Practical 1

Aim: **Perform Geometric Transformations**

Theory:

Often, during image processing, the geometry of the image (such as the width and height of the image) needs to be transformed. This process is called geometric transformation, images are nothing but matrices.

Since images are matrices, if we apply an operation to the images (matrices) and we end up with another matrix, we call this a transformation. This basic idea will be used extensively to understand and apply various kinds of geometric transformations.

Here are the geometric transformations that we are performing in this practical:

Translation: Translation basically means shifting the object's location. It means shifting the object horizontally or vertically by some defined off-set (measured in pixels).
x' = x + A (Eq. 1)
y' = y + B (Eq. 2)
Here, let's say A = 100, B = 80 (Translation in x and y-direction respectively)
(x, y) – point in input image
(x', y') – point in output image
It means that each pixel is shifted 100 pixels in the x-direction and each pixel is shifted 80 pixels in the y-direction.

Rotation: As evident by its name, this technique rotates an image by a specified angle and by the given axis or point. It performs a geometric transform which maps the position of a point in current image to the output image by rotating it by the user-defined angle through the specified axis.
The points that lie outside the boundary of an output image are ignored.

Image scaling or resizing: Scaling means resizing an image which means an image is made bigger or smaller in x- or/and y-direction.

Affine transformation: An affine transformation is a transformation that preserves collinearity and the ratio of distances (for example – the midpoint of a line segment is still the midpoint even after the transformation))

The parallel lines in an original image will be parallel in the output image.

In general, an affine transformation is a composition of translations, rotations, shears, and magnifications.

Perspective transformation: Perspective transformation, also known as homography, is a projective transformation that maps points in one plane to another while preserving straight lines. It can account for changes in perspective and orientation, making it suitable for applications like image stitching, rectifying images, and augmented reality. In computer vision, perspective transformation is often represented as a 3×3 matrix

called the homography matrix. When applied to an image, this matrix can correct perspective distortions, align images from different viewpoints, and create panoramas.

CODE

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load on image
image = cv2.imread('left.jpg')
# Get image dimensions
height, width = image.shape[:2]
# Display the original image
plt.figure(figsize=(6,6))
plt.subplot(2,3,1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.show()
# Define the translation matrix
translation_matrix = np.float32([[1,0,300],[0,1,100]])
# Apply translation transformation
translated_image = cv2.warpAffine(image, translation_matrix,
(width,height))
# Display the translated image
plt.figure(figsize=(6,6))
plt.subplot(2,3,2)
plt.imshow(cv2.cvtColor(translated_image, cv2.COLOR_BGR2RGB))
plt.title('Translated Image')
plt.axis('off')
plt.show()
# Define the rotation matrix
rotation_matrix = cv2.getRotationMatrix2D((width/2, height/2), 60, 1)
# Apply rotation transformation
rotated_image = cv2.warpAffine(image, rotation_matrix, (width,height))
# Display the rotated image
plt.figure(figsize=(6,6))
plt.subplot(2,3,3)
plt.imshow(cv2.cvtColor(rotated_image, cv2.COLOR_BGR2RGB))
plt.title('Rotated Image')
```

```python
plt.axis('off')
plt.show()
# Define the scaling matrix
scaling_matrix = np.float32([[0.8,0,0],[0,0.8,0]])
# Apply Scaling transformation
scaled_image = cv2.warpAffine(image, scaling_matrix, (int(width*0.5),
int(height*0.5)))
# Display the scaled image
plt.figure(figsize=(6,6))
plt.subplot(2,3,4)
plt.imshow(cv2.cvtColor(scaled_image, cv2.COLOR_BGR2RGB))
plt.title('Scaled Image')
plt.axis('off')
plt.show()
# Define affine transformation points
original_points = np.float32([[50,50], [200,50], [50,200]])
transformed_points = np.float32([[10,100], [200,50], [100,250]])
# Compute the affine transformation matrix
affine_matrix = cv2.getAffineTransform(original_points, transformed_points)
# Apply affine transformation
affine_transformed_image = cv2.warpAffine(image, affine_matrix,
(width,height))
# Display the affine transformed image
plt.figure(figsize=(6,6))
plt.subplot(2,3,5)
plt.imshow(cv2.cvtColor(affine_transformed_image, cv2.COLOR_BGR2RGB))
plt.title('Affine Transformed Image')
plt.axis('off')
plt.show()
# Define perspective transformation points
original_points = np.float32([[56,65], [368,52], [28,387], [389,390]])
transformed_points = np.float32([[0,0], [300,0], [0,300], [300,300]])
# Compute the perspective transformation matrix
perspective_matrix = cv2.getPerspectiveTransform(original_points,
transformed_points)
# Apply perspective transformation
perspective_transformed_image = cv2.warpPerspective(image,
perspective_matrix,
(width,height))
# Display the perspective transformed image
```

```
plt.figure(figsize=(6,6))
plt.subplot(2,3,6)
plt.imshow(cv2.cvtColor(perspective_transformed_image, cv2.COLOR_BGR2RGB))
plt.title('Perspective Transformed Image')
plt.axis('off')
plt.show()
```

OUTPUT

Original Image

Scaled Image

Translated Image

Affine Transformed Image

Rotated Image

Perspective Transformed Image

Practical 2

<u>Aim</u>: Perform Image Stitching

<u>Theory:</u>

Image stitching is the process of combining multiple images to create a single larger image, often referred to as a panorama. The images depict the same three-dimensional scene, and they must overlap in some regions. Image stitching aims to create a seamless transition between adjacent images while preserving the geometry and visual quality.

Image stitching is a multi-step technique that involves the following image-processing operations:

- **Feature detection:** Feature detection is the task of identifying distinctive and salient points in an image. These points, generally called keypoints, serve as landmarks for aligning the acquired images.

- **Feature matching:** Once features are detected, the next step is to find corresponding features between the images. The goal is to identify points in one image that correspond to the same real-world location in another image. For each detected feature, a descriptor is computed. The descriptor is a compact numerical representation of the local image information around the feature point. It captures the key characteristics of the feature and is used for matching.

- **Transformation estimation:** Once the pairs of matching features are identified, the transformations that align each image with a reference image are estimated. Such transformation is called homography.

- **Image warping:** This step involves applying the transformations found in the previous step to all the images, except the reference image. In this way, the images are aligned in the same reference system. The process of image warping involves applying a transformation to each pixel's coordinates in the original image and determining its location in the transformed image

- **Blending:** Uneven lighting conditions and exposure differences between the acquired images lead to visible seams in the final panorama. Image blending techniques allow us to mitigate the seam problem.

CODE

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load images
img_ = cv2.imread('right1.jpg')
img1 = cv2.cvtColor(img_, cv2.COLOR_BGR2GRAY)
img = cv2.imread('left1.jpg')
```

```python
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#Initialize SIFT detector
sift = cv2.SIFT_create()
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
# Create a BFMatcher object with distance measurement cv2.NORM_L2
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck = False)
# Perfomr the matching between the SIFT descriptors of the images
matches = bf.knnMatch(des1,des2,k=2)
# Apply the ratio test to find good matches
good = []
for m,n in matches:
 if m.distance < 0.75*n.distance:
     good.append(m)
# Atleast 4 matches are to be there to find the homography
if len(good)>4:
 # prepare source and destination points
 src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
 dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)
 # Compute Homography
 H, status = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
 # Use homography to warp image
 dst = cv2.warpPerspective(img_,H,
(img.shape[1]+img_.shape[1],img.shape[0]))
 # Convert warped image from BGR to RGB for matplotlib
 dst_rgb = cv2.cvtColor(dst, cv2.COLOR_BGR2RGB)
 # Display the warped image
 plt.subplot(122), plt.imshow(dst_rgb), plt.title('Warped Image')
 plt.show()
 # Place the left image on the appropriate position
 dst[0:img.shape[0], 0:img.shape[1]] = img
 # Convert the combined image from BGR to RGB for matplotlib
 combined_rgb = cv2.cvtColor(dst, cv2.COLOR_BGR2RGB)
    # Save the stitched image as output.jpg in the BGR format
 cv2.imwrite('output.jpg',dst)
 # Display the stitched image
 plt.imshow(combined_rgb)
 plt.title('Stitched Image')
 plt.show()
else:
```

```
 raise    AssertionError('Not    enough    matches    are    found    -
{}/{}'.format(len(good),4))
```
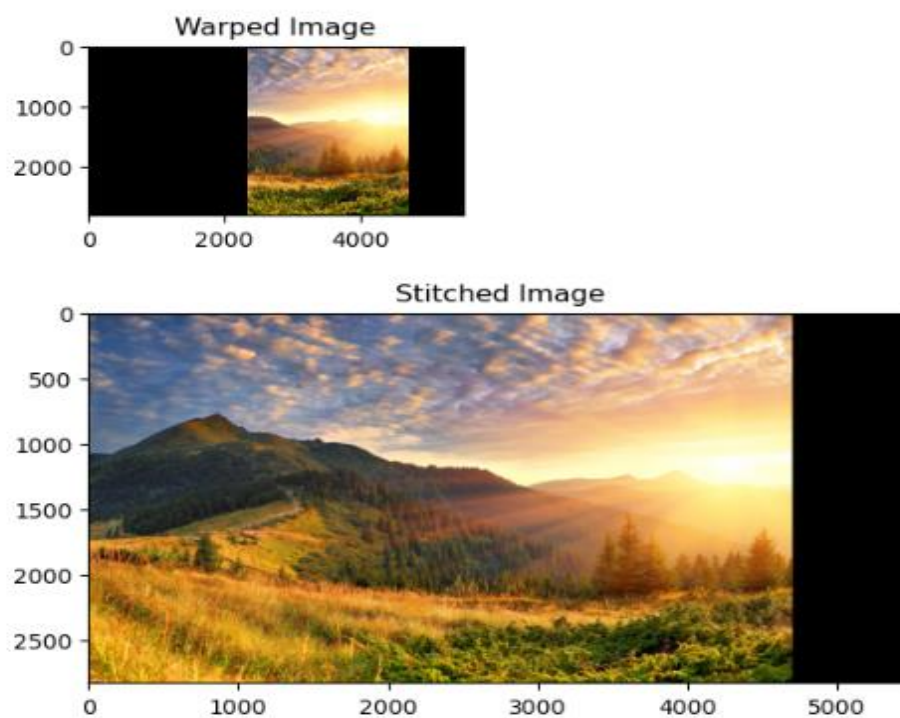
Output

**Left**   **Right**

Wrapped

Practical 3

Aim: **Perform Camera Calibration**

Theory:

Images are captured by cameras, which use lenses to convert 3D objects from the real world into 2D images. However, because lenses are used during the transformation, some distortions are also introduced to the pictures. In order to prevent capturing distorted images, the camera needs to be calibrated, to accurately relate a 3D point in the real world to its matching 2D projection (pixel) in the image. Hence, camera calibration means determining the parameters of the camera to capture an undistorted image which is carried out by the function calibrateCamera() in Opencv.

The object in the real world exists in the World Coordinate System (3D) which when captured by the camera is viewed in Camera Coordinate System (3D). Finally, to project the captured image, the result is viewed in Image Coordinate System (2D).
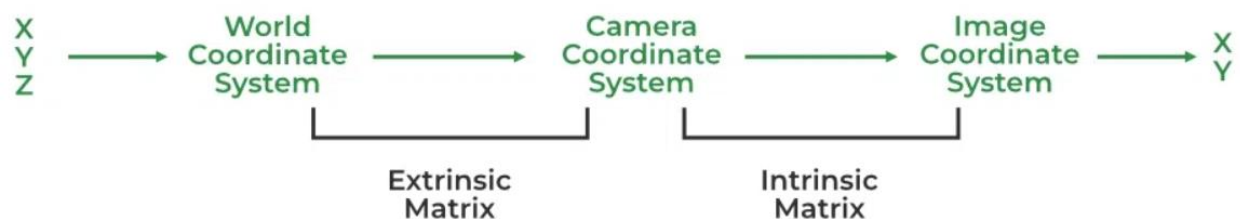
Usually, two types of distortion happen in an image. First radial distortion, causes straight lines to look slightly twisted or bent when a camera photographs them. Second, Tangential distortion, which causes the picture to be extended slightly longer or skewed and causes the objects to appear closer or farther away than they actually are, is most commonly caused by the lens not being parallel to the imaging plane.

The two major types of parameters required for calibration are as follows:
- Extrinsic Parameters: It includes the orientation of the camera in the World Coordinate System by rotation and translation matrix.
- Intrinsic Parameters: It includes the lens system parameters such as focal length, optical center, aperture, field-of-view, resolution, etc

The calibration procedure includes solving the given matrices using basic geometric equation calculations. The equations are chosen depending on the calibration objects. Opencv, to date supports three types of objects for calibration:

- Classical black-white chessboard
- Symmetrical circle pattern
- Asymmetrical circle pattern

CODE

```python
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)


# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.


images = glob.glob('left/*.jpg') #read a series of images


for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #Convert the image to gray


    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6), None)


    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)
```

```python
        corners2=cv2.cornerSubPix(gray, corners, (11,11), (-1,-1),
criteria) #refine the corner locations
        imgpoints.append(corners2)

        # Draw and display the corners
        cv2.drawChessboardCorners(img, (9,6), corners2, ret)
        cv2.imshow('img', img)
        cv2.waitKey(500)
cv2.destroyAllWindows()


#calibration
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1],None,None)


img = cv2.imread('left/left12.jpg')
h, w = img.shape[:2]
newcameramtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1,
(w,h))


# undistort
dst = cv.undistort(img, mtx, dist, None, newcameramtx)


# crop the image
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]


# Plot original and undistorted images

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax[0].set_title('Original Image')
ax[0].axis('off')
ax[1].imshow(cv2.cvtColor(dst, cv2.COLOR_BGR2RGB))
ax[1].set_title('Undistorted Image')
ax[1].axis('off')
plt.show()
```
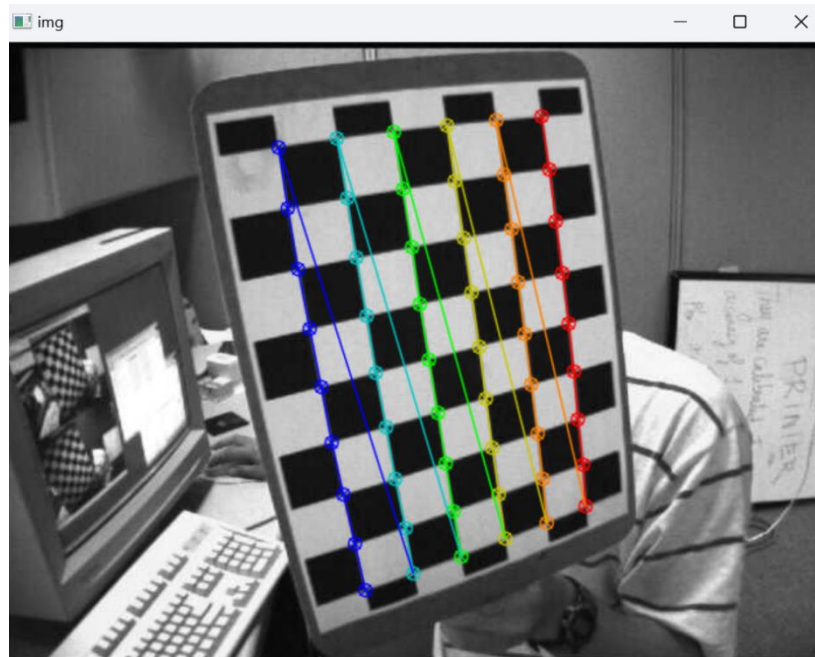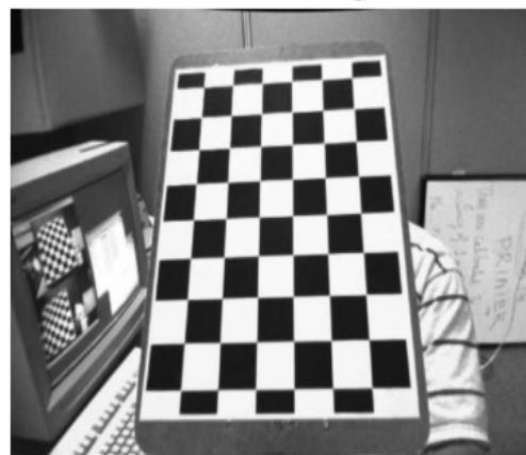
Output:

Original Image

Undistorted Image

Practical 4

Perform the following:
A) Face Detection

<u>Theory:</u> There are some steps that how face detection operates, which are as follows:

Firstly, the image is imported by providing the location of the image. Then the picture is transformed from RGB to Grayscale because it is easy to detect faces in the grayscale. After that, the image manipulation used, in which the resizing, cropping, blurring and sharpening of the images done if needed.

The next step is image segmentation, which is used for contour detection or segments the multiple objects in a single image so that the classifier can quickly detect the objects and faces in the picture.

The next step is to use Haar-Like features algorithm, which is proposed by Voila and Jones for face detection. This algorithm used for finding the location of the human faces in a frame or image. All human faces share some universal properties of the human face like the eyes region is darker than its neighbour pixels and nose region is brighter than eye region.

The haar-like algorithm is also used for feature selection or feature extraction for an object in an image, with the help of edge detection, line detection, centre detection for detecting eyes, nose, mouth, etc. in the picture. It is used to select the essential features in an image and extract these features for face detection.

The next step is to give the coordinates of x, y, w, h which makes a rectangle box in the picture to show the location of the face or we can say that to show the region of interest in the image. After this, it can make a rectangle box in the area of interest where it detects the face.

a) Image Face Detection

<u>CODE</u>

```
!pip install open-cv python
# detect face from input image and save it on the disk.
import cv2
# Load the pre-trained Haar Cascade model for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
# load the image where you want to detect face
image_path = 'face.jpg' # path to your image
image = cv2.imread(image_path)
# convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
# detect faces in the image
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5,
minSize=(30,30))
# draw rectangles around each face
for (x,y,w,h) in faces:
 cv2.rectangle(image,(x,y),(x+w,y+h),(255,0,0),2)
# save the image with faces highlighted
output_path = 'faces_detected.jpg' # corrected file extension
cv2.imwrite(output_path, image)
print(f"Faces Detected: {len(faces)}. Output saved to {output_path}")
```

Output


```
ERROR: Could not find a version that satisfies the requirement open-cv (from versions: none)
ERROR: No matching distribution found for open-cv
Faces Detected: 27. Output saved to faces_detected.jpg
```

b) Live Face Detection

CODE

```python
# detect faces and show it on screen
import cv2
import matplotlib.pyplot as plt
# initialize the Haar Cascade face detection model
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
# start capturing video from the camera (default camera is usually at
index 0)
cap = cv2.VideoCapture(0)
# Capture a single frame
ret, frame = cap.read()
if ret: # check if the frame was successfully captured
 # convert the captured frame to grayscale
 gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
 # detect faces in the grayscale frame
 faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5,
minSize=(30,30))
 # draw rectangles around the detected faces
 for (x,y,w,h) in faces:
     cv2.rectangle(frame, (x,y), (x+w, y+h), (0,255,0), 2)
 # Display the resulting frame with faces highlighted using matplotlib
 plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
 plt.axis('off') # turn off axis labels
 plt.show()
# Release the capture
cap.release()
print('Number of faces detected: ',len(faces))
```
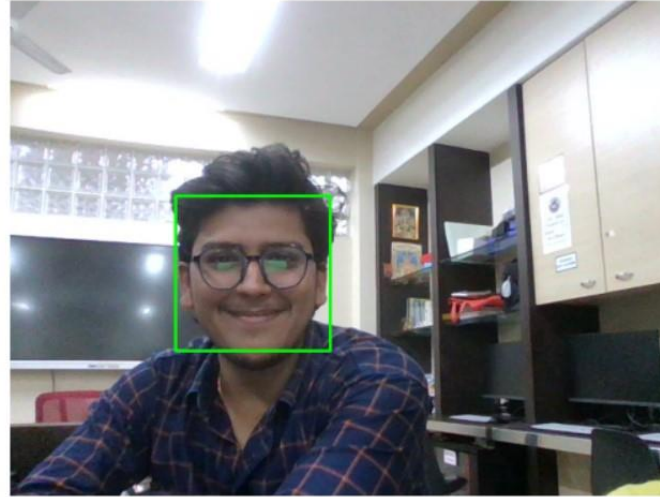
Output


```
Number of faces detected:  1
```

B) Object Detection

Theory: Object detection, within computer vision, involves identifying objects within images or videos. Object detection is merely to recognize the object with bounding box in the image, where in image classification, we can simply categorize(classify) that is an object in the image or not in terms of the likelihood (Probability). It is like object recognition, which involves identifying the object category, but object detection goes a step further and localizes the object within the image. Object detection applications can be performed using two different data analysis techniques:

A. Image Processing

In image processing, the algorithm analyses an image to detect objects based on certain features, such as edges or textures. This approach typically involves applying a series of image processing techniques, such as filtering, thresholding, and segmentation, to extract regions of interest in the image. These regions are then analyzed to determine whether they contain an object or not.

B. Deep Neural Network

In a deep neural network, the algorithm is trained on large, labelled datasets to detect objects. This approach involves using a convolutional neural network (CNN) architecture, which is a type of deep learning algorithm specifically designed for image analysis. The network learns to detect objects by analysing features of the image at different levels of abstraction, using multiple layers of artificial neurons to perform

increasingly complex analyses. The output of the network is a bounding box around the detected object, along with a confidence score that represents how certain the network is that the object is present.

Categories of Object Detection:
• Single-shot detectors: Single-shot detectors combine localization and classification and they perform very quickly.
• Two-stage detectors: Two-stage detectors separate object localization and classification in the head, generally returns higher localization accuracy.

Object detection algorithms and architectures:

R-CNN (region-based convolutional neural network) is a two-stage detector that uses a method called region proposals to generate 2,000 region predictions per image. R-CNN then warps the extracted regions to a uniform size and runs those regions through separate networks for feature extraction and classification.

YOLO (You Only Look Once) is a family of single-stage detection architectures based in Darknet, an open-source CNN framework. YOLO's speed makes it preferable for real-time object detection and has earned it the common descriptor of state-of-the-art object detector. YOLO makes less than 100 bounding box predictions per image, it also produces less background false positives, although it has a higher localization error, generally focuses on speed and accuracy.

CODE

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


args = argparse.Namespace(
    image="C:/dog.jpg",
    weights="yolov3.weights",
    config="yolov3.cfg",
    classes="yolov3.txt"
)
def get_output_layers(net):
    layer_names = net.getLayerNames()
    try:
        output_layers = [layer_names[i-1] for i in
net.getUnconnectedOutLayers()]
    except:
        output_layers = [layer_names[i-1] for i in
net.getUnconnectedOutLayers()]
```

```python
    return output_layers


def draw_prediction(img, class_id, confidence, x, y, x_plus_w, y_plus_h):
    label = str(classes[class_id])
    color = COLORS(class_id)
    cv2.rectangle(img, (x,y),(x_plus_w, y_plus_h), label, color,2)
    cv2.putText(img, label,  (x-10,y-10),cv2.FONT_HERSHEY_COMPLEX, color,
2)
# read input image
image = cv2.imread(args.image)


Width = image.shape[1]
Height = image.shape[0]
scale = 0.00392
classes = None


# read class names from text file
classes = None
with open(args.classes, 'r') as f:
    classes = [line.strip() for line in f.readlines()]


# generate different colors for different classes
COLORS = np.random.uniform(0, 255, size=(len(classes), 3))


# read pre-trained model and config file
net = cv2.dnn.readNet(args.weights, args.config)


# create input blob
blob = cv2.dnn.blobFromImage(image, scale, (416,416), (0,0,0), True,
crop=False)


# set input blob for the network
net.setInput(blob)


# function to get the output layer names
# in the architecture


# function to draw bounding box on the detected object with class name
def draw_bounding_box(img, class_id, confidence, x, y, x_plus_w, y_plus_h):
```

```python
    label = str(classes[class_id])

    color = COLORS[class_id]

    cv2.rectangle(img, (x,y), (x_plus_w,y_plus_h), color, 2)

    cv2.putText(img, label, (x-10,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
color, 2)

# run inference through the network
# and gather predictions from output layers
outs = net.forward(get_output_layers(net))

# initialization
class_ids = []
confidences = []
boxes = []
conf_threshold = 0.5
nms_threshold = 0.4

# for each detetion from each output layer
# get the confidence, class id, bounding box params
# and ignore weak detections (confidence < 0.5)
for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.5:
            center_x = int(detection[0] * Width)
            center_y = int(detection[1] * Height)
            w = int(detection[2] * Width)
            h = int(detection[3] * Height)
            x = center_x - w / 2
            y = center_y - h / 2
            class_ids.append(class_id)
            confidences.append(float(confidence))
            boxes.append([x, y, w, h])
```

```python
# apply non-max suppression
indices = cv2.dnn.NMSBoxes(boxes, confidences, conf_threshold,
nms_threshold)


# go through the detections remaining
# after nms and draw bounding box
for i in indices:
    try:
        box=boxes[i]
    except:
        i=i[0]
        box=boxes[i]

    x = box[0]
    y = box[1]
    w = box[2]
    h = box[3]
    draw_bounding_box(image, class_ids[i], confidences[i], round(x),
round(y), round(x+w), round(y+h))


# display output image
#cv2.imshow("object detection", image)

plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')  # turn off axis labels
plt.show()
# wait until any key is pressed
#cv2.waitKey()

 # save output image to disk
#cv2.imwrite("object-detection.jpg", image)

# release resources
#cv2.destroyAllWindows()
```
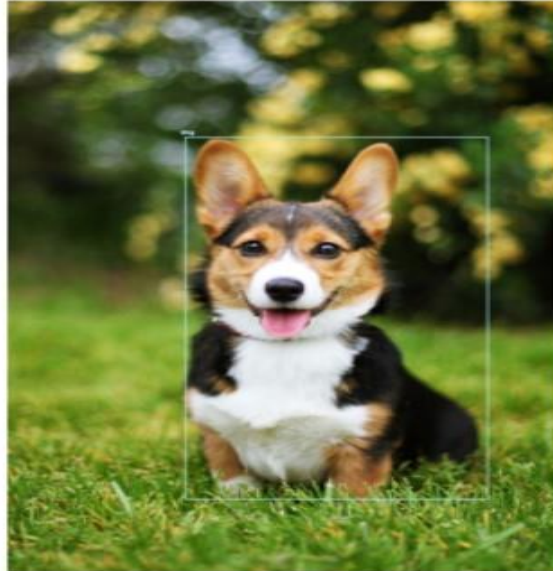
OUTPUT

C) Pedestrian detection

Theory:

Pedestrian detection is a very important area of research because it can enhance the functionality of a pedestrian protection system in Self Driving Cars. We can extract features like head, two arms, two legs, etc, from an image of a human body and pass them to train a machine learning model. After training, the model can be used to detect and track humans in images and video streams. However, OpenCV has a built-in method to detect pedestrians. It has a pre-trained haarcascade fullbody model to detect pedestrians in images and video streams.

Haar Cascade: Haar cascade is an algorithm that can detect objects in images, irrespective of their scale in image and location.

This algorithm is not so complex and can run in real-time. We can train a haar-cascade detector to detect various objects like cars, bikes, buildings, fruits, etc.

Haar cascade uses the cascading window, and it tries to compute features in every window and classify whether it could be an object.

There are some pre-trained haar cascade models like:

- Human face detection
- Eye detection
- Nose / Mouth detection
- Vehicle detection
- Full body detection

In this practical we will be using the full body detection pre-trained model.

Method 1
Code for jupyter

<u>CODE</u>

```
import cv2
import matplotlib.pyplot as plt
pedestrian_cascade    =    cv2.CascadeClassifier(cv2.data.haarcascades    +
'haarcascade_fullbody.xml')
video_path = 'C:/Users/NISHA/Desktop/photos/video.mp4'
cap = cv2.VideoCapture(video_path)
if not cap.isOpened():
    print("Error: Unable to open the video")
    exit()
while True:
    ret, frame = cap.read()
    if not ret:
        break
    gray_frame = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    # Detect pedestrians in the grayscale frame
    pedestrians      =      pedestrian_cascade.detectMultiScale(gray_frame,
scaleFactor=1.1, minNeighbors=5, minSize=(30,30))
    # Draw rectangle around the detected pedestrians
    for (x,y,w,h) in pedestrians:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 2)
    plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
    plt.axis('off')   # Turn off the axis
    plt.show()
cap.release()
```

OUTPUT
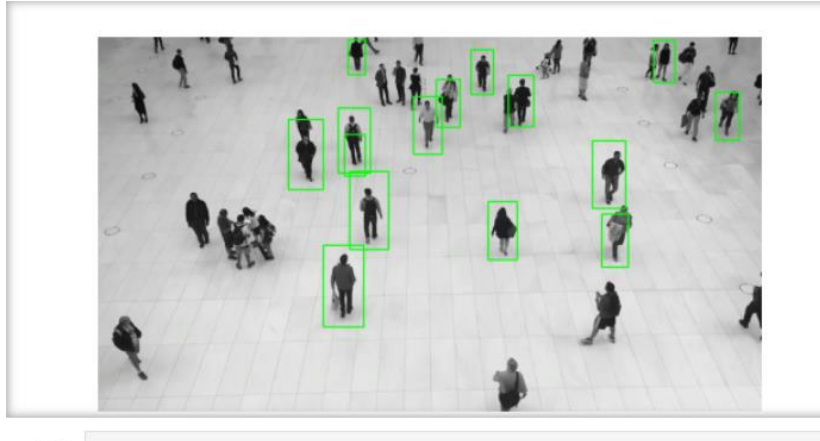
Method 2:
Code for ideal script

CODE

```
import cv2
import matplotlib.pyplot as plt
import matplotlib.animation as animation


#Initialize the Haar Cascade pedestrian detection model


pedestrian_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_fullbody.xml')


# Load the video
video_path = 'D:/VIDEO.mp4'
cap= cv2.VideoCapture(video_path)


#check if the video is opened successfully
if not cap.isOpened():
    print("Error: Unable to open the video.")
    exit()
```

```python
#Function to detect pedestrians and draw bounding boxes
def detect_pedestrians(frame):
    #Convert the frame to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    #Detect pedestrians in the grayscale frame
    pedestrians = pedestrian_cascade.detectMultiScale(gray_frame,
scaleFactor=1.05,minNeighbors=5,minSize=(30,30))

    #Draw rectangles around the detected pedestrains
    for(x,y,w,h) in pedestrians:
        cv2.rectangle( frame,(x,y),(x+w,y+h),(0,255,0),2)
    return frame
fig,ax=plt.subplots()
#Function to update the animation
def update(frame):
    #Read a frME FROM THE VIDEO
    ret,frame = cap.read()
    if not ret:
        ani.event_source.stop()
        return

    #Detect pedestrians and draw boundng boxes
    frame_with_pedestrains=detect_pedestrians(frame)

    ax.clear()
    ax.imshow(cv2.cvtColor(frame_with_pedestrains, cv2.COLOR_BGR2RGB))
    ax.axis('off')

#create the animation
ani=animation.FuncAnimation(fig,update,interval=50)

#Display the animation
plt.show()
#Release the video capture object
```

```
cap.release
```

OUTPUT

D) Face Recognition

Theory:
Facial recognition works in three steps: detection, analysis, and recognition.

Detection
Detection is the process of finding a face in an image. Enabled by computer vision, facial recognition can detect and identify individual faces from an image containing one or many people's faces. It can detect facial data in both front and side face profiles.

Computer vision
Machines use computer vision to identify people, places, and things in images with accuracy at or above human levels and with much greater speed and efficiency. Using complex artificial intelligence (AI) technology, computer vision automates extraction, analysis, classification, and understanding of useful information from image data. The image data takes many forms, such as the following:

- Single images
- Video sequences
- Views from multiple cameras
- Three-dimensional data
-

Analysis
The facial recognition system then analyzes the image of the face. It maps and reads face geometry and facial expressions. It identifies facial landmarks that are key to distinguishing a face from other objects. The facial recognition technology typically looks for the following:

- Distance between the eyes
- Distance from the forehead to the chin
- Distance between the nose and mouth
- Depth of the eye sockets
- Shape of the cheekbones
- Contour of the lips, ears, and chin

The system then converts the face recognition data into a string of numbers or points called a faceprint. Each person has a unique faceprint, similar to a fingerprint. The information used by facial recognition can also be used in reverse to digitally reconstruct a person's face.

Recognition

Facial recognition can identify a person by comparing the faces in two or more images and assessing the likelihood of a face match. For example, it can verify that the face shown in a selfie taken by a mobile camera matches the face in an image of a government-issued ID like a driver's license or passport, as well as verify that the face shown in the selfie does not match a face in a collection of faces previously captured.

<u>CODE</u>

```
!pip install dlib-19.22.99-cp38-cp38-win_amd64.whl
!pip install face_recognition
import face_recognition
from matplotlib.patches import Rectangle
import matplotlib.pyplot as plt


known_image = face_recognition.load_image_file('ryan.jpg')
known_face_encoding = face_recognition.face_encodings(known_image)[0]


unknown_image = face_recognition.load_image_file('test_ryan.jpg')
unknown_face_locations = face_recognition.face_locations(unknown_image)
unknown_face_encodings = face_recognition.face_encodings(unknown_image,
unknown_face_locations)


unknown_image_rgb = unknown_image[:, :, ::-1]


fig,ax= plt.subplots(figsize=(8,6))
ax.imshow(unknown_image_rgb)


for (top, right, bottom, left), unknown_face_encoding in
zip(unknown_face_locations, unknown_face_encodings):
    results=face_recognition.compare_faces([known_face_encoding],
unknown_face_encoding)
```
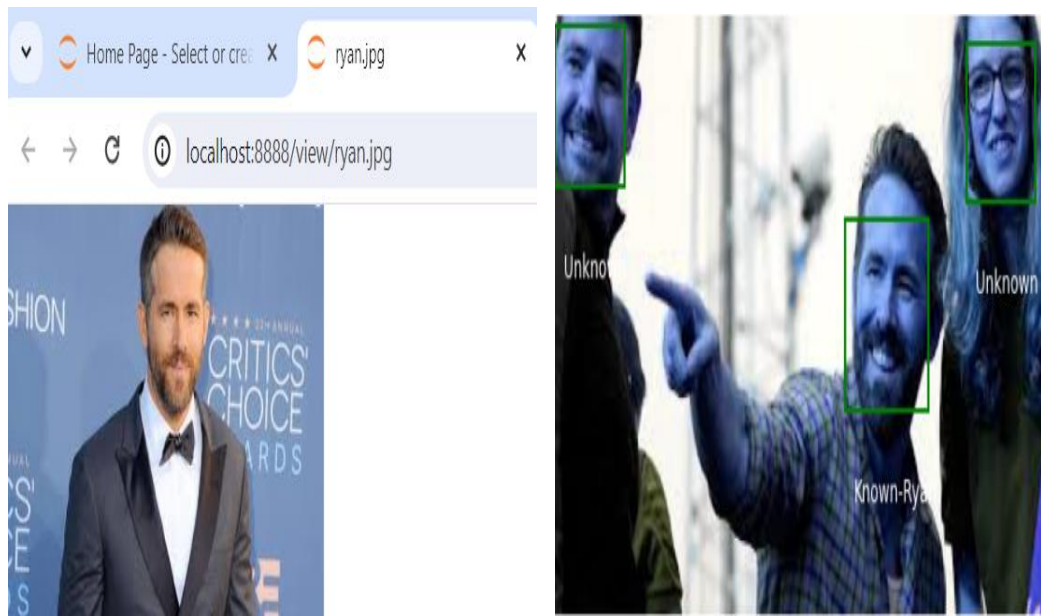
```
    if results[0]:
        name="Known-Ryan"
    else:
        name="Unknown"


    ax.add_patch(Rectangle((left, top), right - left, bottom-top,
fill=False, color='green', linewidth=2))
    ax.text(left+6, bottom+25, name, color='white', fontsize=12)


    plt.axis('off')
```

OUTPUT

Practical 5

Aim: Construct 3D model from Images:
Theory:

Transforming a 2D image into a 3D space using OpenCV refers to the process of converting a two-dimensional image into a three-dimensional spatial representation using the Open Source Computer Vision Library (OpenCV). OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library written in C++ with interfaces for Python, Java, and MATLAB. It provides a wide range of features for processing and evaluating pictures and movies. This transformation involves inferring the depth information from the 2D image, typically through techniques such as stereo vision, depth estimation, or other computer vision algorithms, to create a 3D model with depth perception. This process enables various applications such as 3D reconstruction, depth sensing.

Modules Needed
- Matplotlib: It is a plotting library for Python programming it serves as a visualization utility library, Matplotlib is built on NumPy arrays, and designed to work with the broader SciPy stack.
- Numpy: It is a general-purpose array-processing package. It provides a high-performance multidimensional array and matrices along with a large collection of high-level mathematical functions.
- mpl_toolkits: It provides some basic 3d plotting (scatter, surf, line, mesh) tools.

SIFT Algorithm

The SIFT (Scale-Invariant Feature Transform) algorithm is a computer vision technique used for feature detection and description. It detects distinctive key points or features in an image that are robust to changes in scale, rotation, and affine transformations. SIFT (scale invariant feature transform) works by identifying key points based on their local intensity extrema and computing descriptors that capture the local image information around those key points. These descriptors can then be used for tasks like image matching, object recognition, and image retrieval. SIFT algorithm helps locate the local features in an image, commonly known as the 'keypoints' of the image. These keypoints are scale & rotation invariants that can be used for various computer vision applications, like image matching, object detection, scene detection, etc.

CODE

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def extract_keypoints_and_descriptors(image):
    sift = cv2.SIFT_create()
```

```
    keypoints, descriptors = sift.detectAndCompute(image, None)
    return keypoints, descriptors
def match_keypoints(desc1, desc2):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(desc1, desc2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches
def draw_matches(img1, kp1, img2, kp2, matches):
    return cv2.drawMatches(img1, kp1, img2, kp2, matches[:50], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)


def reconstruct_3d_points(K, kp1, kp2, matches, E):
    points1 = np.float32([kp1[m.queryIdx].pt for m in matches])
    points2 = np.float32([kp2[m.trainIdx].pt for m in matches])
    _, R, t, mask = cv2.recoverPose(E, points1, points2, K)

    # Triangulate points
    P1 = np.hstack((np.eye(3, 3), np.zeros((3, 1))))
    P2 = np.hstack((R, t))

    points_4d_hom = cv2.triangulatePoints(P1, P2, points1.T, points2.T)
    points_4d = points_4d_hom / np.tile(points_4d_hom[-1, :], (4, 1))
    points_3d = points_4d[:3, :].T
    return points_3d
# Load two images of a scene
img1 = cv2.imread('3d1.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('3d2.jpg', cv2.IMREAD_GRAYSCALE)

# Placeholder values for the camera matrix K (Students you should change it
based on your camera parameters.
fx = 1000  # Focal length in pixels
fy = 1000  # Focal length in pixels
cx = img1.shape[1] / 2  # Optical center X coordinate, assuming center of
the image
cy = img1.shape[0] / 2  # Optical center Y coordinate, assuming center of
the image
K = np.array([[fx, 0, cx],
              [0, fy, cy],
              [0, 0, 1]], dtype=float)
# Extract keypoints and descriptors
```

```python
kp1, desc1 = extract_keypoints_and_descriptors(img1)
kp2, desc2 = extract_keypoints_and_descriptors(img2)


# Match keypoints
matches = match_keypoints(desc1, desc2)
# Estimate the essential matrix
points1 = np.float32([kp1[m.queryIdx].pt for m in matches])
points2 = np.float32([kp2[m.trainIdx].pt for m in matches])
F, mask = cv2.findFundamentalMat(points1, points2, cv2.FM_RANSAC)
E = K.T @ F @ K


# Reconstruct 3D points
points_3d = reconstruct_3d_points(K, kp1, kp2, matches, E)
# Visualize 3D points
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(points_3d[:, 0], points_3d[:, 1], points_3d[:, 2])
plt.show()


# Display matched keypoints
matched_img = draw_matches(img1, kp1, img2, kp2, matches)
plt.imshow(matched_img)
plt.show()
```
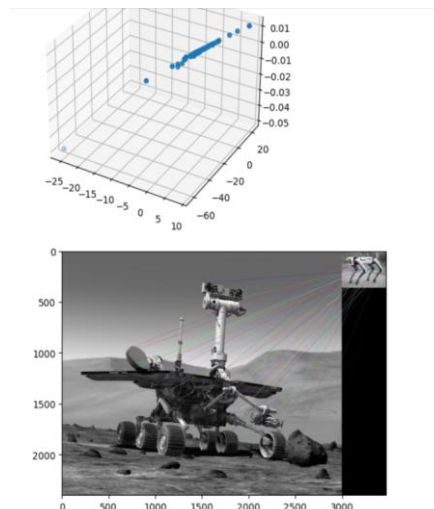
OUTPUT

Practical 6

Aim: Implement object detection and tracking from video

Theory: Object detection, within computer vision, involves identifying objects within images or videos. Object detection is merely to recognize the object with bounding box in the image, where in image classification, we can simply categorize(classify) that is an object in the image or not in terms of the likelihood (Probability). It is like object recognition, which involves identifying the object category, but object detection goes a step further and localizes the object within the image. Object detection applications can be performed using two different data analysis techniques:

A. Image Processing
In image processing, the algorithm analyses an image to detect objects based on certain features, such as edges or textures. This approach typically involves applying a series of image processing techniques, such as filtering, thresholding, and segmentation, to extract regions of interest in the image. These regions are then analyzed to determine whether they contain an object or not.

B. Deep Neural Network
In a deep neural network, the algorithm is trained on large, labelled datasets to detect objects. This approach involves using a convolutional neural network (CNN) architecture, which is a type of deep learning algorithm specifically designed for image analysis. The network learns to detect objects by analysing features of the image at different levels of abstraction, using multiple layers of artificial neurons to perform increasingly complex analyses. The output of the network is a bounding box around the detected object, along with a confidence score that represents how certain the network is that the object is present.

Categories of Object Detection:
- Single-shot detectors: Single-shot detectors combine localization and classification and they perform very quickly.
- Two-stage detectors: Two-stage detectors separate object localization and classification in the head, generally returns higher localization accuracy.
-

Object detection algorithms and architectures
There are a number of machine learning approaches to object detection tasks. Examples include the Viola-Jones framework8 and the histogram of oriented gradients. Recent object detection research and development, however, has focused largely on convolutional neural networks (CNNs).

R-CNN (region-based convolutional neural network) is a two-stage detector that uses a method called region proposals to generate 2,000 region predictions per image. R-CNN then warps the extracted regions to a uniform size and runs those regions through separate networks for feature extraction and classification.
YOLO (You Only Look Once) is a family of single-stage detection architectures based in Darknet, an open-source CNN framework. YOLO's speed makes it preferable for

real-time object detection and has earned it the common descriptor of state-of-the-art object detector. YOLO makes less than 100 bounding box predictions per image, it also produces less background false positives, although it has a higher localization error, generally focuses on speed and accuracy.

CODE

```python
import cv2
import numpy as np
from IPython.display import display, clear_output
import matplotlib.pyplot as plt


net=cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")
classes=[]
with open("coco.names", "r") as f:
    classes=[line.strip() for line in f.readlines()]
layer_names=net.getLayerNames()
output_layers=[layer_names[i-1] for i in net.getUnconnectedOutLayers()]


cap=cv2.VideoCapture('video2.mp4')


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


        height, width, channels = frame.shape


        blob=cv2.dnn.blobFromImage(frame, 0.00392, (416,416), (0,0,0),
True, crop=False)
        net.setInput(blob)
        outs=net.forward(output_layers)


    class_ids=[]
    confidences=[]
    boxes=[]
```

```python
    for out in outs:
        for detection in out:
            scores=detection[5:]
            class_id=np.argmax(scores)
            confidence=scores[class_id]
            if confidence > 0.5:
                center_x=int(detection[0]*width)
                center_y=int(detection[1]*height)
                w=int(detection[2]*width)
                h=int(detection[3]*height)

                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)

        indexes=cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
        for i in range(len(boxes)):
            if i in indexes:
                x,y,w,h=boxes[i]
                label=str(classes[class_ids[i]])
                color=(0,255,0) #GReen
                cv2.rectangle(frame, (x,y), (x+w, y+h), color, 2)
                cv2.putText(frame, label, (x,y+ 30),
cv2.FONT_HERSHEY_PLAIN,1,color,2)

        frame_rgb=cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        plt.figure(figsize=(10,10))
        plt.imshow(frame_rgb)
        plt.axis('off')
        display(plt.gcf())
        clear_output(wait=True)
        plt.close()
```
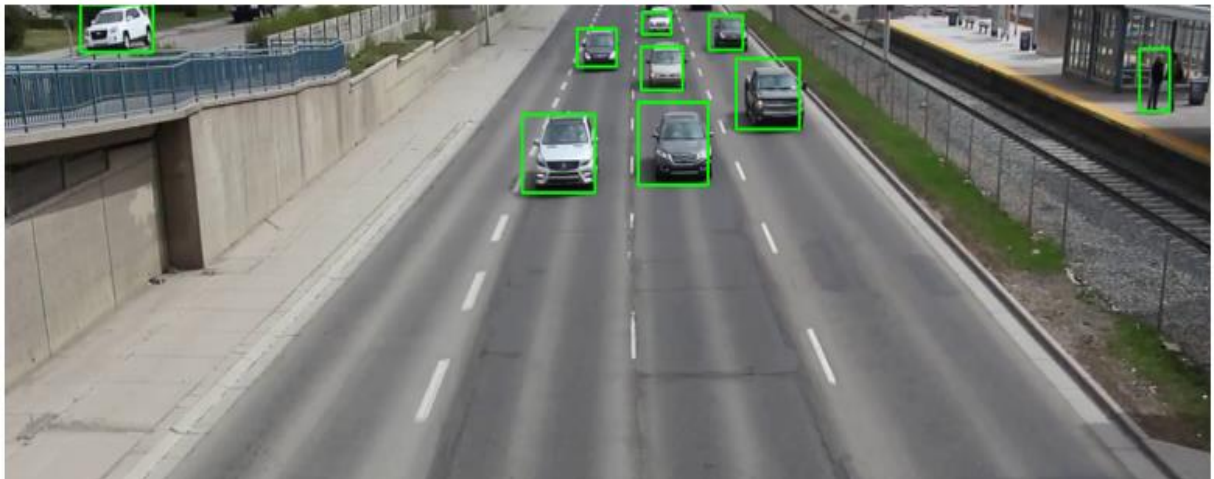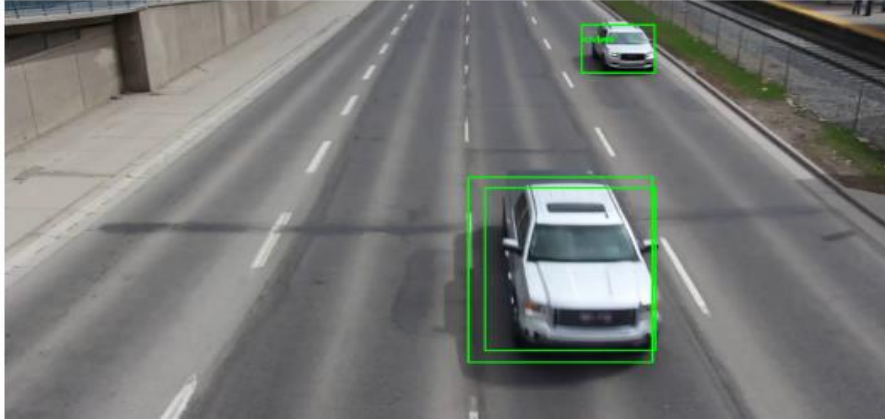
```
finally:
    cap.release()
    print("Stream ended")
```

Output:

Practical 7

Aim: Perform Feature extraction using RANSAC

Theory:

Random sample consensus, or RANSAC, is an iterative method for estimating a mathematical model from a data set that contains outliers. The RANSAC algorithm works by identifying the outliers in a data set and estimating the desired model using data that does not contain outliers.

RANSAC is accomplished with the following steps:

- Randomly selecting a subset of the data set
- Fitting a model to the selected subset
- Determining the number of outliers
- Repeating steps 1-3 for a prescribed number of iterations

In computer vision, RANSAC is used as a robust approach to estimate the fundamental matrix in stereo vision, for finding the commonality between two sets of points for feature-based object detection, and registering sequential video frames for video stabilization.

Advantages and disadvantages of RANSAC:

An advantage of RANSAC is its ability to do robust estimation of the model parameters, i.e., it can estimate the parameters with a high degree of accuracy even when a significant number of outliers are present in the data set. A disadvantage of RANSAC is that there is no upper bound on the time it takes to compute these parameters (except exhaustion). When the number of iterations computed is limited the solution obtained may not be optimal, and it may not even be one that fits the data in a good way. In this way RANSAC offers a trade-off; by computing a greater number of iterations the probability of a reasonable model being produced is increased.

CODE

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


# Load the color images
img1_color = cv2.imread('left1.jpg')  # Query image (color)
img2_color = cv2.imread('right1.jpg')  # Train image (color)


# Convert to grayscale for feature detection
img1 = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)


# Initialize SIFT detector
```

```python
sift = cv2.SIFT_create()
# Detect keypoints and compute descriptors
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)


# FLANN parameters and matcher setup
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)


flann = cv2.FlannBasedMatcher(index_params, search_params)


# Match descriptors using KNN
matches = flann.knnMatch(des1, des2, k=2)
# Ratio test to keep good matches
good = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good.append(m)


# Extract location of good matches
points1 = np.zeros((len(good), 2), dtype=np.float32)
points2 = np.zeros((len(good), 2), dtype=np.float32)


for i, match in enumerate(good):
    points1[i, :] = kp1[match.queryIdx].pt
    points2[i, :] = kp2[match.trainIdx].pt
# Find homography using RANSAC
    H, status = cv2.findHomography(points1, points2, cv2.RANSAC)


# Create a new image that puts the two images side by side
    height = max(img1_color.shape[0], img2_color.shape[0])
    width = img1_color.shape[1] + img2_color.shape[1]
    output = np.zeros((height, width, 3), dtype=np.uint8)


# Place both images within this new image
    output[0:img1_color.shape[0], 0:img1_color.shape[1]] = img1_color
    output[0:img2_color.shape[0],
img1_color.shape[1]:img1_color.shape[1]+img2_color.shape[1]] = img2_color
# Draw lines between the matching points
```

```
for i, (m, color) in enumerate(zip(good, np.random.randint(0, 255,
(len(good), 3)))):
    if status[i]:
        pt1 = tuple(map(int, kp1[m.queryIdx].pt))
        pt2 = tuple(map(int, kp2[m.trainIdx].pt))
        pt2 = (pt2[0] + img1_color.shape[1], pt2[1])  # Shift the point for
img2

        cv2.line(output, pt1, pt2, color.tolist(), 2)


# Convert the result to RGB for matplotlib display and show the final image
output_rgb = cv2.cvtColor(output, cv2.COLOR_BGR2RGB)
# Use matplotlib to display the image
plt.figure(figsize=(15, 5))
plt.imshow(output_rgb)
plt.axis('off')  # Turn off axis numbers and ticks
plt.title('Feature Matching with RANSAC')
plt.show()
```

OUTPUT

Feature Matching with RANSAC

Practical 8

Aim: **Perform Colorization**

Theory: Image colorization is the process of taking an input grayscale (black and white) image and then producing an output colorized image that represents the semantic colors and tones of the input.

Challenges in Image Colorization:

- **Loss of Color Information**: A fundamental challenge in image colorization is the loss of color information during the conversion to grayscale. Color images capture the richness of the real world by encoding the full spectrum of light hitting each pixel. This spectrum is typically represented using three channels: red, green, and blue (RGB). Grayscale images, on the other hand, discard this detailed color data and only retain the luminance information, essentially representing how bright or dark each pixel is. This loss of information creates a significant obstacle for colorization algorithms.
- **Ambiguity in Grayscale Images:** Unlike color images that capture the full spectrum of light, grayscale images only represent luminance, or brightness. This ambiguity makes it difficult for colorization algorithms to definitively assign colors to grayscale pixels, requiring them to rely on additional information or make educated guesses to achieve a realistic outcome.
- **Lack of Semantic Understanding:** Grayscale images lack the rich information that allows us to distinguish between a brown bear and a polar bear, for instance. This semantic ambiguity presents difficulties. Colorization algorithms may assign colors based solely on local image features, potentially leading to unrealistic color choices.
- **Perceptual Color Constancy:** Colorization algorithms need to not only predict colors for grayscale pixels but also ensure those predicted colors appear consistent with the perceived lighting in the scene. Imagine a grayscale image of a landscape at dusk. The algorithm must not only assign colors to the sky and trees, but also account for the warm, yellowish light typical of dusk to achieve a realistic and believable colorization.
- **Color Inconsistency (Color Bleeding):** It occurs when the predicted colors for neighboring regions in the image fail to respect clear boundaries. During colorization, color bleeding might cause the red color of the rose to spill over and contaminate the edges of the leaf, resulting in a greenish tinge. This happens because colorization algorithms often rely on local image features and may struggle to differentiate between distinct objects with similar grayscale values.

In this practical, we will create a program to convert a black & white image i.e grayscale image to a colour image. We are going to use the Caffe colourization model for this program. And you should be familiar with basic OpenCV functions and uses like reading an image or how to load a pre-trained model using dnn module etc. Now let us discuss the procedure that we will follow to implement the program. Like RGB, lab colour has 3 channels L, a, and b. But here instead of pixel values, these have different significances i.e:

- **L-channel:** light intensity
- **a channel:** green-red encoding

- **b channel:** blue-red encoding

And in our program, we will use the L channel of our image as input to our model to predict ab channel values and then rejoin it with the L channel to generate our final image.

Steps:

Load the model and the convolution/kernel points

Read and preprocess the image

Generate model predictions using the L channel from our input image

Use the output -> ab channel to create a resulting image

CODE

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


net=cv2.dnn.readNetFromCaffe('colorization_deploy_v2.prototxt',
'colorization_release_v2.caffemodel')


pts_in_hull= np.load('pts_in_hull.npy', allow_pickle=True)


class8=net.getLayerId("class8_ab")
conv8=net.getLayerId("conv8_313_rh")
pts_in_hull=pts_in_hull.transpose().reshape(2,313,1,1)
net.getLayer(class8).blobs= [pts_in_hull.astype(np.float32)]
net.getLayer(conv8).blobs=[np.full([1,313], 2.606, np.float32)]


image=cv2.imread('b&w_1.jpg')


gray_image=cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)


gray_image=cv2.cvtColor(gray_image, cv2.COLOR_GRAY2RGB)


normalized_image=gray_image.astype('float32') / 255.0


lab_image=cv2.cvtColor(normalized_image, cv2.COLOR_RGB2Lab)
```

```python
resized_l_channel=cv2.resize(lab_image[:,:,0], (224,224))
resized_l_channel-=50


net.setInput(cv2.dnn.blobFromImage(resized_l_channel))
pred=net.forward()[0,:,:,:].transpose((1,2,0))


pred_resized=cv2.resize(pred,(image.shape[1], image.shape[0]))


colorized_image=np.concatenate((lab_image[:,:,0][:,:, np.newaxis],
pred_resized), axis=2)


colorized_image=cv2.cvtColor(colorized_image, cv2.COLOR_Lab2BGR)


colorized_image-np.clip(colorized_image, 0 ,1)


colorized_image=(255*colorized_image).astype('uint8')
cv2.imwrite('path_to_output/colorized_image.jpg', colorized_image)


#cv2.imshow('Colorized Image', colorized_image)
#cv2.waitKey(0)
#cv2.destroyAllWindows()


colorized_image_rgb= cv2.cvtColor(colorized_image, cv2.COLOR_BGR2RGB)


plt.figure(figsize=(8,8))
plt.imshow(colorized_image_rgb)
plt.axis('off')
plt.show()
```
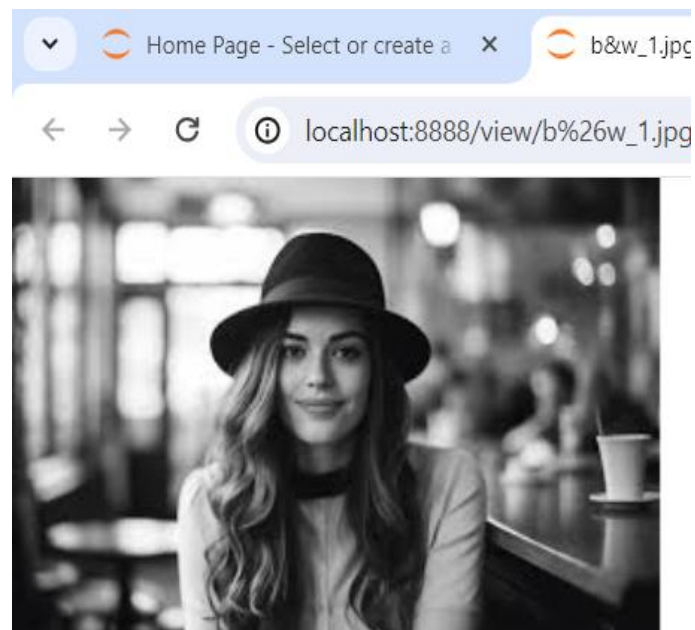
<u>OUTPUT</u>

Practical 9

Aim: **Perform Text Detection and Recognition**

Theory: Computer vision involves extracting information from visual data and allows us to perform complex tasks such as classification, prediction, recognition, and much more. In this practical, we will look at how to detect text using Tesseract in media, a classic optical character recognition application.

Optical character recognition:
Optical character recognition (OCR), is a revolutionary technology that enables machines to interpret and convert images of text into machine-readable formats. It allows us to utilize the potential of printed or handwritten text.

Simply put, the goal of OCR is to convert the human perception of characters and convert them into machine-encoded text.

The concept of optical character recognition is used in text detection, where we aim to identify and recognize the text found within an image or a video.

The Vision API can detect and extract text from images. There are two annotation features that support optical character recognition (OCR):
- TEXT_DETECTION detects and extracts text from any image. For example, a photograph might contain a street sign or traffic sign. The JSON includes the entire extracted string, as well as individual words, and their bounding boxes. In text detection, our goal is to automatically compute the bounding boxes for every region of text in an image.
- DOCUMENT_TEXT_DETECTION also extracts text from an image, but the response is optimized for dense text and documents. The JSON includes page, block, paragraph, word, and break information.

CODE

```
import pytesseract
import cv2
from pytesseract import Output
# Specify the tesseract executable path
pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-
OCR\tesseract.exe'
# Load the image
image = cv2.imread('qoute.jpg')
original_image = image.copy() # Make a copy for displaying the original
later
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```python
# Use Opencv to find text blocks (simple thresholding)
_, thresh = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY_INV)
# Dilate to connect text characters
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5,5))
dilate = cv2.dilate(thresh, kernel, iterations = 3)
# Find Contours
contours, _ = cv2.findContours(dilate, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
# Sort Contours by their y-coordinate first, then x-coordinate
lines = sorted(contours, key=lambda ctr:
(cv2.boundingRect(ctr)[1],cv2.boundingRect(ctr)[0]))
# Go through each contour, crop and read the text
for contour in lines:
    x,y,w,h = cv2.boundingRect(contour)
    # Make sure the contour area is a reasonable size
    if w*h > 50:
        roi = image[y:y+h, x:x+w]
        text = pytesseract.image_to_string(roi, lang='eng', config= '--psm
6')
        cv2.putText(image, text, (x,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.3,
(0,255,0), 1)
        cv2.rectangle(image, (x,y), (x+w,y+h), (0,255,0), 1)
        print(text)
import matplotlib.pyplot as plt
original_image_rgb = cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
# Display the original image
plt.figure(figsize=(10,10))
plt.subplot(1,2,1)
plt.imshow(original_image_rgb)
plt.title('Original Image')
plt.axis('off')
# Display the image with detected text
plt.subplot(1,2,2)
plt.imshow(image_rgb)
plt.title('Image With Text')
```

```
plt.axis('off')

plt.show()
```

OUTPUT

REMEMBER WHEN
YOU WANTED
WHAT YOU HAVE
TODAY.

Practical 10

Aim: Perform Image matting and compositing

Theory: Image Matting is the process of accurately estimating the foreground object in images and videos. It is a very important technique in image and video editing applications, particularly in film production for creating visual effects.

In case of image segmentation, we segment the image into foreground and background by labeling the pixels. Image segmentation generates a binary image, in which a pixel either belongs to foreground or background. However, Image Matting is different from the image segmentation, wherein some pixels may belong to foreground as well as background, such pixels are called partial or mixed pixels.

In order to fully separate the foreground from the background in an image, accurate estimation of the alpha values for partial or mixed pixels is necessary.

Compositing is the process or technique of combining visual elements from separate sources into single images, often to create the illusion that all those elements are parts of the same scene.

In the matting process, a foreground element of arbitrary shape is extracted from a background image. The matte extracted by this process describes the opacity of the foreground element at every point. In the compositing process, the foreground element is placed over a new background image, using the matte to hold out those parts of the new background that the foreground element obscures.

CODE

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


def create_mask_for_foreground(fg_image):

    fg_hsv=cv2.cvtColor(fg_image, cv2.COLOR_BGR2HSV)

    lower_green=np.array([36,25,25])
    upper_green=np.array([86, 255, 255])

    mask=cv2.inRange(fg_hsv, lower_green, upper_green)

    foreground_mask=cv2.bitwise_not(mask)
```

```python
    kernel=np.ones((3,3), np.uint8)
    foreground_mask=cv2.morphologyEx(foreground_mask, cv2.MORPH_OPEN,
kernel, iterations=2)
    foreground_mask=cv2.dilate(foreground_mask, kernel, iterations=4)
    return foreground_mask


def composite_fg_with_new_bg(fg_image, bg_image, fg_mask):
    bg_resized=cv2.resize(bg_image,(fg_image.shape[1], fg_image.shape[0]))
    fg_mask_normalized=fg_mask/255.0

    fg_prepared=cv2.bitwise_and(fg_image, fg_image, mask=fg_mask)
    bg_prepared=cv2.bitwise_and(bg_resized, bg_resized,
mask=cv2.bitwise_not(fg_mask))

    composite_image=cv2.add(fg_prepared, bg_prepared)
    return composite_image


foreground_image_path='greenscreen.jpg'
foreground=cv2.imread(foreground_image_path)
if foreground is None:
    raise ValueError("Error loading foreground image")


background_image_path='bg.jpg'
background=cv2.imread(background_image_path)
if background is None:

    background=np.full(foreground.shape, 255, dtype=foreground.dtype)


foreground_mask=create_mask_for_foreground(foreground)


composite_image=composite_fg_with_new_bg(foreground, background,
foreground_mask)


foreground_rgb=cv2.cvtColor(foreground, cv2.COLOR_BGR2RGB)
foreground_mask_rgb=cv2.cvtColor(foreground_mask, cv2.COLOR_GRAY2RGB)
composite_image_rgb=cv2.cvtColor(composite_image, cv2.COLOR_BGR2RGB)
```

```
plt.figure(figsize=(18,6))

plt.subplot(1,3,1)
plt.imshow(foreground_rgb)
plt.title('Original Image')
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(foreground_mask_rgb, cmap='gray')
plt.title('Foreground Mask')
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(composite_image_rgb)
plt.title('Composite Image')
plt.axis('off')

plt.show()
```

OUTPUT