# Index

| Sr. No | Title | Date | Signature |
|---|---|---|---|
| 1 | Perform Geometric transformation. | | |
| | A. Image Scaling. | | |
| | B. Image Shrinking. | | |
| | C. Image Rotation. | | |
| | D. Affine Transformation. | | |
| | E. Perspective Transformation. | | |
| | F. Shearing X-axis. | | |
| | G. Shearing Y-axis. | | |
| | H. Reflected Image. | | |
| | I. Cropped Image. | | |
| 2 | Perform Image Stitching. | | |
| 3 | Perform Camera Calibration. | | |
| 4 | A. Perform the following Face detection. | | |
| | B. Object detection | | |
| | C. Perform the following Pedestrian detection. | | |
| | D. Perform the following Face Recognition. | | |
| 5 | A. Implement object detection and tracking from video. | | |
| 6 | Perform Colorization. | | |
| 7 | Perform Text Detection and Recognition. | | |
| 8 | Construct 3D model from Images. | | |
| 9 | Perform Feature extraction using RANSAC. | | |
| 10 | Perform Image matting and composition | | |

# PRACTICAL – 1

**Aim:** Perform Geometric transformation.

**Theory:**

**Geometric Transformations:**
Geometric transformation is a fundamental technique used in image processing that involves manipulating the spatial arrangement of pixels in an image. It is used to modify the geometric properties of an image, such as its size, shape, position, and orientation. OpenCV provides two transformation functions, cv2.warpAffine and cv2.warpPerspective, with which you can have all kinds of transformations. cv2.warpAffine takes a 2x3 transformation matrix while cv2.warpPerspective takes a 3x3 transformation matrix as input.

Translation
Translation is the shifting of object's location. If you know the shift in (x,y) direction, let it be $(t_x, t_y)$, you can create the transformation matrix $\mathbf{M}$as follows:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

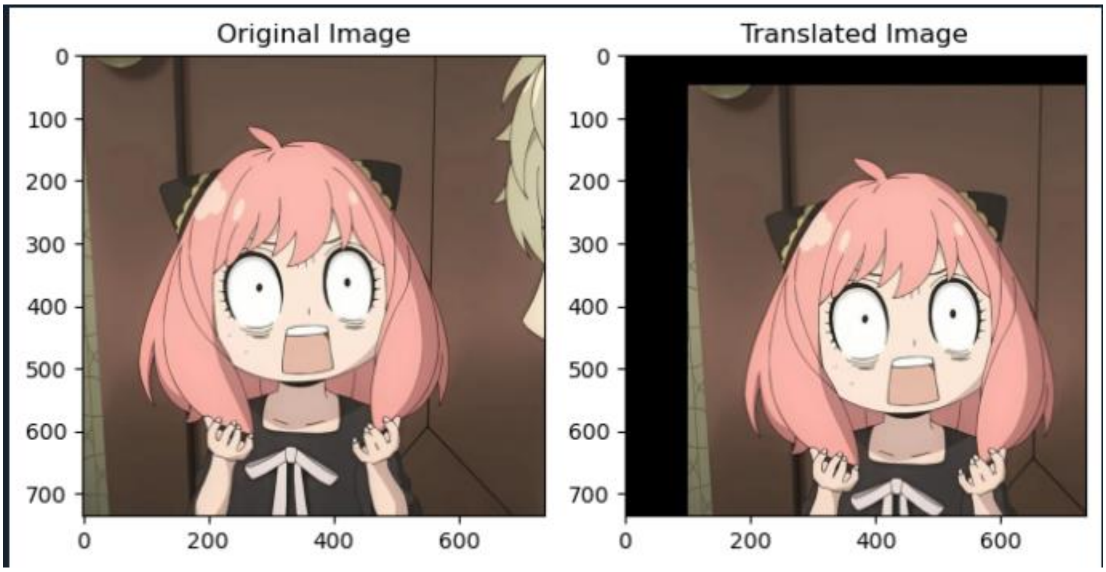
You can take make it into a Numpy array of type np.float32 and pass it into cv2.warpAffine() function. See below example for a shift of (100,50):

**Warning:**
Third argument of the cv2.warpAffine() function is the size of the output image, which should be in the form of (width, height). Remember width = number of columns, and height = number of rows.

## 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:**


Original Image
Translated Image
0
100
200
300
400
500
600
700
0
200
400
600

# PRACTICAL – 1(A)

**Aim:** Image Scaling

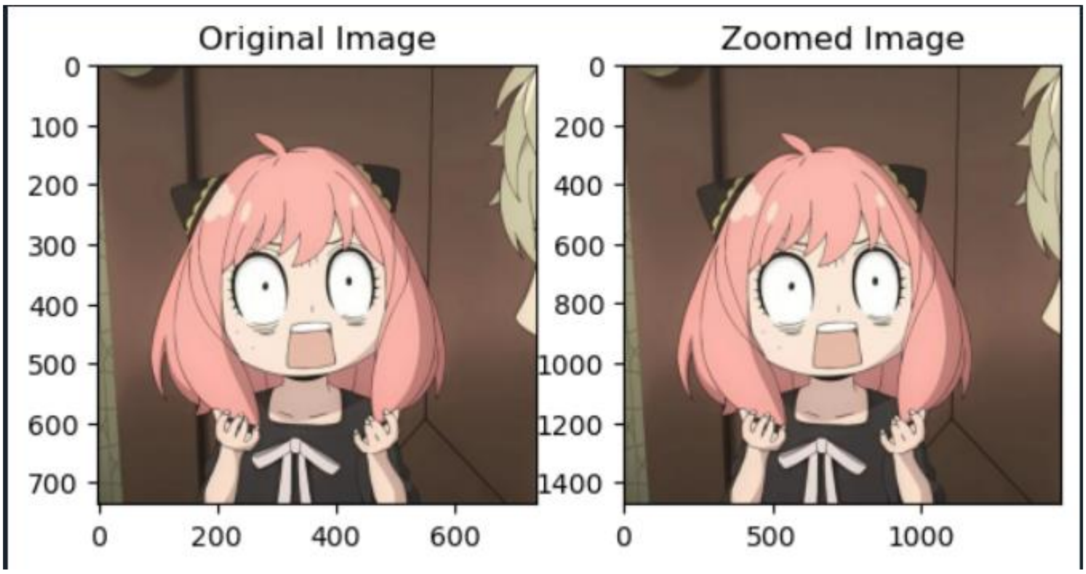
**Theory:**

**Scaling:**
Scaling is just resizing of the image. OpenCV comes with a function cv2.resize() for this purpose. The size of the image can be specified manually, or you can specify the scaling factor. Different interpolation methods are used. Preferable interpolation methods are cv2.INTER_AREA for shrinking and cv2.INTER_CUBIC (slow) & cv2.INTER_LINEAR for zooming. By default, interpolation method used is cv2.INTER_LINEAR for all resizing purposes. You can resize an input image either of following methods:

## 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

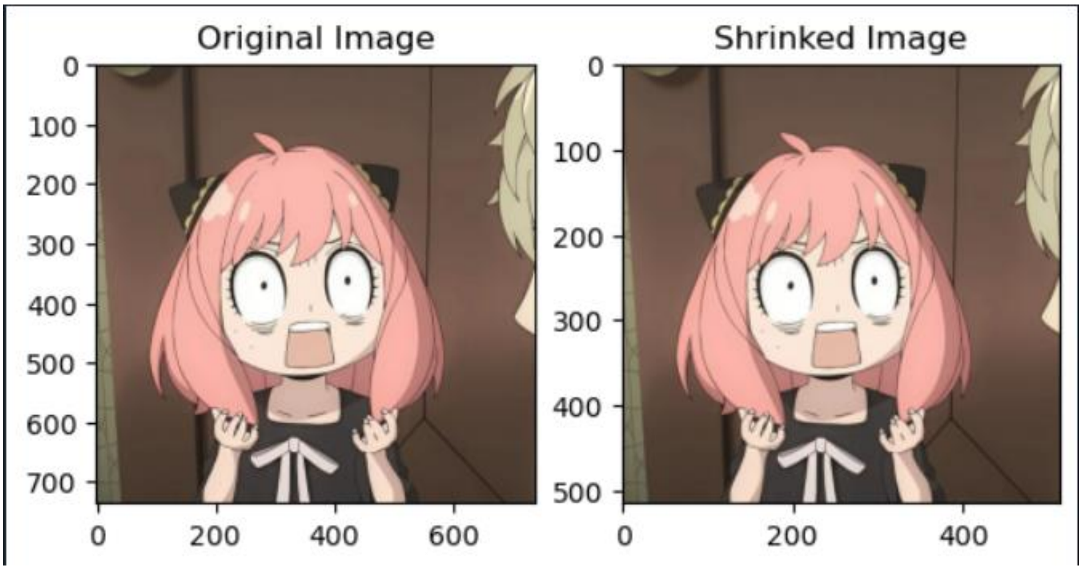
**Theory:**

**Shrinking**
Image shrinking in Python involves reducing the dimensions of an image, effectively making it smaller while maintaining its aspect ratio. This process is commonly done using libraries like PIL (Python Imaging Library) or its fork, Pillow. By resizing the image to smaller dimensions, either by specifying new dimensions or a scaling factor, you can achieve the desired reduction in size.
This is useful for tasks like image compression, thumbnail generation, or preparing images for web display, where smaller file sizes are advantageous. The process typically involves opening the image, calculating the new dimensions, resizing the image accordingly, and then saving the resized image to a new file.

## 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

**Theory:**

**Rotation**

Rotation of an image for an angle $\theta$ is achieved by the transformation matrix of the form

$$M = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

But OpenCV provides scaled rotation with adjustable center of rotation so that you can rotate at any location you prefer. Modified transformation matrix is given by

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha) \cdot center.y \end{bmatrix}$$

where:

$$\alpha = scale \cdot \cos\theta,$$
$$\beta = scale \cdot \sin\theta$$

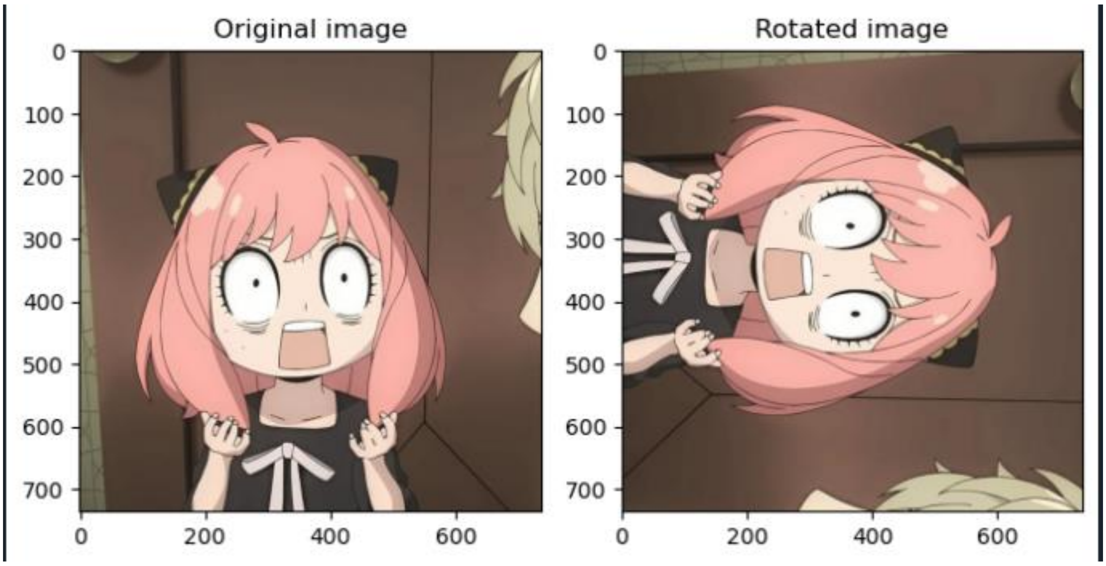
To find this transformation matrix, OpenCV provides a function, **cv2.getRotationMatrix2D**. Check below example which rotates the image by 90 degree with respect to center without any scaling.

## 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
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

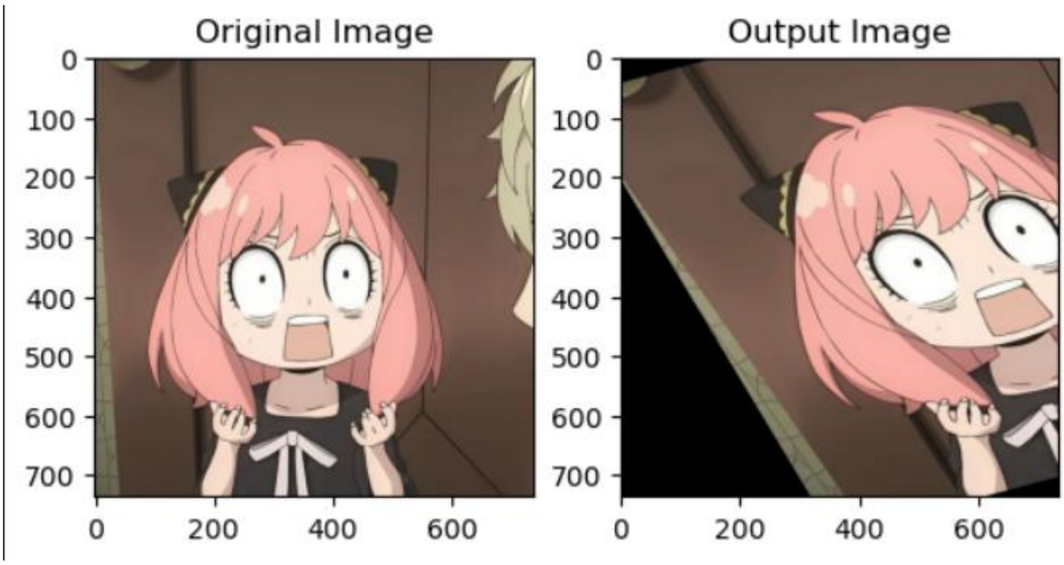
**Theory:**

In **affine** transformation, all parallel lines in the original image will still be parallel in the output image. To find the transformation matrix, we need three points from input image and their corresponding locations in output image. Then **cv2.getAffineTransform** will create a 2x3 matrix which is to be passed to **cv2.warpAffine**.

## 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.

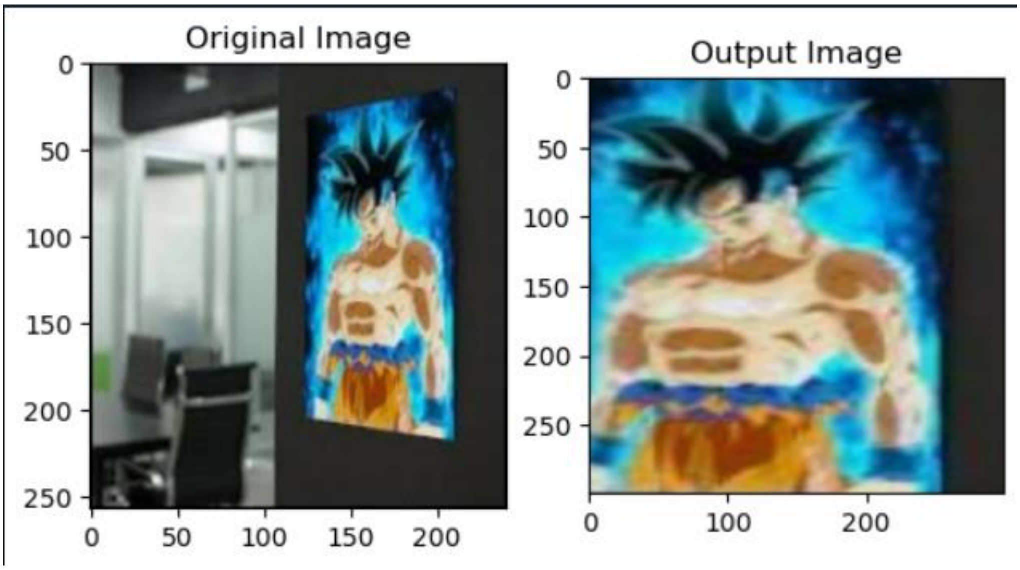
**Theory:**

For perspective transformation, you need a 3x3 transformation matrix. Straight lines will remain straight even after the transformation. To find this transformation matrix, you need 4 points on the input image and corresponding points on the output image. Among these 4 points, 3 of them should not be collinear. Then transformation matrix can be found by the function **cv2.getPerspectiveTransform**. Then apply **cv2.warpPerspective** with this 3x3 transformation matrix.

## 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.

**Theory:**

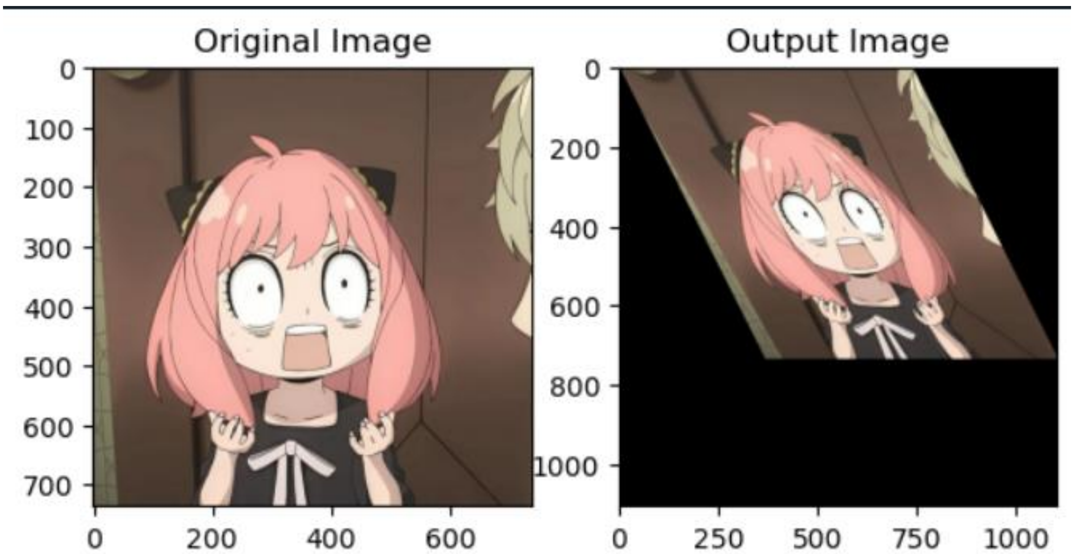
**Shearing:**
deals with changing the shape and size of the 2D object along x-axis and y- axis. It is similar to sliding the layers in one direction to change the shape of the 2D object.It is an ideal technique to change the shape of an existing object in a two dimensional plane. In a two dimensional plane, the object size can be changed along X direction as well as Y direction.

**X- Shear:**
In x shear, the y co-ordinates remain the same but the x co-ordinates changes.

## 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.

**Theory:**

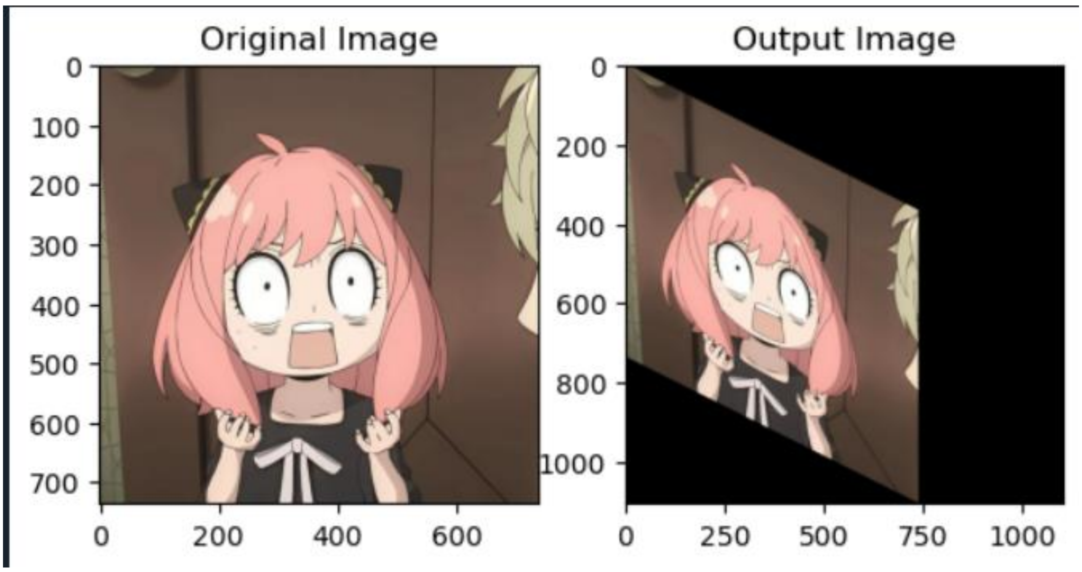
**Shearing:**
deals with changing the shape and size of the 2D object along x-axis and y- axis. It is similar to sliding the layers in one direction to change the shape of the 2D object.It is an ideal technique to change the shape of an existing object in a two dimensional plane. In a two dimensional plane, the object size can be changed along X direction as well as Y direction.

**Y-Shear:**
In y shear, the x co-ordinates remain the same but the y co-ordinates changes.

## 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.

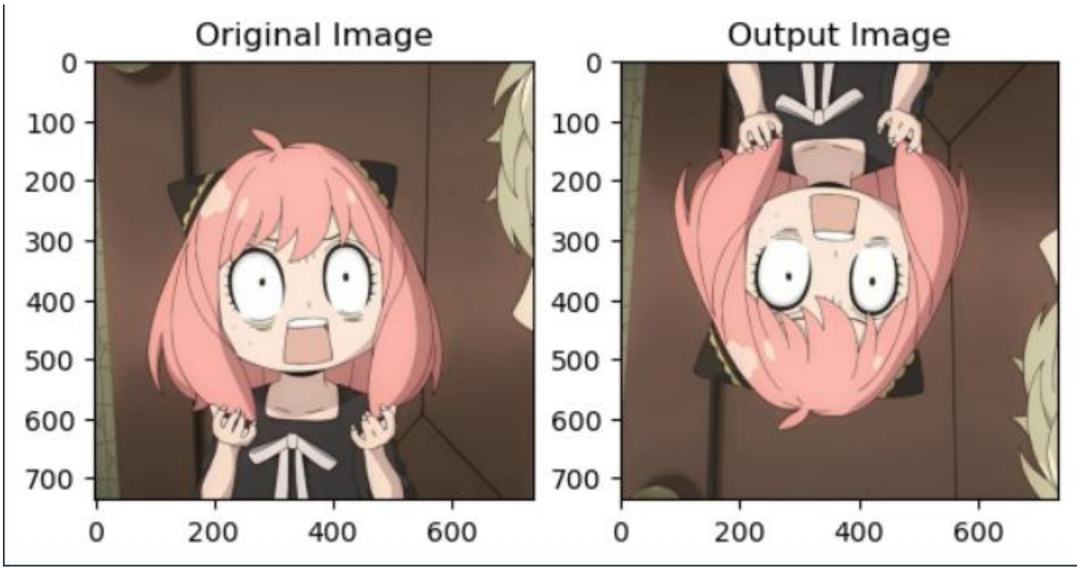
**Theory:**

**Image Reflection:**
Image reflection is used to flip the image vertically or horizontally. For reflection along the x-axis, we set the value of Sy to -1, Sx to 1, and vice-versa for the y-axis reflection.

## 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.

**Theory:**

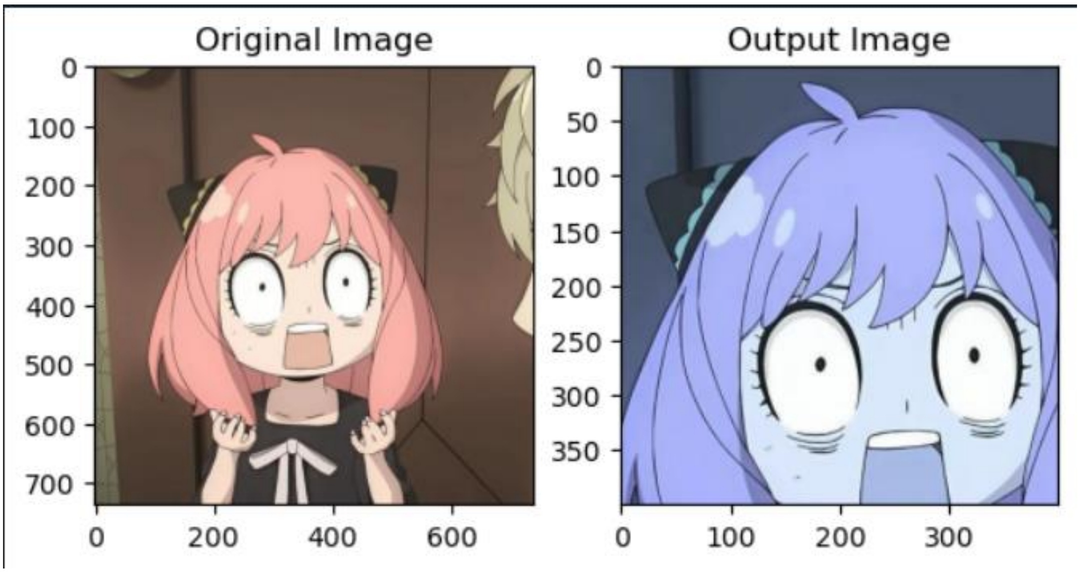
**Image Cropping**

Cropping is the removal of unwanted outer areas from an 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.

**Theory:**

**Image stitching** is the process of combining multiple overlapping images to create a seamless, high-resolution output image. This technique is commonly used to create panoramic images, virtual tours, and even some medical imaging applications.

Image stitching involves several steps:

1. **Feature detection**: Identifying and extracting unique features (e.g., corners, edges) from each input image. Compute the SIFT-key points and descriptors for both the images.
2. **Feature matching**: Finding correspondences between features in the overlapping regions of the input images. Compute distances between every descriptor in one image and every descriptor in the other image. Select the top ‘m’ matches for each descriptor of an image.
3. **Homography estimation**: Estimating the transformation (e.g., rotation, scaling, translation) that aligns the input images. Run RANSAC to estimate homography
4. **Warping**: Applying the estimated transformation to the input images. Warp to align for stitching
5. **Blending**: Combining the warped images into a single seamless output image. Now stitch them together

**Explanation of Code:**

Firstly, we have to find out the features matching in both the images. These best matched features act as the basis for stitching. We extract the key points and sift descriptors for both the images as follows:

```
sift = cv2.SIFT_create()
# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
```

kp1 and kp2 are keypoints, des1 and des2 are the descriptors of the respective images. Now, the obtained descriptors in one image are to be recognized in the image too. We do that as follows:

```
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)
```

The BFMatcher() matches the features which are more similar. When we set parameter k=2, we are asking the knnMatcher to give out 2 best matches for each descriptor.

‘**matches**’ is a list of list, where each sub-list consists of ‘k’ objects. Often in images, there are tremendous chances where the features may be existing in many places of the image. This may mislead us to use trivial features for our experiment. So we filter out through all the matches to obtain the best ones. So we apply ratio test using the top 2 matches obtained above. We consider a match if the ratio defined below is predominantly greater than the specified ratio.

```
# Apply ratio test
good = []
for m in matches:
if m[0].distance < 0.5*m[1].distance:
good.append(m)
matches = np.asarray(good)
```

It’s time to align the images now. As you know that a homography matrix is needed to perform the transformation, and the homography matrix requires at least 4 matches, we do the following.

```
if len(matches[:,0]) >= 4:
src = np.float32([ kp1[m.queryIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)
dst = np.float32([ kp2[m.trainIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)H,
masked = cv2.findHomography(src, dst, cv2.RANSAC, 5.0)
#print H
else:
raise AssertionError(“Can’t find enough keypoints.”)
```

And finally comes the last part, stitching of the images. Now that we found the homography for transformation, we can now proceed to warp and stitch them together:

```
dst = cv2.warpPerspective(img_,H,(img.shape[1] + img_.shape[1],
img.shape[0]))
plt.subplot(122),plt.imshow(dst),plt.title(‘Warped Image’)
plt.show()
plt.figure()
dst[0:img.shape[0], 0:img.shape[1]] = img
cv2.imwrite(‘output.jpg’,dst)
plt.imshow(dst)
plt.show()
```

We get warped image plotted using matplotlib to well visualize the warping.

## 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.

**Theory:**
A camera is an integral part of several domains like robotics, space exploration, etc camera is playing a major role. It helps to capture each and every moment and helpful for many analyses. In order to use the camera as a visual sensor, we should know the parameters of the camera. **Camera Calibration** is nothing but estimating the parameters of a camera, parameters about the camera are required to determine an accurate relationship between a 3D point in the real world and its corresponding 2D projection (pixel) in the image captured by that calibrated camera.
We need to consider both internal parameters like focal length, optical center, and radial distortion coefficients of the lens etc., and external parameters like rotation and translation of the camera with respect to some real world coordinate system.

**Camera Calibration can be done in a step-by-step approach:**
- **Step 1:** First define real world coordinates of 3D points using known size of checkerboard pattern.
- **Step 2:** Different viewpoints of check-board image is captured.

- **Step 3:** findChessboardCorners() is a method in OpenCV and used to find pixel coordinates (u, v) for each 3D point in different images
- **Step 4:** Then calibrateCamera() method is used to find camera parameters.

It will take our calculated (threedpoints, twodpoints, grayColor.shape[::-1], None, None) as parameters and returns list having elements as Camera matrix, Distortion coefficient, Rotation Vectors, and Translation Vectors.
Camera Matrix helps to transform 3D objects points to 2D image points and the Distortion Coefficient returns the position of the camera in the world, with the values of Rotation and Translation vectors

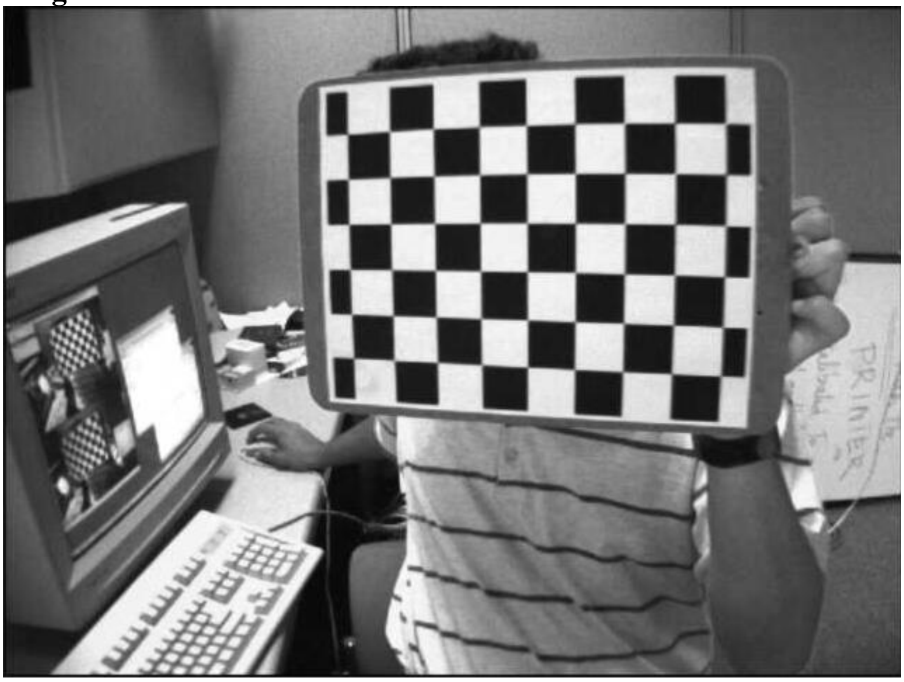
## Code:

```
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.

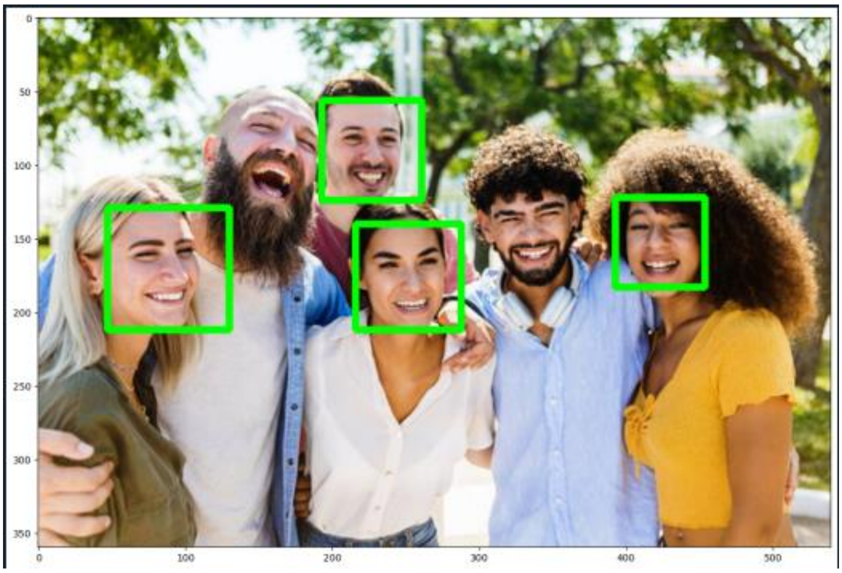
**Theory:**
**Face detection** involves identifying a person's face in an image or video. This is done by analyzing the visual input to determine whether a person's facial features are present.
Since human faces are so diverse, face detection models typically need to be trained on large amounts of input data for them to be accurate. The training dataset must contain a sufficient representation of people who come from different backgrounds, genders, and cultures.
These algorithms also need to be fed many training samples comprising different lighting, angles, and orientations to make correct predictions in real-world scenarios.
These nuances make face detection a non-trivial, time-consuming task that requires hours of model training and millions of data samples.
The OpenCV package comes with pre-trained models for face detection, which means that we don't have to train an algorithm from scratch. More specifically, the library employs a machine learning approach called Haar cascade to identify objects in visual data.
Face detection approach called Haar Cascade for face detection using OpenCV and Python.
**Intro to Haar Cascade Classifiers**
This method was first introduced in the paper Rapid Object Detection Using a Boosted Cascade of Simple Features, written by Paul Viola and Michael Jones.
The idea behind this technique involves using a cascade of classifiers to detect different features in an image. These classifiers are then combined into one strong classifier that can accurately distinguish between samples that contain a human face from those that don't.
The Haar Cascade classifier that is built into OpenCV has already been trained on a large dataset of human faces, so no further training is required. We just need to load the classifier from the library and use it to perform face detection on an input image.
**Installing OpenCV for Python**
To install the OpenCV library, simply open your command prompt or terminal window and run the following command:
pip install opencv-python
**OpenCV for Face Detection in Images**We will build a detector to identify the human face in a photo. Make sure to save the picture to your working directory and rename it to input_image before coding along.

## 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.

**Theory:**

**Object Detection**

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos.

**Haar cascade:**

Basically, the Haar cascade technique is an approach based on machine learning where we use a lot of positive and negative images to train the classifier to classify between the images. Haar cascade classifiers are considered as the effective way to do object detection with the OpenCV library.

**Positive images:** These are the images that contain the objects which we want to be identified from the classifier.

**Negative Images:** These are the images that do not contain any object that we want to be detected by the classifier, and these can be images of everything else.

**Requirements for object detection with Python OpenCV:**

1. Install OpenCV-Python Library
2. Install matplotlib library
3. An Image with Stop Sign.
4. An xml file “Stop_data.xml” to detect stop sign board.

## 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

**Theory:**

**Pedestrian detection**

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 HOG(Histogram of Oriented Gradients) + Linear SVM model to detect pedestrians in images and video streams.

**Histogram of Oriented Gradients**

This algorithm checks directly surrounding pixels of every single pixel. The goal is to check how darker is the current pixel compared to the surrounding pixels. The algorithm draws and arrows showing the direction of the image getting darker. It repeats the process for each and every pixel in the image. At last, every pixel would be replaced by an arrow, these arrows are called Gradients. These gradients show the flow of light from light to dark. By using these gradients algorithms perform further analysis.

**Requirements**

1. opencv-python
2. imutils

## 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.

**Theory:**
Face recognition is different from face detection. In face detection, we had only detected the location of human faces, and we recognized the identity of faces in the face recognition task. In this article, we are going to build a face recognition system using python with the help of face recognition library.
There are many algorithms available in the market for face recognition. This broad computer vision challenge is detecting faces from videos and pictures. Many applications can be built on top of recognition systems. Many big companies are adopting recognition systems for their security and authentication purposes.
**Use Cases of Recognition Systems**
Face recognition systems are widely used in the modern era, and many new innovative systems are built on top of recognition systems.

**There are a few used cases :**
- Finding Missing Person
- Identifying accounts on social media
- Recognizing Drivers in Cars
- School Attendance System

Several methods and algorithms implement facial recognition systems depending on the performance and accuracy.
**Traditional Face Recognition Algorithm**
Traditional face recognition algorithms don't meet modern-day's facial recognition standards. They were designed to recognize faces using old conventional algorithms.
OpenCV provides some traditional facial Recognition Algorithms.
- **Eigenfaces**
- **Scale Invariant Feature Transform (SIFT)**
- **Fisher faces**
- **Local Binary Patterns Histograms (LBPH)**

These methods differ in the way they extract image information and match input and output images.
LBPH algorithm is a simple yet very efficient method still in use but it's slow compared to modern days algorithms.
**Deep Learning For Face Recognition**
There are various deep learning-based facial recognition algorithms available.
- DeepFace
- DeepID series of systems,
- FaceNet
- VGGFace

Generally, face recognizers that are based on landmarks take face images and try to find essential feature points such as eyebrows, corners of the mouth, eyes, nose, lips, etc. There are more than 60 points.
**Steps Involved in Face Recognition**
1. **Face Detection**: Locate the face, note the coordinates of each face located and draw a bounding box around every faces.

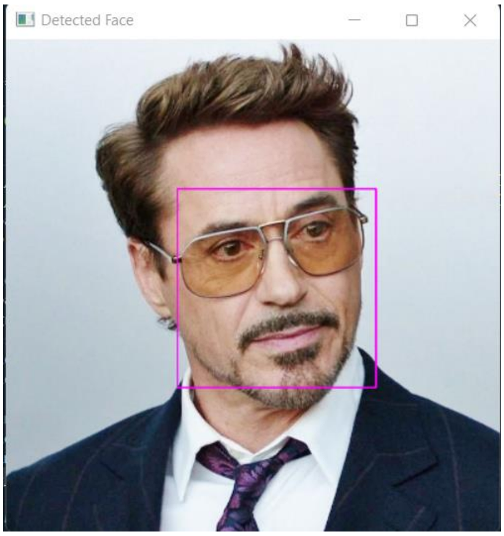
2. **Face Alignments**. Normalize the faces in order to attain fast training.
3. **Feature Extraction**. Local feature extraction from facial pictures for training, this step is performed differently by different algorithms.
4. **Face Recognition**. Match the input face with one or more known faces in our dataset.

## Code:

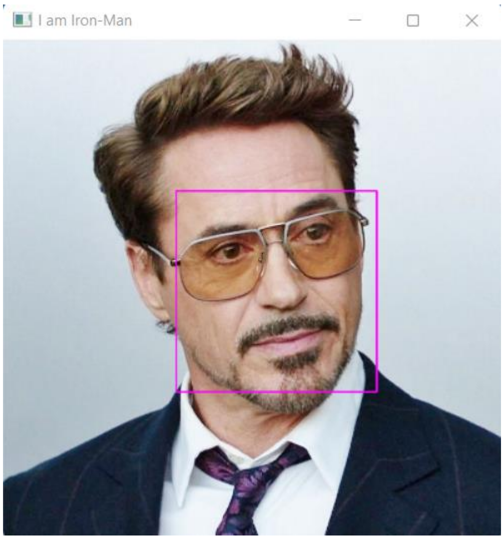
```
import numpy as np
import face_recognition
import os
# Resize helper function
def resize_image(image, scale=0.5):
    width = int(image.shape[1] * scale)
    height = int(image.shape[0] * scale)
    return cv2.resize(image, (width, height))
# Load and check if image exists
image_path_1 = "C:/Users/DELL/Desktop/practicals/sem2/CV
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.

**Theory:**
**Object detection** is the detection on every single frame and frame after frame.
**Object tracking** does frame-by-frame tracking but keeps the history of where the object is at a time after time

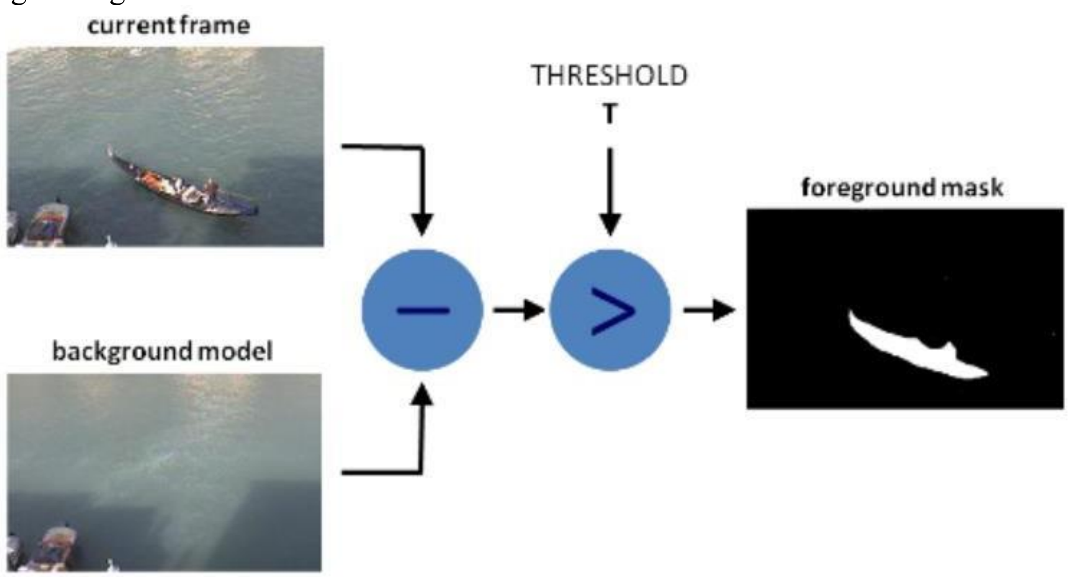
1. **Importing Libraries and Modules:**
   import cv2:
   Imports the OpenCV library used for computer vision tasks. from

   tracker import *:

   Imports all functions and classes from a tracker module, which likely contains the implementation of the EuclideanDistTracker.
2. **Creating Objects and Initializing Video Capture:**
   tracker = EuclideanDistTracker():
   Instantiates the Euclidean Distance Tracker. cap =

   cv2.VideoCapture("highway.mp4"):

   Initializes video capture with the video file "highway.mp4".
3. **Object Detection Initialization:**
   object_detector = cv2.createBackgroundSubtractorMOG2(history=100, varThreshold=40):
   Initializes a background subtractor with MOG2 method to differentiate between foreground (moving objects) and the background.
   Background subtraction (BS) is a common and widely used technique for generating a foreground mask (namely, a binary image containing the pixels belonging to moving objects in the scene) by using static cameras.
   As the name suggests, BS calculates the foreground mask performing a subtraction between the current frame and a background model, containing the static part of the scene or, more in general, everything that can be considered as background given the characteristics of the observed scene.




4. **Processing Video Frames**: The while True loop starts an infinite loop to process video frames until manually stopped.

ret, frame = cap.read(): Reads the next frame from the video.
height, width, _ = frame.shape: Retrieves the dimensions of the frame.

5. **Defining Region of Interest (ROI):**
roi = frame[340: 720,500: 800]: Defines a specific area in the video frame to focus the object detection on. This reduces computation and ignores irrelevant areas.
6. **Object Detection:**
mask = object_detector.apply(roi): Applies the background subtractor to the ROI to get the foreground mask.
_, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY): Applies a threshold to the mask to make it binary, which helps in identifying distinct objects.contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE):
Finds the contours of the detected objects in the binary mask.
7. **Filtering and Storing Detections:**
It iterates through each contour, calculates its area, and if the area is larger than a threshold (100), it calculates a bounding box for the object.These bounding boxes ([x, y, w, h]) are added to the detections list representing detected objects.
8. Object Tracking:boxes_ids = tracker.update(detections): The tracker updates with the current frame's detections and returns the tracked objects with their IDs.The loop then iterates through these tracked objects, drawing their ID and bounding box on the ROI.
9. **Displaying Results:cv2.imshow():**
Displays the ROI, the original frame, and the binary mask in separate windows.key = cv2.waitKey(30): Waits for a key press for 30 ms and breaks the loop if the 'Esc' key is pressed.
10. **Cleanup:cap.release() and cv2.destroyAllWindows():**
Releases the video capture and closes all OpenCV windows.

In summary, this script is a complete system for object detection and tracking in a video. It uses MOG2 for background subtraction to detect moving objects and tracks them using a Euclidean Distance Tracker. It processes and visualizes the results in real-time until the user exits by pressing the 'Esc' key.

## 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()
```

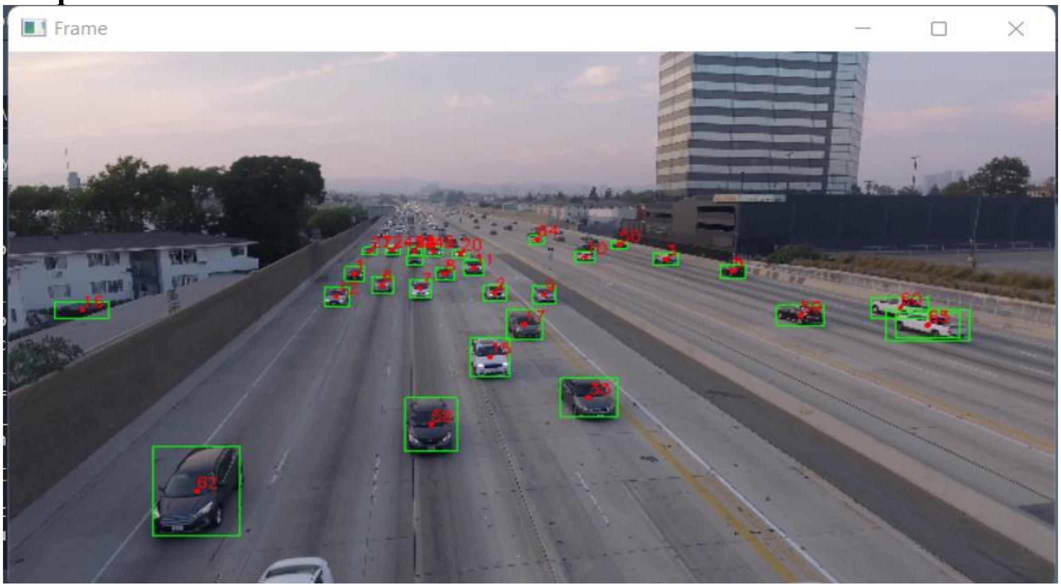
```
count += 1
if not ret:
    break
# Convert frame to grayscale
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
# Apply thresholding to create a binary mask
_, mask = cv2.threshold(gray, 50, 255, cv2.THRESH_BINARY)
# Point current frame
center_points_cur_frame = []
# Detect objects on frame
(class_ids, 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.

**Theory:**
We'll create a program to convert a black & white image i.e grayscale image to a colour image. We're 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.
The procedure that we'll follow to implement the program:
**Steps:**

1. Load the model and the **convolution/kernel points**
2. Read and preprocess the image
3. Generate model predictions using the **L channel** from our input image
4. Use the output -> **ab channel** to create a resulting image

*What is the L channel and ab channel?* Basically like **RGB** colour space, there is something similar, known as **Lab colour space**. And this is the basis on which our program is based. Let's discuss what it is briefly:
**What is Lab Colour Space?**
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'll 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.

Automatic colorization of photos using deep neural networks is a technology that can add color to black and white photos without the need for manual coloring. This technology uses deep neural networks that have been trained on large datasets of color images to learn the relationship between luminance and color, which can then be used to predict the color channels of grayscale images. This technique has many applications, including restoring old photos and enhancing the visual appeal of images.

**Background**

Before the advent of computer technology, adding color to a black-and-white photograph was a manual and time-consuming process that required skilled artists. With the introduction of automated colorization methods, the process became quicker but often inaccurate and still required manual intervention. Deep neural networks are a type of machine learning algorithm inspired by the human brain, and they have made it possible to automatically colorize black-and-white photographs with high accuracy and speed. These networks are trained on large datasets of color images and use what they learned to generate plausible colorizations for grayscale images. Using deep neural networks for automatic colorization has many practical applications, such as restoring old photographs, enhancing medical images, and creating realistic 3D models from 2D images.

**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.

**Theory:**
**Required Installations:**
pip install opencv-python

pip install pytesseract

```
Command Prompt

Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rasik>pip install pytesseract
Collecting pytesseract
  Downloading pytesseract-0.3.10-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: packaging>=21.3 in c:\users\rasik\appdata\local\programs\python\python311\lib\site-packag
es (from pytesseract) (23.2)
Requirement already satisfied: Pillow>=8.0.0 in c:\users\rasik\appdata\local\programs\python\python311\lib\site-packages
 (from pytesseract) (10.1.0)
Downloading pytesseract-0.3.10-py3-none-any.whl (14 kB)
Installing collected packages: pytesseract
Successfully installed pytesseract-0.3.10

C:\Users\rasik>
```

Follow the instructions given in below Link:

https://builtin.com/articles/python-tesseract

OpenCV package is used to read an image and perform certain image processing techniques. Python-tesseract is a wrapper for Google's Tesseract-OCR Engine which is used to recognize text from images.
Python-tesseract is an optical character recognition (OCR) tool for python. That is, it will recognize and "read" the text embedded in images.
Python-tesseract is a wrapper for Google's Tesseract-OCR Engine. It is also useful as a stand-alone invocation script to tesseract, as it can read all image types supported by the Pillow and Leptonica imaging libraries, including jpeg, png, gif, bmp, tiff, and others. Additionally, if used as a script, Python-tesseract will print the recognized text instead of writing it to a file.

**Approach:**
After the necessary imports, a sample image is read using the imread function of opencv.

**Applying image processing for the image:**
The colorspace of the image is first changed and stored in a variable. For color conversion we use the function cv2.cvtColor(input_image, flag). The second parameter flag determines the type of conversion. We can chose among cv2.COLOR_BGR2GRAY and cv2.COLOR_BGR2HSV. cv2.COLOR_BGR2GRAY helps us to convert an RGB image to gray scale image and cv2.COLOR_BGR2HSV is used to convert an RGB image to HSV (Hue, Saturation, Value) color-space image. Here, we use cv2.COLOR_BGR2GRAY. A threshold is applied to the converted image using cv2.threshold function.
**There are 3 types of thresholding:**

1. Simple Thresholding
2. Adaptive Thresholding
3. Otsu's Binarization

For more information on thresholding, refer Thresholding techniques using OpenCV.
cv2.threshold() has 4 parameters, first parameter being the color-space changed image,

followed by the minimum threshold value, the maximum threshold value and the type of thresholding that needs to be applied.

**To get a rectangular structure:**

cv2.getStructuringElement() is used to define a structural element like elliptical, circular, rectangular etc. Here, we use the rectangular structural element (cv2.MORPH_RECT). cv2.getStructuringElement takes an extra size of the kernel parameter. A bigger kernel would make group larger blocks of texts together. After choosing the correct kernel, dilation is applied to the image with cv2.dilate function. Dilation makes the groups of text to be detected more accurately since it dilates (expands) a text block.

**Finding Contours:**

cv2.findContours() is used to find contours in the dilated image. There are three arguments in cv.findContours(): the source image, the contour retrieval mode and the contour approximation method.

This function returns contours and hierarchy. Contours is a python list of all the contours in the image. Each contour is a Numpy array of (x, y) coordinates of boundary points in the object. Contours are typically used to find a white object from a black background. All the above image processing techniques are applied so that the Contours can detect the boundary edges of the blocks of text of the image. A text file is opened in write mode and flushed. This text file is opened to save the text from the output of the OCR.

**Applying OCR:**

Loop through each contour and take the x and y coordinates and the width and height using the function cv2.boundingRect(). Then draw a rectangle in the image using the function cv2.rectangle() with the help of obtained x and y coordinates and the width and height. There are 5 parameters in the cv2.rectangle(), the first parameter specifies the input image, followed by the x and y coordinates (starting coordinates of the rectangle), the ending coordinates of the rectangle which is (x+w, y+h), the boundary color for the rectangle in RGB value and the size of the boundary. Now crop the rectangular region and then pass it to the tesseract to extract the text from the image. Then we open the created text file in append mode to append the obtained text and close the file.

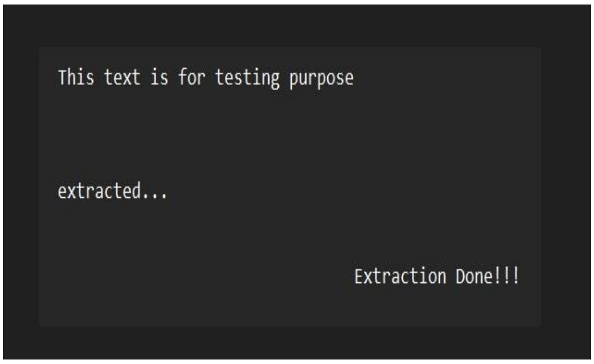
## Code:

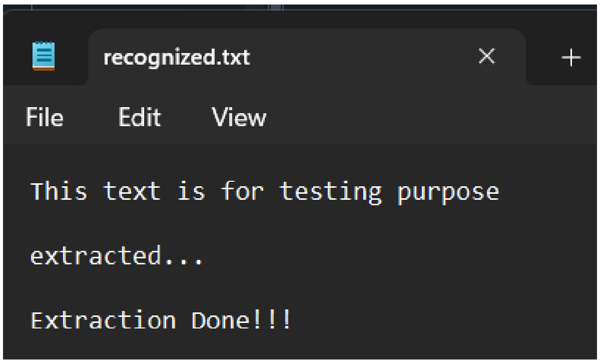
```
import cv2
import pytesseract

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.

**Theory:**
Transforming a 2D image into a 3D environment requires depth estimation, which can be a difficult operation depending on the amount of precision and information required. OpenCV supports a variety of depth estimation approaches, including stereo vision and depth from focus/defocus.

**In this practical we'll see a basic technique utilizing stereovision:**

**Transform a 2D image into a 3D space using OpenCV:**

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). 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, and augmented reality.

**Importance of transformations of a 2D image into a 3D space**
Transforming 2D images into 3D space becomes crucial in various fields due to its numerous applications and benefits:
**Depth Perception:** We are able to detect depth by transforming 2D pictures into 3D space. This makes it possible to use augmented reality, object recognition, and scene understanding.
**3D Reconstruction**: Converting 2D photos into 3D space makes it easier to recreate 3D scenes, which is crucial in industries like robotics, computer vision, and the preservation of cultural assets.
**Stereo Vision**: Stereo vision depends on converting 2D images into 3D space. It entails taking pictures from various angles and calculating depth from the difference between matching spots. It is employed in 3D modeling, autonomous navigation, and depth sensing, among other applications.
**Medical Imaging**: Improved visualization, diagnosis, and treatment planning are possible in medical imaging when 2D medical scans—such as CT or MRI scans—are converted into 3D space.
**Virtual Reality and Simulation**: In virtual reality, simulation, and gaming, realistic 3D worlds must be constructed from 2D photos or video. This requires translating 2D visuals into 3D space.

**How you get a 3D image from a 2D?**
In conventional photography, you can either utilize a mirror and attach a camera to it to create an immediate 3D effect, or you can take a shot, step to your right (or left), and then shoot another, ensuring that all components from the first photo are present in the second.
However, if you just move a 2D picture left by 10 pixels, nothing changes. This is because you are shifting the entire environment, and no 3D information is saved. Instead, there must be a bigger shift distance between the foreground and backdrop. In other words, the farthest point distant from the lens remains motionless while the nearest

point moves.
How is this related to using a 2D image to create a 3D image?
We need a method to move the pixels since an picture becomes three-dimensional when the foreground moves more than the background.
Fortunately, a technique known as depth detection exists that generates what is known as a depth map.
Now remember that this is only an estimate. Furthermore, it won't reach every nook and corner. All that depth detection does is use cues like shadows, haze, and depth of focus to determine if an object is in the forefront or background.

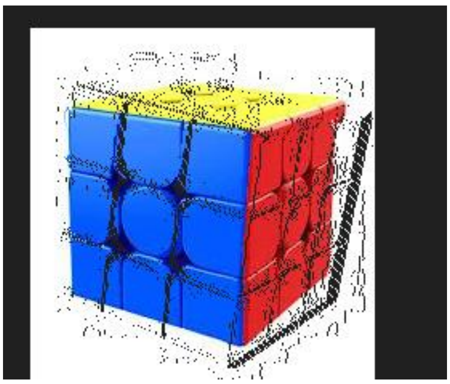
## Code:

```
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.

**Theory:**

**Image registration** is a digital image processing technique that helps us align different images of the same scene. For instance, one may click the picture of a book from various angles. Below are a few instances that show the diversity of camera angles. Now, we may want to "align" a particular image to the same angle as a reference image. In the images above, one may consider the first image to be an "ideal" cover photo, while the second and third images do not serve well for book cover photo purposes. The image registration algorithm helps us align the second and third pictures to the same plane as the first one.

**How does image registration work?**

Alignment can be looked at as a simple coordinate transform. The algorithm works as follows:

- Convert both images to grayscale.
- Match features from the image to be aligned, to the reference image and store the coordinates of the corresponding key points. Keypoints are simply the selected few points that are used to compute the transform (generally points that stand out), and descriptors are histograms of the image gradients to characterize the appearance of a keypoint. In this post, we use ORB (Oriented FAST and Rotated BRIEF) implementation in the OpenCV library, which provides us with both key points as well as their associated descriptors.
- Match the key points between the two images. In this post, we use BFMatcher, which is a brute force matcher. BFMatcher.match() retrieves the best match, while BFMatcher.knnMatch() retrieves top K matches, where K is specified by the user.
- Pick the top matches, and remove the noisy matches.
- Find the homomorphy transform.
- Apply this transform to the original unaligned image to get the output image.

**Applications of Image Registration –**

Some of the useful applications of image registration include:

- Stitching various scenes (which may or may not have the same camera alignment) together to form a continuous panoramic shot.
- Aligning camera images of documents to a standard alignment to create realistic scanned documents.
- Aligning medical images for better observation and analysis.

## Code:

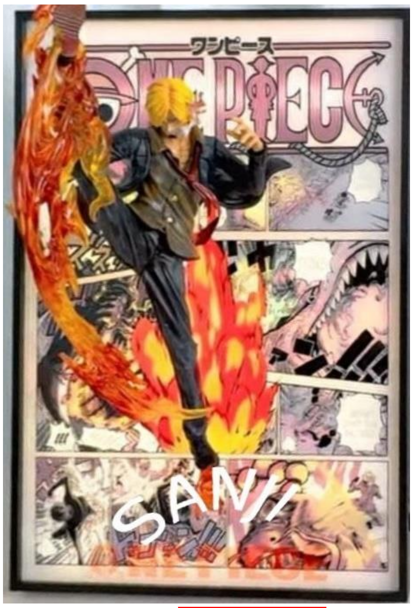
```
import cv2
import numpy as np

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.

**Theory:**
Image matting and composition are two interconnected image processing techniques. Image matting focuses on separating a foreground object from its background, often by estimating the opacity (alpha value) of each pixel. This results in an alpha matte, which is used in the subsequent image composition step. Image composition then combines this extracted foreground with a different background, effectively blending them together.

**Image Matting:**

- **Goal:**

To accurately separate a foreground object from its background.

- **Method:**

Involves estimating the alpha value for each pixel, representing the degree to which it belongs to the foreground.

- **Output:**

An alpha matte, which is a grayscale image where pixel values indicate the foreground's opacity.

- **Key considerations:**

Handling fine details (hair, fur, transparent objects) and similar foreground/background colors are challenges in image matting.
Image Composition:

- **Goal:**

To combine a foreground element with a new background.

- **Method:**

Uses the alpha matte generated during image matting to blend the foreground and background pixels.

- **Process:**

The alpha values determine how much of the foreground and background colors contribute to the final composite image.

- **Example:**

Replacing a background in a photo or creating visual effects in movies often involves image matting and composition.

In essence, image matting provides the necessary information (the alpha matte) to perform image composition, allowing for seamless integration of different image elements.

## 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:**