ESTD. 2001

PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

for

CCS349-IMAGE AND VIDEO ANALYSIS LABORATORY

(Regulation 2021, V Semester)

(Odd Semester)

ACADEMIC YEAR: 2023 – 2024

PREPARED BY

B.Gunasundari,

Assistant Professor / CSE

PRATHYUSHA ENGINEERING COLLEGE

Exp.no. 1

T-pyramid of an image

Aim:

To Write a program that computes the T-pyramid of an image.

Algorithm:

1. Import the required libraries, OpenCV (cv2) and matplotlib.
2. Load an image "img1.jpg" from the specified file path.
3. Initialize a variable **layer** as a copy of the loaded image. This copy will be repeatedly downsampled to create the pyramid levels.
4. Iterate through a loop four times (from i = 0 to 3).
5. Inside the loop:
    - Create a subplot in a 2x2 grid (total of 4 subplots).
    - Downsample the **layer** using **cv2.pyrDown()**. This operation reduces the image size by half, creating a new level of the pyramid.
    - Display the downsampled image using **plt.imshow()** within the current subplot.
6. After displaying all four levels of the image pyramid, the code attempts to display each level individually using **cv2.imshow()** with a window title based on the current loop index **i**. However, there is a mistake in the code where the window title is not updated correctly, and it always displays "str(i)" as the title.
7. Finally, the code calls **cv2.waitKey(0)** to keep the OpenCV windows open until a key is pressed, and then it closes all OpenCV windows with **cv2.destroyAllWindows()**.

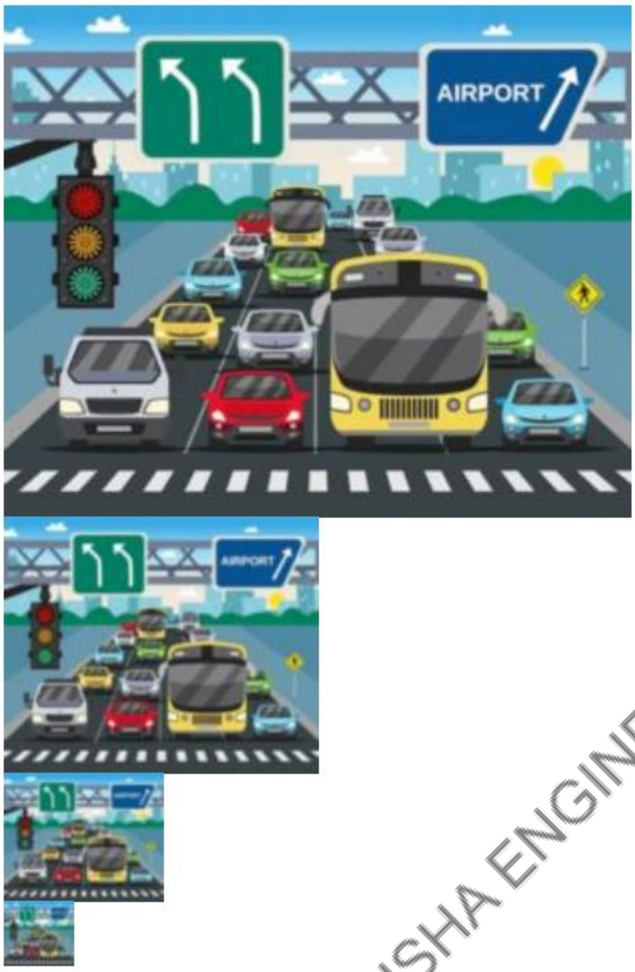
Program:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread("E:\Backup 14.4.23\image\img1.jpg")

layer = img.copy()

for i in range(4):
   plt.subplot(2, 2, i + 1)
   layer = cv2.pyrDown(layer)
   plt.imshow(layer)
   cv2.imshow("str(i)", layer)
   cv2.waitKey(0)
```

PRATHYUSHA ENGINEERING COLLEGE

```
cv2.destroyAllWindows()
```

Output:



PRATHYUSHA ENGINEERING COLLEGE

Result:

Thus the program that computes the T-pyramid of an image is executed successfully.

Exp. No.2

QUAD TREE

Aim:

To Write a program that derives the quad tree representation of an image using the homogeneity criterion of equal intensity

Algorithm:

1. Define a Quadtree Node structure to represent each node in the quadtree. Each node should contain the following information:
    - Position (x, y): The top-left corner of the node within the image.

    - Size: The width and height of the node.
    - Color: The dominant color of the node.
    - Children: An array or a dictionary to store child nodes.
    - Termination Condition: A condition that determines when to stop subdividing.
2. Initialize the quadtree by creating the root node, which represents the entire image.
3. Define the termination condition, which could be based on a threshold for color similarity, a maximum depth, or any other criterion. If the termination condition is met, mark the current node as a leaf node.
4. If the termination condition is not met, subdivide the current node into four quadrants, each representing a subregion of the image:
    - Divide the current node's size by 2.
    - Create four child nodes, one for each quadrant.
    - Determine the dominant color for each quadrant.
    - Recursively apply the quadtree algorithm to each child node.
5. Repeat the subdivision process for each child node until the termination condition is met for each leaf node.

PRATHYUSHA ENGINEERING COLLEGE

Program:

```
import numpy as np
import cv2
from PIL import Image, ImageDraw

MAX_DEPTH = 8
DETAIL_THRESHOLD = 13
SIZE_MULT = 1

def average_colour(image):
   # convert image to np array
   image_arr = np.asarray(image)

   # get average of whole image
   avg_color_per_row = np.average(image_arr, axis=0)
   avg_color = np.average(avg_color_per_row, axis=0)

   return (int(avg_color[0]), int(avg_color[1]), int(avg_color[2]))

def weighted_average(hist):
   total = sum(hist)
   error = value = 0

   if total > 0:
      value = sum(i * x for i, x in enumerate(hist)) / total
```

```
        error = sum(x * (value - i) ** 2 for i, x in enumerate(hist)) / total
        error = error ** 0.5

    return error

def get_detail(hist):
    red_detail = weighted_average(hist[:256])
    green_detail = weighted_average(hist[256:512])
    blue_detail = weighted_average(hist[512:768])

    detail_intensity = red_detail * 0.2989 + green_detail * 0.5870 + blue_detail * 0.1140

    return detail_intensity

class Quadrant():
    def __init__(self, image, bbox, depth):
        self.bbox = bbox
        self.depth = depth
        self.children = None
        self.leaf = False

        # crop image to quadrant size
        image = image.crop(bbox)
        hist = image.histogram()

        self.detail = get_detail(hist)
        self.colour = average_colour(image)

    def split_quadrant(self, image):
        left, top, width, height = self.bbox

        # get the middle coords of bbox
        middle_x = left + (width - left) / 2
        middle_y = top + (height - top) / 2

        # split root quadrant into 4 new quadrants
        upper_left = Quadrant(image, (left, top, middle_x, middle_y), self.depth+1)
        upper_right = Quadrant(image, (middle_x, top, width, middle_y), self.depth+1)
        bottom_left = Quadrant(image, (left, middle_y, middle_x, height), self.depth+1)
        bottom_right = Quadrant(image, (middle_x, middle_y, width, height), self.depth+1)

        # add new quadrants to root children
        self.children = [upper_left, upper_right, bottom_left, bottom_right]
```

PRATHYUSHA ENGINEERING COLLEGE

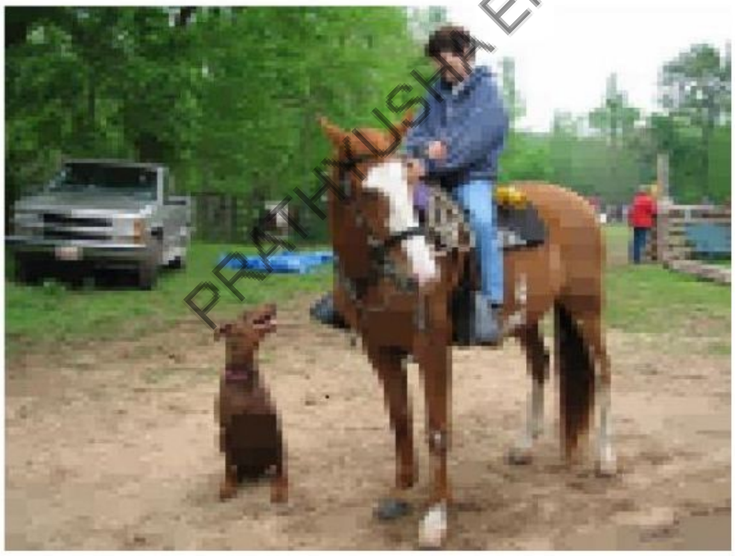
```
class QuadTree():
    def __init__(self, image):
        self.width, self.height = image.size

        # keep track of max depth achieved by recursion
        self.max_depth = 0

        # start compression
        self.start(image)

    def start(self, image):
        # create initial root
        self.root = Quadrant(image, image.getbbox(), 0)

        # build quadtree
        self.build(self.root, image)

    def build(self, root, image):
        if root.depth >= MAX_DEPTH or root.detail <= DETAIL_THRESHOLD:
            if root.depth > self.max_depth:
                self.max_depth = root.depth

            # assign quadrant to leaf and stop recursing
            root.leaf = True
            return

        # split quadrant if there is too much detail
        root.split_quadrant(image)

        for children in root.children:
            self.build(children, image)

    def create_image(self, custom_depth, show_lines=False):
        # create blank image canvas
        image = Image.new('RGB', (self.width, self.height))
        draw = ImageDraw.Draw(image)
        draw.rectangle((0, 0, self.width, self.height), (0, 0, 0))

        leaf_quadrants = self.get_leaf_quadrants(custom_depth)

        # draw rectangle size of quadrant for each leaf quadrant
        for quadrant in leaf_quadrants:
            if show_lines:
                draw.rectangle(quadrant.bbox, quadrant.colour, outline=(0, 0, 0))
```

PRATHYUSHA ENGINEERING COLLEGE

```
        else:
            draw.rectangle(quadrant.bbox, quadrant.colour)

    return image

def get_leaf_quadrants(self, depth):
    if depth > self.max_depth:
        raise ValueError('A depth larger than the trees depth was given')

    quandrants = []

    # search recursively down the quadtree
    self.recursive_search(self, self.root, depth, quandrants.append)

    return quandrants

def recursive_search(self, tree, quadrant, max_depth, append_leaf):
    # append if quadrant is a leaf
    if quadrant.leaf == True or quadrant.depth == max_depth:
        append_leaf(quadrant)

    # otherwise keep recursing
    elif quadrant.children != None:
        for child in quadrant.children:
            self.recursive_search(tree, child, max_depth, append_leaf)

def create_gif(self, file_name, duration=1000, loop=0, show_lines=False):
    gif = []
    end_product_image = self.create_image(self.max_depth, show_lines=show_lines)

    for i in range(self.max_depth):
        image = self.create_image(i, show_lines=show_lines)
        gif.append(image)

    # add extra images at end
    for _ in range(4):
        gif.append(end_product_image)

    gif[0].save(
        file_name,
        save_all=True,
        append_images=gif[1:],
        duration=duration, loop=loop)
```
PRATHYUSHA ENGINEERING COLLEGE

```
if __name__ == '__main__':
    #image_path = "./images/eye.jpg"
    image_path = "E:\Backup 14.4.23\image\img1.jpg"

    # load image
    image = Image.open(image_path)
    image = image.resize((image.size[0] * SIZE_MULT, image.size[1] * SIZE_MULT))

    # create quadtree
    quadtree = QuadTree(image)

    # create image with custom depth
    depth = 7
    image = quadtree.create_image(depth, show_lines=False)
    quadtree.create_gif("mountain_quadtree.gif", show_lines=True)

    # show image
    # image.show()

    image.save("E:\Backup 14.4.23\image\img111.jpg")
```

Output:



Result:
Thus the program that derives the quad tree representation of an image using the homogeneity criterion of equal intensity is executed successfully.

PRATHYUSHA ENGINEERING COLLEGE

Exp. No.3

GEOMETRIC TRANSFORMATION OF IMAGE

Aim:

To develop programs for the following geometric transforms: (a) Rotation (b) Change of scale (c) Skewing (d) Affine transform calculated from three pairs of corresponding points

Algorithm:

(a) Rotation

1. Import the Pillow library:
    - The code starts by importing the **Image** module from the Pillow library.
2. Open the original image:
    - It opens an image from the file path "E:\Backup 14.4.23\image\img1.jpg" and assigns it to the **Original_Image** variable.
3. Rotate the image by 180 degrees:
    - The **rotate** method is used to rotate the original image by 180 degrees, and the result is stored in **rotated_image1**.
4. Rotate the image by 90 degrees (counter-clockwise):
    - The **transpose** method is used with the argument **Image.ROTATE_90** to rotate the original image by 90 degrees counter-clockwise (also known as a counterclockwise or left rotation). The result is stored in **rotated_image2**.
5. Rotate the image by 60 degrees:
    - The **rotate** method is used to rotate the original image by 60 degrees, and the result is stored in **rotated_image3**.
6. Display the rotated images:
    - The **show** method is called on each of the rotated images to display them.

PRATHYUSHA ENGINEERING COLLEGE

(b) Change of scale

1. Import the OpenCV library:
    - The code starts by importing the OpenCV library.
2. Read the original image:
    - It reads an image from the file path "E:\Backup 14.4.23\image\img1.jpg" using **cv2.imread** with the flag **cv2.IMREAD_UNCHANGED**. This flag loads the image as-is, including the alpha channel if it exists.
3. Print the original image dimensions:

- The code prints the original image's dimensions (height, width, and number of channels) using **img.shape**.

4. Calculate the new dimensions for resizing:
   - The code calculates the new dimensions for resizing based on a specified scale percentage. In this case, the image is resized to 40% of its original size.
5. Resize the image:
   - The **cv2.resize** function is used to resize the image to the new dimensions (**dim**) using the specified interpolation method (**cv2.INTER_AREA**). The interpolation method is often used for downscaling to ensure better quality.
6. Print the resized image dimensions:
   - The code prints the dimensions of the resized image using **resized.shape**.
7. Display the resized image:
   - The resized image is displayed using **cv2.imshow**.
8. Wait for a key press and close the window:
   - The code waits for a key press with **cv2.waitKey()**.
   - It then closes all OpenCV windows using **cv2.destroyAllWindows()**.

(c) Skewing

1. Import the necessary libraries:
   - **numpy** for numerical operations.
   - **skimage** for image processing.
   - **deskew** to perform the skew detection and correction.
2. Read an image:
   - It reads an image from the file path "E:\imageoutput.jpg" using **io.imread** from scikit-image.
3. Convert the image to grayscale:
   - The code converts the color image to grayscale using **rgb2gray** from scikit-image.
4. Determine the skew angle:
   - The **determine_skew** function from the **deskew** library is used to automatically determine the skew angle in the grayscale image.
5. Rotate the image to correct the skew:
   - The code rotates the original image by the determined angle using **rotate** from scikit-image. This corrects the skew in the image.
   - The result is multiplied by 255 to ensure that pixel values remain in the range [0, 255].
6. Save the corrected image:
   - The corrected image is saved to "E:\imageoutput1.jpg" using **io.imsave**. It's cast to the **np.uint8** data type to ensure the correct data type for image saving.

(d) Affine transform calculated from three pairs of corresponding points

1. Import necessary libraries:

PRATHYUSHA ENGINEERING COLLEGE

- The code imports OpenCV (**cv2**) for image processing, NumPy (**np**) for numerical operations, and Matplotlib (**plt**) for displaying images.

2. Read and convert the image:
    - The code reads an image from the file path "E:/img.jpg" using **cv2.imread**. It is then converted to the RGB color space using **cv2.cvtColor**.
3. Define source and target points:
    - **pt1** contains the coordinates of the three vertices of a triangular region in the source image.
    - **pt2** contains the corresponding coordinates for the three vertices in the output image, defining how the triangular region should be transformed.
4. Create a transformation matrix:
    - The **cv2.getAffineTransform** function is used to calculate the affine transformation matrix (**Mat**) based on the source (**pt1**) and target (**pt2**) points.
5. Apply the affine transformation:
    - **cv2.warpAffine** is used to apply the affine transformation to the original image (**img**) using the transformation matrix (**Mat**). The output is stored in the **dst** variable.
6. Display the original and transformed images:
    - The code uses Matplotlib to display the original image and the transformed image side by side.
7. Show the images:
    - The images are displayed using **plt.show()**.

Program:

(a) Rotation

```
from PIL import Image

Original_Image = Image.open("E:\Backup 14.4.23\image\img1.jpg")

# Rotate Image By 180 Degree
rotated_image1 = Original_Image.rotate(180)


rotated_image2 = Original_Image.transpose(Image.ROTATE_90)

rotated_image3 = Original_Image.rotate(60)

rotated_image1.show()
rotated_image2.show()
rotated_image3.show()
```

PRATHYUSHA ENGINEERING COLLEGE

(b) Change of scale

```
import cv2

img = cv2.imread("E:\Backup 14.4.23\image\img1.jpg", cv2.IMREAD_UNCHANGED)

print('Original Dimensions : ',img.shape)

scale_percent = 40 # percent of original size
width = int(img.shape[1] * scale_percent / 100)
height = int(img.shape[0] * scale_percent / 100)
dim = (width, height)

# resize image
resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

print('Resized Dimensions : ',resized.shape)

cv2.imshow("Resized image", resized)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

(c) Skewing

```
import numpy as np
from skimage import io
from skimage.color import rgb2gray
from skimage.transform import rotate

from deskew import determine_skew

image = io.imread('E:\imageoutput.jpg')
grayscale = rgb2gray(image)
angle = determine_skew(grayscale)
rotated = rotate(image, angle, resize=True) * 255
io.imsave('E:\imageoutput1.jpg', rotated.astype(np.uint8))
```

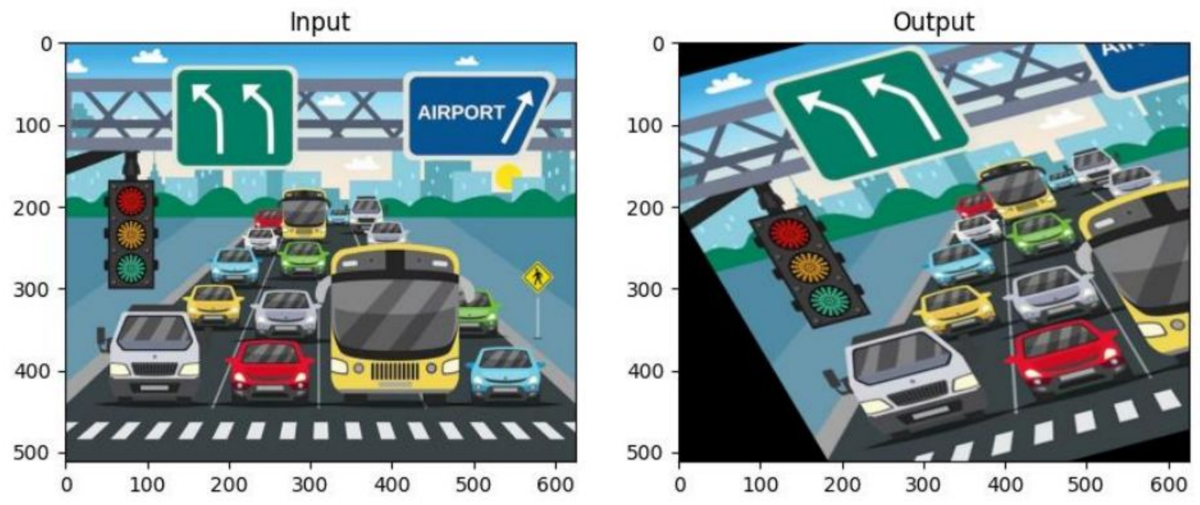
PRATHYUSHA ENGINEERING COLLEGE

(d) Affine transform calculated from three pairs of corresponding points

```
# Importing OpenCV
import cv2
# Importing numpy
import numpy as np
# Importing matplotlib.pyplot
import matplotlib.pyplot as plt
# Reading the image
img = cv2.imread(r"E:/img.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, ch = img.shape

# Coordinates of triangular vertices in the source image
pt1 = np.float32([[50, 50],
                  [200, 50],
                  [50, 200]])
# Coordinates of the corresponding triangular vertices in the output image
pt2 = np.float32([[10, 100],
                  [200, 50],
                  [100, 250]])
# Creating a transformation matrix
Mat = cv2.getAffineTransform(pt1, pt2)
dst = cv2.warpAffine(img, Mat, (cols, rows))
plt.figure(figsize=(10,10))
# Plotting the input image
plt.subplot(121)
plt.imshow(img)
plt.title('Input')
# Plotting the output image
plt.subplot(122)
plt.imshow(dst)
plt.title('Output')

plt.show()
```

output:

PRATHYUSHA ENGINEERING COLLEGE



Result:

Thus the programs for the geometric transforms: (a) Rotation (b) Change of scale (c) Skewing (d) Affine transform calculated from three pairs of corresponding points is executed successfully.

PRATHYUSHA ENGINEERING COLLEGE

Exp. No.4

Object Detection and Recognition

Aim:

To develop a program to implement Object Detection and Recognition

Algorithm:

1. Import necessary libraries:
    - **cv2** for OpenCV functions.
    - **google.colab.patches** for displaying images in a Colab notebook.
2. Load and resize an input image:
    - Read an image from a file named 'image.jpg'.
    - Resize the image to a size of 640x480 pixels.
3. Define the paths to the model and class label files:
    - **weights** contains the path to the frozen inference graph file (the pre-trained model).
    - **model** contains the path to the model configuration file.
    - **coco_names.txt** contains the class labels for the COCO dataset.

4. Load the MobileNet SSD model:
    - Use **cv2.dnn.readNetFromTensorflow** to load the model using the provided weights and model files.
5. Load class labels:
    - Read class labels from the 'coco_names.txt' file and store them in the **class_names** list.
6. Create a blob from the input image:
    - Prepare the image for inference using **cv2.dnn.blobFromImage**.
7. Pass the blob through the network:
    - Set the blob as input to the network.
    - Use **net.forward()** to obtain the output predictions.
8. Process the detection results:
    - Loop over the detected objects in the output.
    - For each detection, check the confidence score (probability).
    - If the confidence is below 50%, continue to the next detection.
9. Draw bounding boxes and labels:
    - Extract the (x, y) coordinates of the bounding box.
    - Draw a green rectangle around the detected object.
    - Extract the class ID to identify the object's name.
    - Draw the object's name and the probability as text above the bounding box.
10. Display the resulting image:
    - Use **cv2_imshow** to display the image with bounding boxes and labels.

- **cv2.waitKey()** waits for a key press

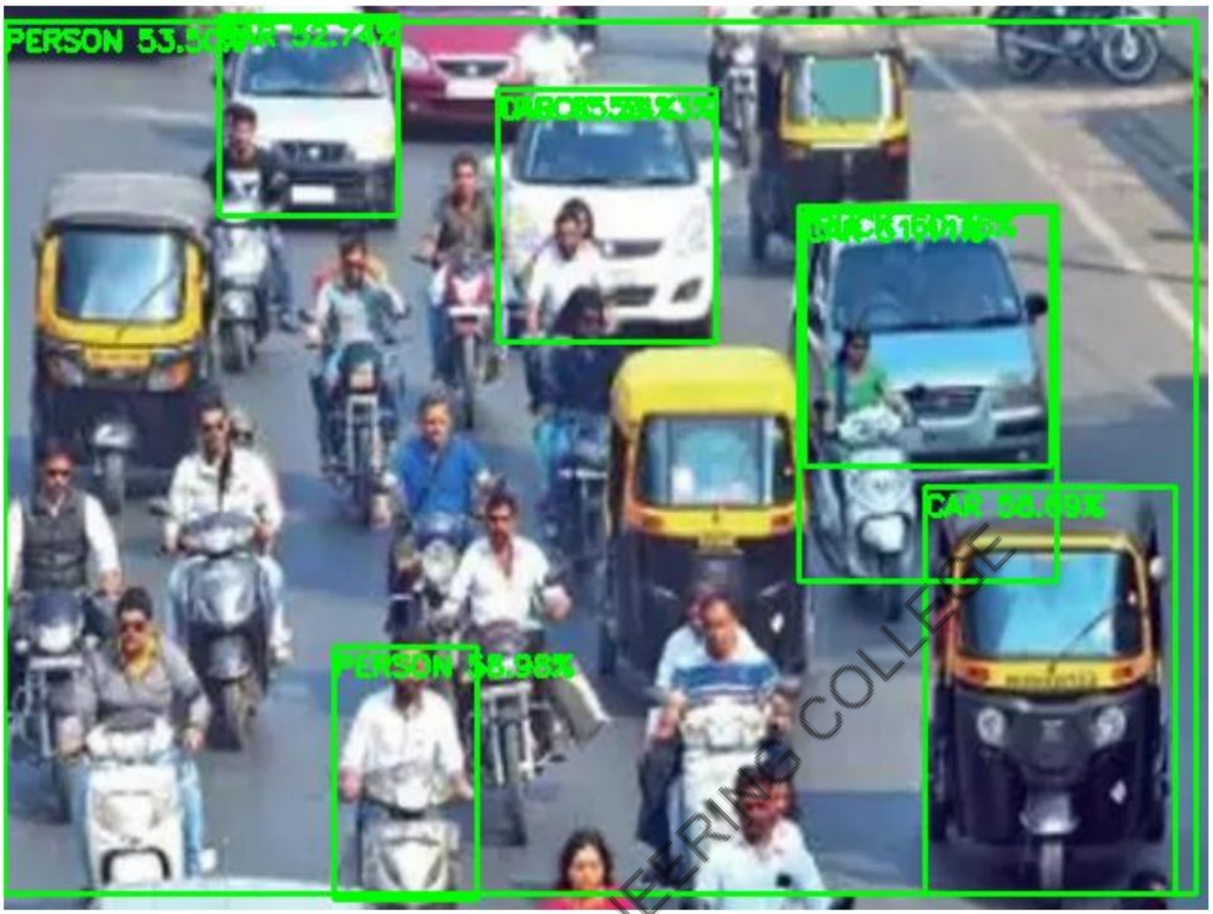
PRATHYUSHA ENGINEERING COLLEGE

Program:

```
from google.colab.patches import cv2_imshow
import cv2

image = cv2.imread('image.jpg')
image = cv2.resize(image, (640, 480))
h = image.shape[0]
w = image.shape[1]

# path to the weights and model files
weights = "frozen_inference_graph.pb"
model = "ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt"
# load the MobileNet SSD model trained  on the COCO dataset
net = cv2.dnn.readNetFromTensorflow(weights, model)

# load the class labels the model was trained on
class_names = []
with open("coco_names.txt", "r") as f:
    class_names = f.read().strip().split("\n")

# create a blob from the image
blob = cv2.dnn.blobFromImage(
    image, 1.0/127.5, (320, 320), [127.5, 127.5, 127.5])
# pass the blog through our network and get the output predictions
net.setInput(blob)
output = net.forward()  # shape: (1, 1, 100, 7)

# loop over the number of detected objects
for detection in output[0, 0, :, :]:  # output[0, 0, :, :] has a shape of: (100, 7)
    # the confidence of the model regarding the detected object
    probability = detection[2]

    # if the confidence of the model is lower than 50%,
    # we do nothing (continue looping)
    if probability < 0.5:
        continue

    # perform element-wise multiplication to get
    # the (x, y) coordinates of the bounding box
    box = [int(a * b) for a, b in zip(detection[3:7], [w, h, w, h])]
    box = tuple(box)
    # draw the bounding box of the object
    cv2.rectangle(image, box[:2], box[2:], (0, 255, 0), thickness=2)

    # extract the ID of the detected object to get its name
    class_id = int(detection[1])
    # draw the name of the predicted object along with the probability
    label = f"{class_names[class_id - 1].upper()} {probability * 100:.2f}%"
    cv2.putText(image, label, (box[0], box[1] + 15),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

cv2_imshow(image)
cv2.waitKey()
```

Output:

PRATHYUSHA ENGINEERING COLLEGE



Result:

Thus the program to implement Object Detection and Recognition is executed successfully and output is verified.

PRATHYUSHA ENGINEERING COLLEGE

Exp. No. 5

Aim:

To develop a program for motion analysis using moving edges.

Algorithm:

1. Import necessary libraries:
    - **cv2** for OpenCV functions.
    - **numpy** for numerical operations.
    - **google.colab.patches** for displaying images in a Colab notebook.
2. Open a video file for reading:
    - **cv2.VideoCapture** is used to open a video file named "yolodetection.mp4" for reading.
3. Get video properties:

- Retrieve the frame width and frame height of the video.

4. Define the codec for the output video:
    - **cv2.VideoWriter_fourcc** is used to specify the codec for the output video. In this case, it's set to 'XVID'.
5. Create a VideoWriter object for the output video:
    - A VideoWriter object is created to write the processed video to "output.mp4" with a frame rate of 5.0 frames per second and a frame size of 1280x720.
6. Read the first two frames of the video:
    - **cap.read()** is used to read the first two frames of the video.
7. Start processing the video in a loop:
    - The code enters a loop that processes each frame of the video.
8. Calculate the difference between consecutive frames:
    - Calculate the absolute difference between **frame1** and **frame2** to identify areas of motion.
9. Convert the difference frame to grayscale:
    - Convert the difference frame to grayscale using **cv2.cvtColor**.
10. Apply Gaussian blur:
    - Apply Gaussian blur to the grayscale frame to reduce noise.
11. Apply thresholding:
    - Apply a threshold to the blurred frame to create a binary image where motion areas are white.
12. Dilate the thresholded image:
    - Dilate the thresholded image to make the white areas more prominent.
13. Find contours of motion:
    - Find contours in the dilated image.
14. Iterate through the detected contours:
    - For each contour, check its area. If the area is less than 900, it's likely not significant motion and is skipped.
    - If the area is significant, draw a green bounding box around the moving object and add a "Movement" status text.
15. Resize the frame:
    - Resize the frame to a fixed size of 1280x720.
16. Write the frame to the output video:
    - Write the processed frame to the output video using **out.write()**.
17. Display the frame with bounding boxes:
    - Display the frame with bounding boxes using **cv2_imshow**.
18. Update the frames for the next iteration:
    - Set **frame1** to the previous **frame2**.
    - Read the next frame into **frame2**.
19. Check for the 'Esc' key (27) to exit the loop:
    - Check if the 'Esc' key is pressed to exit the loop.
20. Release resources:
    - Release OpenCV windows and the video capture and writer objects

PRATHYUSHA ENGINEERING COLLEGE

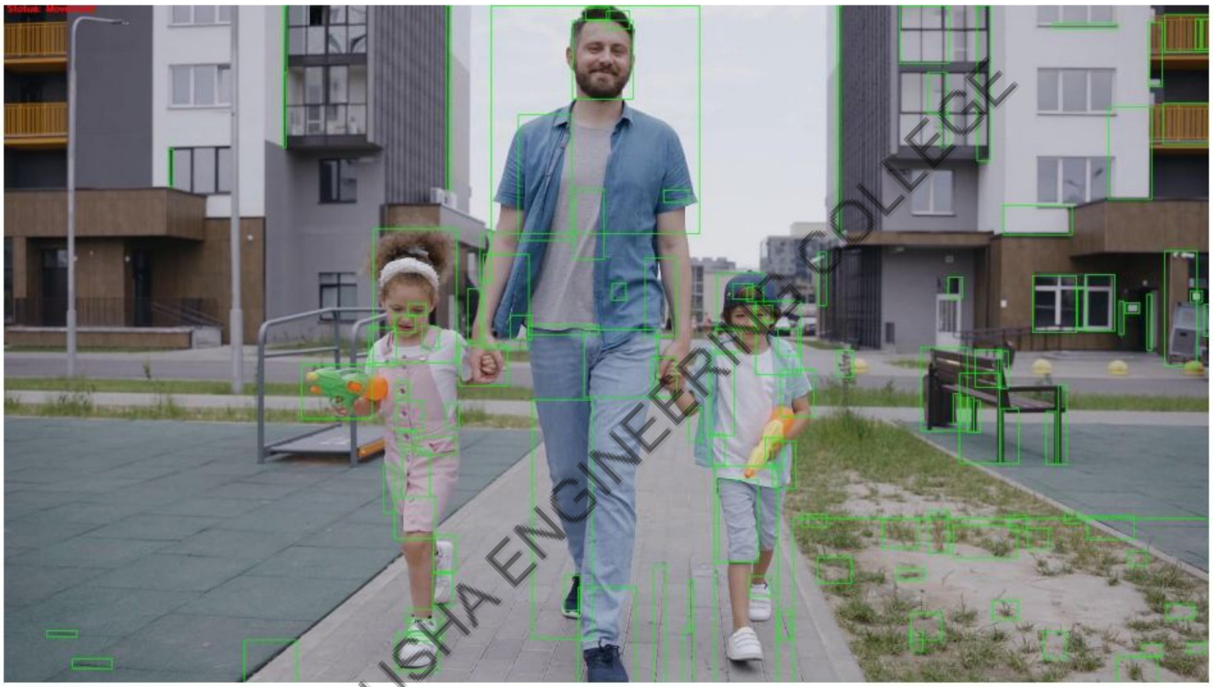
Program:

```
from google.colab.patches import cv2_imshow


import cv2
import numpy as np

cap = cv2.VideoCapture('/content/yolodetection.mp4')
frame_width = int( cap.get(cv2.CAP_PROP_FRAME_WIDTH))

frame_height =int( cap.get( cv2.CAP_PROP_FRAME_HEIGHT))

fourcc = cv2.VideoWriter_fourcc('X','V','I','D')

out = cv2.VideoWriter("output.mp4", fourcc, 5.0, (1280,720))

ret, frame1 = cap.read()
ret, frame2 = cap.read()
print(frame1.shape)
while cap.isOpened():
  diff = cv2.absdiff(frame1, frame2)
  gray = cv2.cvtColor(diff, cv2.COLOR_BGR2GRAY)
  blur = cv2.GaussianBlur(gray, (5,5), 0)
  _, thresh = cv2.threshold(blur, 20, 255, cv2.THRESH_BINARY)
  dilated = cv2.dilate(thresh, None, iterations=3)
          contours,    _    =    cv2.findContours(dilated,    cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

  for contour in contours:
    (x, y, w, h) = cv2.boundingRect(contour)

    if cv2.contourArea(contour) < 900:
      continue
    cv2.rectangle(frame1, (x, y), (x+w, y+h), (0, 255, 0), 2)
            cv2.putText(frame1,    "Status:    {}".format('Movement'),    (10,    20),
cv2.FONT_HERSHEY_SIMPLEX,
          1, (0, 0, 255), 3)
  #cv2.drawContours(frame1, contours, -1, (0, 255, 0), 2)

  image = cv2.resize(frame1, (1280,720))
  out.write(image)
  cv2_imshow(frame1)
  frame1 = frame2
```

PRATHYUSHA ENGINEERING COLLEGE

```
    ret, frame2 = cap.read()

    if cv2.waitKey(40) == 27:
        break

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

Output:



Result:
Thus the program for motion analysis using moving edges is executes successfully and output is verified.

PRATHYUSHA ENGINEERING COLLEGE

Exp. No.6

FACIAL DETECTION AND RECOGNITION

Aim:

To develop a program for Facial Detection and Recognition

Algorithm:

1. Install the required libraries:

- The code begins by installing the **face_recognition** library and the **dlib** library. These libraries are used for face recognition and deep learning-based image processing.

2. Import necessary libraries:
    - **face_recognition** for facial recognition functionality.
    - **cv2** for OpenCV functions.
    - **numpy** for numerical operations.
    - **os** for file and directory operations.
3. Define the path to the directory containing known face images:
    - The path variable points to a directory named "train" which contains known face images.
4. Initialize lists for known names and their encodings:
    - Two lists, **known_names** and **known_name_encodings**, are created to store the names of known individuals and their face encodings.
5. Load known face images and compute face encodings:
    - Loop through the images in the specified directory.
    - Load each image using **fr.load_image_file**.
    - Compute the face encoding for each image using **fr.face_encodings**.
    - Add the name and encoding to the respective lists.
6. Load and process the test image:
    - Load the test image using **cv2.imread**.
    - Use **fr.face_locations** and **fr.face_encodings** to locate and encode the faces in the test image.
7. Compare face encodings to known faces:
    - For each detected face in the test image, compare its encoding to the known face encodings using **fr.compare_faces**.
    - Find the best match using **np.argmin** on the computed face distances.
8. Label and draw bounding boxes around recognized faces:
    - If a match is found, label the recognized face with the corresponding name.
    - Draw a bounding box around the recognized face and display the name.
9. Display the processed image with recognized faces:
    - Display the image with bounding boxes and recognized names using **cv2_imshow**.
10. Save the output image:

- Save the processed image with recognized faces to the specified output path using **cv2.imwrite**.

11. Wait for a key press and close OpenCV windows:

- Wait for a key press (0) to keep the window open.
- Release OpenCV resources and close the window using **cv2.waitKey** and **cv2.destroyAllWindows**.

PRATHYUSHA ENGINEERING COLLEGE

Program:

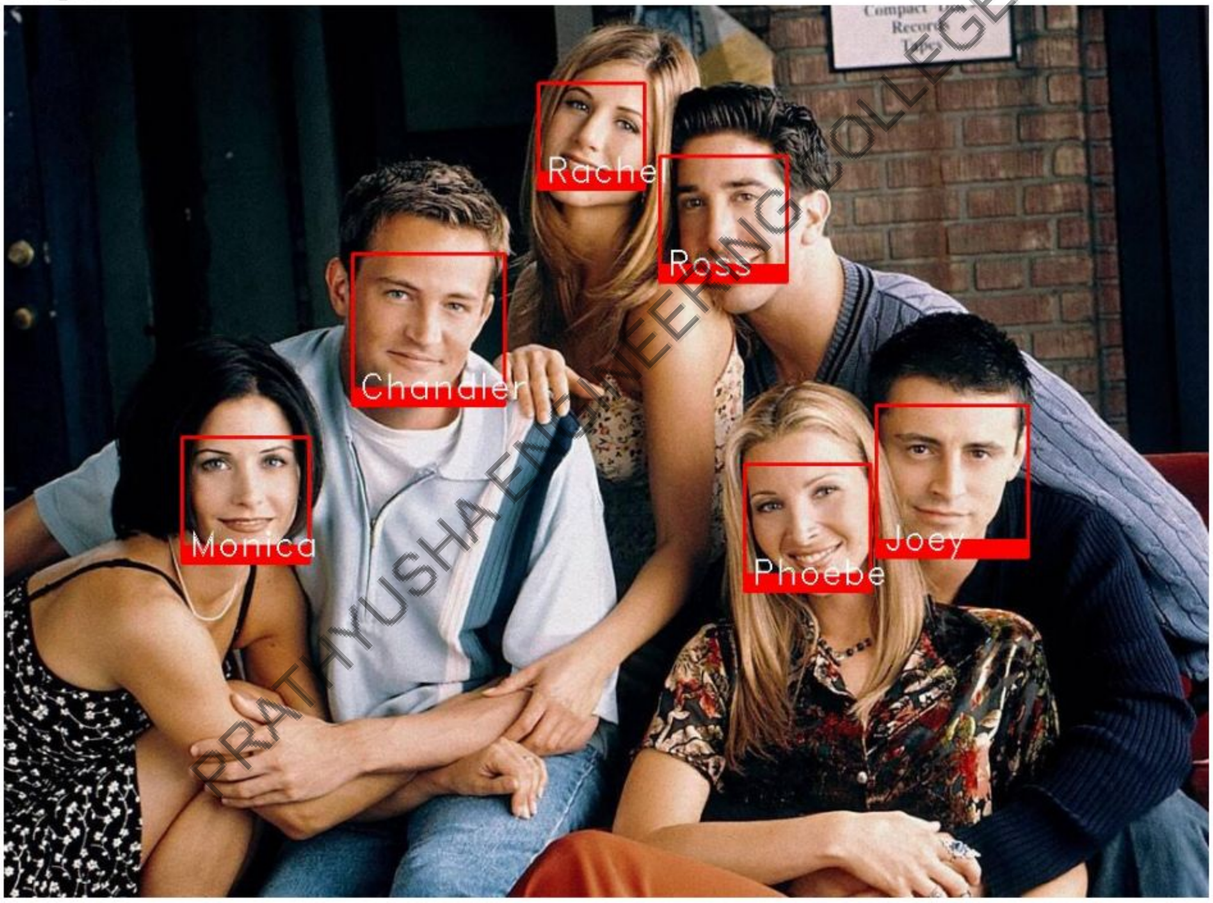
```
!pip install face_recognition
from google.colab.patches import cv2_imshow

!pip install dlib
```

```
import face_recognition as fr
import cv2
import numpy as np
import os

path = "/content/drive/MyDrive/facer/train/"

known_names = []
known_name_encodings = []

images = os.listdir(path)
for _ in images:
    image = fr.load_image_file(path + _)
    image_path = path + _
    encoding = fr.face_encodings(image)[0]

    known_name_encodings.append(encoding)
    known_names.append(os.path.splitext(os.path.basename(image_path))[0].capitalize())

print(known_names)

test_image = "/content/drive/MyDrive/facer/test/test.jpg"
image = cv2.imread(test_image)
# image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

face_locations = fr.face_locations(image)
face_encodings = fr.face_encodings(image, face_locations)

for (top, right, bottom, left), face_encoding in zip(face_locations, face_encodings):
    matches = fr.compare_faces(known_name_encodings, face_encoding)
    name = ""

    face_distances = fr.face_distance(known_name_encodings, face_encoding)
    best_match = np.argmin(face_distances)

    if matches[best_match]:
```

PRATHYUSHA ENGINEERING COLLEGE

```
        name = known_names[best_match]

    cv2.rectangle(image, (left, top), (right, bottom), (0, 0, 255), 2)
    cv2.rectangle(image, (left, bottom - 15), (right, bottom), (0, 0, 255), cv2.FILLED)
    font = cv2.FONT_HERSHEY_DUPLEX
    cv2.putText(image, name, (left + 6, bottom - 6), font, 1.0, (255, 255, 255), 1)

cv2_imshow(image)
cv2.imwrite("/content/drive/MyDrive/facer/output.jpg", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



PRATHYUSHA ENGINEERING COLLEGE

Result:

Thus the program for Facial Detection and Recognition is executed successfully and output is verified.

Exp. no:7

HAND GESTURE RECOGNITION

Aim:

To develop a program to recognize hand gesture.

Algorithm:

1. Import necessary libraries:
    - **cv2** for OpenCV functions.
    - **mediapipe** for hand tracking and landmarks detection.
2. Open a video capture source:
    - Capture video from the default camera (webcam) using **cv2.VideoCapture(0)**.
3. Initialize MediaPipe Hand tracking:
    - Create instances of **mpHands.Hands()** for hand tracking and **mpDraw** for drawing landmarks.
4. Define finger and thumb coordinates:
    - **fingerCoordinates** is a list of tuples that define the landmarks for the fingertips. Each tuple contains two landmark indices: the tip and the base of each finger.
    - **thumbCoordinate** is a tuple that defines the landmarks for the thumb tip and base.
5. Start an infinite loop for video processing:
    - Continuously capture frames from the camera.
6. Read and process the captured frame:
    - Read a frame from the camera using **cap.read()**.
    - Convert the frame from BGR to RGB color space, as MediaPipe requires RGB input.
7. Process hand landmarks:
    - Use **hands.process(imgRGB)** to process the RGB image and detect hand landmarks.
    - Extract the landmarks from the results if hands are detected.
8. Draw hand landmarks and connections:
    - If hands are detected, draw the hand landmarks and connections on the frame using **mpDraw.draw_landmarks**.
9. Extract and visualize hand points:
    - Extract the (x, y) coordinates of hand landmarks and store them in **handPoints**.
    - Draw circles at the detected hand points on the frame.
10. Count the number of raised fingers:
    - Check the relative positions of specific finger tip and base landmarks to determine if a finger is raised. Increment **upCount** for each raised finger.

PRATHYUSHA ENGINEERING COLLEGE

- Additionally, check the thumb position to see if it is raised.

11. Display the finger count:
    - Draw the finger count on the frame using **cv2.putText**.
12. Display the processed frame:
    - Show the processed frame with hand landmarks and finger count using **cv2.imshow**.
13. Wait for a key press and update the frame:
    - Wait for 1 millisecond using **cv2.waitKey(1)** to allow the frame to be displayed and updated in the loop.
14. Close the OpenCV window:
    - The loop continues until you press a key to exit the program. When a key is pressed, the program closes the OpenCV window.

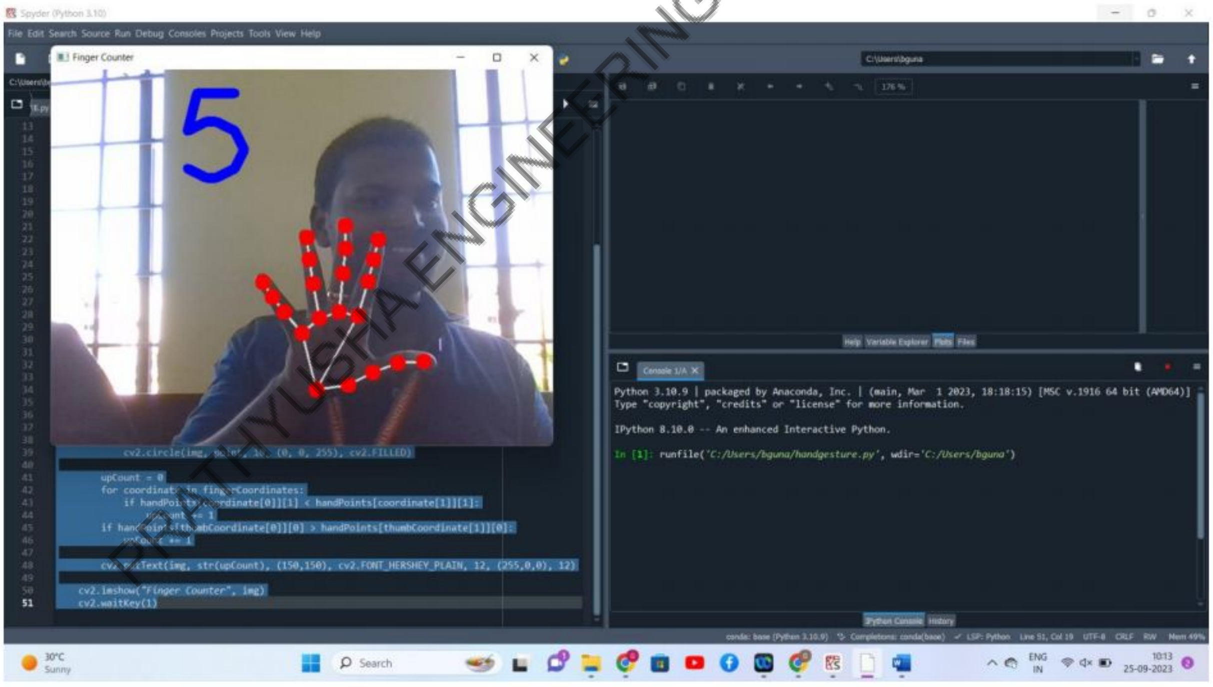
PRATHYUSHA ENGINEERING COLLEGE

Program:

```
import cv2
import mediapipe as mp

cap = cv2.VideoCapture(0)
mpHands = mp.solutions.hands
hands = mpHands.Hands()
mpDraw = mp.solutions.drawing_utils
fingerCoordinates = [(8, 6), (12, 10), (16, 14), (20, 18)]
thumbCoordinate = (4,2)
while True:
  success, img = cap.read()
  imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  results = hands.process(imgRGB)
  multiLandMarks = results.multi_hand_landmarks

  if multiLandMarks:
    handPoints = []
    for handLms in multiLandMarks:
      mpDraw.draw_landmarks(img, handLms, mpHands.HAND_CONNECTIONS)

      for idx, lm in enumerate(handLms.landmark):
        # print(idx,lm)
        h, w, c = img.shape
        cx, cy = int(lm.x * w), int(lm.y * h)
        handPoints.append((cx, cy))

    for point in handPoints:
            cv2.circle(img, point, 10, (0, 0, 255), cv2.FILLED)

        upCount = 0
        for coordinate in fingerCoordinates:
            if handPoints[coordinate[0]][1] < handPoints[coordinate[1]][1]:
                upCount += 1
        if handPoints[thumbCoordinate[0]][0] > handPoints[thumbCoordinate[1]][0]:
            upCount += 1

        cv2.putText(img, str(upCount), (150,150), cv2.FONT_HERSHEY_PLAIN, 12, (255,0,0), 12)

    cv2.imshow("Finger Counter", img)
    cv2.waitKey(1)
```

PRATHYUSHA ENGINEERING COLLEGE

Output:


Spyder (Python 3.10)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Finger Counter
5
Help Variable Explorer Plots Files
Console 1/A
Python Console History
30°C Sunny
Search
25-09-2023

```
            cv2.circle(img, point, 10, (0, 0, 255), cv2.FILLED)

        upCount = 0
        for coordinate in fingerCoordinates:
            if handPoints[coordinate[0]][1] < handPoints[coordinate[1]][1]:
                upCount += 1
        if handPoints[thumbCoordinate[0]][0] > handPoints[thumbCoordinate[1]][0]:
            upCount += 1

        cv2.putText(img, str(upCount), (150,150), cv2.FONT_HERSHEY_PLAIN, 12, (255,0,0), 12)

    cv2.imshow("Finger Counter", img)
    cv2.waitKey(1)
```

```
Python 3.10.9 | packaged by Anaconda, Inc. | (main, Mar  1 2023, 18:18:15) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.10.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/bguna/handgesture.py', wdir='C:/Users/bguna')
```

Result:

Thus the program to recognize hand gesture is executed successfully and output is verified.

ADDITIONAL EXPERIMENTS

Exp. No.:8

Aim:

To develop a program for detecting an edges of an image.

Algorithm:

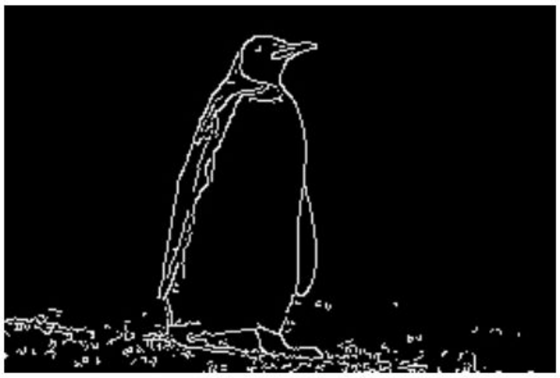
1. Import the OpenCV library:
   - The code starts by importing the OpenCV library.
2. Load an image:
   - It loads an image named "penguin.jpg" using **cv2.imread** and assigns it to the **image** variable.
3. Apply Canny edge detection:
   - The Canny edge detection algorithm is applied to the loaded image using the **cv2.Canny** function. The parameters **200** and **300** are used as the low and high thresholds, respectively, for edge detection.
4. Save the resulting image:
   - The edge-detected image is saved with the name 'edges_Penguins.jpg' using **cv2.imwrite**.
5. Display the edge-detected image:
   - The code uses **cv2.imshow** to display the edge-detected image.

Program:

```
import cv2
image = cv2.imread("penguin.jpg")
cv2.imwrite('edges_Penguins.jpg',cv2.Canny(image,200,300))

cv2.imshow('edges', cv2.imread('edges_Penguins.jpg'))
```

Output:

PRATHYUSHA ENGINEERING COLLEGE



Result:

Thus the program for detecting an edges of an image is executed successfully and output is verified.

Exp. No.9

SMOOTHING AND BLURRING

Aim:
To develop a program to apply smoothing and blurring to an image.

Algorithm:

1. Import OpenCV and NumPy:
   - The code starts by importing the OpenCV library as **cv2** and the NumPy library as **np**.
2. Read an image:
   - It reads an image from the file path "E:\Backup 14.4.23\image\lab\pen.jpg" using **cv2.imread** and stores it in the **image** variable.
3. Create a kernel for averaging (blur):
   - The code defines a 5x5 kernel of ones (all values set to 1) using NumPy.
   - The division by 25 is to normalize the kernel so that the sum of the values is 1, making it an average filter.
4. Apply the filter to the image:
   - The **cv2.filter2D** function is used to apply the filter to the input image. It takes the source image (**image**), the depth (**ddepth**), and the kernel (**kernel2**) as

PRATHYUSHA ENGINEERING COLLEGE

parameters. The **ddepth** of -1 indicates that the output image should have the same depth as the input image.

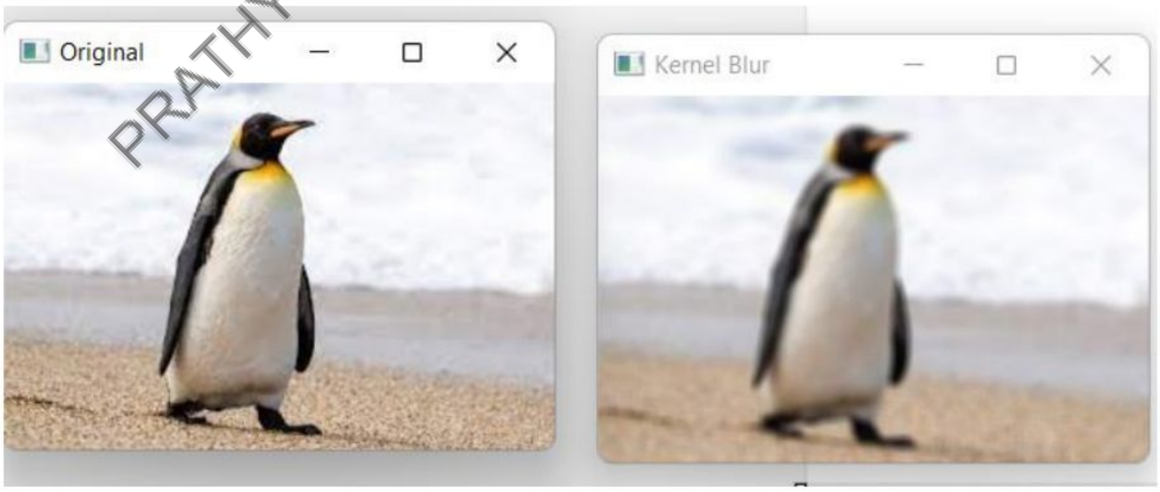
5. Display the original and filtered images:
   - The code displays both the original image and the filtered (blurred) image using **cv2.imshow**.
6. Wait for a key press and close the windows:
   - The code waits for a key press with **cv2.waitKey()**.
   - It then closes all OpenCV windows using **cv2.destroyAllWindows()**.

Program:

```
import cv2
import numpy as np
 # Reading the image
image = cv2.imread("E:\Backup 14.4.23\image\lab\pen.jpg")
 # Creating the kernel with numpy
kernel2 = np.ones((5, 5), np.float32)/25
 # Applying the filter
img = cv2.filter2D(src=image, ddepth=-1, kernel=kernel2)
 # showing the image
cv2.imshow('Original', image)
cv2.imshow('Kernel Blur', img)
 cv2.waitKey()
cv2.destroyAllWindows()
```

PRATHYUSHA ENGINEERING COLLEGE

Output:



Result:
Thus the program to apply smoothing and blurring to an image is executed successfully and output is verified.