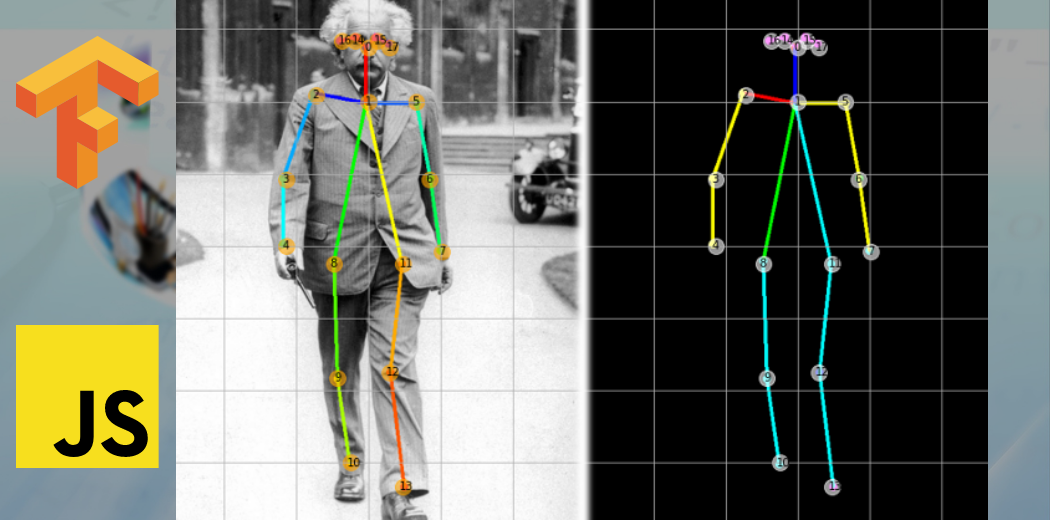
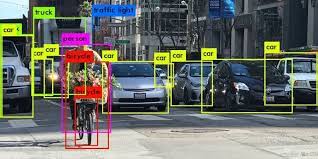
**UNIT - 2**

**Points Patches Lines and Segments**

In computer vision, points, patches, lines, and segments are fundamental elements used to represent and analyze the content of images. Each of these entities serves a specific purpose in image processing and computer vision tasks.

* Points: As mentioned earlier, points represent specific pixel locations in an image.
* They are often used to mark key interest points, corners, or landmarks.
* Points can be detected using various techniques, such as the Harris corner detection or the Scale-Invariant Feature Transform (SIFT).
* Points are essential for tasks like feature detection, object recognition, camera calibration, and object tracking.



Patches: Patches, also known as image regions or windows, are small square or rectangular areas extracted from a larger image.

These regions usually contain local information, and they are essential for capturing localized features within an image.

Patches are commonly used for feature description, object recognition, texture analysis, and training Convolutional Neural Networks (CNNs) in computer vision tasks.



Lines: Lines are used to represent linear structures or edges in an image. They are detected through edge detection techniques like the Canny edge detection algorithm.

Lines are critical for tasks like shape detection, boundary extraction, and object contour analysis.

In some cases, line segments or line pairs may be used to represent more complex structures or curves in an image.



Segments: Segments, also known as image regions or image segmentation, refer to the process of dividing an image into meaningful and coherent regions or segments.

* Each segment typically corresponds to a distinct object or region of interest in the image.
* Image segmentation is widely used in various computer vision applications such as object recognition, object tracking, image editing, and medical imaging.

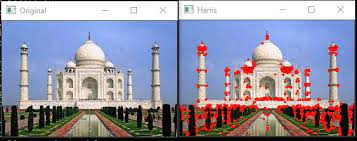
Applications of these elements in computer vision can vary depending on the specific task.

For example, in object recognition, points and patches are used for feature extraction, lines can help in edge detection, and image segmentation can aid in isolating and identifying objects within the scene.

**Feature Detection, Description, Mapping and Extraction**

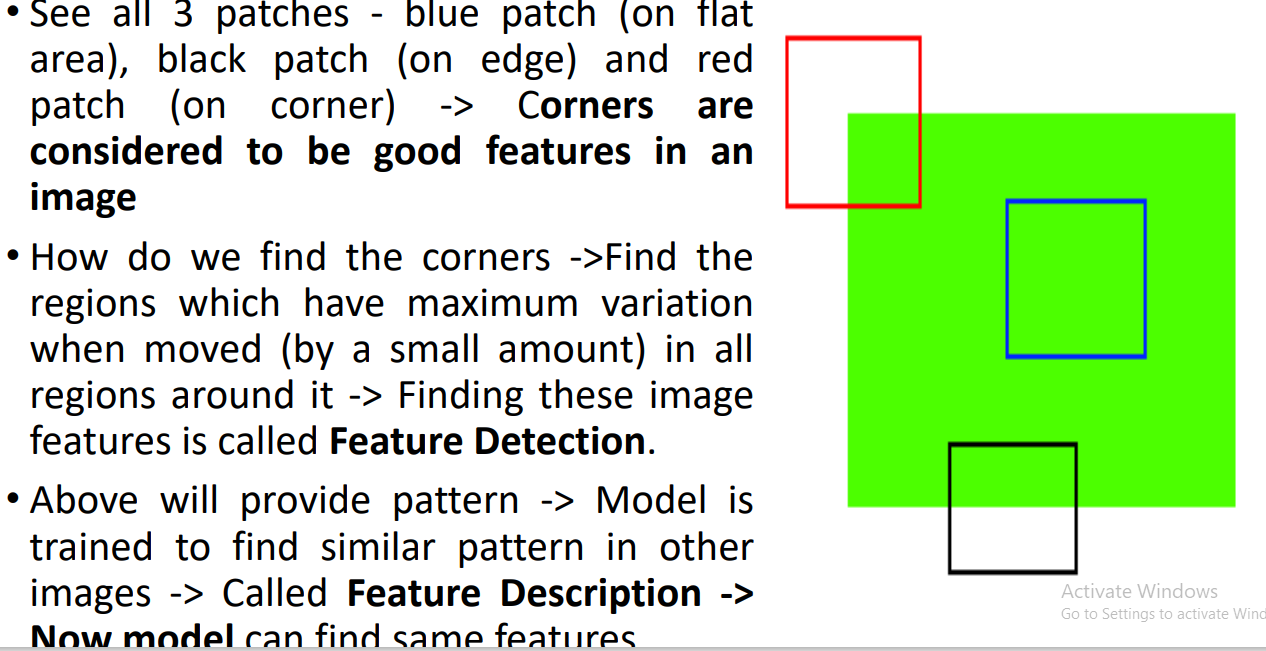
**Feature Detection**

* Feature detection involves identifying distinctive points or regions within an image that can be used to differentiate it from other images.
* These features are typically areas where intensity, color, or texture change significantly.
* Common methods for feature detection include Harris corner detection, Shi-Tomasi corner detection, and scale-invariant feature transform (SIFT).
* The goal is to identify points of interest that are likely to remain recognizable under different viewing conditions or transformations.



**Feature Description**

* Once distinctive points or regions are detected, feature description involves creating compact and robust representations of these points that can be used for matching across images.
* These descriptions are typically vectors that encode information about the local neighborhood of each feature point.
* Descriptors should be invariant to changes like rotation, scale, and illumination.
* Examples of descriptors include SIFT descriptors, Speeded-Up Robust Features (SURF) descriptors, and Histogram of Oriented Gradients (HOG) descriptors.



**Feature Mapping**

* Feature matching is the process of establishing correspondences between features detected in different images.
* The goal is to identify which features in one image correspond to features in another image, allowing for comparison and analysis across images.
* Various algorithms can be used for feature matching, such as nearest neighbor matching, where each feature in one image is paired with its nearest neighbor in the other image based on a distance metric (e.g., Euclidean distance) calculated from their descriptors.



**Feature Extraction**

* Feature extraction involves the extraction of relevant information from the matched feature points for subsequent tasks.
* Once correspondences are established, this step aims to extract meaningful information, such as the transformation parameters between images (e.g., for image registration), or 3D point correspondences (e.g., for 3D reconstruction).

**Example for implementation of feature detection**

**import matplotlib.pyplot as plt**

**import numpy as np**

**import cv2**

**my\_image\_color = cv2.imread('c:/images/arrow.jpg')**

**my\_image\_gray = np.float32(cv2.cvtColor(my\_image\_color,cv2.COLOR\_BGR2GRAY))**

# cornerHarris arguments are

**#img - Input image, it should be grayscale and float32 type.**

**#blockSize - It is the size of neighbourhood considered for corner detection**

**#ksize - Aperture parameter of Sobel derivative used.**

**#k - Harris detector free parameter in the equation.**

**my\_image\_corner\_objects = cv2.cornerHarris(my\_image\_gray,2,3,0.04)**

**#result is dilated(make or become wider, larger, or more open) for marking the corners**

**my\_image\_corner\_objects = cv2.dilate(my\_image\_corner\_objects,None)**

**# Threshold for an optimal value, it may vary depending on the image.**

**my\_image\_color[my\_image\_corner\_objects>0.01\*my\_image\_corner\_objects.max()] = [255,0,0] #**

**cv2.imshow('my\_image',my\_image\_color); cv2.waitKey(0); cv2.destroyAllWindows()**

**EDGE DETECTION**

Edge detection is a fundamental technique in computer vision used to identify the boundaries of objects within an image. It plays a crucial role in various image processing tasks such as object recognition, image segmentation, and feature extraction. The goal of edge detection is to find areas in an image where the intensity changes sharply, which often correspond to object boundaries or significant image features.

Canny edge detection is a popular edge detection algorithm that was developed by John F. Canny in 1986. It is known for its ability to accurately detect edges while suppressing noise, and it consists of multiple stages that work together to produce high-quality edge maps. Here's an overview of the key steps in the Canny edge detection algorithm:

**Gaussian Blurring**: The first step involves applying Gaussian blurring to the input image. This helps reduce noise and prevents small variations in intensity from being detected as edges. The blurring is achieved by convolving the image with a Gaussian kernel.

**Gradient Calculation**: After blurring, the gradient magnitude and direction are calculated for each pixel in the image. This is typically done using Sobel operators, which compute the gradients in the horizontal and vertical directions.

**Non-Maximum Suppression**: In this step, each pixel is examined in its gradient direction. If a pixel's gradient magnitude is larger than its neighbors in the direction of the gradient, it is retained as a potential edge pixel; otherwise, it is suppressed (set to zero).

**Double Thresholding**: The gradient magnitudes are now thresholded using two thresholds: a high threshold (strong edge threshold) and a low threshold (weak edge threshold). Pixels with gradient magnitudes above the high threshold are considered strong edge pixels, while those between the low and high thresholds are considered weak edge pixels.

**Edge Tracking by Hysteresis**: This step aims to link weak edge pixels that are likely part of the same edge structure. Starting from strong edge pixels, connected weak edge pixels are considered part of the same edge. This helps to overcome gaps in the detected edges and create continuous edge contours.

**Edge Map Generation**: Finally, the edge map is generated based on the processed strong and connected weak edge pixels. Strong edge pixels are usually part of the final edge map, while weak edge pixels that are not connected to strong edges are discarded.

Canny edge detection produces a binary edge map where edge pixels are marked with white and non-edge pixels with black. This edge map highlights the regions in the image where significant intensity changes occur, often corresponding to object boundaries or important features.

One of the advantages of the Canny edge detection algorithm is its ability to handle noise effectively and produce thin and well-connected edges. However, it also requires parameter tuning for the thresholds, and the choice of these thresholds can impact the results. Additionally, Canny may not be the most efficient algorithm for real-time applications due to its multiple stages and computations involved.

**Example for Implementation of Edge Detection**

import cv2

import numpy as np

# Read the image

image\_path = 'c:/images/tigerlion.jpg' # Replace with the actual image path

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

# Apply Gaussian blurring

blurred = cv2.GaussianBlur(image, (5, 5), 0)

# Apply Canny edge detection

edges = cv2.Canny(blurred, threshold1=100, threshold2=200)

# Display the original image, blurred image, and edge map

cv2.imshow('Original Image', image)

cv2.imshow('Blurred Image', blurred)

cv2.imshow('Edge Map', edges)

cv2.waitKey(0)

cv2.destroyAllWindows()

**CONTOUR DETECTION**

Contour detection is a process in computer vision that involves identifying and extracting the outlines or boundaries of objects and regions within an image. The goal of contour detection is to segment an image into its constituent parts by detecting the curves that mark the transitions between different areas of the image with distinct characteristics.

Here's how contour detection typically works:

**Edge Detection:** Contour detection often starts with edge detection. Edge detection algorithms identify points in an image where the intensity or color changes significantly. These points are usually located at the boundaries between objects or regions. Common edge detection techniques include the Sobel operator, Canny edge detection, and Prewitt operator.

**Gradient Computation:** Once the edges are detected, gradient information is computed to determine the direction and strength of the intensity change at each edge point. Gradients provide information about the orientation of the edges.

**Contour Tracing:** The detected edges are connected to form continuous curves or contours. This is achieved by tracing the gradient directions and linking neighboring edge points that have a coherent gradient direction.

**Contour Simplification:** Depending on the application, contours may need to be simplified to reduce the number of points and create smoother curves. Simplification techniques like the Douglas-Peucker algorithm can be applied to achieve this.

**Contour Representation:** Contours can be represented in various ways, such as sequences of points, parametric equations, or B-spline curves. The choice of representation depends on the specific application.

Contour detection has a wide range of applications in computer vision and image analysis:

**Object Detection and Recognition:** Contour detection is a key step in recognizing objects by their shapes. It helps segment objects from the background and provides a structural representation for object recognition algorithms.

**Boundary Extraction:** In medical imaging, contour detection is used to delineate boundaries of organs, tumors, and other anatomical structures.

**Image Segmentation:** Contour detection is essential for segmenting an image into meaningful regions, which can be further analyzed or manipulated.

**Gesture and Handwriting Recognition:** Contour detection can be used to interpret gestures or handwritten characters by recognizing the shapes created by the user's movements.

**Industrial Inspection:** In quality control, contour detection can identify defects or irregularities in manufactured products by comparing detected contours with expected shapes.

**Robotics and Navigation:** Robots and autonomous vehicles can use contour information for obstacle avoidance and navigation.

Example

import cv2

image = cv2.imread('image\_path.jpg')

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

edges = cv2.Canny(gray, threshold1, threshold2)

contours, \_ = cv2.findContours(edges, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

cv2.drawContours(image, contours, -1, (0, 255, 0), 2) # Draw all contours in green with a thickness of 2 pixels

cv2.imshow('Contours', image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Mean Shift Segmentation**

* Mean Shift is a non-parametric image segmentation technique that doesn't require prior knowledge of the number of clusters or their shapes.
* Mean Shift can be used to group similar pixels together, effectively segmenting an image into different regions based on color or feature similarity.
* Mean Shift has been applied to various tasks such as object tracking, image segmentation, and even in some cases, feature space reduction.

Algorithm for Mean Shift

1. **Kernel Density Estimation**: A kernel function is defined (e.g., Gaussian). Each data point is considered as a potential center. The kernel weights nearby points based on distance.
2. **Center Shifting**: Iteratively, each center is shifted towards the mean of the points within the kernel's neighborhood, based on the weighted distances.
3. **Convergence**: The process continues until centers converge to local modes (clusters) in the data distribution.

**Example**

Import numpy as np

import matplotlib.pyplot as plt

# Define Gaussian kernel

def gaussian\_kernel(x, center, bandwidth):

return np.exp(-0.5 \* np.linalg.norm((x - center) / bandwidth) \*\* 2)

# Generate synthetic data

np.random.seed(0)

data = np.concatenate((np.random.randn(100, 2), np.random.randn(100, 2) + [5, 5]))

# Initialize centers

centers = data.copy()

# Mean Shift iterations

num\_iterations = 10

bandwidth = 1.5

for \_ in range(num\_iterations):

new\_centers = []

for center in centers:

weighted\_sum = np.zeros\_like(center)

total\_weight = 0

for point in data:

weight = gaussian\_kernel(point, center, bandwidth)

weighted\_sum += point \* weight

total\_weight += weight

new\_centers.append(weighted\_sum / total\_weight)

centers = np.array(new\_centers)

# Visualize the data and centers

plt.scatter(data[:, 0], data[:, 1], c='blue', label='Data Points')

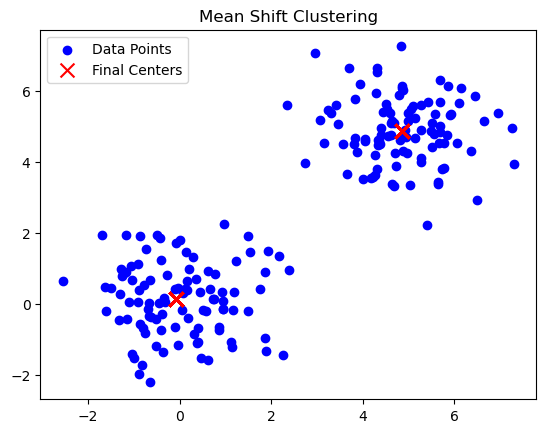
plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=100, label='Final Centers')

plt.title('Mean Shift Clustering')

plt.legend()

plt.show()

Output:



**Normalized Cut**

* Normalized Cuts is a graph-based image segmentation technique that emphasizes both similarity within segments and dissimilarity between them.
* It views image segmentation as a graph partitioning problem, aiming for balanced segments.

Algorithm

1. **Graph Representation**: The first step involves constructing a graph representation of the image. Each pixel becomes a node in the graph, and the edges between nodes are weighted based on the similarity between pixel values or other feature descriptors. Common choices for the similarity measure include color similarity, texture, and gradient magnitude.
2. **Graph Partitioning**: The goal of Graph Cut is to divide the graph into segments or clusters such that the similarity within each segment is high, and the similarity between segments is low. This is done by finding a cut that minimizes the cost function, which is defined based on the weights of the edges and the degrees of the nodes.
3. **Normalized Cut Criterion**: The normalized cut criterion takes into account the size of the segments. It ensures that segments are not too small or too large by normalizing the cut cost with the segment sizes. The goal is to find a partition that minimizes the normalized cut criterion, which encourages balanced segment sizes and low dissimilarity between segments.
4. **Recursive Approach**: Since the Normalized Cuts algorithm aims to divide the image into smaller regions, it can be applied recursively. After the initial segmentation, the segments with high normalized cut costs can be further split until a stopping criterion is met.

Example

Perform Normalized Graph cut on the Region Adjacency Graph.Given an image’s labels and its similarity RAG, recursively perform a 2-way normalized cut on it. All nodes belonging to a subgraph that cannot be cut further are assigned a unique label in the output.

from skimage import data, segmentation, color

from skimage.future import graph

from matplotlib import pyplot as plt

img = data.coffee()

labels1 = segmentation.slic(img, compactness=30, n\_segments=400,

start\_label=1)

out1 = color.label2rgb(labels1, img, kind='avg', bg\_label=0)

g = graph.rag\_mean\_color(img, labels1, mode='similarity')

labels2 = graph.cut\_normalized(labels1, g)

out2 = color.label2rgb(labels2, img, kind='avg', bg\_label=0)

fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True, figsize=(6, 8))

ax[0].imshow(out1)

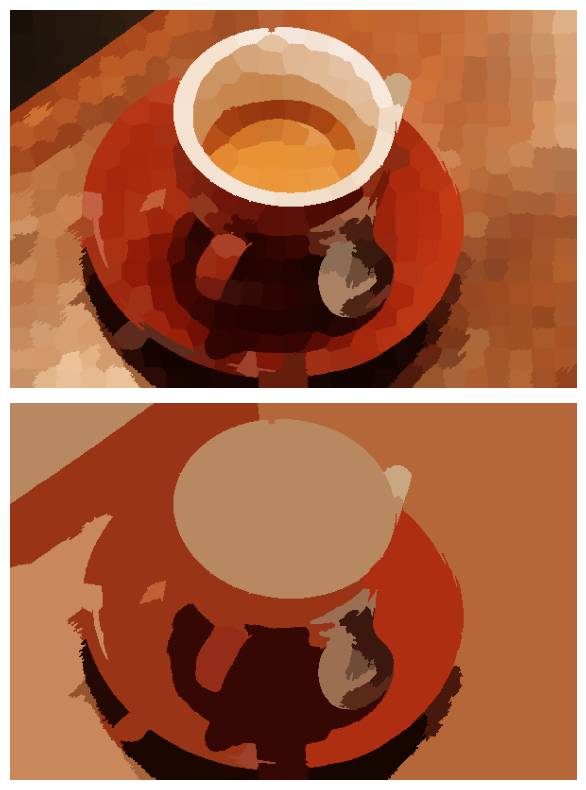
ax[1].imshow(out2)

for a in ax:

a.axis('off')

plt.tight\_layout()

Output:



**Pose Estimation**

* When you have a camera observing a scene, and you know the 3D positions of certain points in the scene along with their corresponding 2D projections in the camera image.
* The goal is to determine the camera's pose, which includes its rotation and translation with respect to the scene.

**DLT (Direct Linear Transformation) Algorithm**

* The DLT algorithm works by setting up a system of linear equations that relate the 2D image coordinates to the 3D world coordinates.
* This system of equations is typically over-determined, as you have more equations than unknowns due to the presence of multiple 2D-3D point correspondences.
* The DLT algorithm then solves this system of equations using linear algebra techniques, usually involving methods like singular value decomposition (SVD) or QR decomposition.

**Basic Steps of Algorithm**

* **Setup Equations**: For each 2D-3D correspondence, express the relationship between the camera projection matrix, the 3D point in homogeneous coordinates, and the 2D point in homogeneous coordinates. This results in a set of linear equations.
* **Construct Matrix**: Collect all the linear equations into a matrix equation of the form A \* P = 0, where A is the coefficient matrix and P is a vector containing the elements of the camera projection matrix.
* **Solve Linear System**: Solve the homogeneous linear system of equations A \* P = 0. This is typically done using techniques like singular value decomposition (SVD).
* **Extract Projection Matrix**: After obtaining the solution vector P, reshape it into a 3x4 matrix, representing the camera projection matrix.
* **Decompose Projection Matrix**: The camera projection matrix can be decomposed into intrinsic parameters (focal length, skew, principal point) and extrinsic parameters (rotation matrix and translation vector).
* **Normalization (Optional)**: It's common to apply a normalization step to improve the stability of the algorithm. This involves scaling and shifting the 2D and 3D points to have zero mean and unit standard deviation.

**Intrinsic Parameters**

**f\_x s c\_x**

**0 f\_y c\_y**

**0 0 1**

**f\_x f\_y**  are focal length parameters

focal length is the distance between lens and image sensor

**c\_x and c\_y** are the coordinates of the principal point (the point where the optical axis intersects the image plane).

**s** represents the skew coefficient or shear coefficient.

* In most well-calibrated cameras, the skew coefficient s is close to zero, meaning that the camera's imaging sensor is aligned with the camera's lens system and there's no skew between the image axes.

**Remarks about DLT**

* The DLT algorithm provides an initial estimate of the camera pose based on the linear equations, but it doesn't handle noise, outliers, or non-linear distortions well. Therefore, it's often used as an initialization step for more sophisticated techniques, such as non-linear optimization methods like Levenberg-Marquardt, to refine the pose estimate based on the actual image data.

Example:

cube\_3d = [

[0, 0, 0],

[1, 0, 0],

[1, 1, 0],

[0, 1, 0],

[0, 0, 1],

[1, 0, 1],

[1, 1, 1],

[0, 1, 1]

]

cube\_2d = [

[100, 100],

[200, 100],

[200, 200],

[100, 200],

[120, 80],

[220, 80],

[220, 180],

[120, 180]

]

import numpy as np

A = []

for i in range(len(cube\_3d)):

X, Y, Z = cube\_3d[i]

u, v = cube\_2d[i]

A.append([-X, -Y, -Z, -1, 0, 0, 0, 0, u\*X, u\*Y, u\*Z, u])

A.append([0, 0, 0, 0, -X, -Y, -Z, -1, v\*X, v\*Y, v\*Z, v])

A = np.array(A)

\_, \_, V = np.linalg.svd(A)

P = V[-1, :].reshape(3, 4) # Reshape the last row of V into a 3x4 matrix

K, RT = np.linalg.qr(np.linalg.inv(P[:, :3]))

K /= K[2, 2] # Normalize the intrinsic matrix

R = RT[:, :3]

t = RT[:, ]

print("Intrinsic Matrix K:")

print(K)

print("Rotation Matrix R:")

print(R)

print("Translation Vector t:")

print(t)