Project 2

ENPM673

Vishnu Mandala
119452608

Problem 1:

In this problem, you will perform camera pose estimation using homography. Given this video your task is to compute the rotation and translation between the camera and a coordinate frame whose origin is located on any one corner of the sheet of paper. In order to do so, you must:

- **Design an image processing pipeline to extract the paper on the ground and then extract all of its corners using the Hough Transformation technique .**
- **Once you have all the corner points, you will have to compute homography between real world points and pixel coordinates of the corners. You must write your own function to compute homography.**
- **Decompose the obtained homography matrix to get the rotation and translation**

Note: ***If you decide to resize the image frames, you need to accordingly modify your intrinsic matrix too. Refer to this discussion.***

Data:
The dimensions of the paper is 21.6 cm x 27.9 cm.
Intrinsic Matrix of the camera-

1.38E+03	0	9.46E+02
0	1.38E+03	5.27E+02
0	0	1

Solution:

Camera pose estimation refers to the process of determining the position and orientation of a camera with respect to a coordinate frame. In this problem, we are given a video that captures an image of a sheet of paper on the ground, and we need to compute the rotation and translation between the camera and a coordinate frame whose origin is located on any one corner of the sheet of paper. This can be achieved using homography, which is a transformation that maps points in one plane to corresponding points in another plane.

To perform camera pose estimation using homography, we need to follow the following steps:

1. Design an image processing pipeline to extract the paper on the ground and then extract all of its corners using the Hough Transformation technique.

   Hough Transformation is a computer vision technique used to detect lines and other geometric shapes in an image. It works by representing the lines in the image as points in a parameter space, where each point represents a line in the image. The parameter space is called the Hough space.

   To understand the concept of Hough Transformation, let's consider the standard form of a line equation:

   $$y = mx + c$$

   where m is the slope of the line and
   c is the y-intercept.

   We can rewrite this equation in polar coordinates as:

   $$\rho = xcos\theta + ysin\theta$$

   where ρ is the perpendicular distance of the line from the origin
   θ is the angle between the line and the x-axis.

   We can represent each line in the image as a point in the Hough space, where the x-axis represents θ and the y-axis represents ρ. The Hough space is a 2D parameter space, where each point represents a line in the image. To detect lines using Hough Transformation, we need to follow these steps:

i. **Edge Detection:** We need to first detect edges in the image using techniques like Canny Edge Detection. This gives us a binary image with white pixels representing the edges.
ii. **Hough Accumulator:** We then create a 2D array called the Hough Accumulator that represents the Hough parameter space. Each point (ρ, θ) in this parameter space corresponds to a line in the image. We initialize the accumulator to all zeros.
iii. **Voting:** For each edge pixel in the binary image, we calculate its distance ρ to the origin and the angle θ between the line passing through the origin and the edge pixel and the x-axis. We then increment the corresponding cell in the Hough Accumulator.
iv. **Thresholding:** Once we have accumulated all the votes, we threshold the accumulator to only keep the points that have a high number of votes. These points correspond to the lines in the image.
v. **Line Extraction:** We extract the lines from the thresholded accumulator by converting each point (ρ, θ) back to the image space and drawing the corresponding line.

Once we obtain multiple lines along the edges, we can use multiple thresholds for accumulator, ρ and θ to filter out the lines and extract only 4 lines along the edges of the paper. We then find the intersections of the current line with all the other lines in the list to obtain the corners of the paper. This is done by solving the system of equations:

$$aX = b$$

where a is a 2x2 matrix representing the slopes of the lines
b is a 2x1 vector representing the intercepts of the lines
X is a 2x1 vector representing the intersection point.

2. Once you have all the corner points, you will have to compute homography between real world points and pixel coordinates of the corners. You must write your own function to compute homography.

Homography refers to a mathematical relationship between two images or sets of points in an image. In computer vision, homography is often used to align and merge two images, or to project a 3D image onto a 2D plane. In the context of the given task, homography is used to calculate the transformation between the four corner points in the real world and their corresponding pixel coordinates in the image. The homography matrix can be used to transform any point in the real world to its corresponding pixel coordinates in the image, or vice versa.

The homography matrix is a 3x3 matrix that represents the transformation between the two sets of points. The process of computing the homography matrix involves solving a set of linear equations, typically using methods such as Linear Least Squares or Direct Linear Transform (DLT). Once the homography matrix is computed, it can be used to transform any point between the two coordinate systems.

Mathematical Concept –
Assume that we have N pairs of corresponding points in the real world and image coordinate systems:

Real world points: P1, P2, ..., PN (in 3D)
Image points: p1, p2, ..., pN (in 2D)

We want to find a 3x3 matrix H such that for any point P in the real world, its corresponding point p in the image can be calculated as:

$$p = H * P$$

or, in homogeneous coordinates:

$$p' = H * P'$$

where p' and P' are the homogeneous representations of p and P, respectively.

To compute H, we need to solve the following set of linear equations:

[x1 y1 1 0 0 0 -x1'x1 -x1'y1] [h1] [0]
[0 0 0 x1 y1 1 -y1'x1 -y1'y1] [h2] = [0]
[...] [..] [.]
[xn yn 1 0 0 0 -xn'xn -xn'yn] [h8] [0]
[0 0 0 xn yn 1 -yn'xn -yn'yn] [h9] [1]

where (x1, y1) and (x1', y1') are the coordinates of the first corresponding point in the real world and image, respectively, and so on for all N pairs of corresponding points.

The solution to this set of equations (which can be found using Singular Value Decomposition) gives us the elements of the homography matrix H. Once we have H, we can use it to transform any point between the real world and image coordinate systems.

Singular Value Decomposition:

Singular Value Decomposition (SVD) is a matrix factorization technique used to decompose a matrix into its constituent parts. SVD is used in various applications like image compression, data analysis, and machine learning. Given a matrix A, we can decompose it using SVD as:

$$A = U\Sigma V^T$$

where U and V are orthogonal matrices

Σ is a diagonal matrix containing the singular values of A.

The singular values are the square roots of the eigenvalues of A^TA or AA^T.

SVD can be used to solve linear systems of equations, compute the pseudoinverse of a matrix, and reduce the dimensionality of a dataset. To compute the SVD of a matrix, we need to follow these steps:

i. **Compute the singular values of A:** We first compute the singular values of A by computing the eigenvalues of A^TA or AA^T and taking the square root of them.
ii. **Compute the right singular vectors:** We then compute the right singular vectors of A by computing the eigenvectors of A^TA.
iii. **Compute the left singular vectors:** We compute the left singular vectors of A by computing the eigenvectors of AA^T
iv. **Construct the matrices U, Σ, and V:** We then construct the matrices U, Σ, and V using the singular values and singular vectors we computed in the previous steps.

The final column of V is extracted and reshaped to form a homography matrix.

3. Decompose the obtained homography matrix to get the rotation and translation

When we obtain a homography matrix from a set of corresponding points in two images, it represents the 2D transformation that maps points from one image to the other. However, this transformation does not directly provide us with the rotation and translation between the cameras that captured the images. To extract the rotation and translation, we need to decompose the homography matrix.

There are different methods to decompose a homography matrix, but one common approach is to use the Singular Value Decomposition (SVD) of the matrix. Given a homography matrix H, we can decompose it as follows:

$$H = K[R|t]K^{-1}$$

where K is the intrinsic matrix of the camera.

R is the rotation matrix that represents the orientation of the image points with respect to the real world

t is the translation vector that represents the position of the image points with respect to the real world

| denotes concatenation.

The intrinsic matrix is used to describe the properties of the camera such as focal length, image center and skew. The inverse of the intrinsic matrix is used to cancel out the effect of the camera parameters on the rotation and translation.

To obtain the decomposition using SVD, we first need to normalize the homography matrix by dividing it by its last element. This ensures that the resulting matrix has a determinant of 1 and simplifies the decomposition. Then, we perform SVD on the normalized matrix

The rotation matrix R can be obtained from U and V:

$$R = U * W * V^T$$

where W is a 3x3 diagonal matrix with 1s and determinant equal to 1 or -1.
If the determinant is -1, we can change the sign of the last column of R to make it positive.

We can obtain the translation vector t from H and R:

$$t = H[:, 2]/||R[:, 0]||$$

where H[:,2] represents the third column of H
||R[:,0]|| represents the Euclidean norm of the first column of R.

Function Descriptions:

```
def extract_paper_corners(img):
    gray = cv2.cvtColor(cv2.bitwise_and(img, img, mask=cv2.inRange(img, (200, 120, 100), (255, 255, 255))),
cv2.COLOR_BGR2GRAY)  # Convert the image to grayscale and extract the white paper using color masking
    gray = cv2.bilateralFilter(gray, 9, 75, 75)     # Apply bilateral filter to the grayscale image
    blurred = cv2.GaussianBlur(gray, (3, 3), 1)     # Apply Gaussian blur to the grayscale image
    closed = cv2.Canny(blurred, 750, 770)           # Apply Canny edge detection to the blurred image

    # Define the range of theta and rho
    height, width = closed.shape
    theta_range = np.deg2rad(np.arange(-90, 90, 1))
    rho_max = int(math.ceil(math.sqrt(height**2 + width**2)))
    rho_range = np.arange(-rho_max, rho_max + 1, 1)

    # Create accumulator array and accumulate votes
    accumulator = np.zeros((len(rho_range), len(theta_range)))
    for y, x in np.argwhere(closed):
        rho_vals = np.round(x * np.cos(theta_range) + y * np.sin(theta_range)).astype(int) + rho_max
        accumulator[rho_vals, np.arange(len(theta_range))] += 1

    peaks = np.argwhere(accumulator > 120)  # Select the peaks with votes greater than 120 (threshold) and
store them in peaks

    # Extract the (rho, theta) values corresponding to the peaks
    lines = []
    max_votes = [0] * 4
    final_lines = [None] * 4
    # Find the lines that are similar to the lines in lines
    for rho_idx, theta_idx in peaks:
        rho, theta = rho_range[rho_idx], theta_range[theta_idx]
        similar = False
        for i, (prev_rho, prev_theta) in enumerate(lines):
            if abs(rho - prev_rho) < 200 and abs(theta - prev_theta) < np.deg2rad(120):
                similar = True
                break
        if not similar:
            lines.append((rho, theta))
            votes = accumulator[rho_idx, theta_idx]
            if votes > min(max_votes):
                idx = max_votes.index(min(max_votes))
                max_votes[idx] = votes
                final_lines[idx] = (rho, theta)

    intersections = []
    for i, (rho, theta) in enumerate(lines):
        # Convert from polar coordinates to Cartesian coordinates and plot the line
        x0, y0 = rho * np.cos(theta), rho * np.sin(theta)
        x1, y1 = np.array([x0, y0]) + 2000 * np.array([-np.sin(theta), np.cos(theta)])
        x2, y2 = np.array([x0, y0]) - 2000 * np.array([-np.sin(theta), np.cos(theta)])
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 2)

        #Find the intersection of the current line with all the other lines
        for rho1, theta1 in lines[i+1:]:
            a = np.array([[np.cos(theta), np.sin(theta)], [np.cos(theta1), np.sin(theta1)]])
            b = np.array([rho, rho1])
```

```
            try:
                intersection = tuple(map(int, np.linalg.solve(a, b)))
                intersections.append(intersection)
            except np.linalg.LinAlgError:
                pass

    corners = np.array(intersections)   # Convert the list of intersections to a numpy array

    return corners
```

- **def extract_paper_corners(img):**: Define a function **extract_paper_corners** that takes an input image **img**.
- **gray = cv2.cvtColor(cv2.bitwise_and(img, img, mask=cv2.inRange(img, (200, 120, 100), (255, 255, 255))), cv2.COLOR_BGR2GRAY)**: Convert the input image to grayscale and extract the white paper using color masking. This is done using **cv2.inRange** to create a binary mask that identifies white areas in the input image, and **cv2.bitwise_and** to apply the mask to the input image. The resulting image is then converted to grayscale using **cv2.cvtColor**.
- **gray = cv2.bilateralFilter(gray, 9, 75, 75)**: Apply bilateral filter to the grayscale image. Bilateral filtering is a nonlinear edge-preserving smoothing filter that removes noise while preserving edges.
- **blurred = cv2.GaussianBlur(gray, (3, 3), 1)**: Apply Gaussian blur to the grayscale image. Gaussian blur is a linear smoothing filter that removes high-frequency noise.
- **closed = cv2.Canny(blurred, 750, 770)**: Apply Canny edge detection to the blurred image. Canny edge detection is a popular edge detection algorithm that uses gradient information to detect edges in an image.
- **height, width = closed.shape**: Get the height and width of the Canny edge detection output image.
- **theta_range = np.deg2rad(np.arange(-90, 90, 1))**: Define the range of theta (angle) values to use in the Hough transform. The **np.arange** function generates an array of values from -90 to 89, with a step size of 1, and **np.deg2rad** converts the values to radians.
- **rho_max = int(math.ceil(math.sqrt(height**2 + width**2)))**: Define the maximum distance (rho) value to use in the Hough transform. The value is calculated as the square root of the sum of the squared height and width of the Canny edge detection output image, rounded up to the nearest integer using **math.ceil**.
- **rho_range = np.arange(-rho_max, rho_max + 1, 1)**: Define the range of rho (distance) values to use in the Hough transform. The **np.arange** function generates an array of values from **-rho_max** to **rho_max**, with a step size of 1.
- **accumulator = np.zeros((len(rho_range), len(theta_range)))**: Create an accumulator array to store the Hough transform output.
- **peaks = np.argwhere(accumulator > 120)**: This line identifies the coordinates of the accumulator array where the votes are greater than 120, indicating the presence of a line segment. It stores these coordinates as peaks.
- **lines = []**, **max_votes = [0] * 4**, **final_lines = [None] * 4**: These lines initialize three lists: **lines**, **max_votes**, and **final_lines**. **lines** will store all the line segments that are found, **max_votes** will store the maximum number of votes received by each line segment, and **final_lines** will store the four line segments that best match the four sides of the paper. Each element of **max_votes** and **final_lines** is initialized to 0 and **None**, respectively.
- **for rho_idx, theta_idx in peaks: ...**: This loop iterates over all the peaks, which represent the line segments that were detected in the image. For each peak, it extracts the corresponding **(rho, theta)** values from the **rho_range** and **theta_range** arrays. It then checks if this line segment is similar to any of the previously detected line segments in **lines**. If it is not similar, it adds the line segment to **lines** and updates the **max_votes** and **final_lines** lists if the line segment has more votes than one of the existing line segments in **final_lines**.
- **x0, y0 = rho * np.cos(theta), rho * np.sin(theta)**: This line converts the polar coordinates **(rho, theta)** to Cartesian coordinates **(x0, y0)**, which represents a point on the line.
- **x1, y1 = np.array([x0, y0]) + 2000 * np.array([-np.sin(theta), np.cos(theta)])** and **x2, y2 = np.array([x0, y0]) - 2000 * np.array([-np.sin(theta), np.cos(theta)])**: These lines calculate the endpoints of the line segment by adding and subtracting 2000 times a unit vector perpendicular to the line direction. The length of 2000 was chosen empirically to ensure that the line segment extends beyond the paper boundaries.
- **cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 2)**: This line draws the line segment on the input image **img** using the endpoints **(x1, y1)** and **(x2, y2)**.
- **for rho1, theta1 in lines[i+1:]: ...**: This loop iterates over all pairs of line segments in **lines** and finds the intersection point between them. It uses the **np.linalg.solve()** function to solve a system of linear equations to find the intersection point.
- **corners = np.array(intersections)**: This line converts the list of intersection points to a numpy array, which represents the four corners of the paper in the image coordinate system.
- **return corners:** Returns the **corners** array, which can be used to warp the image and extract the paper region.

```
# Define the function to compute the homography matrix
def compute_homography(real_world_points, image_points):
    # Build the A matrix
    A = []
    for i in range(4):
        x, y = real_world_points[i]
        u, v = image_points[i]
        A.append([x, y, 1, 0, 0, 0, -u*x, -u*y, -u])
        A.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])
    A = np.array(A)

    # Solve for the homography matrix using linear least squares
    _, _, V = np.linalg.svd(A)  # Compute the SVD of A
    H = (V[-1,:] / V[-1,-1]).reshape((3,3)) # Extract the last column of V and reshape it to a 3x3 matrix

    return H
```

- **def compute_homography(real_world_points, image_points):** Define the function **compute_homography** that takes in two parameters: **real_world_points**, a list of four points in the real world, and **image_points**, a list of their corresponding points in the image.
- **A = []**: Create an empty list to store the elements of the A matrix.
- **for i in range(4):** Loop over the four points in **real_world_points** and **image_points**.
- **x, y = real_world_points[i]**: Unpack the **i**-th point in **real_world_points** into the variables **x** and **y**.
- **u, v = image_points[i]**: Unpack the **i**-th point in **image_points** into the variables **u** and **v**.
- **A.append([x, y, 1, 0, 0, 0, -u*x, -u*y, -u])**: Append a list to **A** that contains the elements of the first row of the matrix equation Ax = 0, where x is the homography matrix.
- **A.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])**: Append a list to **A** that contains the elements of the second row of the matrix equation Ax = 0.
- **A = np.array(A)**: Convert the list of lists **A** to a numpy array.
- **_, _, V = np.linalg.svd(A)**: Compute the singular value decomposition (SVD) of **A** and store the right singular vectors in **V**.
- **H = (V[-1,:] / V[-1,-1]).reshape((3,3))**: Extract the last column of **V** and divide it by its last element to obtain the homography matrix **H**. Reshape the resulting 9-element vector into a 3x3 matrix.
- **return H**: Return the homography matrix **H** from the **compute_homography** function.

```
# Define the function to decompose the homography matrix
def decompose_homography(H):
    K = np.array([[1.38e+03, 0, 9.46e+02], [0, 1.38e+03, 5.27e+02], [0, 0, 1]])  # Camera matrix
    H = np.linalg.inv(K) @ H    # Compute the homography matrix with respect to the camera frame
    R = H[:, :3]    # Extract the first three columns of the homography matrix

    # Normalize the first three columns to obtain the rotation matrix
    U, _, Vt = np.linalg.svd(R)
    R = np.dot(U, Vt)

    t = H[:, 2] / np.linalg.norm(H[:, :2])      # Extract the last column of the homography matrix and
normalize it
    t /= H[2, 2]                                # Divide the last column by the last element of the last row
of the homography matrix
    t = -np.dot(R, t)                           # Compute the translation vector by multiplying the rotation
matrix with the last column of the homography matrix
    return R, t
```

- **def decompose_homography(H):** Define the function decompose_homography that takes in one parameter: H, the homography matrix.
- **K = np.array([[1.38e+03, 0, 9.46e+02], [0, 1.38e+03, 5.27e+02], [0, 0, 1]])**: This line defines the camera matrix K as a numpy array.
- **H = np.linalg.inv(K) @ H**: This line computes the homography matrix with respect to the camera frame by multiplying the inverse of the camera matrix K with H.
- **R = H[:, :3]**: This line extracts the first three columns of the homography matrix H to obtain the rotation matrix R.
- **U, _, Vt = np.linalg.svd(R)**: This line performs the singular value decomposition (SVD) of R to obtain its left-singular vectors U and right-singular vectors Vt.

- **R = np.dot(U, Vt)**: This line normalizes the first three columns of H to obtain the rotation matrix R. It does this by multiplying U and Vt together to obtain an orthonormal matrix.
- **t = H[:, 2] / np.linalg.norm(H[:, :2])**: This line extracts the last column of H and normalizes it by dividing it by the Euclidean norm of the first two columns of H.
- **t /= H[2, 2]**: This line divides the last column of H by the last element of the last row of H.
- **t = -np.dot(R, t)**: This line computes the translation vector t by multiplying the rotation matrix R with the normalized last column of H and negating the result.
- **return R,t:** Returns the rotation matrix R and the translation vector t as a tuple.

Full Code:

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

paper_width, paper_height = 21.6, 27.9  # Define the dimensions of the paper in cm
real_world_corners = np.array([[0, 0], [0, paper_height], [paper_width, paper_height], [paper_width, 0]])   #
Define the coordinates of the corners of the paper in the real world coordinate system

# Define the function to extract the corners of the paper
def extract_paper_corners(img):
    gray = cv2.cvtColor(cv2.bitwise_and(img, img, mask=cv2.inRange(img, (200, 120, 100), (255, 255, 255))),
cv2.COLOR_BGR2GRAY)  # Convert the image to grayscale and extract the white paper using color masking
    gray = cv2.bilateralFilter(gray, 9, 75, 75)     # Apply bilateral filter to the grayscale image
    blurred = cv2.GaussianBlur(gray, (3, 3), 1)     # Apply Gaussian blur to the grayscale image
    closed = cv2.Canny(blurred, 750, 770)           # Apply Canny edge detection to the blurred image

    # Define the range of theta and rho
    height, width = closed.shape
    theta_range = np.deg2rad(np.arange(-90, 90, 1))
    rho_max = int(math.ceil(math.sqrt(height**2 + width**2)))
    rho_range = np.arange(-rho_max, rho_max + 1, 1)

    # Create accumulator array and accumulate votes
    accumulator = np.zeros((len(rho_range), len(theta_range)))
    for y, x in np.argwhere(closed):
        rho_vals = np.round(x * np.cos(theta_range) + y * np.sin(theta_range)).astype(int) + rho_max
        accumulator[rho_vals, np.arange(len(theta_range))] += 1

    peaks = np.argwhere(accumulator > 120)  # Select the peaks with votes greater than 120 (threshold) and
store them in peaks

    # Extract the (rho, theta) values corresponding to the peaks
    lines = []
    max_votes = [0] * 4
    final_lines = [None] * 4
    # Find the lines that are similar to the lines in lines
    for rho_idx, theta_idx in peaks:
        rho, theta = rho_range[rho_idx], theta_range[theta_idx]
        similar = False
        for i, (prev_rho, prev_theta) in enumerate(lines):
            if abs(rho - prev_rho) < 200 and abs(theta - prev_theta) < np.deg2rad(120):
                similar = True
                break
        if not similar:
            lines.append((rho, theta))
            votes = accumulator[rho_idx, theta_idx]
            if votes > min(max_votes):
                idx = max_votes.index(min(max_votes))
                max_votes[idx] = votes
                final_lines[idx] = (rho, theta)

    intersections = []
    for i, (rho, theta) in enumerate(lines):
        # Convert from polar coordinates to Cartesian coordinates and plot the line
        x0, y0 = rho * np.cos(theta), rho * np.sin(theta)
```

```
        x1, y1 = np.array([x0, y0]) + 2000 * np.array([-np.sin(theta), np.cos(theta)])
        x2, y2 = np.array([x0, y0]) - 2000 * np.array([-np.sin(theta), np.cos(theta)])
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 2)

        #Find the intersection of the current line with all the other lines
        for rho1, theta1 in lines[i+1:]:
            a = np.array([[np.cos(theta), np.sin(theta)], [np.cos(theta1), np.sin(theta1)]])
            b = np.array([rho, rho1])
            try:
                intersection = tuple(map(int, np.linalg.solve(a, b)))
                intersections.append(intersection)
            except np.linalg.LinAlgError:
                pass

    corners = np.array(intersections)   # Convert the list of intersections to a numpy array

    return corners

# Define the function to compute the homography matrix
def compute_homography(real_world_points, image_points):
    # Build the A matrix
    A = []
    for i in range(4):
        x, y = real_world_points[i]
        u, v = image_points[i]
        A.append([x, y, 1, 0, 0, 0, -u*x, -u*y, -u])
        A.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])
    A = np.array(A)

    # Solve for the homography matrix using linear least squares
    _, _, V = np.linalg.svd(A)  # Compute the SVD of A
    H = (V[-1,:] / V[-1,-1]).reshape((3,3)) # Extract the last column of V and reshape it to a 3x3 matrix

    return H

# Define the function to decompose the homography matrix
def decompose_homography(H):
    K = np.array([[1.38e+03, 0, 9.46e+02], [0, 1.38e+03, 5.27e+02], [0, 0, 1]])  # Camera matrix
    H = np.linalg.inv(K) @ H    # Compute the homography matrix with respect to the camera frame
    R = H[:, :3]    # Extract the first three columns of the homography matrix

    # Normalize the first three columns to obtain the rotation matrix
    U, _, Vt = np.linalg.svd(R)
    R = np.dot(U, Vt)

    t = H[:, 2] / np.linalg.norm(H[:, :2])      # Extract the last column of the homography matrix and
normalize it
    t /= H[2, 2]                                # Divide the last column by the last element of the last row
of the homography matrix
    t = -np.dot(R, t)                           # Compute the translation vector by multiplying the rotation
matrix with the last column of the homography matrix
    return R, t

cap = cv2.VideoCapture('project2.avi')
roll_list, pitch_list, yaw_list, x_list, y_list, z_list = [], [], [], [], [], []

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

    corners = extract_paper_corners(frame)
    for corner in corners:
        cv2.circle(frame, tuple(corner), 5, (0, 0, 255), -1)    # Draw the corners on the image

    H = compute_homography(real_world_corners, corners)
    R, t = decompose_homography(H)
    print("\nRotation:\n", R, "\nTranslation:\n", t, '\n')  # Print the rotation and translation matrices

    roll, pitch, yaw = [math.atan2(R[i, j], R[i, k]) for i, j, k in [(0, 1, 2), (1, 2, 0), (2, 0, 1)]]  #
Compute the roll, pitch, and yaw angles from the rotation matrix
```

```
    tx, ty, tz = t  # Extract the translation vector components

    roll_list.append(roll)
    pitch_list.append(pitch)
    yaw_list.append(yaw)
    x_list.append(tx)
    y_list.append(ty)
    z_list.append(tz)

    cv2.imshow('frame', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()

fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, sharex=True)

# Plot the roll, pitch, and yaw data in the first subplot
ax1.plot(roll_list, label='Roll')
ax1.plot(pitch_list, label='Pitch')
ax1.plot(yaw_list, label='Yaw')
ax1.legend()
ax1.set_ylabel('Angle (rad)')

# Plot the x, y, and z data in the second subplot
ax2.plot(x_list, label='X')
ax2.plot(y_list, label='Y')
ax2.plot(z_list, label='Z')
ax2.legend()
ax2.set_ylabel('Translation (cm)')
ax2.set_xlabel('Frame')

# Show the plot
plt.show()
```

Plot:

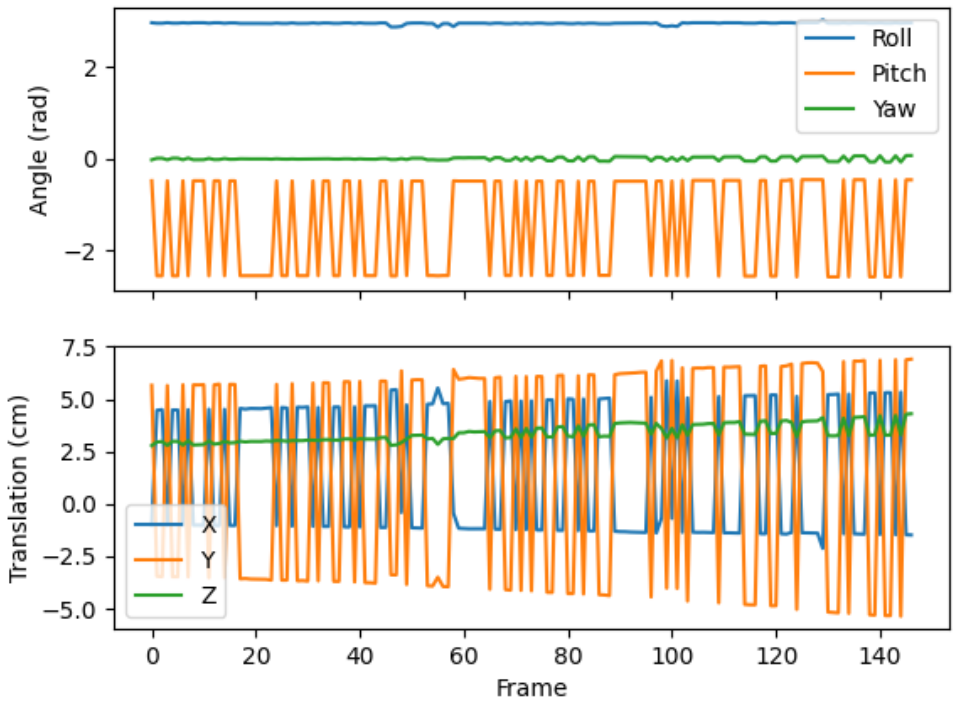

Figure 1 Plots with the Rotation and Translation of the Paper Frame

Hough Lines on the frame of the video:

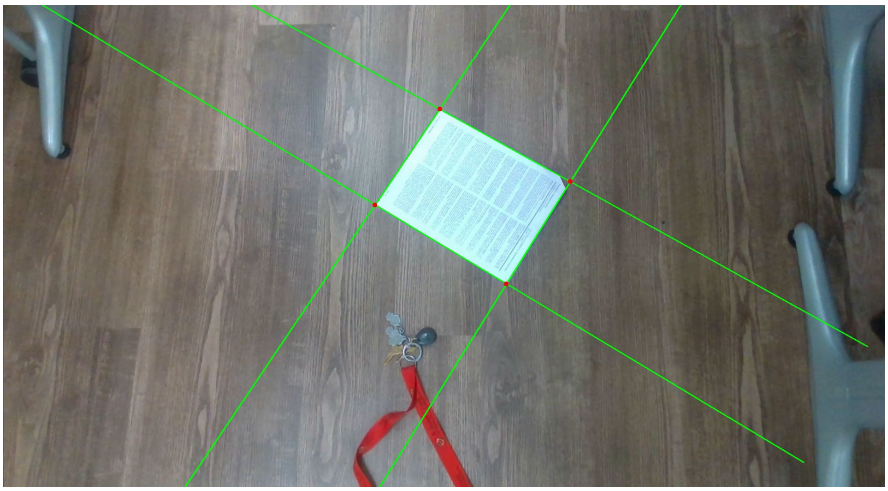

Figure 2 Lines with corners identified

Terminal Output:
(Only the initial matrices)

```
PS C:\Users\manda\OneDrive - University of Maryland\Perception For Autonmous Robots\Project 2> & "C:/Program
Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Perception For Autonmous
Robots/Project 2/problem1.py"

Rotation:
 [[-0.47551727  0.14923454 -0.86695581]
 [ 0.87931474  0.11003532 -0.46335496]
 [-0.02624719  0.98266031  0.18354783]]
Translation:
 [-1.0023014   5.66139708  2.78463452]


Rotation:
 [[ 0.55341795  0.15025172 -0.81923927]
 [-0.8328733   0.10823348 -0.54277765]
 [ 0.00711584  0.98270542  0.185039  ]]
Translation:
 [ 4.46735501 -3.45086053  2.95554201]
```

Problems Encountered:
As the output plot demonstrates, the corners are not always computed perfectly, which causes the translation and rotation to vary sometimes. But for most of the video, the output is good

Pipeline:

1. Import the necessary libraries: OpenCV (cv2), NumPy, math, and matplotlib.pyplot
2. Define the dimensions of a paper in centimeters and the real-world coordinates of its corners.
3. Define a function named "extract_paper_corners" that takes an image and returns its corners.
    a. Convert the image to grayscale and extract the white paper using color masking.
    b. Apply a bilateral filter to the grayscale image.
    c. Apply a Gaussian blur to the grayscale image.
    d. Apply Canny edge detection to the blurred image.
    e. Create an accumulator array and accumulate votes.
    f. Select the peaks with votes greater than 120 (threshold) and store them in peaks.
    g. Extract the (rho, theta) values corresponding to the peaks.
    h. Find the lines that are similar to the lines in lines.
    i. Convert from polar coordinates to Cartesian coordinates and plot the line.
    j. Find the intersection of the current line with all the other lines.

    k. Convert the list of intersections to a NumPy array and return it.
4. Define a function named "compute_homography" that takes the real-world points and image points and returns the homography matrix.
    a. Build the A matrix.
    b. Solve for the homography matrix using linear least squares.
5. Define a function named "decompose_homography" that takes the homography matrix and returns the intrinsic and extrinsic camera parameters.
    a. Compute the inverse of the camera matrix using the homography matrix.
    b. Normalize the columns of the camera matrix.
    c. Compute the rotation matrix and the translation vector using the QR decomposition of the camera matrix.
    d. Compute the extrinsic and intrinsic camera parameters.
6. Load an image and extract its corners using the "extract_paper_corners" function.
7. Define the image points of the corners of the image in the same order as the real-world corners.
8. Compute the homography matrix using the "compute_homography" function and the real-world and image points.
9. Decompose the homography matrix using the "decompose_homography" function and print the Rotation and Translation Matrices
10. Compute the Roll, Pitch, Yaw and Translation vectors from matrices and plot them

Problem 2:

You are given four images which were taken from the same camera position (only rotation, no translation) you will need to stitch these images to create a panoramic image.

To solve this problem, you will need to:

- **Extract features from each frame (You can use any feature extractor).**
- **Match the features between each consecutive image and visualize them.**
- **Compute the homographies between the pairs of images**
- **Combine these frames together using the computed homographies.**

Figure 3 Given Images

Solution:

Creating a panoramic image by stitching images involves Feature Extraction, Feature Matching, Homography Computation and Combining the images.

1. **Feature Extraction:** Extracting features from each frame using SIFT (Scale-Invariant Feature Transform) algorithm. SIFT extracts keypoints (locations in the image) and descriptors (features at these keypoints) that are invariant to scale, rotation, and illumination changes.

   SIFT Algorithm –

   i. **Scale-Space Extrema Detection:** SIFT uses a Difference of Gaussian (DoG) filter to detect scale-space extrema in an image. The DoG filter is created by subtracting one Gaussian filtered image from another at different scales.

   $$DoG(x, y, \sigma) = G(x, y, k\sigma) - G(x, y, \sigma)$$

   where G(x, y, σ) is a Gaussian filter with standard deviation σ
   k is the factor between adjacent scales (typically k = √2).

   The DoG filter is applied to the image at different scales to detect keypoint locations that are invariant to scale.

   ii. **Keypoint Localization**: Once the keypoints are detected, SIFT refines their locations by fitting a second-order Taylor series to the DoG function at each keypoint location.

   $$D(x) = D + \frac{1}{2}dT(x - p) + (x - p)TH(x - p)$$

   where D is the DoG value at the keypoint location
   p is the keypoint location
   dT is the gradient vector
   H is the Hessian matrix.

   The keypoint location is refined by solving for the maximum of the above function.

iii. **Orientation Assignment:** SIFT assigns an orientation to each keypoint by computing a histogram of gradient orientations in a circular region around the keypoint location. The histogram has 36 bins (10 degrees per bin) and is weighted by the gradient magnitude. The orientation with the highest magnitude in the histogram is chosen as the dominant orientation.

iv. **Descriptor Computation:** SIFT computes a 128-dimensional descriptor for each keypoint by dividing the region around the keypoint into 16x16 subregions, and computing a histogram of gradient orientations in each subregion. Each subregion contributes an 8-dimensional histogram, resulting in a 128-dimensional descriptor. The descriptor is normalized and thresholded to improve its robustness to changes in illumination.

Other methods such as Features from accelerated segment test (FAST) and Oriented FAST and Rotated BRIEF (ORB) can also be used to extract features.

2. **Feature Matching:** Match the features between consecutive images using the Brute Force Matcher algorithm. Brute force matching compares each descriptor in one image with every descriptor in the other image to find the best matches.

   Brute-Force Algorithm –

   We have two sets of features A = {a1, a2, ..., an} and B = {b1, b2, ..., bm}. Each feature is represented by a vector of features or descriptors, such as SIFT.

   For each feature a_i in set A, we calculate the Euclidean distance d_{ij} to every feature b_j in set B. The Euclidean distance is defined as:

$$d_{ij} = \sqrt{(a_{i1} - b_{j1})^2 + (a_{i2} - b_{j2})^2 + \ldots + (a_{id} - b_{jd})^2}$$

   where a_{i1}, a_{i2}, ..., a_{id} are the d-dimensional features of a_i
   $b_{j1}b_{j2}$, ..., b_{jd} are the d-dimensional features of b_j.

   We then find the closest match for each feature ai by selecting the feature b_j with the smallest distance d_{ij}. This can be represented as:

$$\min(d_{ij}) \text{ for } j \text{ in } \{1,2,...,m\}$$

   where $\min(d_{ij})$ is the minimum distance between a_i and all the features in set B.

   The brute force matcher returns the set of closest matches for each feature in set A.To improve the accuracy of the matching, a ratio test can be applied to remove ambiguous matches. The ratio test compares the distance of the best match to the distance of the second best match. If the ratio of these distances is below a certain threshold, the match is considered valid.

   Overall, the Brute Force Matcher algorithm is simple and straightforward, but can be computationally expensive for large sets of features. It is typically used as a baseline method for feature matching and can be improved upon with more advanced methods like FLANN (Fast Library for Approximate Nearest Neighbors).

3. **Computing Homographies:** Using RANSAC (Random Sample Consensus) algorithm to estimate the homographies between pairs of images. A homography is a transformation that maps points in one image to corresponding points in another image, taking into account the rotation and scaling between the two images. RANSAC is an iterative algorithm that finds the best homography by randomly selecting a set of correspondences and then checking if they are consistent with the model.
   (Theory of Homography is already described in the previous problem)

RANSAC Algorithm -

i. Randomly select a set of correspondences (matches) between two images.
ii. Compute the homography that maps the points in one image to the corresponding points in the other image.
iii. Check if the homography is consistent with the correspondences by computing the number of inliers (matches that agree with the homography) and outliers (matches that do not agree with the homography).
iv. Repeat steps for multiple iterations and select the homography with the largest number of inliers.

4. **Combining Images:** Finally, we combine the images together using the computed homographies. We apply the homography to each image, warping it to the same coordinate system as the first image. The warpPerspective function in python takes in an input image, a 3x3 transformation matrix, and the desired output image size. The transformation matrix is used to specify the desired perspective transformation. The function then applies the transformation matrix to the input image and generates a transformed output image of the specified size. The transformation matrix is the homography matrix we obtained. Then we simply concatenate the images horizontally to create a panoramic image.

Function Descriptions:

```
def load(image):
    img = cv2.cvtColor(cv2.imread(image), cv2.COLOR_BGR2RGB)    # Read the image and convert it to RGB
    img = cv2.resize(img, (int(img.shape[1] * 0.2),int(img.shape[0] * 0.2)), interpolation = cv2.INTER_AREA)
# Resize the image
    kp, des = cv2.SIFT_create().detectAndCompute(img, None)   # Extract the keypoints and descriptors
    return img, kp, des
```

- **def load(image)**: Function definition that takes a single argument **image**, which is the file path of the image to be loaded.
- **img = cv2.cvtColor(cv2.imread(image), cv2.COLOR_BGR2RGB)**: Reads the image from the file path specified by **image** using OpenCV's **imread** function, then converts the color space from BGR to RGB using OpenCV's **cvtColor** function. The resulting RGB image is stored in the variable **img**.
- **img = cv2.resize(img, (int(img.shape[1] * 0.2),int(img.shape[0] * 0.2)), interpolation = cv2.INTER_AREA):** Resizes the image to 20% of its original size using OpenCV's **resize** function. The new dimensions are computed by multiplying the original dimensions by 0.2. The **interpolation** parameter specifies the interpolation method used for resizing, in this case **cv2.INTER_AREA**.
- **kp, des = cv2.SIFT_create().detectAndCompute(img, None)**: Extracts the keypoints and descriptors from the resized image using the SIFT algorithm. First, an instance of the SIFT detector is created using **cv2.SIFT_create()**. Then, the **detectAndCompute** function is called on the SIFT detector with the resized image **img** and a **None** mask as arguments. The function returns two outputs: **kp**, which is a list of keypoints detected in the image, and **des**, which is a 2D numpy array of feature descriptors, with each row representing a single keypoint.
- **return img, kp, des**: Returns the resized image **img**, the list of keypoints **kp**, and the array of descriptors **des** as a tuple.

```
# Function to match the keypoints and find the homography
def matching(img1, img2, kp1, kp2, des1, des2):
    matches = cv2.BFMatcher().knnMatch(des1, des2, k=2) # Match keypoints using the brute-force matcher
    good = [m for m,n in matches if m.distance < 0.75*n.distance]   # Select the good matches using the ratio
test
    draw_params = dict(matchColor=(0,255,0), singlePointColor=None, flags=2)    # Draw the matches
    img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)    # Extract the matched
keypoints
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)
    H = findHomography(dst_pts, src_pts)
    return img_matches, H
```

- **matches = cv2.BFMatcher().knnMatch(des1, des2, k=2)**: This command matches the keypoints in the two images using the brute-force matcher. It takes as input the descriptors for each image (des1 and des2) and returns a list of matches for each keypoint. The k parameter sets the number of nearest neighbors to return, which is set to 2 here.

- **good = [m for m,n in matches if m.distance < 0.75*n.distance]**: This command selects the good matches from the list of matches returned by the brute-force matcher. It does this by using the ratio test, which checks that the distance between the two closest matches is less than 0.75 times the distance between the two farthest matches. This helps to filter out false matches.
- **draw_params = dict(matchColor=(0,255,0), singlePointColor=None, flags=2)**: This command defines the parameters for drawing the matches between the two images. The matchColor parameter sets the color of the lines that connect the matching keypoints, while singlePointColor sets the color of the keypoints themselves. The flags parameter specifies additional drawing options.
- **img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)**: This command draws the matches between the two images using the parameters specified in draw_params. It takes as input the two images (img1 and img2), their corresponding keypoints (kp1 and kp2), the list of good matches (good), and the drawing parameters.
- **src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)**: This command extracts the matched keypoints from the first image (img1). It does this by taking the index of each good match (m.queryIdx) and retrieving the corresponding keypoint (kp1) from the list of keypoints for that image. The result is a list of points (in pixel coordinates) that correspond to the matched keypoints.
- **dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)**: This command extracts the matched keypoints from the second image (img2) in a similar way to the previous command.
- **H = findHomography(dst_pts, src_pts)**: This command finds the homography matrix (H) that maps the points in the second image (dst_pts) to their corresponding points in the first image (src_pts). The homography matrix is computed using the RANSAC algorithm, which helps to filter out outliers and improve the accuracy of the mapping.
- **return img_matches, H**: This command returns the image with the drawn matches (img_matches) and the homography matrix (H) to the calling function.

```
# Function to find homography using RANSAC algorithm
def findHomography(src_pts, dst_pts, num_iter=100, num_pts=4, threshold=5):
    best_H, best_count = None, 0    # Initialize the best homography and the inlier count
    src_pts, dst_pts = np.hstack((src_pts.reshape(-1,2), np.ones((len(src_pts), 1)))),
np.hstack((dst_pts.reshape(-1,2), np.ones((len(dst_pts), 1))))   # Convert the input points to homogeneous
coordinates

    # Perform RANSAC iterations
    for i in range(num_iter):
        idx = np.random.choice(len(src_pts), num_pts, replace=False)    # Randomly sample num_pts points
        src_sample, dst_sample = src_pts[idx, :], dst_pts[idx, :]

        # Construct the matrix A
        A = np.asarray([[x, y, 1, 0, 0, 0, -u*x, -u*y, -u] for (x, y, _), (u, v, _) in zip(src_sample,
dst_sample)] + [[0, 0, 0, x, y, 1, -v*x, -v*y, -v] for (x, y, _), (u, v, _) in zip(src_sample, dst_sample)])
        _, _, V = np.linalg.svd(A)  # Perform SVD on matrix A
        H = V[-1, :].reshape(3, 3)  # Extract the homography from the last row of V
        count = sum(np.linalg.norm(np.dot(H, src_pts[j]) / np.dot(H, src_pts[j])[2] - dst_pts[j]) < threshold
for j in range(len(src_pts)))    # Count the number of inliers for the current homography
        if count > best_count: best_H, best_count = H, count   # Update the best homography and the inlier
count
    return best_H
```

- **def findHomography(src_pts, dst_pts, num_iter=100, num_pts=4, threshold=5):** Defines a function called **findHomography** that takes as input two sets of corresponding points (**src_pts** and **dst_pts**), the number of RANSAC iterations to perform (**num_iter**), the number of points to randomly sample in each iteration (**num_pts**), and the inlier threshold distance (**threshold**).
- **best_H, best_count = None, 0**: Initializes the best homography matrix (**best_H**) and the number of inliers for that matrix (**best_count**) to **None** and **0**, respectively.
- **src_pts, dst_pts = np.hstack((src_pts.reshape(-1,2), np.ones((len(src_pts), 1)))), np.hstack((dst_pts.reshape(-1,2), np.ones((len(dst_pts), 1))))**: Converts the input points to homogeneous coordinates by stacking a column of ones to the end of each row and reshaping the resulting array to have shape **(n, 3)**.
- **for i in range(num_iter):** Loops **num_iter** times to perform RANSAC iterations.
- **idx = np.random.choice(len(src_pts), num_pts, replace=False)**: Randomly samples **num_pts** indices without replacement from the set of indices of the input points.

- **src_sample, dst_sample = src_pts[idx, :], dst_pts[idx, :]**: Selects the corresponding points at the sampled indices from the input sets of points.
- **A = np.asarray([[x, y, 1, 0, 0, 0, -u*x, -u*y, -u] for (x, y, _), (u, v, _) in zip(src_sample, dst_sample)] + [[0, 0, 0, x, y, 1, -v*x, -v*y, -v] for (x, y, _), (u, v, _) in zip(src_sample, dst_sample)])**: Constructs the matrix **A** by stacking the rows corresponding to each pair of corresponding points selected in the previous step. Each row has the form **[x, y, 1, 0, 0, 0, -u*x, -u*y, -u]** or **[0, 0, 0, x, y, 1, -v*x, -v*y, -v]**, where **(x, y)** and **(u, v)** are the homogeneous coordinates of a pair of corresponding points.
- **_, _, V = np.linalg.svd(A)**: Performs singular value decomposition (SVD) on **A** and returns the singular values in a vector _ and the right singular vectors in the columns of the matrix **V**.
- **H = V[-1, :].reshape(3, 3)**: Extracts the homography matrix from the last row of **V** and reshapes it to have shape **(3, 3)**.
- **count = sum(np.linalg.norm(np.dot(H, src_pts[j]) / np.dot(H, src_pts[j])[2] - dst_pts[j]) < threshold for j in range(len(src_pts))):** For each iteration of RANSAC, this line computes the number of inliers for the current homography H. It does this by first computing the transformed point np.dot(H, src_pts[j]), then normalizing it by dividing by its third coordinate (i.e., np.dot(H, src_pts[j])[2]). This results in a homogeneous coordinate representation of the transformed point. The code then computes the Euclidean distance between this transformed point and the corresponding point in dst_pts. If this distance is less than threshold value, the point is considered an inlier. The sum function counts the total number of inliers over all src_pts.
- **if count > best_count: best_H, best_count = H, count: This** line updates the best homography and the number of inliers found so far. If the current homography H has more inliers than the previous best homography, best_H and best_count are updated accordingly.
- **return best_H:** The best homography is returned as the output of the RANSAC algorithm.

```
def trim(frame):
    while not np.any(frame[0]): frame = frame[1:]
    while not np.any(frame[-1]): frame = frame[:-2]
    while not np.any(frame[:,0]): frame = frame[:,1:]
    while not np.any(frame[:,-1]): frame = frame[:,:-2]
    return frame
```

- **def trim(frame):** Defines a function called **trim** that takes one input argument called **frame**.
- **while not np.any(frame[0]): frame = frame[1:]**: This loop removes all rows from the top of the frame that contain only black pixels. The **np.any** function checks if any element of the first row of the **frame** matrix is non-zero. If there are no non-zero elements, then the loop continues to remove the first row of the matrix until a row with non-zero elements is found.
- **while not np.any(frame[-1]): frame = frame[:-2]**: This loop removes all rows from the bottom of the frame that contain only black pixels. The **np.any** function checks if any element of the last row of the **frame** matrix is non-zero. If there are no non-zero elements, then the loop continues to remove the last row of the matrix until a row with non-zero elements is found.
- **while not np.any(frame[:,0]): frame = frame[:,1:]**: This loop removes all columns from the left of the frame that contain only black pixels. The **np.any** function checks if any element of the first column of the **frame** matrix is non-zero. If there are no non-zero elements, then the loop continues to remove the first column of the matrix until a column with non-zero elements is found.
- **while not np.any(frame[:,-1]): frame = frame[:,:-2]**: This loop removes all columns from the right of the frame that contain only black pixels. The **np.any** function checks if any element of the last column of the **frame** matrix is non-zero. If there are no non-zero elements, then the loop continues to remove the last column of the matrix until a column with non-zero elements is found.
- **return frame**: Returns the trimmed frame.

```
# Function to stitch the images
def stitch_images(img2, img1, H):
    width = img2.shape[1] + img1.shape[1]
    height = max(img2.shape[0], img1.shape[0])
    stitched_img = cv2.warpPerspective(img2, H, (width,height))  # Warp the image using the homography matrix
    stitched_img[0:img1.shape[0], 0:img1.shape[1]] = img1
    return stitched_img
```

- **width = img2.shape[1] + img1.shape[1]**: This line calculates the width of the output image by adding the widths of **img2** and **img1**.

- **height = max(img2.shape[0], img1.shape[0])**: This line calculates the height of the output image by taking the maximum of the heights of **img2** and **img1**.
- **stitched_img = cv2.warpPerspective(img2, H, (width,height))**: This line uses the **warpPerspective** function from the OpenCV library to warp **img2** using the homography matrix **H**. The resulting image is stored in **stitched_img**.
- **stitched_img[0:img1.shape[0], 0:img1.shape[1]] = img1**: This line pastes **img1** onto the left side of **stitched_img**, with its top-left corner aligned with the top-left corner of **stitched_img**.
- **return stitched_img**: This line returns the final stitched image.

Full Code:

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

# Function to load the images and extract the keypoints and descriptors
def load(image):
    img = cv2.cvtColor(cv2.imread(image), cv2.COLOR_BGR2RGB)    # Read the image and convert it to RGB
    img = cv2.resize(img, (int(img.shape[1] * 0.2),int(img.shape[0] * 0.2)), interpolation = cv2.INTER_AREA)
# Resize the image
    kp, des = cv2.SIFT_create().detectAndCompute(img, None)   # Extract the keypoints and descriptors
    return img, kp, des

# Function to match the keypoints and find the homography
def matching(img1, img2, kp1, kp2, des1, des2):
    matches = cv2.BFMatcher().knnMatch(des1, des2, k=2) # Match keypoints using the brute-force matcher
    good = [m for m,n in matches if m.distance < 0.75*n.distance]   # Select the good matches using the ratio
test
    draw_params = dict(matchColor=(0,255,0), singlePointColor=None, flags=2)    # Draw the matches
    img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)    # Extract the matched
keypoints
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)
    H = findHomography(dst_pts, src_pts)
    return img_matches, H

# Function to find homography using RANSAC algorithm
def findHomography(src_pts, dst_pts, num_iter=100, num_pts=4, threshold=5):
    best_H, best_count = None, 0    # Initialize the best homography and the inlier count
    src_pts, dst_pts = np.hstack((src_pts.reshape(-1,2), np.ones((len(src_pts), 1)))),
np.hstack((dst_pts.reshape(-1,2), np.ones((len(dst_pts), 1))))   # Convert the input points to homogeneous
coordinates

    # Perform RANSAC iterations
    for i in range(num_iter):
        idx = np.random.choice(len(src_pts), num_pts, replace=False)    # Randomly sample num_pts points
        src_sample, dst_sample = src_pts[idx, :], dst_pts[idx, :]

        # Construct the matrix A
        A = np.asarray([[x, y, 1, 0, 0, 0, -u*x, -u*y, -u] for (x, y, _), (u, v, _) in zip(src_sample,
dst_sample)] + [[0, 0, 0, x, y, 1, -v*x, -v*y, -v] for (x, y, _), (u, v, _) in zip(src_sample, dst_sample)])
        _, _, V = np.linalg.svd(A)  # Perform SVD on matrix A
        H = V[-1, :].reshape(3, 3)  # Extract the homography from the last row of V
        count = sum(np.linalg.norm(np.dot(H, src_pts[j]) / np.dot(H, src_pts[j])[2] - dst_pts[j]) < threshold
for j in range(len(src_pts)))    # Count the number of inliers for the current homography
        if count > best_count: best_H, best_count = H, count   # Update the best homography and the inlier
count
    return best_H

# Function to trim the black borders of the stitched image
def trim(frame):
    while not np.any(frame[0]): frame = frame[1:]
    while not np.any(frame[-1]): frame = frame[:-2]
    while not np.any(frame[:,0]): frame = frame[:,1:]
    while not np.any(frame[:,-1]): frame = frame[:,:-2]
    return frame
```

```
# Function to stitch the images
def stitch_images(img2, img1, H):
    width = img2.shape[1] + img1.shape[1]
    height = max(img2.shape[0], img1.shape[0])
    stitched_img = cv2.warpPerspective(img2, H, (width,height))  # Warp the image using the homography matrix
    stitched_img[0:img1.shape[0], 0:img1.shape[1]] = img1
    return stitched_img

img1, kp1, des1 = load('image_1.jpg')
img2, kp2, des2 = load('image_2.jpg')
img3, kp3, des3 = load('image_3.jpg')
img4, kp4, des4 = load('image_4.jpg')

# Draw the matches between the consecutive images
img_matches12, H12 = matching(img1, img2, kp1, kp2, des1, des2)
img_matches23, H23 = matching(img2, img3, kp2, kp3, des2, des3)
img_matches34, H34 = matching(img3, img4, kp3, kp4, des3, des4)

# Display the matched features
plt.figure(figsize=(20,10))
for i, img_match, title in zip(range(1,4), [img_matches12, img_matches23, img_matches34], ['Matches between
Image 1 and Image 2', 'Matches between Image 2 and Image 3', 'Matches between Image 3 and Image 4']):
    plt.subplot(1,3,i),plt.imshow(img_match),plt.title(title)
plt.show()

# Stitch the images
stitched_img12 = stitch_images(img2, img1, H12)
stitched_img34 = stitch_images(img4, img3, H34)
stitched_img1234 = stitch_images(trim(stitched_img34), trim(stitched_img12), np.dot(H23, H12))

# Display the final stitched image
plt.imshow(trim(stitched_img1234))
plt.show()
```

Matches between Images:

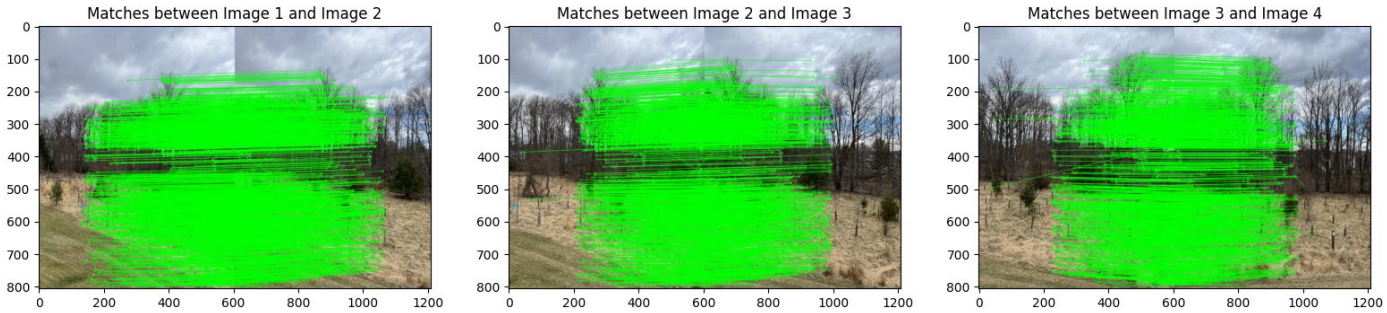

Figure 4 Matches between images

Final Stitched Panoramic Image:

Figure 5 Stitched Image

Pipeline:

- Import required libraries: cv2, numpy, and matplotlib.pyplot.
- Define a function named load that takes an image as input, reads and converts the image to RGB format, resizes it to 20% of its original size, extracts the keypoints and descriptors of the image using the SIFT algorithm, and returns the image, keypoints, and descriptors.
- Define a function named matching that takes two images and their respective keypoints and descriptors as input. It matches the keypoints using the brute-force matcher, selects the good matches using the ratio test, draws the matches, extracts the matched keypoints, finds the homography between the two images, and returns the image with drawn matches and the homography matrix.
- Define a function named findHomography that takes the source and destination points, number of iterations, number of points, and threshold as input. It initializes the best homography and the inlier count, converts the input points to homogeneous coordinates, performs RANSAC iterations, randomly samples num_pts points, constructs matrix A, performs SVD on matrix A, extracts the homography from the last row of V, counts the number of inliers for the current homography, and updates the best homography and the inlier count. Finally, it returns the best homography.
- Define a function named trim that takes an image as input. It removes the black borders of the stitched image and returns the trimmed image.
- Define a function named stitch_images that takes two images and their homography matrix as input. It calculates the width and height of the stitched image, warps the second image using the homography matrix, pastes the first image on top of the warped second image, and returns the stitched image.
- Load four images and extract their keypoints and descriptors using the load function.
- Call the matching function for each pair of consecutive images and get the image with drawn matches and the homography matrix for each pair.
- Display the matched features using matplotlib.pyplot for each pair of consecutive images.
- Warp the images using the homography matrices and stitch them together using the stitch_images function.
- Display the stitched image.