CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

Approved by AICTE | Accredited by NBA CSE, MCT, MECH | nirf 101 – 150 Band

CO-PO Mapping for Lab- Experiments

Academic Year: 2024 – 2025 **Year/Semester: III/VI**

Course Code/ Title: AD2603/Computer Vision Laboratory

Regulation: 2022

S. No.	Laboratory Experiments
1	OpenCV Installation and working with Python
2	Basic Image Processing - loading images, Cropping, Resizing, Thresholding, Contour analysis, Blob detection
3	Image Annotation – Drawing lines, text circle, rectangle, ellipse on images
4	Image Enhancement - Understanding Color spaces, color space conversion, Histogram equalization, Convolution, Image smoothing, Gradients, Edge Detection
5	Image Features and Image Alignment – Fourier, Hough, Extract ORB Image features, Feature matching, Feature matching-based image alignment
6	Image segmentation using Graphcut / Grabcut
7	Camera Calibration with circular grid
8	Pose Estimation
9	3D Reconstruction – Creating Depth map from stereo images

LAB MANUAL

Academic Year: 2024 – 2025 **Year/Semester: III/VI**

Course Code/ Title: AD2603/Computer Vision Laboratory

Regulation: 2022

SL. NO.	NAME OF THE PROGRAM	DATE	MARKS	SIGNATURE
1	OpenCV Installation and working with Python			
2	Basic Image Processing - loading images, Cropping, Resizing, Thresholding, Contour analysis, Blob detection			
3	Image Annotation – Drawing lines, text circle, rectangle, ellipse on images			
4	Image Enhancement - Understanding Color spaces, color space conversion, Histogram equalization, Convolution, Image smoothing, Gradients, Edge Detection			
5	Image Features and Image Alignment – Fourier, Hough, Extract ORB Image features, Feature matching, Feature matching-based image alignment			
6	Image segmentation using Graphcut / Grabcut			
7	Camera Calibration with circular grid			
8	Pose Estimation			
9	3D Reconstruction – Creating Depth map from stereo images			
10	Object Detection and Tracking using Kalman Filter, Camshift			

List of Equipment, Instruments, and Systems for the Laboratory Exercises

1. **Hardware Requirements**:
   - Desktop or laptop computers with adequate processing power (minimum: Intel i5 or equivalent, 8GB RAM).
   - Cameras (e.g., USB webcams or DSLR cameras) for image and video capture.
   - Calibration grids (e.g., circular grid or chessboard pattern printouts) for camera calibration experiments.
2. **Software and Libraries**:
   - Python programming environment (e.g., Anaconda, PyCharm, or Jupyter Notebook).
   - OpenCV library (latest stable version).
   - NumPy, Matplotlib, PyTorch, TensorFlow and other Python packages for numerical operations and visualization.
3. **Input Data and Resources**:
   - Sample images and videos for processing tasks (e.g., high-resolution images, stereo image pairs, video clips).
   - 3D object models or structured datasets for reconstruction and pose estimation tasks.
4. **Other Equipment**:
   - Projector or monitor for displaying results during demonstrations.
   - Lighting setup (optional) to ensure consistent illumination for image capture.
   - Storage devices (e.g., external hard drives) for saving datasets and results.
5. **Additional Tools (Optional)**:
   - Graphics processing unit (GPU) for accelerated computations (if required for real-time tasks).
   - Tripods or mounting systems for steady camera placement during experiments.

Verified by

Consumables for the Laboratory Experiments

The experiments outlined generally involve digital and software-based processing, so consumables are minimal. However, the following may be needed:

1. **Consumables for Camera Calibration**:
   - **Printed Calibration Grids**: Chessboard or circular grid patterns printed on high-quality paper. These may require reprinting periodically due to wear and tear.
2. **Consumables for Image and Video Capture**:
   - **Paper or Cardstock**: For background or target objects in experiments.
   - **Markers or Tape**: To mark object positions or fix calibration grids during capture.
3. **Power and Connectivity**:
   - **Batteries or Chargers**: For cameras and other electronic devices, if applicable.

   - **Cables**: USB or HDMI cables for camera-to-computer connectivity, which may occasionally require replacement.

Verified by

Experiment 1: OpenCV Installation and Working with Python

Aim: To install OpenCV and use it with Python for basic operations.

Procedure:

1. Open the Command Prompt (cmd) and check the Python version installed using:

   python --version

   If Python is not installed, download and install Python along with PIP.

2. Install OpenCV using the following command:

   pip install opencv-python

3. Verify the OpenCV installation by running:
4. import cv2
   print(cv2.version__)

Result: Successfully installed OpenCV and confirmed its functionality in Python.

Faculty Notes:

- Ensure students have administrative privileges to install software.
- Verify the compatibility of Python versions with OpenCV.

Experiment 2: Basic Image Processing

Aim: To perform basic image processing tasks such as loading images, cropping, resizing, thresholding, and contour analysis.

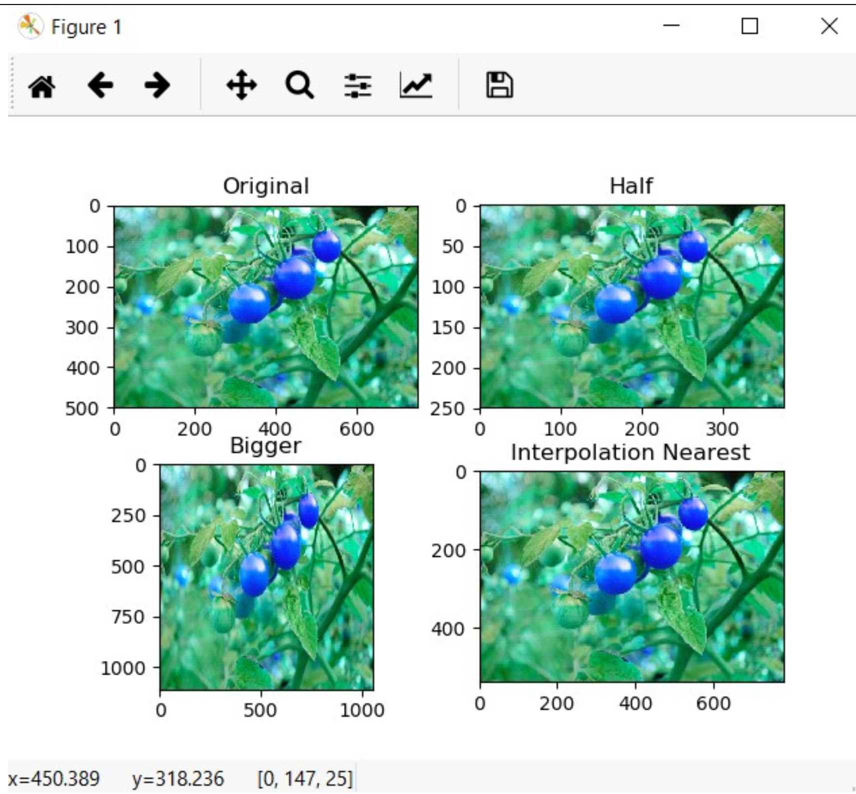
Algorithm:

1. Start the program.
2. Load an image using cv2.imread().
3. Resize the image to different dimensions.
4. Display the original and resized images using Matplotlib.
5. Crop a region of interest (ROI) from the image.
6. Display the original and cropped images using OpenCV.
7. Convert the image to grayscale.
8. Apply binary thresholding and display the thresholded images.
9. Find contours in the thresholded image.
10. Display the contours on the original image and stop

Programs:

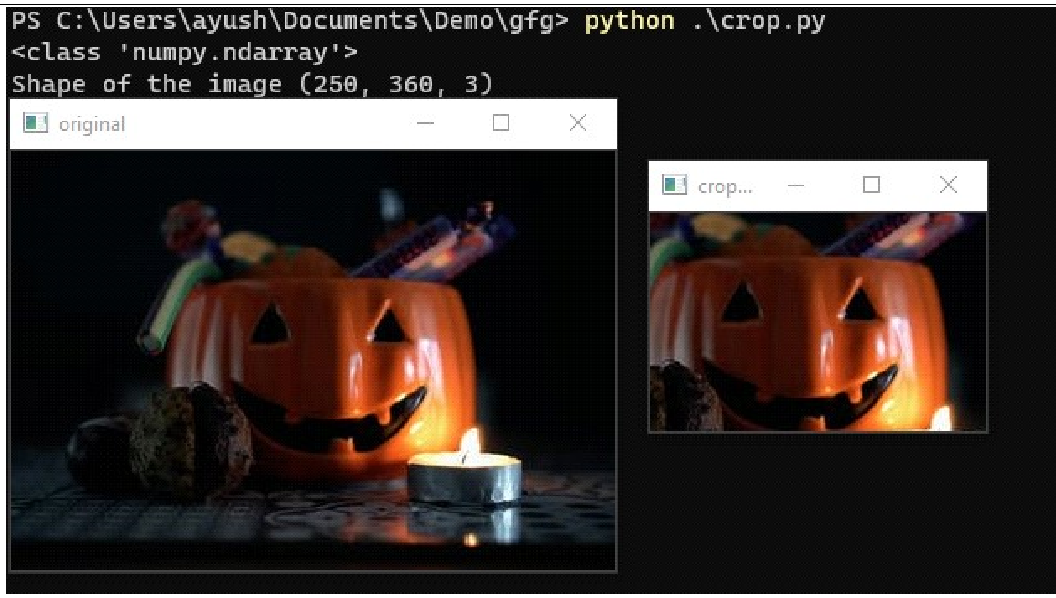
i) Resizing the Image

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

# Load the image
image_path = r"path_to_image"
image = cv2.imread(image_path, 1)

# Resize the image to create variations
half = cv2.resize(image, (0, 0), fx=0.1, fy=0.1)  # 10% of the original size
bigger = cv2.resize(image, (1050, 1610))          # Resized to 1050x1610
stretch_near = cv2.resize(image, (780, 540), interpolation=cv2.INTER_LINEAR)

# Titles and images for displaying
titles = ["Original", "Half", "Bigger", "Interpolation Nearest"]
images = [image, half, bigger, stretch_near]

# Plotting the images
plt.figure(figsize=(10, 8))
for i in range(len(images)):
    plt.subplot(2, 2, i + 1)
    plt.title(titles[i])
    plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB))
    plt.axis('off')
plt.tight_layout()
plt.show()
```



ii) Cropping the Image

```
import cv2

# Read Input Image
img = cv2.imread("test.jpeg")
# Check the type of read image
print(type(img))
print("Shape of the image", img.shape)
crop = img[50:180, 100:300]
cv2.imshow('original', img)
cv2.imshow('cropped', crop)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



iii) Thresholding

```
import cv2
import numpy as np

# Load the image
image1 = cv2.imread('input1.jpg')

# Convert to grayscale
img = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
ret, thresh1 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY)
ret, thresh2 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY_INV)
cv2.imshow('Binary Threshold', thresh1)
cv2.imshow('Binary Threshold Inverted', thresh2)
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

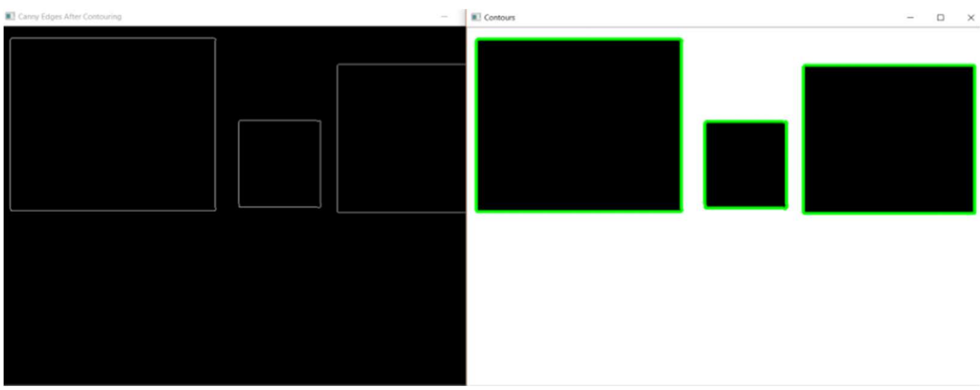
iv) Contour Analysis

```
import cv2
import numpy as np

# Load a simple image with shapes
image = cv2.imread('shapes.jpg')
cv2.waitKey(0)

# Convert to Grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Find Canny edges
edged = cv2.Canny(gray, 30, 200)
cv2.waitKey(0)

# Find contours
```

```
contours, hierarchy = cv2.findContours(edged, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
cv2.imshow('Canny Edges After Contouring', edged)
cv2.waitKey(0)
print("Number of Contours found = " + str(len(contours)))

# Draw all contours
cv2.drawContours(image, contours, -1, (0, 255, 0), 3)
cv2.imshow('Contours', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

- Resized images displayed with variations in dimensions.
- Cropped region of interest (ROI) from the image displayed.
- Thresholded images (binary and inverted) displayed.
- Contours displayed on the original image.



Result: Thus, basic image processing tasks, including loading, cropping, resizing, thresholding, and contour analysis, were successfully performed.

Faculty Notes:

- Ensure students use proper file paths for images.
- Encourage experimentation with different thresholding techniques and contour parameters.
- Explain the importance of resizing and thresholding in preprocessing for computer vision tasks.

Experiment 3: Image Annotation

Aim: To perform image annotation by drawing lines and rectangles on images.

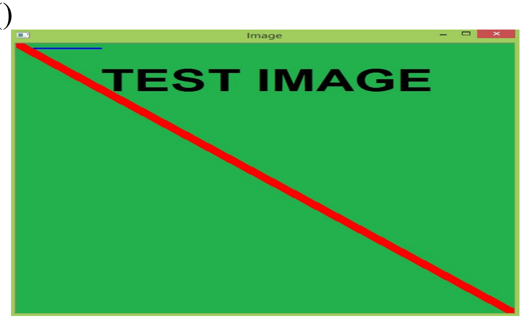
Algorithm:

1. Start the program.
2. Load an image using cv2.imread().
3. Define the window name for displaying the image.
4. Specify the start and end coordinates for the line or rectangle.
5. Choose a color for the line or rectangle in BGR format.
6. Set the line thickness.
7. Use cv2.line() or cv2.rectangle() to draw the annotation.
8. Display the annotated image using cv2.imshow().
9. Wait for a key press to close the window.

Programs:

i) Drawing Lines with OpenCV

```
import cv2

path = 'image_path'
image = cv2.imread(path)
start_point = (0, 0)
end_point = (250, 250)
color = (0, 255, 0)
thickness = 9

image = cv2.line(image, start_point, end_point, color, thickness)
cv2.imshow('Image with Line', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

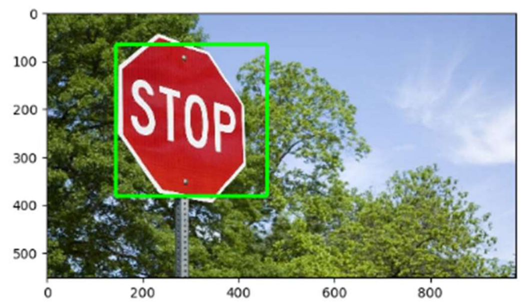


ii) Drawing Rectangles on Image

```
import cv2

path = 'image_path'
image = cv2.imread(path)
start_point = (5, 5)
end_point = (220, 220)
```

```
color = (255, 0, 0)
thickness = 2

image = cv2.rectangle(image, start_point, end_point, color, thickness)
cv2.imshow('Image with Rectangle', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Expected Output:



Annotated images with lines and rectangles drawn.

Result: Successfully annotated images using lines and rectangles.

Faculty Notes:

- Encourage students to explore other shapes like circles and polygons.
- Discuss practical applications of image annotation in computer vision.

Experiment 4: Image Enhancement

Aim: To perform image enhancement techniques using OpenCV.

Algorithm:

1. Start the program.
2. Import necessary libraries.
3. Read an image using cv2.imread().
4. Apply histogram equalization or edge detection.
5. Display input and enhanced images using cv2.imshow().
6. Close all OpenCV windows with cv2.destroyAllWindows().

Programs:

i) Histogram Equalization

```
import cv2
import numpy as np

img = cv2.imread('image_path', 0)
equ = cv2.equalizeHist(img)
res = np.hstack((img, equ))
cv2.imshow('Enhanced Image', res)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

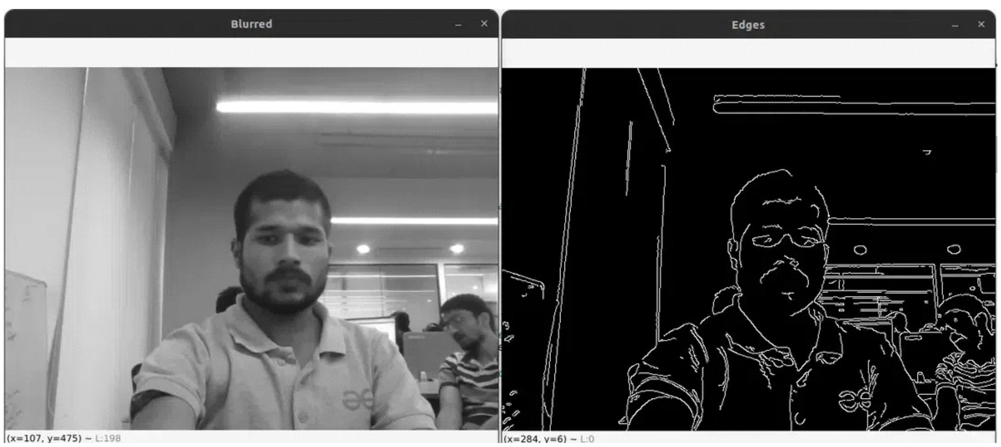
ii) Edge Detection

```
import cv2
```

```
def main():
    cap = cv2.VideoCapture(0)
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        blurred, edges = cv2.GaussianBlur(frame, (5, 5), 0), cv2.Canny(frame, 50, 150)
        cv2.imshow('Blurred', blurred)
        cv2.imshow('Edges', edges)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
main()
```

Expected Output:

Enhanced images with improved contrast or detected edges.



Result: Successfully performed image enhancement techniques.

Faculty Notes:

- Encourage students to test different parameters for edge detection and histogram equalization.
- Discuss applications like medical imaging and object detection.

Experiment 5: Image Transforms

Aim: To perform image transformation techniques like Fourier and Hough transformations using OpenCV.

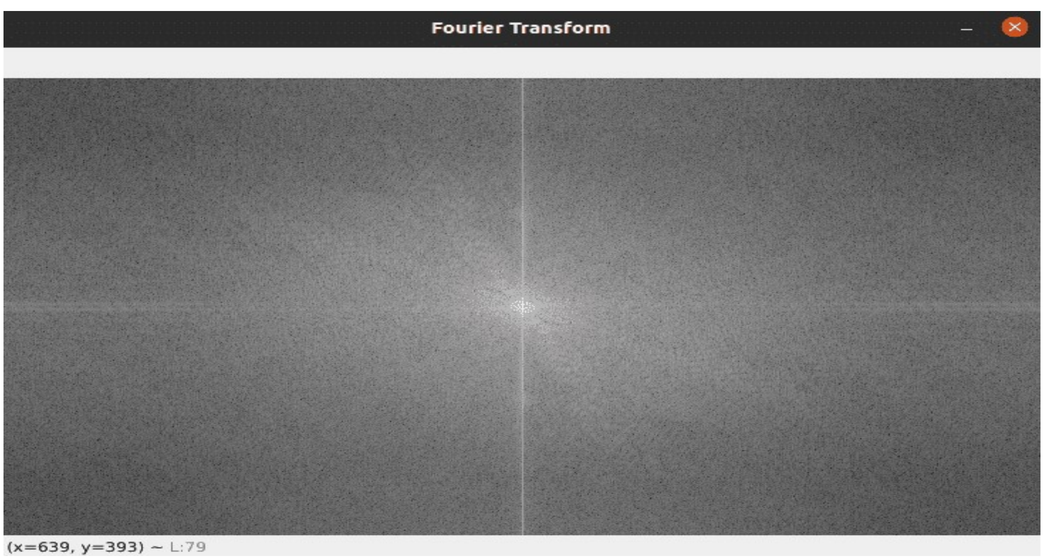
Algorithm:

1. Start the program.
2. Import necessary libraries.
3. Load an image and convert it to grayscale.
4. Apply the required transformation (e.g., Fourier or Hough).
5. Display the transformed image using cv2.imshow().
6. Close all OpenCV windows with cv2.destroyAllWindows().

Programs:

i) Fourier Transformation

```
import cv2
import numpy as np

image = cv2.imread('image_path')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
fourier = cv2.dft(np.float32(gray), flags=cv2.DFT_COMPLEX_OUTPUT)
magnitude = 20 * np.log(cv2.magnitude(fourier[:, :, 0], fourier[:, :, 1]))
cv2.imshow('Fourier Transform', magnitude)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



ii) Hough Transformation

```
import cv2
```

```
image = cv2.imread('image_path')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150)
circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1, 20, param1=50, param2=30)
cv2.imshow('Hough Transform', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Expected Output:

Transformed images with applied Fourier or Hough transformations.

Result: Successfully performed image transformation techniques.

Faculty Notes:

- Discuss applications of Fourier transforms in signal processing and image compression.
- Explore variations of Hough transforms for different shapes

Experiment 6: Image Segmentation

Aim: To perform image segmentation using the GrabCut algorithm.

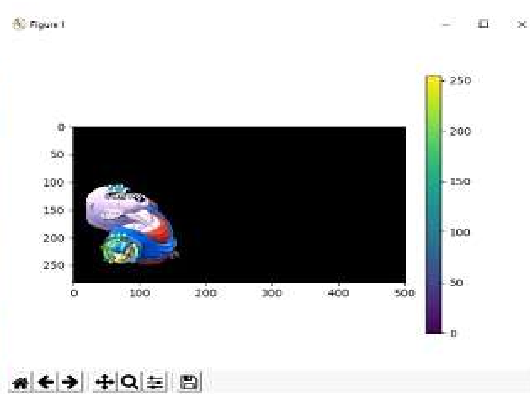
Algorithm:

1. Start the program.
2. Import the necessary libraries, including OpenCV and NumPy.
3. Load the input image.
4. Define an initial mask as a black image.
5. Initialize the background and foreground models.
6. Define a region of interest (ROI).
7. Apply the GrabCut algorithm.
8. Generate the final binary mask.
9. Multiply the input image with the mask to get the segmented image.
10. Display the result using Matplotlib.

Program:

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

image = cv2.imread('image.jpg')
mask = np.zeros(image.shape[:2], np.uint8)
backgroundModel = np.zeros((1, 65), np.float64)
foregroundModel = np.zeros((1, 65), np.float64)
rectangle = (50, 50, 450, 290)
cv2.grabCut(image, mask, rectangle, backgroundModel, foregroundModel, 5,
cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
segmented_image = image * mask2[:, :, np.newaxis]
plt.imshow(cv2.cvtColor(segmented_image, cv2.COLOR_BGR2RGB))
plt.show()
```

Expected Output:



Result: Successfully performed image segmentation using the GrabCut algorithm.

Faculty Notes:

- Highlight the importance of accurate ROI selection.
- Discuss practical applications such as object detection and photo editing

Experiment 7: Camera Calibration with Circular Grid

Aim: To perform camera calibration using a circular grid to determine camera parameters like distortion coefficients and projection errors.

Algorithm:

1. Start the program.
2. Import the necessary libraries, including NumPy and OpenCV, and define termination criteria.
3. Define the real-world coordinates of the circular grid.
4. Initialize vectors to store 3D and 2D points.
5. Load a set of images for calibration.
6. For each image, convert it to grayscale and find the positions of circles in the grid pattern using cv.findCirclesGrid().
7. If circles are detected, append the 3D and 2D points to their respective vectors and draw corners on the image.
8. Perform camera calibration using cv.calibrateCamera() with the object and image points.
9. Display the calibration results, including the projection error, camera matrix, distortion coefficients, rotation vectors, and translation vectors.
10. End the program.

Program:

```
# imports
import numpy as np
import cv2 as cv
import glob

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Real-world coordinates of the circular grid
obj3d = np.zeros((44, 3), np.float32)
a = [0, 36, 72, 108, 144, 180, 216, 252, 288, 324, 360]
b = [0, 72, 144, 216, 36, 108, 180, 252]
for i in range(44):
    obj3d[i] = (a[i // 4], b[i % 8], 0)

# Vectors to store 3D and 2D points
obj_points = []
img_points = []

# Load images from the directory
images = glob.glob('./Images/*.png')
for f in images:
    img = cv.imread(f)
```

```
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Detect circular grid patterns
    ret, corners = cv.findCirclesGrid(gray, (4, 11), None,
flags=cv.CALIB_CB_ASYMMETRIC_GRID)

    if ret:
        obj_points.append(obj3d)
        corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
        img_points.append(corners2)

        # Draw and save corners on the image
        cv.drawChessboardCorners(img, (4, 11), corners2, ret)
        cv.imwrite('output.jpg', img)
        cv.imshow('img', img)
        cv.waitKey(0)
cv.destroyAllWindows()

# Camera calibration
ret, camera_mat, distortion, rotation_vecs, translation_vecs = cv.calibrateCamera(
    obj_points, img_points, gray.shape[::-1], None, None
)

# Display results
print("Error in projection: \n", ret)
print("\nCamera matrix: \n", camera_mat)
print("\nDistortion coefficients: \n", distortion)
print("\nRotation vectors: \n", rotation_vecs)
print("\nTranslation vectors: \n", translation_vecs)
```

Output:

- **Error in projection:** 0.28397138993192417
- **Camera matrix:**
- [[2.98018946e+03 0.00000000e+00 -2.07790644e+02]
- [0.00000000e+00 2.98680309e+03 5.80328416e+02]
  [0.00000000e+00 0.00000000e+00 1.00000000e+00]]

Distortion coefficients:

[[-1.38990879e+00 1.28121501e+01 -1.76642504e-02 4.92392900e-02 -6.65051660e+01]]

Rotation vectors and Translation vectors: Displayed as arrays for each image.

Result: Thus, camera calibration using a circular grid was successfully performed, and the necessary camera parameters were determined.

Faculty Notes:

1. Ensure students understand the role of real-world coordinates and image coordinates in calibration.
2. Discuss the importance of camera calibration in real-world applications like robotics and augmented reality.
3. Encourage students to test calibration with different grid sizes and evaluate results.

Experiment 8: Pose Estimation

Aim: To estimate human pose using MediaPipe Pose.

Algorithm:

1. Start the program.
2. Import necessary libraries.
3. Initialize the MediaPipe Pose model.
4. Process each video frame.
5. Detect pose landmarks.
6. Draw landmarks and connections.
7. Display results and close upon exit.

Program:

```
import cv2
import mediapipe as mp

mp_drawing = mp.solutions.drawing_utils
mp_pose = mp.solutions.pose

pose = mp_pose.Pose(min_detection_confidence=0.5, min_tracking_confidence=0.5)
cap = cv2.VideoCapture('video.mp4')

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = pose.process(rgb_frame)

    mp_drawing.draw_landmarks(frame, results.pose_landmarks,
mp_pose.POSE_CONNECTIONS)
    cv2.imshow('Pose Estimation', frame)

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

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

Expected Output:

Video with overlaid pose landmarks and connections.

Result: Successfully performed pose estimation using MediaPipe Pose.

Faculty Notes:

- Discuss applications in sports analytics and rehabilitation.
- Encourage experimentation with confidence thresholds.

Experiment 9: Creating Depth Map from Stereo Images

Aim: To create a depth map using stereo image pairs.

Algorithm:

1. Start the program.
2. Import necessary libraries.
3. Load stereo image pairs as grayscale.
4. Initialize a StereoBM object.
5. Compute the disparity map.
6. Display the depth map.

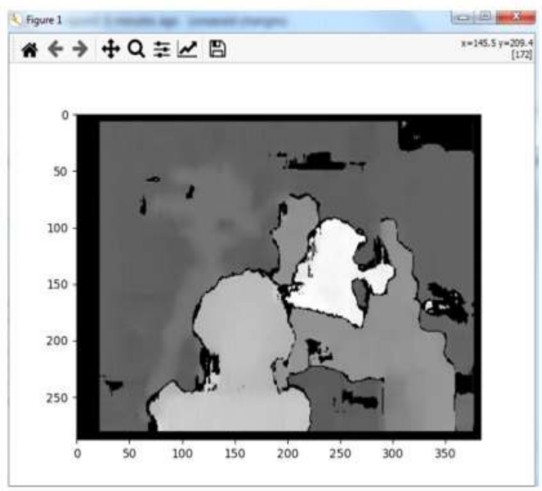
Program:

```
import cv2

import numpy as np

from matplotlib import pyplot as plt

# Load left and right stereo images in grayscale

imgL = cv2.imread('left_image.jpg', cv2.IMREAD_GRAYSCALE)

imgR = cv2.imread('right_image.jpg', cv2.IMREAD_GRAYSCALE)

# Check if the images are loaded successfully

if imgL is None or imgR is None:

    raise FileNotFoundError("One or both image files could not be loaded. Please check the
file paths.")

# Create a StereoBM object

num_disparities = 16 * 5  # Must be divisible by 16

block_size = 15           # Must be an odd number >= 5

stereo = cv2.StereoBM_create(numDisparities=num_disparities, blockSize=block_size)

# Compute disparity map

disparity = stereo.compute(imgL, imgR)

# Normalize disparity map for better visualization
```

```
disparity_normalized = cv2.normalize(disparity, None, alpha=0, beta=255,
norm_type=cv2.NORM_MINMAX)
disparity_normalized = np.uint8(disparity_normalized)
# Display the disparity map
plt.figure(figsize=(10, 7))
plt.imshow(disparity_normalized, cmap='gray')
plt.title('Disparity Map')
plt.axis('off')
plt.show()
```

Expected Output:

- Grayscale depth map indicating depth differences.



Result: Successfully created a depth map from stereo images.

Faculty Notes:

- Discuss applications in 3D reconstruction and autonomous vehicles.
- Encourage students to experiment with disparity and block size values.10

Experiment 10: Object Detection Using CamShift and OpenCV

Aim:

To perform object detection with the CamShift algorithm and OpenCV.

Algorithm:

1. Start the program.
2. Import the necessary libraries, including NumPy and OpenCV, and read the input video.
3. Take the first frame of the video and set up the initial region of interest (ROI) for tracking.
4. Convert the ROI from BGR to HSV format and perform a masking operation to create a histogram of the ROI.
5. Set up the termination criteria for CamShift tracking.
6. Enter a loop to process each frame of the video.
7. Resize the video frames if needed and apply thresholding and color space conversion.
8. Calculate the back projection of the frame and apply CamShift to get the new location of the tracked object.
9. Draw the tracking window on the video frame.
10. Display the processed frame and exit the loop when the 'Esc' key is pressed.

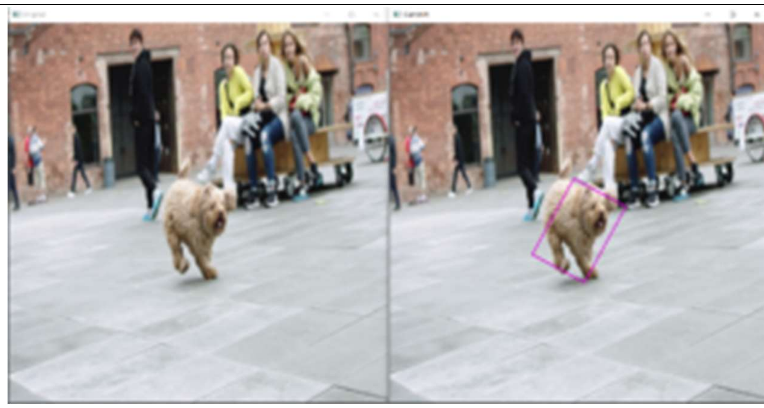
Program:

```
import numpy as np
import cv2 as cv

# Read the input video
cap = cv.VideoCapture('sample.mp4')

# Take the first frame of the video
ret, frame = cap.read()

# Setup initial region of tracker
x, y, width, height = 400, 440, 150, 150
track_window = (x, y, width, height)

# Set up the Region of Interest (ROI) for tracking
roi = frame[y:y + height, x:x + width]

# Convert ROI from BGR to HSV format
hsv_roi = cv.cvtColor(roi, cv.COLOR_BGR2HSV)

# Perform masking operation
mask = cv.inRange(hsv_roi, np.array((0., 60., 32.)), np.array((180., 255., 255)))
roi_hist = cv.calcHist([hsv_roi], [0], mask, [180], [0, 180])
cv.normalize(roi_hist, roi_hist, 0, 255, cv.NORM_MINMAX)

# Setup the termination criteria: 15 iterations or at least 2-pixel movement
term_crit = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 15, 2)
```

```
while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Resize the video frames
    frame = cv.resize(frame, (720, 720), interpolation=cv.INTER_CUBIC)

    # Perform thresholding on the video frames
    ret1, frame1 = cv.threshold(frame, 180, 155, cv.THRESH_TOZERO_INV)

    # Convert from BGR to HSV format
    hsv = cv.cvtColor(frame1, cv.COLOR_BGR2HSV)

    # Calculate back projection
    dst = cv.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)

    # Apply CamShift to get the new location
    ret2, track_window = cv.CamShift(dst, track_window, term_crit)

    # Draw the tracking window on the video frame
    pts = cv.boxPoints(ret2)
    pts = np.int0(pts)
    result = cv.polylines(frame, [pts], True, (0, 255, 255), 2)
    cv.imshow('CamShift', result)

    # Exit on 'Esc' key press
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

# Release the video capture object
cap.release()

# Close all OpenCV windows
cv.destroyAllWindows()
```

Output:

The program detects and tracks a selected object in a video using the CamShift algorithm, displaying the tracked object with a yellow bounding box on each processed frame.



Result:

Thus, object detection using the CamShift algorithm with OpenCV is successfully performed