a scene where an object (e.g., a person) moves into and out of a defined rectangular region.

Expected Output Two windows will appear:

1. "Surveillance Event Detection": Displays the video feed. A blue rectangle will outline the predefined Region of Interest (ROI). When a moving object enters this ROI, a green bounding box will appear around it, and the text "EVENT DETECTED!" will be displayed prominently on the frame.
2. "Foreground Mask": Shows the binary mask of moving objects. Messages will be printed to the console when an event (object entering/exiting ROI) is detected. The program will terminate upon pressing 'q'.

Lab 15: Underwater Object Detection

Aim To develop a vision-based system specifically tailored for detecting objects in challenging underwater environments, addressing common issues like color distortion, low contrast, and scattering. This lab will focus on a conceptual approach involving image enhancement and then applying a general object detection model.

Procedure

1. **Prepare Underwater Images/Video:** Obtain images or video footage captured underwater (e.g., `underwater_fish.jpg`, `underwater_wreck.mp4`). These can be challenging to find, so you might use simulated or publicly available datasets.
2. **Install Libraries:** Ensure OpenCV and potentially deep learning frameworks (if using a pre-trained model) are installed.
3. **Write the Source Code:** Create a Python file (e.g., `lab15.py`) and write the provided source code. This code will demonstrate a simple color correction/enhancement technique followed by a conceptual object detection step.
4. **Run the Program:** Execute the Python script:
5. `python lab15.py`

6. **Observe Output:** Windows will display the original underwater image/frame, the enhanced version, and then the enhanced version with (conceptual) object detections. Press 'q' to quit.

Source Code

```
import cv2
import numpy as np
import os

# Placeholder for a conceptual object detection function
# In a real scenario, this would involve loading a pre-trained model (e.g.,
YOLO, SSD)
# trained on underwater datasets or fine-tuned for such environments.
def conceptual_object_detection(image):
    """
    A placeholder function for object detection.
    In a real application, this would use a deep learning model.
    For this conceptual lab, it will just draw a dummy bounding box.
    """
    # Simulate a detection for demonstration
    h, w, _ = image.shape
    # Example: Detect a "fish" in the center if image is large enough
    if w > 300 and h > 200:
        # Dummy bounding box coordinates (x, y, w, h)
        dummy_bbox = (w // 4, h // 4, w // 2, h // 2)
        x, y, bw, bh = dummy_bbox
        cv2.rectangle(image, (x, y), (x + bw, y + bh), (0, 255, 255), 2) #
Yellow box
        cv2.putText(image, 'Detected Object (Concept)', (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 2)
    return image

def underwater_image_enhancement(image):
    """
    Applies a simple color balance and contrast enhancement
    suitable for underwater images (conceptual).
```

```
    More advanced techniques might involve dehazing, CLAHE, or specific color
correction algorithms.
    """
    # Convert to LAB color space for better color manipulation
    lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
    l, a, b = cv2.split(lab)

    # Apply CLAHE (Contrast Limited Adaptive Histogram Equalization) to the L-
channel
    # This enhances contrast locally without over-amplifying noise
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
    cl = clahe.apply(l)

    # Merge the enhanced L-channel back with original a and b channels
    limg = cv2.merge([cl, a, b])
    enhanced_bgr = cv2.cvtColor(limg, cv2.COLOR_LAB2BGR)

    # Simple color balance (e.g., increasing red/green to counter blue/green
dominance)
    # This is a very basic adjustment and might need fine-tuning
    # enhanced_bgr = np.clip(enhanced_bgr * np.array([1.0, 1.1, 1.2]), 0,
255).astype(np.uint8) # BGR: Blue, Green, Red

    return enhanced_bgr

def underwater_object_detection_system():
    """
    Develops a conceptual vision-based system for underwater object detection,
    incorporating image enhancement and a placeholder for object detection.
    """
    # Use an image file for demonstration, or a video for a continuous stream
    input_source = 'underwater_fish.jpg' # Replace with your underwater
image/video file
    # If using a video, uncomment the line below and comment out the image
loading part
    # cap = cv2.VideoCapture(input_source)

    if not os.path.exists(input_source):
        print(f"Error: Input source '{input_source}' not found. Please provide a
valid underwater image or video.")
        print("Using a dummy black image for demonstration.")
        dummy_img = np.zeros((480, 640, 3), dtype=np.uint8)
        cv2.putText(dummy_img, "No Image Found", (200, 240),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)
        cv2.imshow('Original Underwater Image', dummy_img)
        cv2.imshow('Enhanced Underwater Image', dummy_img)
        cv2.imshow('Detected Objects (Conceptual)', dummy_img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
        return

    # --- Image Processing for a single image ---
    if input_source.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp',
'.tiff')):
        frame = cv2.imread(input_source)
        if frame is None:
            print(f"Error: Could not load image {input_source}.")
            return

        print(f"Processing single image: {input_source}")

        # 1. Image Enhancement
        enhanced_frame = underwater_image_enhancement(frame.copy())

        # 2. Object Detection (Conceptual)
        detected_frame = conceptual_object_detection(enhanced_frame.copy())
```

```
        # Display results
        cv2.imshow('Original Underwater Image', frame)
        cv2.imshow('Enhanced Underwater Image', enhanced_frame)
        cv2.imshow('Detected Objects (Conceptual)', detected_frame)

        print("Displaying results. Press any key to close.")
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    # --- Video Processing (Conceptual) ---
    else: # Assume it's a video file or webcam
        cap = cv2.VideoCapture(input_source)
        if not cap.isOpened():
            print(f"Error: Could not open video stream {input_source}.")
            return

        print(f"Processing video stream: {input_source}. Press 'q' to quit.")

        while True:
            ret, frame = cap.read()
            if not ret:
                print("End of video stream or failed to grab frame. Exiting...")
                break

            # 1. Image Enhancement
            enhanced_frame = underwater_image_enhancement(frame.copy())

            # 2. Object Detection (Conceptual)
            detected_frame = conceptual_object_detection(enhanced_frame.copy())

            # Display results
            cv2.imshow('Original Underwater Frame', frame)
            cv2.imshow('Enhanced Underwater Frame', enhanced_frame)
            cv2.imshow('Detected Objects (Conceptual)', detected_frame)

            if cv2.waitKey(30) & 0xFF == ord('q'):
                break

        cap.release()
        cv2.destroyAllWindows()
        print("Underwater object detection system stopped.")

if __name__ == "__main__":
    print("Starting Lab 15: Underwater Object Detection.")
    print("This is a conceptual lab. Real-world underwater detection requires
specialized datasets and models.")
    underwater_object_detection_system()
    print("Lab 15 completed.")
```

Input An underwater image file (e.g., `underwater_fish.jpg`) or a video file (e.g., `underwater_wreck.mp4`) in the same directory as the script.

Expected Output For an image input: Three windows will appear showing the original underwater image, the enhanced version (with improved color and contrast), and the enhanced version with a conceptual bounding box drawn around a detected object. For a video input: Three continuous windows will display the original frames, enhanced frames, and frames with conceptual detections. The program will terminate upon pressing any key (for images) or 'q' (for videos).