ENPM673

PERCEPTION FOR AUTONOMOUS ROBOTS

PROJECT-2 REPORT

VENKATA SAI SRICHARAN KASTURI

UID:119444788

1Q)

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.

Problems encountered:

1) Faced issues while producing hough lines.
2) Extracting the intersection points(corners), since the multiple points are detected.

Math:

Hough transform algorithm

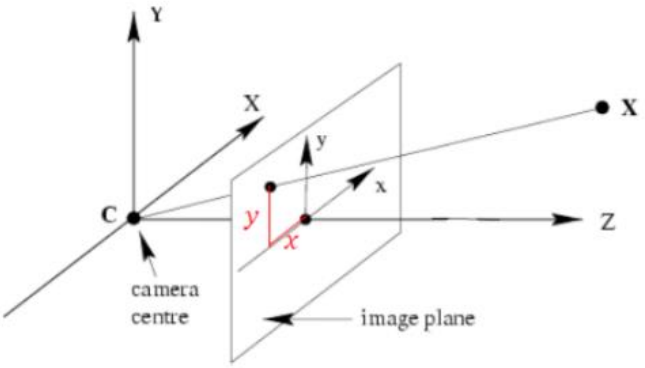
Typically use a different parameterization

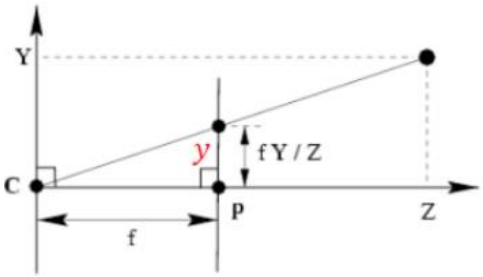
$$d = xcos\theta + ysin\theta$$

- d is the perpendicular distance from the line to the origin
- θ is the angle this perpendicular makes with the x axis
- Why?

Basic Hough transform algorithm

1. Initialize H[d, θ]=0 New discretized image in d, θ domain
2. for each edge point I[x,y] in the image
   for θ = 0 to 180
   $d = xcos\theta + ysin\theta$
   H[d, θ] += 1
3. Find the value(s) of (d, θ) where H[d, θ] is maximum
4. The detected line in the image is given by $d = xcos\theta + ysin\theta$





That gives: $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Calibration matrix } \boldsymbol{K}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$

In short $\boldsymbol{x} = \boldsymbol{K}\,\tilde{\boldsymbol{X}}$ (here $\tilde{\boldsymbol{X}}$ means inhomogeneous coordinates)

Intrinsic Camera Calibration means we know $\boldsymbol{K}$ (we do that later)

We can go from image points into the 3D world: $\boxed{\tilde{\boldsymbol{X}} = \boldsymbol{K}^{-1}\,\boldsymbol{x}}$

Adding principal point into $\boldsymbol{K}$

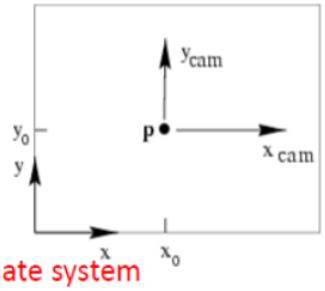
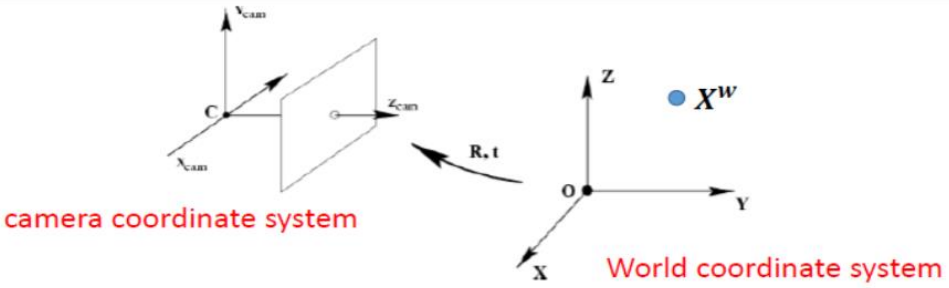


Image coordinate system

Principal point (p_x, p_y)

Projection with principal point : $y = f\frac{Y}{Z} + p_y = \frac{fY+Zp_y}{Z}$ and $x = f\frac{X}{Z} + p_x = \frac{fX+Zp_x}{Z}$

That gives: $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$



camera coordinate system

World coordinate system

Given a 3D homogenous point X^W in world coordinate system

1) Translate from world to camera coordinate system:

$$\tilde{X}^{c\prime} = \tilde{X}^{w} - \tilde{C}$$

$$\tilde{X}^{c\prime} = \underbrace{(I_{3\times3} \mid -\tilde{C})}_{3\times4 \text{ matrix}} X^{w} \quad \text{where } I_{3\times3} \text{ is 3x3 identity matrix}$$

Last column is translation

2) Rotate world coordinate system into camera coordinate system

$$\tilde{X}^{c} = R\,(I_{3\times3} \mid -\tilde{C})\,X^{w}$$

3) Apply camera matrix

$$x = K\,R\,(I_{3\times3} \mid -\tilde{C})\,X$$

Tilda means inhomogeneous vector

Solving for Homographies

$$\begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} \cong \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Camera coordinates

$$x_i' = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + h_{22}}$$

$$y_i' = \frac{h_{10}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + h_{22}}$$

$$\begin{aligned} x_i'(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{00}x_i + h_{01}y_i + h_{02} \\ y_i'(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{10}x_i + h_{11}y_i + h_{12} \end{aligned}$$

$$\begin{bmatrix} x_i & y_i & 1 & 0 & 0 & 0 & -x_i'x_i & -x_i'y_i & -x_i' \\ 0 & 0 & 0 & x_i & y_i & 1 & -y_i'x_i & -y_i'y_i & -y_i' \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Camera pose estimation

Assume all points lie in one plane with Z=0:

$$\boldsymbol{X} = (X, Y, 0, 1)$$

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{PX} \\ &= \boldsymbol{K}\left[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{r}_3\boldsymbol{t}\right]\begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} \\ &= \boldsymbol{K}\left[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}\right]\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \\ &= \boldsymbol{H}\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \end{aligned}$$

$$\boldsymbol{H} = \lambda\boldsymbol{K}\left[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}\right]$$

$$\boldsymbol{K}^{-1}\boldsymbol{H} = \lambda\left[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}\right]$$

–r_1 and r_2 are unit vectors ⇨ find lambda

–Use this to compute t

–Rotation matrices are orthogonal ⇨ find r3

$$\boldsymbol{P} = \boldsymbol{K}\left[\begin{array}{cccc} \boldsymbol{r}_1 & \boldsymbol{r}_2 & (\boldsymbol{r}_1 \times \boldsymbol{r}_2) & \boldsymbol{t} \end{array}\right]$$

Steps to be followed:

1)Open video file and read frame by frame.

2)Apply blurring to the current frame using a 10x10 kernel.

3)Convert the blurred frame from BGR color space to HSV color space.

4)Define a range of white color and create a mask using cv.inRange().

5)Detect edges in the masked image using cv.Canny() function.

6)Apply Hough transform to find the lines in the edge image using the hough_lines() function.

7)Find the four intersection points of the lines detected in the previous step using get_intersection() function.

8)Store these points as corners of the object in the image.

9)Use the corners of the object to calculate homography using find_homography() function.

10)Use the homography matrix to calculate the pose of the object in the image using find_rotation_translation() function.

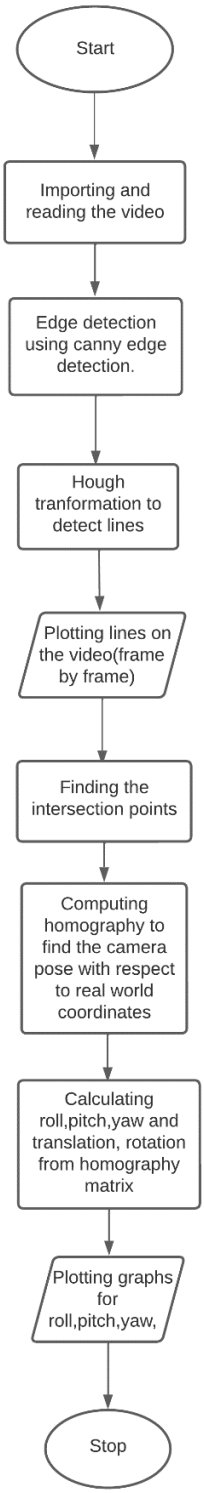
11)Extract the roll, pitch, and yaw values from the pose matrix and store them in the respective lists.

12)Extract the x, y, and z values from the pose matrix and store them in the respective lists.

13)Display the frame with the detected lines and intersection points.

14)Continue the loop until all frames have been processed.

15)Release the video object and close all windows.


Start
Importing and reading the video
Edge detection using canny edge detection.
Hough tranformation to detect lines
Plotting lines on the video(frame by frame)
Finding the intersection points
Computing homography to find the camera pose with respect to real world coordinates
Calculating roll,pitch,yaw and translation, rotation from homography matrix
Plotting graphs for roll,pitch,yaw,
Stop

Pipeline

Code:

```
import numpy as np
from matplotlib import pyplot as plt
import cv2 as cv,os
import math
CURRENT_DIR = os.path.dirname(__file__)
path_video=os.path.join(CURRENT_DIR,'project2.avi')
video= cv.VideoCapture(path_video)

coordinates = np.array([[0, 0], [0, 27.9], [21.6,27.9], [21.6, 0]])

def find_homography(p1, p2):

    if p1.shape[0] < 4 or p2.shape[0] < 4:
        raise ValueError("Not enough points to find homography")


    A = []
    for i in range(p1.shape[0]):
        x, y = p1[i]
        u, v = p2[i]
        A.append([-x, -y, -1, 0, 0, 0, x*u, y*u, u])
        A.append([0, 0, 0, -x, -y, 1, x*v, y*v, v])
    A = np.array(A)

    # SVD
    _, _, Vt = np.linalg.svd(A)
    H = Vt[-1].reshape(3, 3)
    H /= H[2, 2]

    return H

def find_rotation_translation(H, K):

    K_inv = np.linalg.inv(K)
# Vectors
    r1 = H[:, 0:1] / np.linalg.norm(H[:, 0])
    r2 = H[:, 1:2] / np.linalg.norm(H[:, 1])
    r3 = np.cross(r1.T, r2.T).T

# Rotational matrix
    R = np.hstack([r1,r2,r3])

# Translation
```

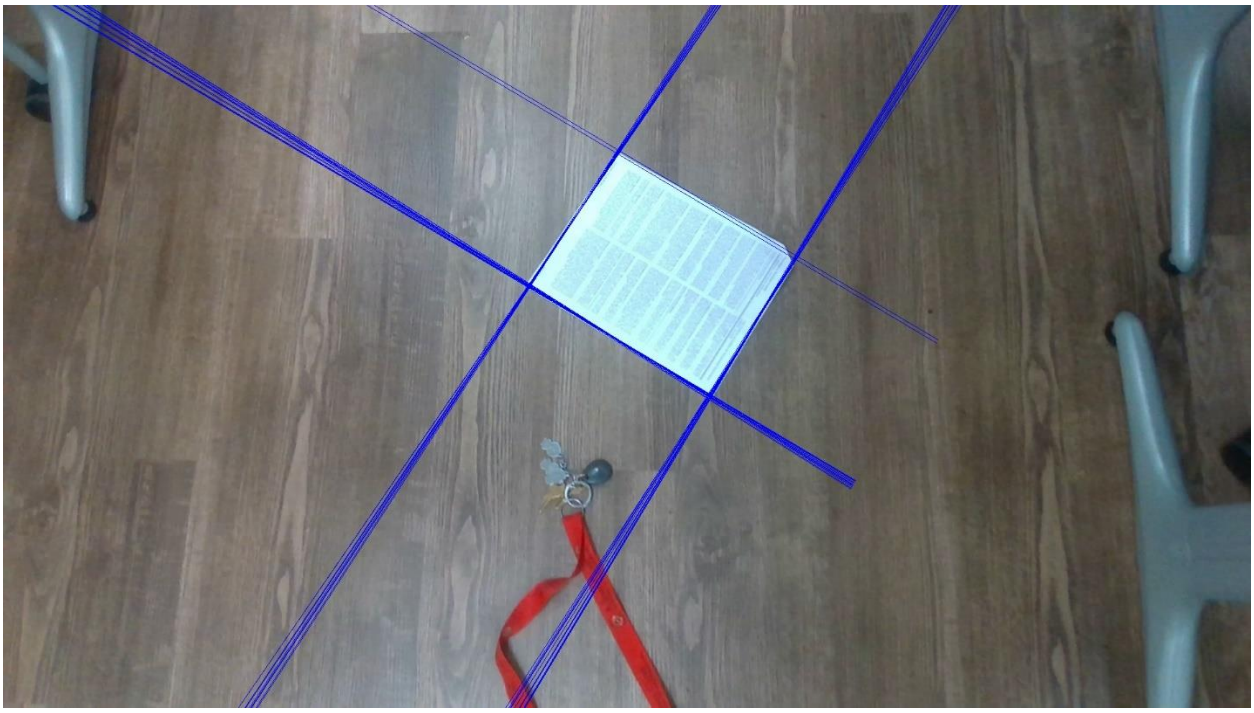
```
    t = K_inv @ H[:, 2] / np.linalg.norm(K_inv @ H[:, 2])

# Pose
    P = np.hstack([R, t.reshape(-1, 1)])

    return P

def hough_lines(edges):

    theta = np.arange(-90, 90)
    cos_t = np.cos(theta)
    sin_t = np.sin(theta)
    a=edges.shape[0]
    b=edges.shape[1]
    c=int(np.ceil(np.sqrt(a**2+b**2)))
    perpendicular_range=np.arange(-c,c+1,1)
    accumulator = np.zeros((len(perpendicular_range), len(theta)))
    y_indexes, x_indexes = np.nonzero(edges)

    for i in range(len(x_indexes)):
        x = x_indexes[i]
        y = y_indexes[i]

        for j in range(len(theta)):
            rho = int(round(x * cos_t[j] + y * sin_t[j])) + c
            accumulator[rho, j] += 1
# Maximum votes
    rhos, thetas = np.nonzero(accumulator >=55)
# Conversion
    lines = []
    for i in range(len(rhos)):
        rho = perpendicular_range[rhos[i]]
        angle = theta[thetas[i]]
        lines.append((rho, angle))


# Polar to cartesian
    cartesian_lines = []
    for line in lines:
        rho, theta = line
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * (rho)
        y0 = b * (rho)
        x1 = int(x0 + 1500 * (-b))
```

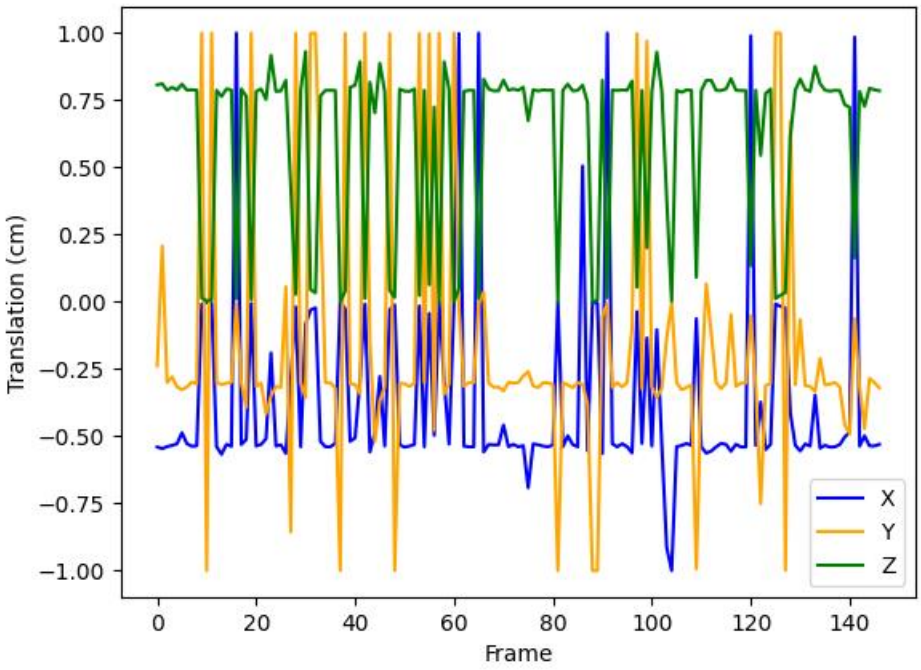
```
        y1 = int(y0 + 1500 * (a))
        x2 = int(x0 - 1500* (-b))
        y2 = int(y0 - 1500* (a))
        cartesian_lines.append([(x1, y1), (x2, y2)])
        cv.line(frame,(x1,y1),(x2,y2),(255,0,0),1)
# Intersection points
    intersections = []
    for i in range(len(cartesian_lines)):
        for j in range(i+1, len(cartesian_lines)):
            line1 = cartesian_lines[i]
            line2 = cartesian_lines[j]
            intersection = get_intersection(line1, line2)
            if intersection is not None:
                intersections.append(intersection)


    intersections.sort()
    corners = np.array(intersections)

    return corners[:4,:]

def get_intersection(line1, line2):

    x1, y1 = line1[0]
    x2, y2 = line1[1]
    x3, y3 = line2[0]
    x4, y4 = line2[1]

    denominator = (x1-x2)*(y3-y4) - (y1-y2)*(x3-x4)
    if denominator == 0:

        return None
    else:
        px = ((x1*y2-y1*x2)*(x3-x4)-(x1-x2)*(x3*y4-y3*x4))/denominator
        py = ((x1*y2-y1*x2)*(y3-y4)-(y1-y2)*(x3*y4-y3*x4))/denominator
        return (int(px), int(py))

roll_list, pitch_list, yaw_list = [], [], []
x_list, y_list, z_list = [], [], []

while(video.isOpened()):

    ret,frame=video.read()
    if ret==True:
```

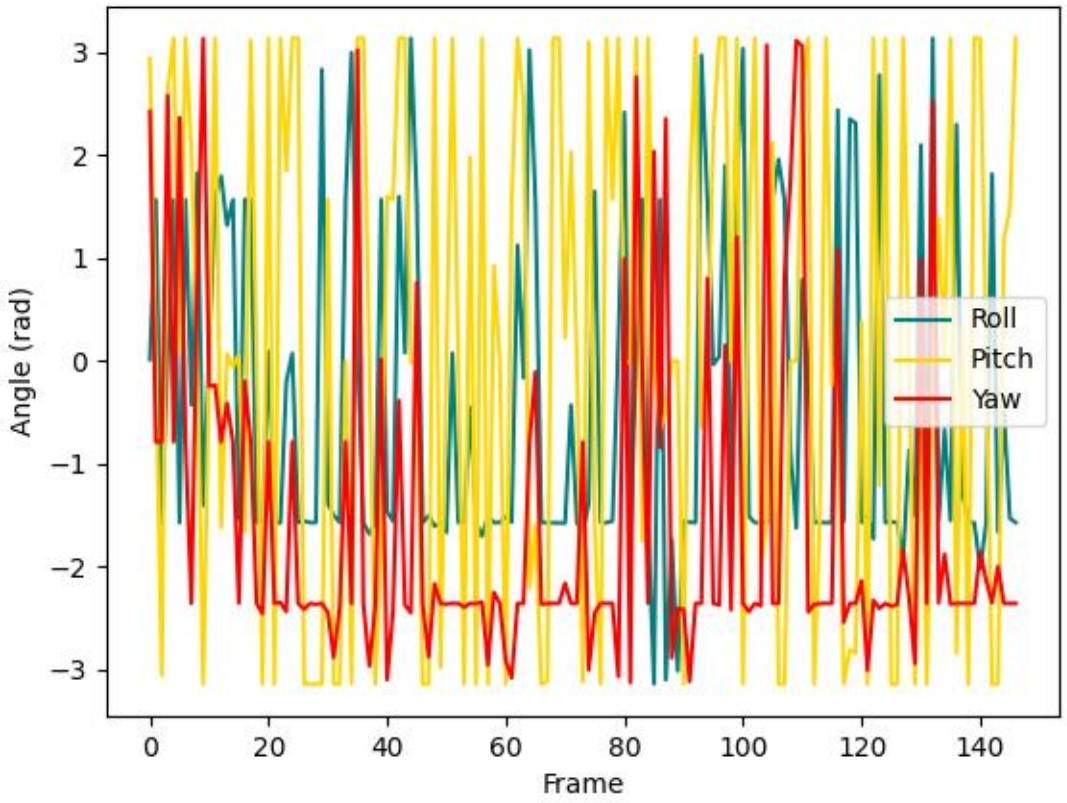
```
        blur = cv.blur(frame,(10,10))
        hsv=cv.cvtColor(blur,cv.COLOR_BGR2HSV)
        whitepaper1=np.array([0,0,235])
        whitepaper2=np.array([180,100,255])
        mask=cv.inRange(hsv,whitepaper1,whitepaper2)
        edge=cv.Canny(mask,10,200)
        g=hough_lines(edge)
        T=find_rotation_translation(find_homography(g,coordinates),K =
np.array([[1.38e+03, 0, 9.46e+02], [0, 1.38e+03, 5.27e+02], [0, 0, 1]]))
        R=T[:,:3]
        t=T[:,3]

        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)]]
        tx, ty, tz = t

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

        cv.imshow('lines',frame)
        # cv.imshow('frame',edge)
        key =cv.waitKey(1)
        if key==ord('q'):
            break
    else:
        break
video.release()
cv.destroyAllWindows()

plt.plot(roll_list, label='Roll',color='Teal')
plt.plot(pitch_list, label='Pitch',color='Gold')
plt.plot(yaw_list, label='Yaw',color='Red')
plt.legend()
plt.xlabel('Frame')
plt.ylabel('Angle (rad)')
plt.show()

plt.plot(x_list, label='X',color='Blue')
plt.plot(y_list, label='Y',color='Orange')
plt.plot(z_list, label='Z',color='Green')
plt.legend()
```

```
plt.xlabel('Frame')
plt.ylabel('Translation (cm)')
plt.show()
```

Output:




Translation (cm)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100
120
140
Frame
X
Y
Z


Angle (rad)
3
2
1
0
−1
−2
−3
0
20
40
60
80
100
120
140
Frame
Roll
Pitch
Yaw

Q2)

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.

Problems encountered:

1)Stitching the images. Blank spaces occurred for the final panorama.

2)Pano1 in the final result is not being displayed correctly.(Note: Using the built in homography function the result is good)

Math:

Solving for Homographies

$$\begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} \cong \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Camera coordinates

$$x_i' = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + h_{22}}$$
$$y_i' = \frac{h_{10}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + h_{22}}$$

$$x_i'(h_{20}x_i + h_{21}y_i + h_{22}) = h_{00}x_i + h_{01}y_i + h_{02}$$
$$y_i'(h_{20}x_i + h_{21}y_i + h_{22}) = h_{10}x_i + h_{11}y_i + h_{12}$$

$$\begin{bmatrix} x_i & y_i & 1 & 0 & 0 & 0 & -x_i'x_i & -x_i'y_i & -x_i' \\ 0 & 0 & 0 & x_i & y_i & 1 & -y_i'x_i & -y_i'y_i & -y_i' \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

2/23/2023

Steps to be followed:

1)Import necessary libraries: Begin by importing the necessary libraries, such as OpenCV, NumPy, and Matplotlib. This will allow you to perform the required operations.

2)Load the images: Load the images that you want to stitch together. In this case, four images are loaded and stored in the image_paths list.

3)Resize the images: Resize the images to ensure that they have a uniform size. In this code, the images are resized to a maximum dimension of 500 pixels while maintaining the aspect ratio.

4)Detect keypoints and descriptors: Use SIFT (Scale-Invariant Feature Transform) to detect keypoints and compute descriptors for each image.

5)Match the keypoints: Use FLANN (Fast Library for Approximate Nearest Neighbors) to match the keypoints between the images.

6)Draw matches: Draw the matched keypoints between the images using the drawMatchesKnn function.

7)Compute homography: Compute the homography matrix for each pair of images using the matches between them.

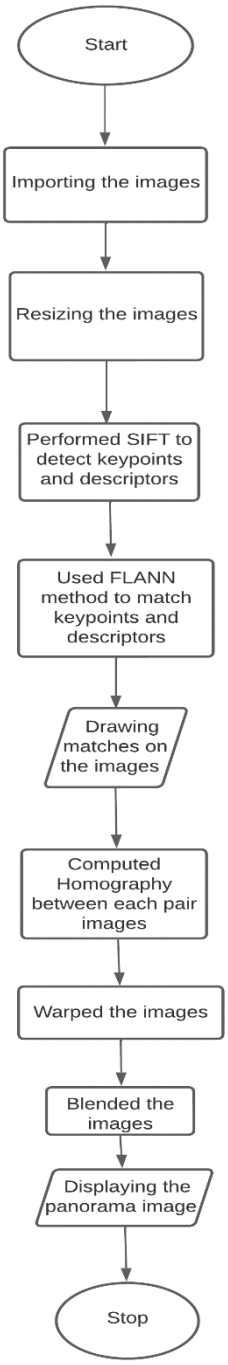
8)Compute the size of the output image: Use the homography matrix to compute the size of the output image.

9)Warp the images: Warp the images using the computed homography matrix.

10)Blend the images: Blend the warped images together to create a seamless panorama.

11)Display the final panorama: Display the final panorama using Matplotlib.

12)Save the output: Save the final panorama to a file for later use.


Start
Importing the images
Resizing the images
Performed SIFT to detect keypoints and descriptors
Used FLANN method to match keypoints and descriptors
Drawing matches on the images
Computed Homography between each pair images
Warped the images
Blended the images
Displaying the panorama image
Stop

Pipeline

Code:

```
import cv2 as cv,numpy as np,os
import matplotlib.pyplot as plt
sift = cv.SIFT_create()

CURRENT_DIR = os.path.dirname(__file__)

# Importing images
image_paths=[]

for i in range(1,5):
    image_paths.append(os.path.join(CURRENT_DIR, f'image_{i}.jpg'))

img1 = cv.imread(image_paths[0], cv.IMREAD_GRAYSCALE)
h, w = img1.shape[:2]
ratio = float(500) / max(h, w)
new_h = int(ratio * h)
new_w = int(ratio * w)

# Resize the image
img1 = cv.resize(img1, (new_w, new_h))

img2 = cv.imread(image_paths[1], cv.IMREAD_GRAYSCALE)
# Resize the image
img2 = cv.resize(img2, (new_w, new_h))

img3 = cv.imread(image_paths[2], cv.IMREAD_GRAYSCALE)
# Resize the image
img3 = cv.resize(img3, (new_w, new_h))

img4 = cv.imread(image_paths[3], cv.IMREAD_GRAYSCALE)
# Resize the image
img4 = cv.resize(img4, (new_w, new_h))

def trimmer(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
```

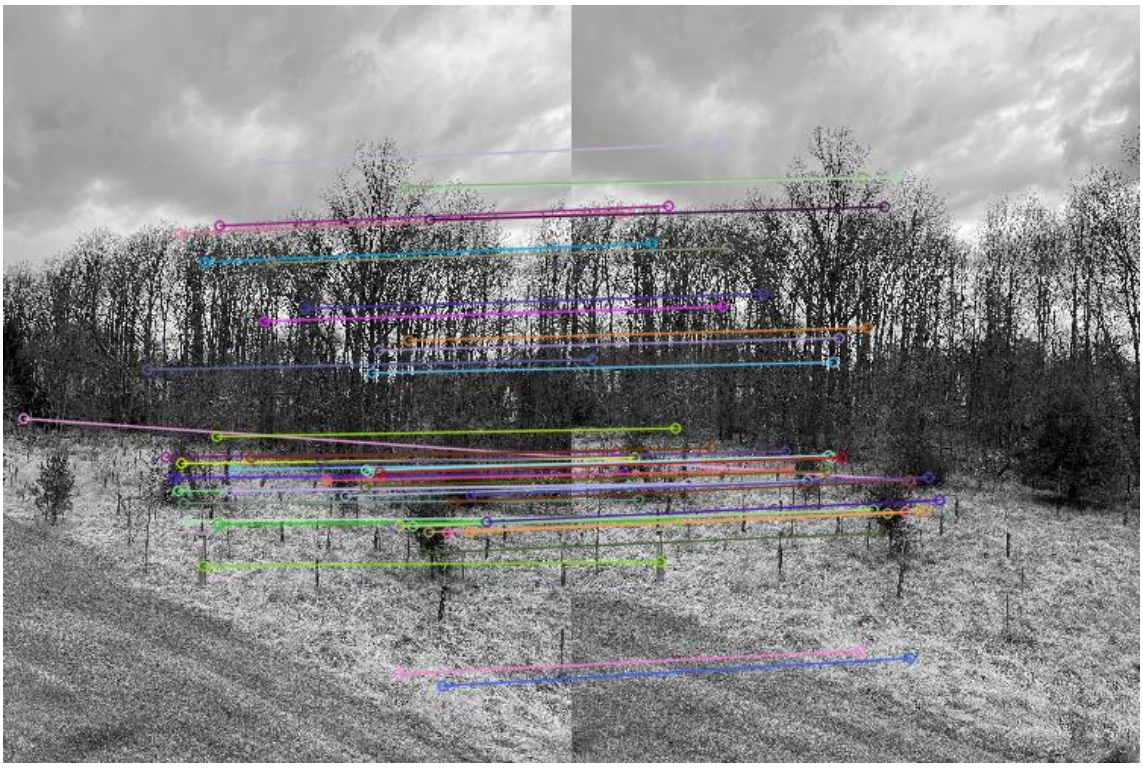
```
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)
kp3, des3 = sift.detectAndCompute(img3, None)
kp4, des4 = sift.detectAndCompute(img4, None)

FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)

flann = cv.FlannBasedMatcher(index_params, search_params)

matches12 = flann.knnMatch(des1, des2, k=2)
matches23 = flann.knnMatch(des2, des3, k=2)
matches34 = flann.knnMatch(des3, des4, k=2)

good12 = []
if len(matches12) > 0:
    for m, n in matches12:
        if m.distance < 0.7 * n.distance:
            good12.append([m])

good23 = []
if len(matches23) > 0:
    for m, n in matches23:
        if m.distance < 0.7 * n.distance:
            good23.append([m])

good34 = []
if len(matches34) > 0:
    for m, n in matches34:
        if m.distance < 0.7 * n.distance:
            good34.append([m])

img_matches12 = cv.drawMatchesKnn(img1, kp1, img2, kp2, good12, None,
flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
img_matches23 = cv.drawMatchesKnn(img2, kp2, img3, kp3, good23, None,
flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
img_matches34 = cv.drawMatchesKnn(img3, kp3, img4, kp4, good34, None,
flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

cv.imshow('Matches 1-2', img_matches12)
cv.imshow('Matches 2-3', img_matches23)
cv.imshow('Matches 3-4', img_matches34)
cv.waitKey(0)
```

```
MIN_MATCH_COUNT = 10




def compute_homography(kp1, kp2,matches):
    if len(matches) < MIN_MATCH_COUNT:
        return None

    src_pts = np.float32([ kp1[m[0].queryIdx].pt for m in matches ]).reshape(-
1,1,2)
    dst_pts = np.float32([ kp2[m[0].trainIdx].pt for m in matches ]).reshape(-
1,1,2)
    src_pts = np.hstack((src_pts.reshape(-1,2), np.ones((len(src_pts), 1))))
    dst_pts = np.hstack((dst_pts.reshape(-1,2), np.ones((len(dst_pts), 1))))


    if src_pts.shape[0] < 4 or dst_pts.shape[0] < 4:
        raise ValueError("Not enough points to find homography")


    A = []
    for i in range(len(src_pts)):
        x, y = src_pts[i,0:2]
        u, v = dst_pts[i,0:2]
        A.append([x,y,1, 0, 0, 0, -x*u, -y*u, -u])
        A.append([0, 0, 0, x*0.5, y, 1, -x*v, -y*v, -v])
    A = np.array(A)

    _, _, Vt = np.linalg.svd(A)
    H = Vt[-1].reshape(3, 3)
    H /= H[2, 2]

    return H

H12 = compute_homography(kp1, kp2, good12)
H23 = compute_homography(kp2, kp3, good23)
H34 = compute_homography(kp3, kp4, good34)


# Compute the size of the output image
corners1 = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
corners2 = cv.perspectiveTransform(corners1, H12)
```

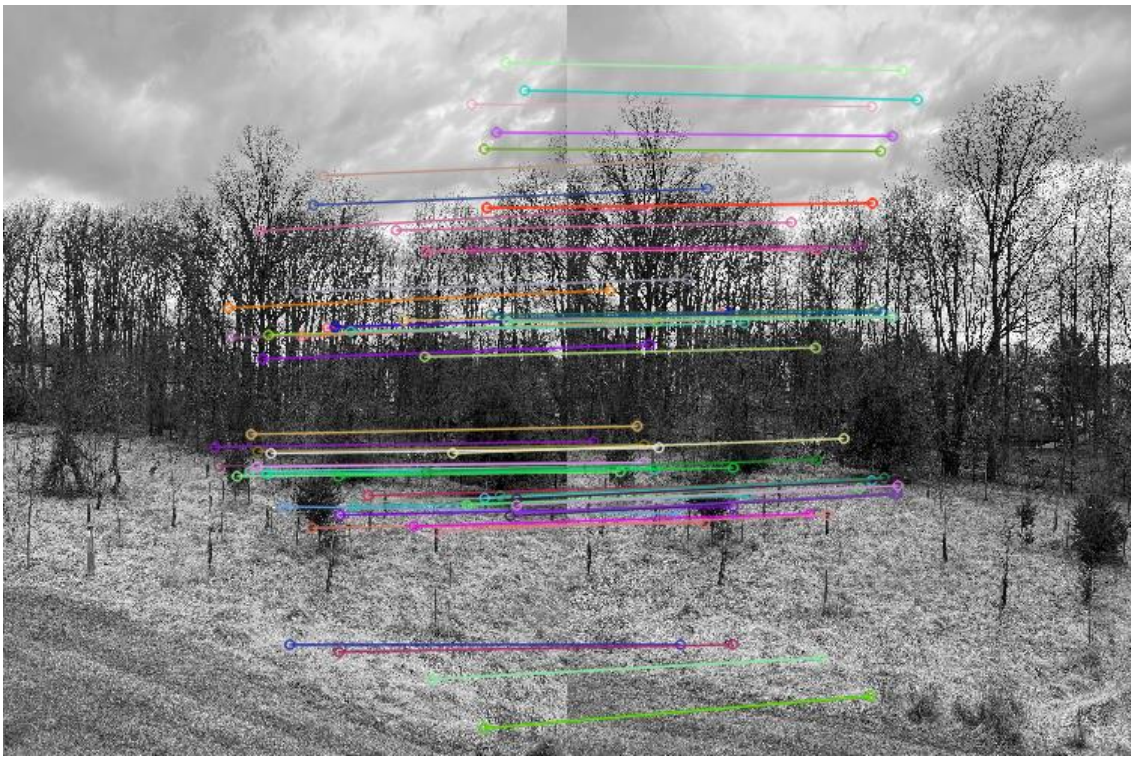
```
corners3 = cv.perspectiveTransform(corners2, H23)
corners4 = cv.perspectiveTransform(corners3, H34)

x_min = int(min(corners1[:,:,0].min(), corners2[:,:,0].min(),
corners3[:,:,0].min(), corners4[:,:,0].min()))
y_min = int(min(corners1[:,:,1].min(), corners2[:,:,1].min(),
corners3[:,:,1].min(), corners4[:,:,1].min()))
x_max = int(max(corners1[:,:,0].max(), corners2[:,:,0].max(),
corners3[:,:,0].max(), corners4[:,:,0].max()))
y_max = int(max(corners1[:,:,1].max(), corners2[:,:,1].max(),
corners3[:,:,1].max(), corners4[:,:,1].max()))

# Create a translation matrix to shift the image
translation_matrix = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]])

# Warp the images using the homographies
pano1 = cv.warpPerspective(img1, translation_matrix.dot(H12), (x_max - x_min,
y_max - y_min))
pano1=trimmer(pano1)
# pano1=cv.cvtColor(pano1,cv.COLOR_GRAY2BGR)
pano2 = cv.warpPerspective(img2, translation_matrix.dot(H23), (x_max - x_min,
y_max - y_min))
pano2=trimmer(pano2)
pano3 = cv.warpPerspective(img3, translation_matrix.dot(H34), (x_max - x_min,
y_max - y_min))
pano3=trimmer(pano3)

cv.imshow('Panorama1', pano1)
cv.imshow('Panorama2', pano2)
cv.imshow('Panorama3', pano3)
cv.waitKey(0)

print(pano1.shape,pano2.shape,pano3.shape)
# Combine the images
pano =
np.zeros((pano1.shape[0]+pano2.shape[0]+pano3.shape[0],pano1.shape[1]+pano2.shape
[1]+pano3.shape[1]), dtype=np.uint8)
# print(pano.shape)
pano[:pano1.shape[0], :pano1.shape[1]] = cv.flip(pano1,0)
pano[:pano1.shape[0], :pano1.shape[1]] = pano1
# print(pano2.shape,pano1.shape[1]+pano2.shape[1])
pano[:pano3.shape[0], pano1.shape[1]+pano2.shape[1]:] = cv.flip(pano3,0)
pano[:pano3.shape[0], pano1.shape[1]+pano2.shape[1]:] = pano3
pano[:pano2.shape[0], pano1.shape[1]:pano1.shape[1]+pano2.shape[1]] =
cv.flip(pano2,0)
```

```
pano[:pano2.shape[0], pano1.shape[1]:pano1.shape[1]+pano2.shape[1]] = pano2

# Show the resulting panorama
pano=trimmer(pano)
print(pano.shape,'shape')
pano=pano[100:550,:]
cv.imshow('Panorama', pano)

cv.waitKey(0)
```

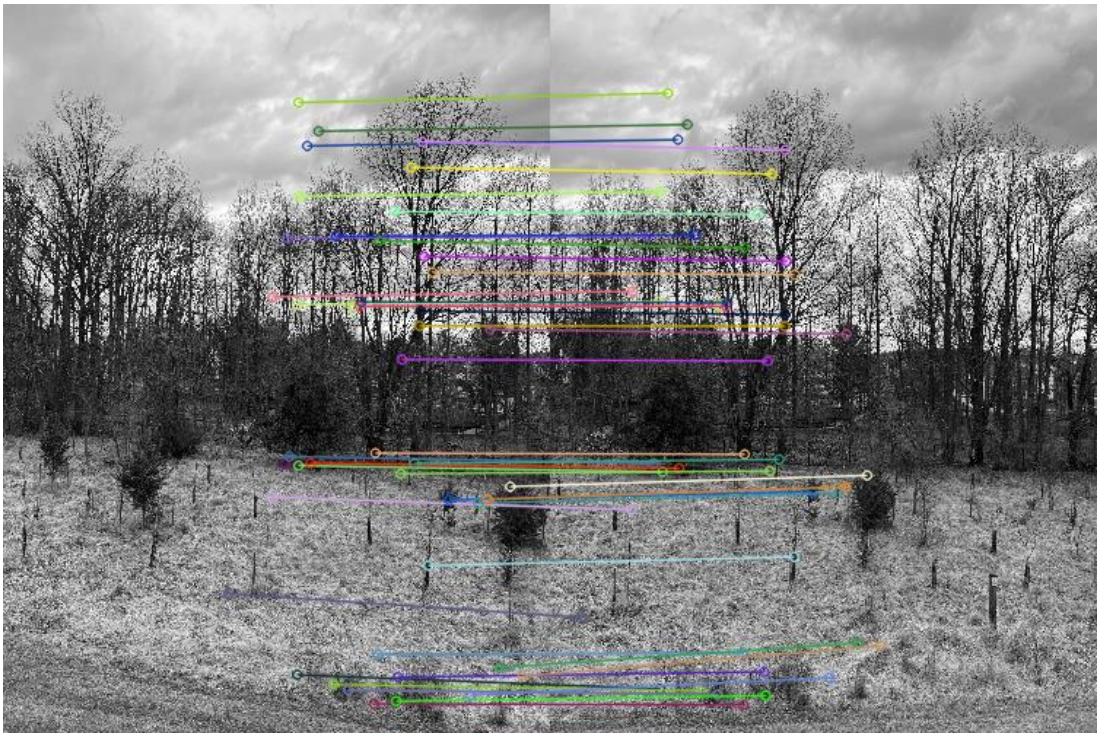
Output:



Matches 1-2



Matches2-3



Matches3-4

Final panorama