Index

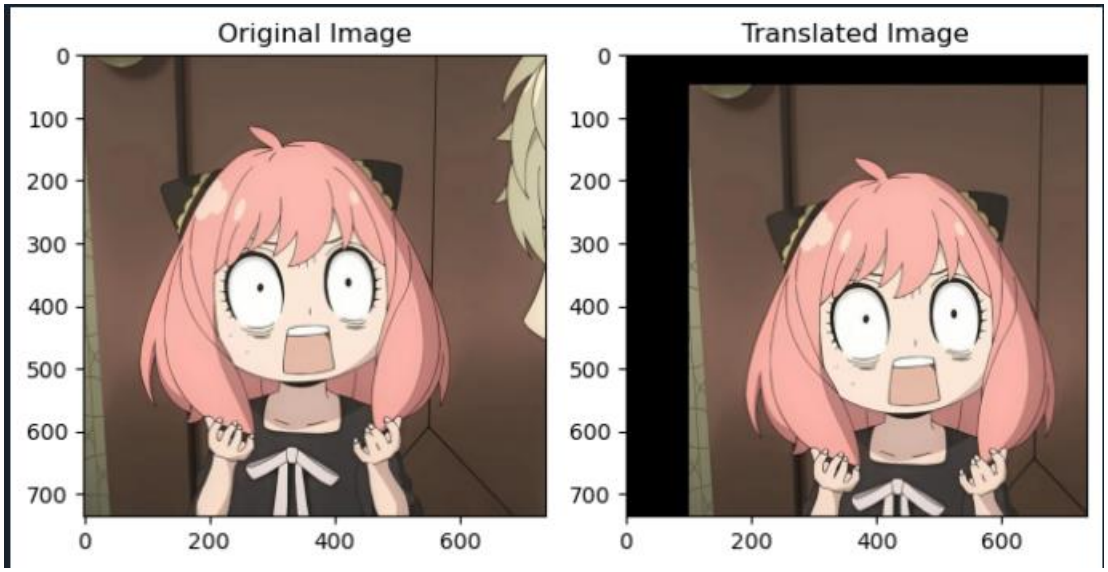
PRACTICAL – 1

Aim: Perform Geometric transformation.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
M = np.float32([[1, 0, 100], [0, 1, 50]])
dst = cv2.warpAffine(img_rgb, M, (cols, rows))
fig, axs = plt.subplots(1, 2, figsize=(7, 4))
axs[0].imshow(img_rgb)
axs[0].set_title('Original Image')
axs[1].imshow(dst)
axs[1].set_title('Translated Image')
plt.tight_layout()
plt.show()
```

Output:



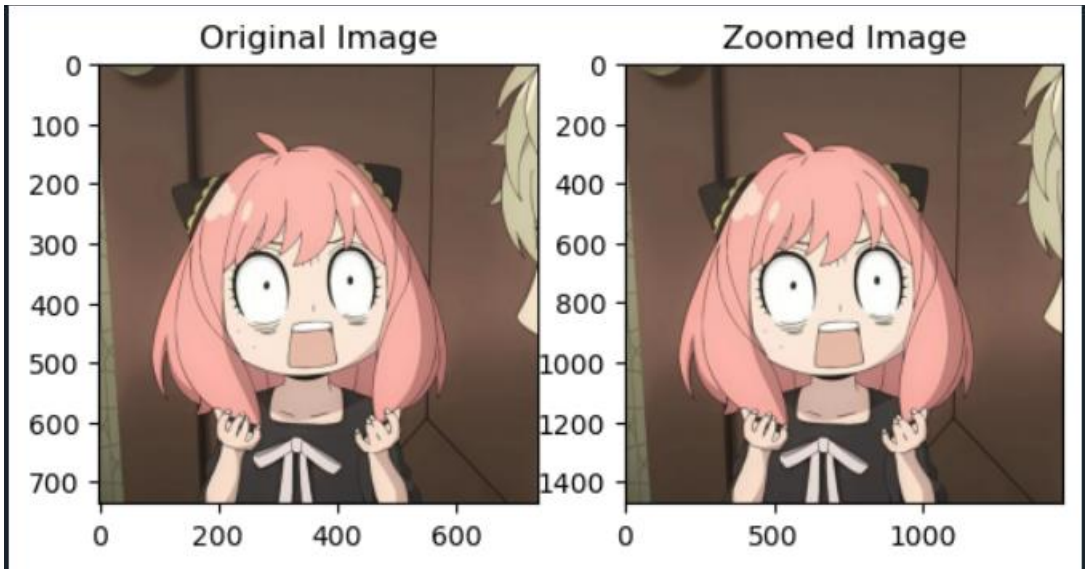
PRACTICAL – 1(A)

Aim: Image Scaling

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
resize_img = cv2.resize(img_rgb, (0, 0), fx=2, fy=2, interpolation=cv2.INTER_CUBIC)
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(resize_img), plt.title('Zoomed Image')
plt.show()
```

Output:



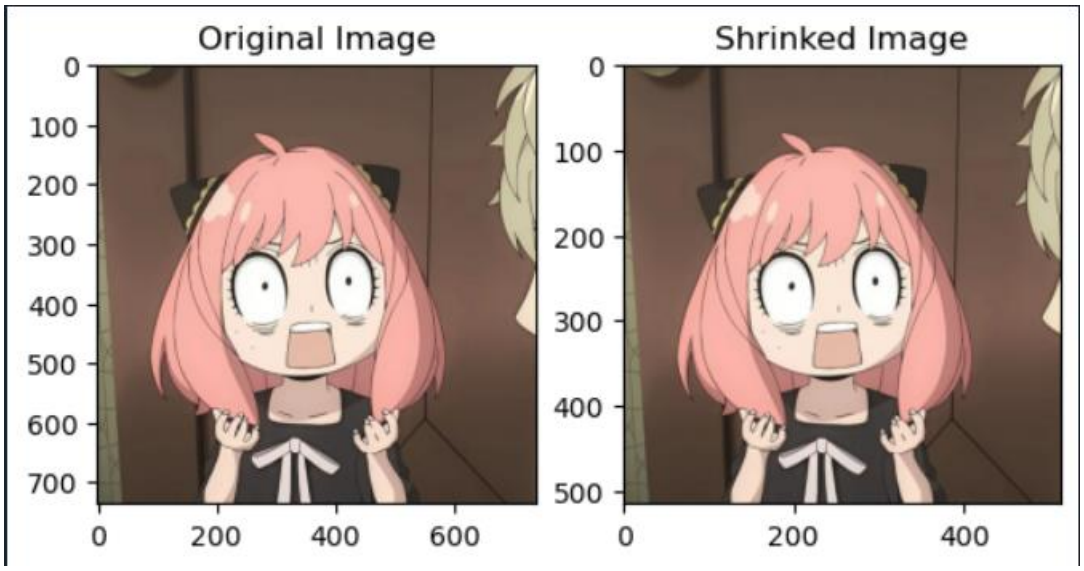
PRACTICAL – 1(B)

Aim: Image Shrinking

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
resize_img = cv2.resize(img_rgb, (0,0), fx=0.7, fy=0.7, interpolation=cv2.INTER_AREA)
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(resize_img), plt.title('Shrinked Image')
plt.show()
```

Output:



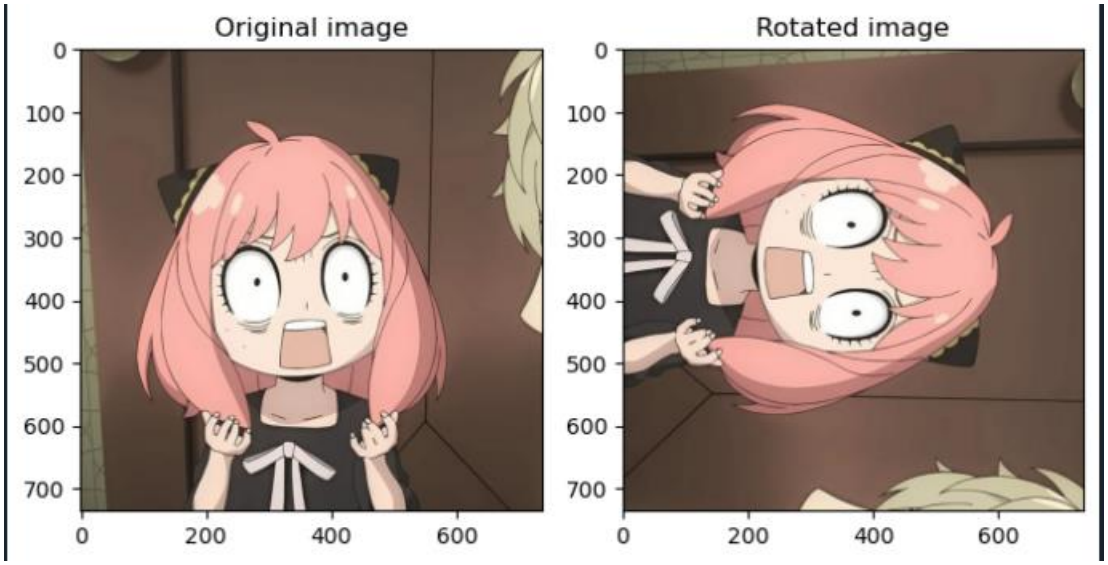
PRACTICAL – 1(C)

Aim: Image Rotation

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
center = (cols // 2, rows // 2)
angle = -90
scale = 1
rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale)
rotated_image = cv2.warpAffine(img_rgb, rotation_matrix, (cols, rows))
fig, axs = plt.subplots(1, 2, figsize=(7, 4))
axs[0].imshow(img_rgb)
axs[0].set_title("Original image")
axs[1].imshow(rotated_image)
axs[1].set_title("Rotated image")
plt.tight_layout()
plt.show()
```

Output:



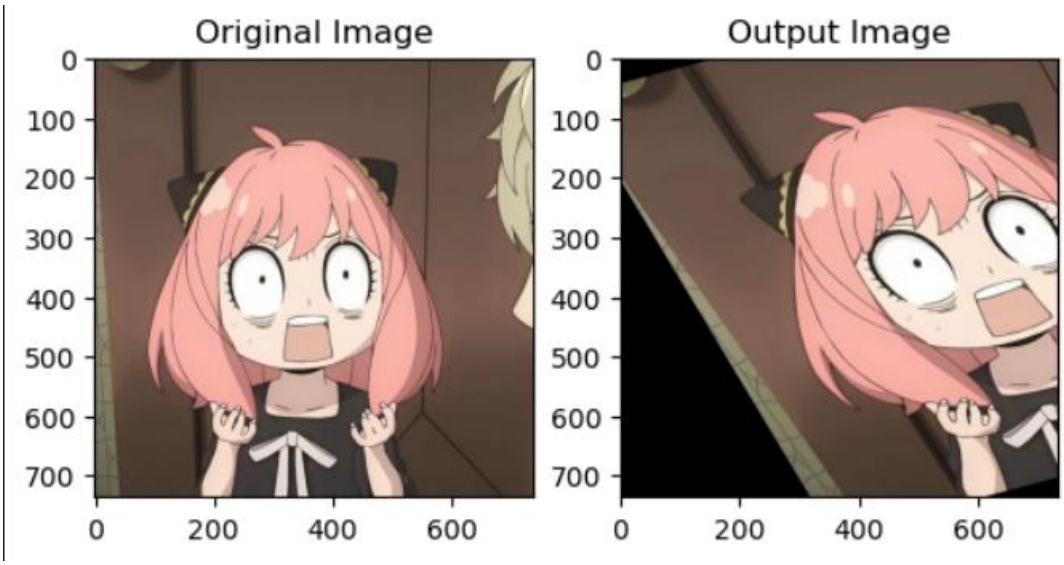
PRACTICAL – 1(D)

Aim: Affine Transformation

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
pts1 = np.float32([[50,50],[200,50],[50,200]])
pts2 = np.float32([[10,100],[200,50],[100,250]])
M = cv2.getAffineTransform(pts1,pts2)
dst = cv2.warpAffine(img_rgb, M, (cols, rows))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



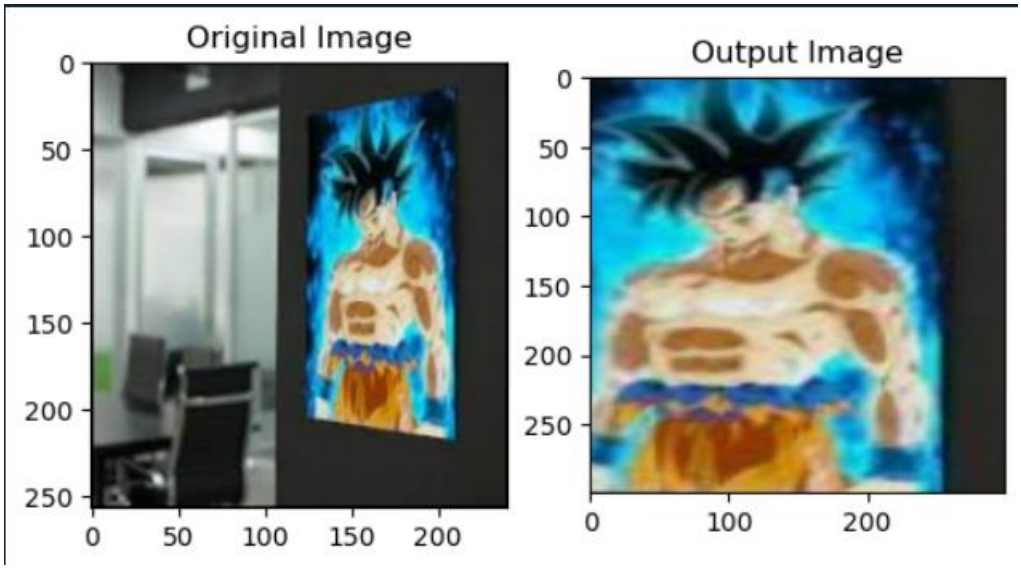
PRACTICAL – 1(E)

Aim: Perspective Transformation.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/1e.png")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
pts1 = np.float32([[133,34],[226,16],[133,206],[226,219]])
pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])
M = cv2.getPerspectiveTransform(pts1, pts2)
dst = cv2.warpPerspective(img_rgb, M, (300,300))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



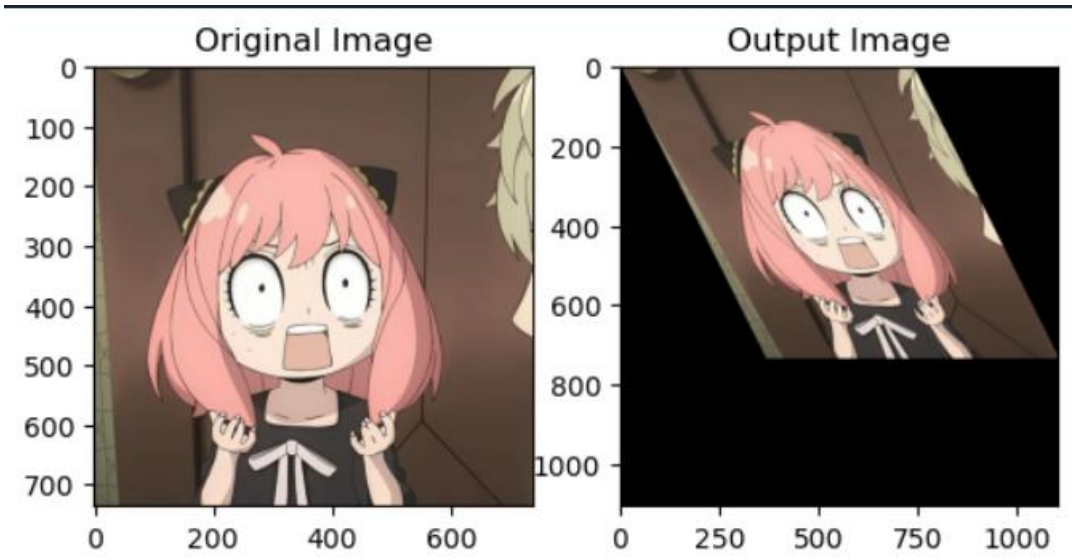
PRACTICAL – 1(F)

Aim: Shearing X-axis.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread(("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg"))
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
M = np.float32([[1, 0.5, 0], [0, 1, 0], [0, 0, 1]])
dst = cv2.warpPerspective(img_rgb, M, (int(cols * 1.5), int(rows * 1.5)))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



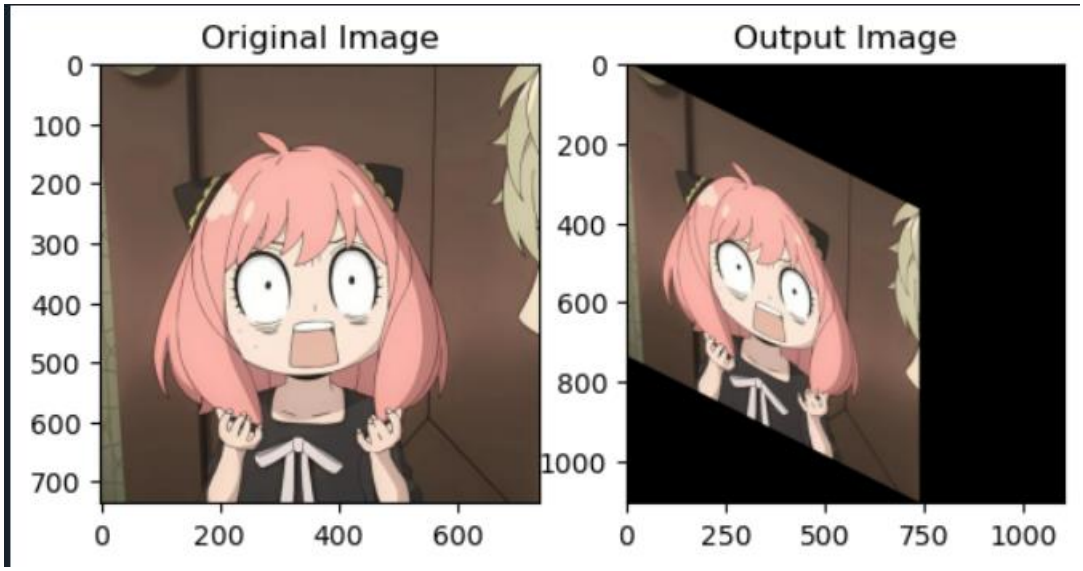
PRACTICAL – 1(G)

Aim: Shearing Y-axis.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
M = np.float32([[1, 0, 0], [0.5, 1, 0], [0, 0, 1]])
dst = cv2.warpPerspective(img_rgb, M, (int(cols * 1.5), int(rows * 1.5)))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



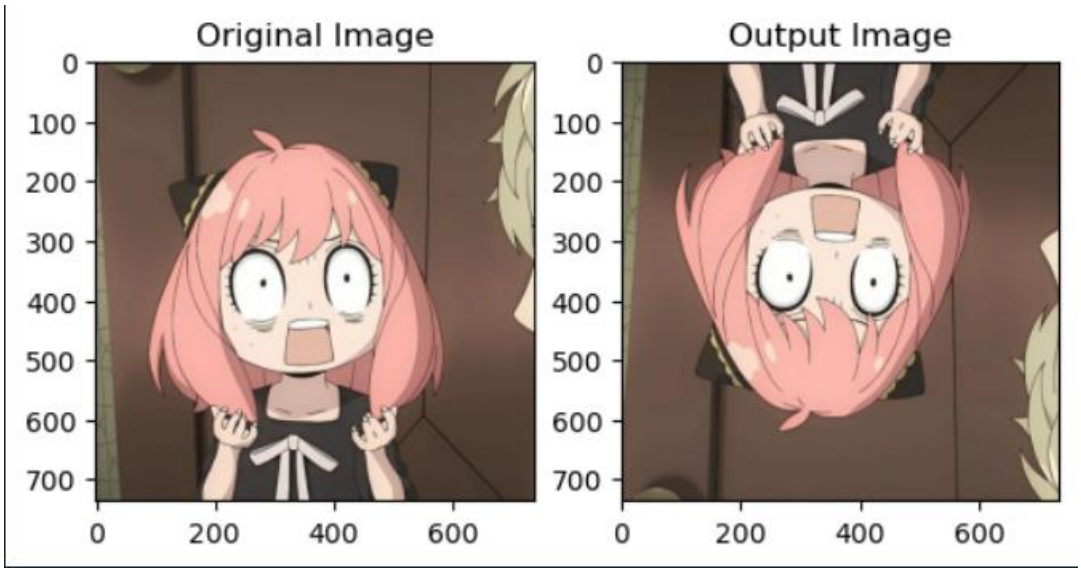
PRACTICAL – 1(H)

Aim: Reflected Image.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
M = np.float32([[1, 0, 0], [0, -1, rows], [0, 0, 1]])
dst = cv2.warpPerspective(img_rgb, M, (cols, rows))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



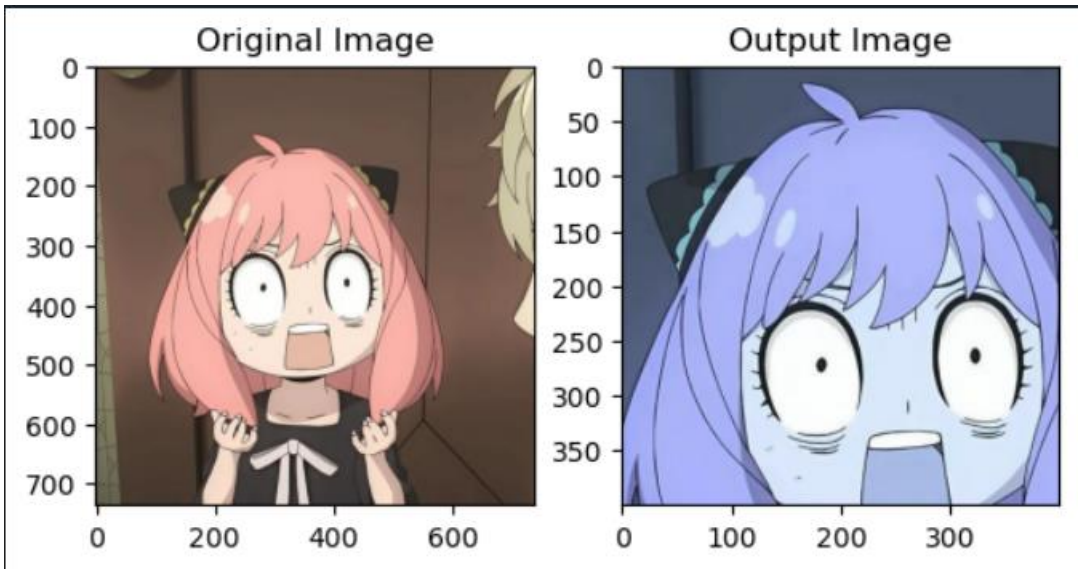
PRACTICAL – 1(I)

Aim: Cropped Image.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical1/anya1.jpg")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
dst = img[100:500, 100:500]
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



PRACTICAL – 2

Aim: Perform Image Stitching.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load images
img1 = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical2/right.jpg")
img2 = cv2.imread("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical2/left.jpg")
# Convert to grayscale
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
# SIFT feature detector
sift = cv2.SIFT_create()
kp1, des1 = sift.detectAndCompute(gray1, None)
kp2, des2 = sift.detectAndCompute(gray2, None)
# BFMatcher with KNN
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1, des2, k=2)
# Apply Lowe's ratio test
good_matches = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good_matches.append(m)
if len(good_matches) > 4:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
    # Compute homography
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    # Get dimensions for output
    height, width, _ = img2.shape
    panorama_width = width + img1.shape[1]
    # Warp first image
    result = cv2.warpPerspective(img1, H, (panorama_width, height))
    result[0:height, 0:width] = img2  # Overlay second image
    # Save and display
    cv2.imwrite("C:/Users/DELL/Desktop/practicals/sem2/CV
```

```
Practicals/practical2/result.jpg", result)
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.title("Stitched Panorama")
    plt.axis("off")
    plt.show()
else:
    print("Not enough keypoints found for stitching.")
```

Output:

PRACTICAL – 3

Aim: Perform Camera Calibration.

Code:

```
import numpy as np
import cv2 as cv
# Termination criteria for corner refinement
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
# Prepare object points (3D points)
objp = np.zeros((6*7, 3), np.float32)
objp[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)
# Lists to store object points and image points
objpoints = []
imgpoints = []
# Manually enter image paths
image_paths = [
    "C:/Users/DELL/Desktop/practicals/sem2/CV Practicals/practical3/ChessBoard.jpeg"
    #"C:/Users/ADMIN/Desktop/chess22.jpg"
]  # Add more image paths as needed
for fname in image_paths:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, (7,6), None)
    if ret:
        objpoints.append(objp)
        corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
        imgpoints.append(corners2)
        cv.drawChessboardCorners(img, (7,6), corners, ret)
        cv.imshow('img', img)
        cv.waitKey(500)
cv.destroyAllWindows()
# Camera calibration
ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None,
None)
# Print calibration results
print("Camera matrix: ")
print(mtx)
print("Distortion coefficients: ")
print(dist)
print("Rotation Vectors: ")
print(rvecs)
print("Translation Vectors: ")
print(tvecs)
# Read an image for undistortion
undistort_img_path = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical3/ChessBoard.jpeg"
img = cv.imread(undistort_img_path)
h, w = img.shape[:2]
newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w, h), 1, (w, h))
dst = cv.undistort(img, mtx, dist, None, newcameramtx)
```

```
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
# Save the undistorted image
cv.imwrite('C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical3/calibresult.png', dst)
print("Undistorted image saved as calibresult.png")
```

Output:

Original

Result

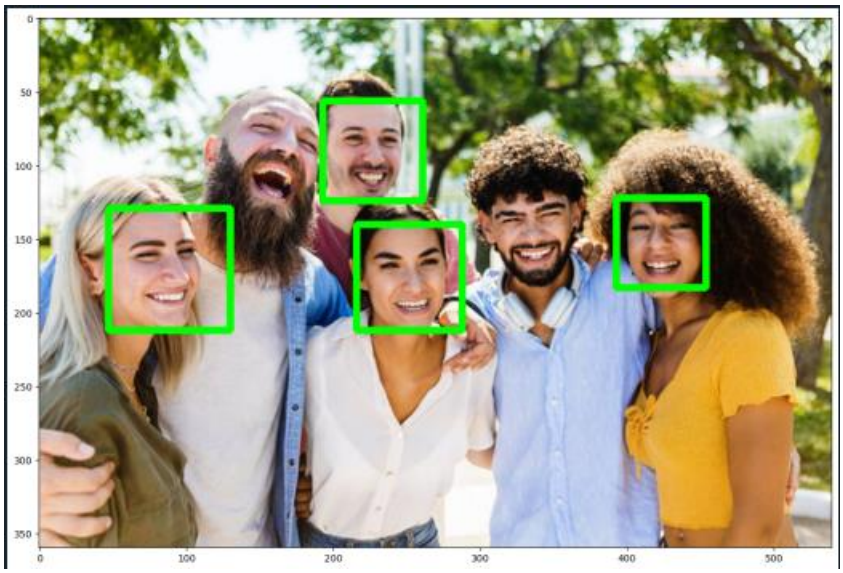
PRACTICAL – 4(A)

Aim: Perform the following Face detection.

Code:

```
import cv2
import matplotlib.pyplot as plt
imagePath = 'C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4a/grpimg.jpg'
img = cv2.imread(imagePath)
print(img.shape)
gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
face_classifier = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
face = face_classifier.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5,
minSize=(40, 40))
for (x, y, w, h) in face:
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 4)
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(20,10))
plt.imshow(img_rgb)
plt.show()
```

Output:



PRACTICAL – 4(B)

Aim: Perform the following Object detection.

Code:

```
import cv2
from matplotlib import pyplot as plt
# Load the image
image_path = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4b/practical4b-i/stop_sign.jpg"
imaging = cv2.imread(image_path)
# Check if image loaded correctly
if imaging is None:
    print("Error: Image not found! Check the file path.")
else:
    # Convert to grayscale
    imaging_gray = cv2.cvtColor(imaging, cv2.COLOR_BGR2GRAY)
    imaging_rgb = cv2.cvtColor(imaging, cv2.COLOR_BGR2RGB)
    # Load the Haar Cascade XML file
    xml_path = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4b/practical4b-i/stop_data.xml"
    xml_data = cv2.CascadeClassifier(xml_path)
    # Check if the XML file loaded properly
    if xml_data.empty():
        print("Error: XML file not found! Check the file path.")
    else:
        # Detect objects
        detecting = xml_data.detectMultiScale(imaging_gray, minSize=(30, 30))
        if len(detecting) > 0:
            for (x, y, w, h) in detecting:
                cv2.rectangle(imaging_rgb, (x, y), (x + w, y + h), (0, 255, 0), 9)
        # Display the image
        plt.imshow(imaging_rgb)
        plt.axis("off")  # Hide axes
        plt.show()
```

Output:

PRACTICAL – 4(C)

Aim: Perform the following Pedestrian detection

Code:

```
import cv2
import imutils
# Initialize HOG descriptor and set the default people detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
# Load the image
image_path = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4c/Pedestrian_image.jpg"
image = cv2.imread(image_path)
if image is None:
    print("Error: Image not found! Check the file path.")
    exit()
# Resize the image for better processing
image = imutils.resize(image, width=min(400, image.shape[1]))
# Detect people in the image
(regions, _) = hog.detectMultiScale(image, winStride=(4, 4), padding=(4, 4), scale=1.05)
# Draw rectangles around detected people
for (x, y, w, h) in regions:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 0, 255), 2)
# Display the output image
cv2.imshow("Detected Pedestrians", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

PRACTICAL – 4(D)

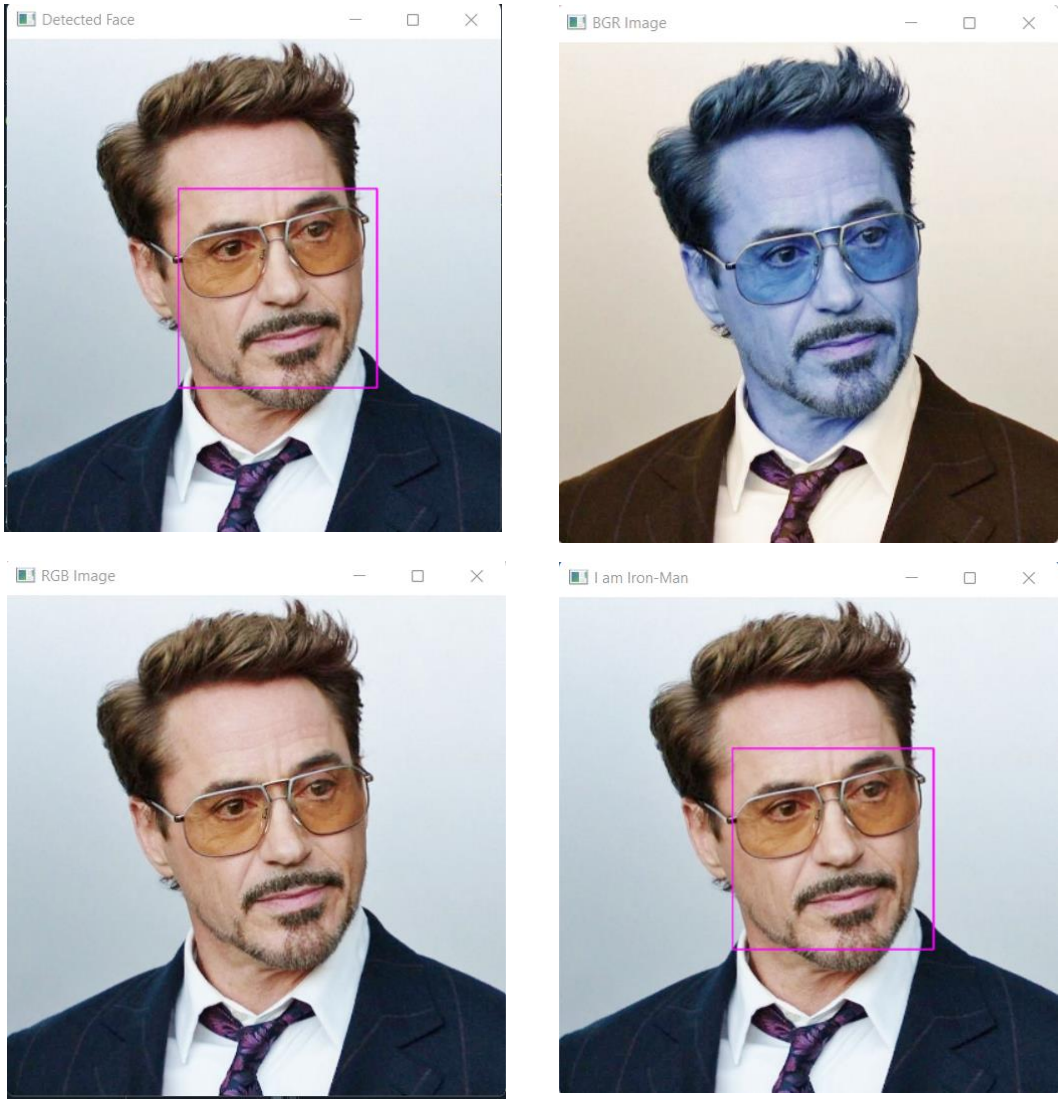
Aim: Perform the following Face Recognition.

Code:

```
import numpy as np
import face_recognition
import os
# Resize helper function
def resize_image(image, scale=0.5):
    width = int(image.shape[1] * scale)
    height = int(image.shape[0] * scale)
    return cv2.resize(image, (width, height))
# Load and check if image exists
image_path_1 = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4d/tonystark.jpg"
image_path_2 = "C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical4/practical4d/rdj_image.jpg"
if not os.path.exists(image_path_1) or not os.path.exists(image_path_2):
    print("Error: One or both image files not found! Check the file paths.")
    exit()
# Load images and convert color
img_bgr = face_recognition.load_image_file(image_path_1)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
# Show BGR and RGB images (resized)
cv2.imshow('BGR Image', resize_image(img_bgr))
cv2.imshow('RGB Image', resize_image(img_rgb))
cv2.waitKey(0)
# Detect faces in the first image
img_modi = face_recognition.load_image_file(image_path_1)
img_modi_rgb = cv2.cvtColor(img_modi, cv2.COLOR_BGR2RGB)
faces = face_recognition.face_locations(img_modi_rgb)
if len(faces) == 0:
    print("No face detected in the first image!")
    exit()
face = faces[0]
copy = img_modi_rgb.copy()
cv2.rectangle(copy, (face[3], face[0]), (face[1], face[2]), (255, 0, 255), 2)
# Show detected face (resized)
cv2.imshow('Detected Face', resize_image(copy))
cv2.waitKey(0)
# Face recognition and comparison
train_encode = face_recognition.face_encodings(img_modi_rgb)[0]
test = face_recognition.load_image_file(image_path_2)
test_rgb = cv2.cvtColor(test, cv2.COLOR_BGR2RGB)
faces_test = face_recognition.face_locations(test_rgb)
if len(faces_test) == 0:
    print("No face detected in the second image!")
    exit()
test_encode = face_recognition.face_encodings(test_rgb)[0]
# Compare faces
```

```
match_result = face_recognition.compare_faces([train_encode], test_encode)
print("Do the faces match?", match_result[0])
# Draw rectangle on detected face and show (resized)
cv2.rectangle(img_modi_rgb, (face[3], face[0]), (face[1], face[2]), (255, 0, 255), 2)
cv2.imshow('I am Iron-Man', resize_image(img_modi_rgb))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:




```
In [1]: runfile('C:/Users/DELL/Desktop/practicals/sem2/CV Practicals/practical4/practical4d/practical
4d.py', wdir='C:/Users/DELL/Desktop/practicals/sem2/CV Practicals/practical4/practical4d')
Do the faces match? True
```

PRACTICAL – 5

Aim: Implement object detection and tracking from video.

Code:

```
import cv2
import numpy as np
from object_detection import ObjectDetection
import math
# Initialize Object Detection
od = ObjectDetection()
cap = cv2.VideoCapture("C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical5/practical5a/los_angeles.mp4")
# Initialize count
count = 0
center_points_prev_frame = []
tracking_objects = {}
track_id = 0
while True:
    ret, frame = cap.read()
    count += 1
    if not ret:
        break
    # Convert frame to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Apply thresholding to create a binary mask
    _, mask = cv2.threshold(gray, 50, 255, cv2.THRESH_BINARY)
    # Point current frame
    center_points_cur_frame = []
    # Detect objects on frame
    (class_ids, scores, boxes) = od.detect(frame)
    for box in boxes:
        (x, y, w, h) = box
        cx = int((x + x + w) / 2)
        cy = int((y + y + h) / 2)
        center_points_cur_frame.append((cx, cy))
        # Draw bounding boxes
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
    # Only at the beginning we compare previous and current frame
    if count <= 2:
        for pt in center_points_cur_frame:
            for pt2 in center_points_prev_frame:
                distance = math.hypot(pt2[0] - pt[0], pt2[1] - pt[1])
                if distance < 20:
                    tracking_objects[track_id] = pt
                    track_id += 1
    else:
        tracking_objects_copy = tracking_objects.copy()
        center_points_cur_frame_copy = center_points_cur_frame.copy()
        for object_id, pt2 in tracking_objects_copy.items():
            object_exists = False
```

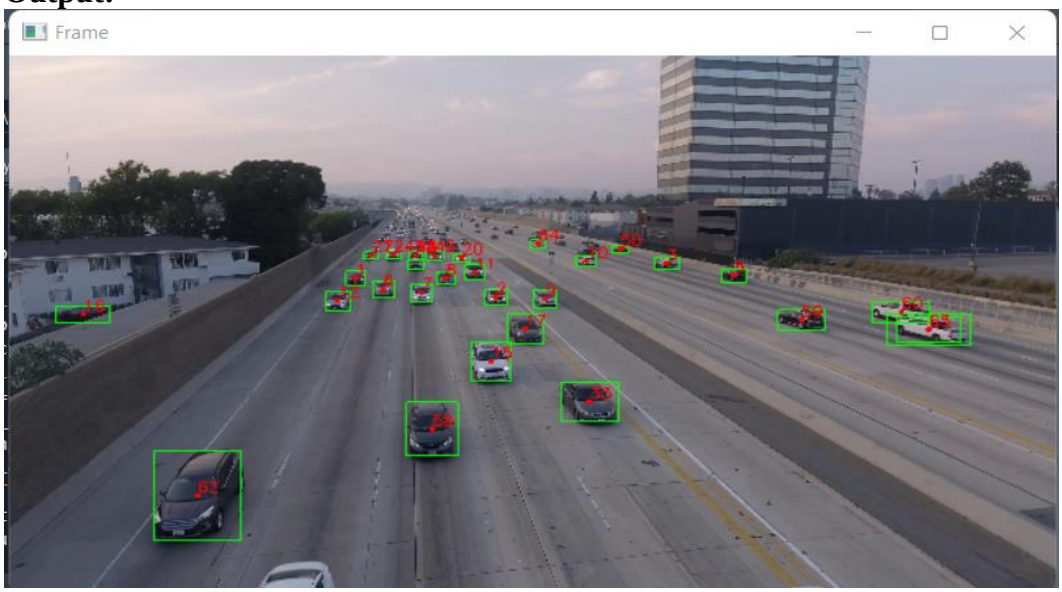
```
            for pt in center_points_cur_frame_copy:
                distance = math.hypot(pt2[0] - pt[0], pt2[1] - pt[1])
                # Update IDs position
                if distance < 20:
                    tracking_objects[object_id] = pt
                    object_exists = True
                    if pt in center_points_cur_frame:
                        center_points_cur_frame.remove(pt)
                    continue
            # Remove IDs lost
            if not object_exists:
                tracking_objects.pop(object_id)
        # Add new IDs found
        for pt in center_points_cur_frame:
            tracking_objects[track_id] = pt
            track_id += 1
    for object_id, pt in tracking_objects.items():
        cv2.circle(frame, pt, 5, (0, 0, 255), -1)
        cv2.putText(frame, str(object_id), (pt[0], pt[1] - 7), 0, 1, (0, 0, 255), 2)
    print("Tracking objects")
    print(tracking_objects)
    print("CUR FRAME LEFT PTS")
    print(center_points_cur_frame)
    # Resize frames before displaying
    frame_resized = cv2.resize(frame, (640, 360))  # Adjust the resolution as needed
    mask_resized = cv2.resize(mask, (300, 200))  # Adjust the resolution as needed
    cv2.imshow("Frame", frame_resized)
    cv2.imshow("Mask", mask_resized)
    # Make a copy of the points
    center_points_prev_frame = center_points_cur_frame.copy()
    key = cv2.waitKey(1)
    if key == 27:
        break
cap.release()
cv2.destroyAllWindows()
```

object_detection.py

```
import cv2
import numpy as np
class ObjectDetection:
    def __init__(self, weights_path="C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical5/practical5a/yolov3.weights",
cfg_path="C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical5/practical5a/yolov3.cfg"):
        print("Loading Object Detection")
        print("Running opencv dnn with YOLOv3")
        self.nmsThreshold = 0.4
        self.confThreshold = 0.5
        self.image_size = 608
        # Load Network
```

```
        net = cv2.dnn.readNet(weights_path, cfg_path)
        # Enable GPU CUDA
        net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
        net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
        self.model = cv2.dnn_DetectionModel(net)
        self.classes = []
        self.load_class_names()
        self.colors = np.random.uniform(0, 255, size=(80, 3))
        self.model.setInputParams(size=(self.image_size, self.image_size), scale=1/255)
    def load_class_names(self, classes_path="C:/Users/DELL/Desktop/practicals/sem2/CV
Practicals/practical5/practical5a/classes.txt"):
        with open(classes_path, "r") as file_object:
            for class_name in file_object.readlines():
                class_name = class_name.strip()
                self.classes.append(class_name)
        self.colors = np.random.uniform(0, 255, size=(80, 3))
        return self.classes
    def detect(self, frame):
        return self.model.detect(frame, nmsThreshold=self.nmsThreshold,
confThreshold=self.confThreshold)
```

Output:




```
In [1]: runfile('C:/Users/DELL/Desktop/practicals/sem2/CV Practicals/practical5/practical5a/practical
5a new.py', wdir='C:/Users/DELL/Desktop/practicals/sem2/CV Practicals/practical5/practical5a')
Loading Object Detection
Running opencv dnn with YOLOv3
Tracking objects
{}
CUR FRAME LEFT PTS
[(440, 742), (881, 474), (643, 434), (943, 459), (1116, 438), (1268, 435), (794, 436), (613, 473),
(687, 449), (754, 461), (1878, 590), (1426, 466), (766, 649), (843, 423), (744, 415), (135, 520),
(930, 532), (1000, 393), (859, 570), (1754, 636), (669, 392), (704, 394), (741, 405), (822, 400),
(1100, 389), (568, 890), (1157, 398), (1347, 978), (776, 394)]
Tracking objects
{0: (434, 746), 1: (642, 435), 2: (882, 475), 3: (1265, 434), 4: (764, 655), 5: (1421, 464), 6: (687,
450), 7: (754, 462), 8: (794, 436), 9: (943, 460), 10: (1114, 435), 11: (843, 423), 12: (612, 473),
13: (744, 415), 14: (744, 415), 15: (860, 566), 16: (135, 520), 17: (930, 533), 18: (998, 392), 19:
(1864, 587), 20: (822, 400), 21: (669, 391), 22: (741, 404), 23: (741, 404), 24: (704, 394), 25:
(1153, 396), 26: (1350, 982), 27: (665, 399), 28: (776, 394), 29: (1100, 390)}
CUR FRAME LEFT PTS
```

PRACTICAL – 6

Aim: Perform Colorization.

Code:

```
import numpy as np
import cv2
from cv2 import dnn

proto_file = 'C:/Users/DELL/Downloads/colorization_deploy_v2.prototxt'
model_file = 'C:/Users/DELL/Downloads/colorization_release_v2.caffemodel'
hull_pts = 'C:/Users/DELL/Downloads/pts_in_hull.npy'
img_path = 'C:/Users/DELL/Downloads/goku_pika.webp'

net = dnn.readNetFromCaffe(proto_file, model_file)
kernel = np.load(hull_pts)
img = cv2.imread(img_path)
scaled = img.astype("float32") / 255.0
lab_img = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
class8 = net.getLayerId("class8_ab")
conv8 = net.getLayerId("conv8_313_rh")
pts = kernel.transpose().reshape(2, 313, 1, 1)
net.getLayer(class8).blobs = [pts.astype("float32")]
net.getLayer(conv8).blobs = [np.full((1, 313), 2.606, dtype="float32")]
resized = cv2.resize(lab_img, (224, 224))
L = cv2.split(resized)[0]
L -= 50
net.setInput(cv2.dnn.blobFromImage(L))
ab_channel = net.forward()[0, :, :, :].transpose((1, 2, 0))
ab_channel = cv2.resize(ab_channel, (img.shape[1], img.shape[0]))
L = cv2.split(lab_img)[0]
colorized = np.concatenate((L[:, :, np.newaxis], ab_channel), axis=2)
colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
colorized = np.clip(colorized, 0, 1)
colorized = (255 * colorized).astype("uint8")
img = cv2.resize(img, (250, 500))
colorized = cv2.resize(colorized, (250, 500))
result = cv2.hconcat([img, colorized])

cv2.imshow("Grayscale -> Colour", result)
cv2.waitKey(0)


# Use following link to download proto_file, model_file, hull_pts files
"https://storage.openvinotoolkit.org/repositories/datumaro/models/colorization/"
# Another link "https://github.com/abhilipsaJena/image_colorization-OpenCV/tree/main"
```

Output:

PRACTICAL – 7

Aim: Perform Text Detection and Recognition.

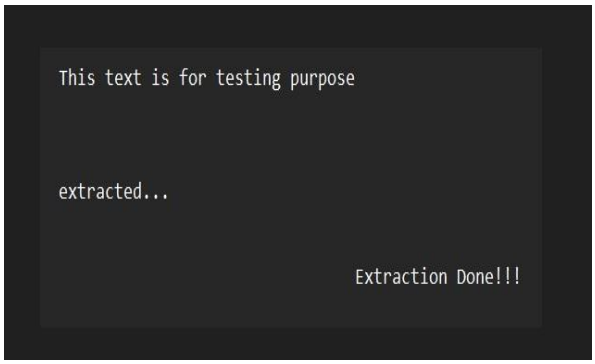
Code:

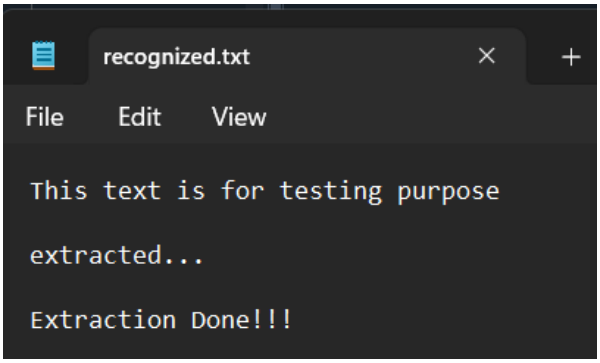
```
import cv2
import pytesseract

pytesseract.pytesseract.tesseract_cmd = 'C:/Program Files/Tesseract-OCR/tesseract.exe'
img = cv2.imread("C:/Users/DELL/Downloads/textforextract.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh1 = cv2.threshold(gray, 0, 255, cv2.THRESH_OTSU |
cv2.THRESH_BINARY_INV)
rect_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (18, 18))
dilation = cv2.dilate(thresh1, rect_kernel, iterations=1)
contours, hierarchy = cv2.findContours(dilation, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
im2 = img.copy()
file = open('C:/Users/DELL/Downloads/recognized.txt', 'w+') #output file location
file.write("")
file.close()
for cnt in contours:
    x, y, w, h = cv2.boundingRect(cnt)
    rect = cv2.rectangle(im2, (x, y), (x + w, y + h), (0, 255, 0), 2)
    cropped = im2[y:y + h, x:x + w]
    file = open('C:/Users/DELL/Downloads/recognized.txt', 'a')  #output file location
    text = pytesseract.image_to_string(cropped)
    file.write(text)
    file.write("\n")
    file.close()

# To download tesseract.exe use following link "https://github.com/UB-
Mannheim/tesseract/wiki"
```

Output:





PRACTICAL – 8

Aim: Construct 3D model from Images.

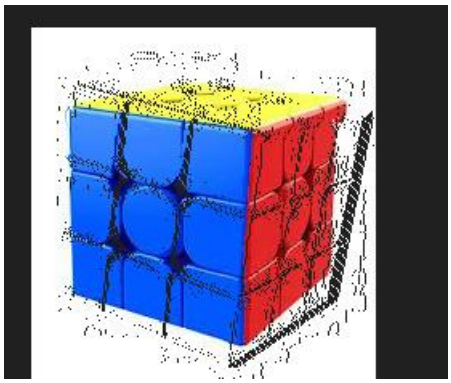
Code:

```
from PIL import Image
import numpy as np
import os

def shift_image(img, depth_img, shift_amount=10):
    img = img.convert("RGBA")
    data = np.array(img)
    depth_img = depth_img.convert("L")
    depth_data = np.array(depth_img)
    deltas = ((depth_data / 255.0) * float(shift_amount)).astype(int)
    shifted_data = np.zeros_like(data)
    height, width, _ = data.shape
    for y, row in enumerate(deltas):
        for x, dx in enumerate(row):
            if x + dx < width and x + dx >= 0:
                shifted_data[y, x + dx] = data[y, x]
    shifted_image = Image.fromarray(shifted_data.astype(np.uint8))
    return shifted_image

img = Image.open("C:/Users/DELL/Downloads/cube1.jpeg")
depth_img = Image.open("C:/Users/DELL/Downloads/cube2.jpeg")
shifted_img = shift_image(img, depth_img, shift_amount=10)
shifted_img.show()
```

Output:



PRACTICAL – 9

Aim: Perform Feature extraction using RANSAC.

Code:

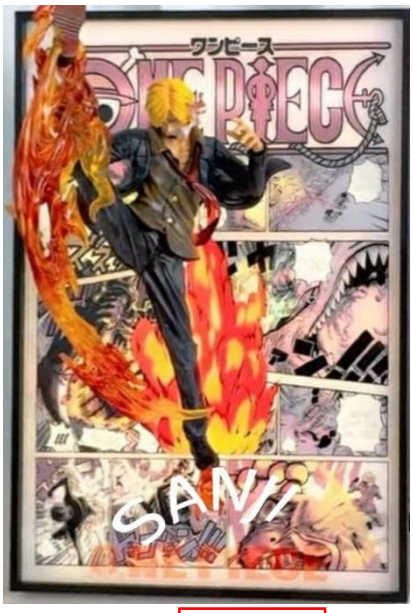
```
import cv2
import numpy as np

img1_color = cv2.imread("C:/Users/DELL/Downloads/wii1.jpeg")
img2_color = cv2.imread("C:/Users/DELL/Downloads/wii2.jpeg")
img1 = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)
height, width = img2.shape
orb_detector = cv2.ORB_create(5000)
kp1, d1 = orb_detector.detectAndCompute(img1, None)
kp2, d2 = orb_detector.detectAndCompute(img2, None)
matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = matcher.match(d1, d2)
matches = sorted(matches, key=lambda x: x.distance)
matches = matches[:int(len(matches) * 0.9)]
no_of_matches = len(matches)
p1 = np.zeros((no_of_matches, 2))
p2 = np.zeros((no_of_matches, 2))
for i in range(len(matches)):
    p1[i, :] = kp1[matches[i].queryIdx].pt
    p2[i, :] = kp2[matches[i].trainIdx].pt
homography, mask = cv2.findHomography(p1, p2, cv2.RANSAC)
transformed_img = cv2.warpPerspective(img1_color, homography, (width, height))
cv2.imwrite('C:/Users/DELL/Downloads/output.jpg', transformed_img)
```

Output:

Input

Reference



Output

PRACTICAL – 10

Aim: Perform Image matting and composition.

Code:

```
import cv2
import numpy as np

image_path = "C:/Users/Admin/Downloads/girl.jpg"
background_path = "C:/Users/Admin/Downloads/home.jpeg"
output_path = "C:/Users/Admin/Downloads/result.jpeg"
def grabcut_matting(image_path, background_path, output_path):
    # Load the input image and background
    img = cv2.imread(image_path)
    bg = cv2.imread(background_path)
    # Check if images are loaded successfully
    if img is None:
        print(f"Error loading image: {image_path}")
        return
    if bg is None:
        print(f"Error loading background: {background_path}")
        return
    # Resize background to match the input image size
    bg = cv2.resize(bg, (img.shape[1], img.shape[0]))
    # Create initial mask
    mask = np.zeros(img.shape[:2], np.uint8)
    # Define a rectangle containing the foreground object (manually adjustable)
    rect = (50, 50, img.shape[1] - 100, img.shape[0] - 100)
    # Allocate memory for models (needed by GrabCut)
    bgdModel = np.zeros((1, 65), np.float64)
    fgdModel = np.zeros((1, 65), np.float64)
    # Apply GrabCut
    cv2.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
    # Prepare the mask for compositing
    mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
    mask3 = cv2.merge([mask2, mask2, mask2])
    # Extract the foreground
    foreground = img * mask3
    cv2.imshow('Foreground', foreground)
    # Extract the background where the mask is 0
    background = bg * (1 - mask3)
    # Combine foreground and new background
    result = cv2.add(foreground, background)
    # Save the result to output path
    cv2.imwrite(output_path, result)
    cv2.imshow('Composited Image', result)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
# Call the function with paths
grabcut_matting(image_path, background_path, output_path)
```

Output: