PSET 3

These are some of the libraries/modules you will require for this homework.

```
# Install OpenCV version for SIFT.
!pip install opencv-contrib-python
```

```
Requirement already satisfied: opencv-contrib-python in
/opt/homebrew/lib/python3.11/site-packages (4.9.0.80)
Requirement already satisfied: numpy>=1.21.2 in
/opt/homebrew/lib/python3.11/site-packages (from opencv-contrib-
python) (1.26.1)

[notice] A new release of pip is available: 23.3.2 -> 24.0
[notice] To update, run: python3.11 -m pip install --upgrade pip
```

```
!pip install gdown==v4.6.0
```

```
Requirement already satisfied: gdown==v4.6.0 in
/opt/homebrew/lib/python3.11/site-packages (4.6.0)
Requirement already satisfied: filelock in
/opt/homebrew/lib/python3.11/site-packages (from gdown==v4.6.0)
(3.13.1)
Requirement already satisfied: requests[socks] in
/opt/homebrew/lib/python3.11/site-packages (from gdown==v4.6.0)
(2.31.0)
Requirement already satisfied: six in
/opt/homebrew/lib/python3.11/site-packages (from gdown==v4.6.0)
(1.16.0)
Requirement already satisfied: tqdm in
/opt/homebrew/lib/python3.11/site-packages (from gdown==v4.6.0)
(4.66.1)
Requirement already satisfied: beautifulsoup4 in
/opt/homebrew/lib/python3.11/site-packages (from gdown==v4.6.0)
(4.12.2)
Requirement already satisfied: soupsieve>1.2 in
/opt/homebrew/lib/python3.11/site-packages (from beautifulsoup4-
>gdown==v4.6.0) (2.5)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/homebrew/lib/python3.11/site-packages (from requests[socks]-
>gdown==v4.6.0) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/opt/homebrew/lib/python3.11/site-packages (from requests[socks]-
>gdown==v4.6.0) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/homebrew/lib/python3.11/site-packages (from requests[socks]-
>gdown==v4.6.0) (2.1.0)
Requirement already satisfied: certifi>=2017.4.17 in
```

```
/opt/homebrew/lib/python3.11/site-packages (from requests[socks]-
>gdown==v4.6.0) (2023.11.17)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/opt/homebrew/lib/python3.11/site-packages (from requests[socks]-
>gdown==v4.6.0) (1.7.1)

[notice] A new release of pip is available: 23.3.2 -> 24.0
[notice] To update, run: python3.11 -m pip install --upgrade pip
```

```
import zipfile
import os

# Path to the zip file on your desktop
zip_file_path = "/Users/krishpatel/Desktop/homework3_data.zip"

# Destination folder where you want to extract the contents
extract_folder = "/Users/krishpatel/Desktop"

# Extract the zip file
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extract_folder)
```

```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import copy
import os
import random

import numpy as np
import scipy
import cv2
from PIL import Image
import matplotlib.pyplot as plt
from IPython.display import display
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

These are some functions which will be useful throughout the homework for visualizations.

```
def display_color(x: np.array, normalized:bool = False):
    plt.figure(figsize=(10,10))
    if not normalized:
        plt.imshow(x,vmin=0,vmax=1)
    else:
        plt.imshow(x/x.max(),vmin=0,vmax=1)
    return plt
```

```
def plot_correspondences(image1, image2, correspondences, color):
    image = np.concatenate((image1, image2), axis=1)
    for correspondence in correspondences:
        point1, point2 = correspondence
        point1 = (int(round(point1[0])), int(round(point1[1])))
        point2 = (int(round(point2[0])), int(round(point2[1])))
        cv2.circle(image, point1, 10, color, 2, cv2.LINE_AA)
        cv2.circle(image, tuple([point2[0] + image1.shape[1],
point2[1]]), 10,
                   color, 2, cv2.LINE_AA)
        cv2.line(image, point1, tuple([point2[0] + image1.shape[1],
point2[1]]),
                 color, 2)
    plot = display_color(image)
    return plot
```

Question 3

Image Stitching

In this question, you will be stitching together two images of the same scene (images assumed to be in left to right order) taken from different camera viewpoints to form a panorama of the scene. This task will require implementing a pipeline with the following steps:

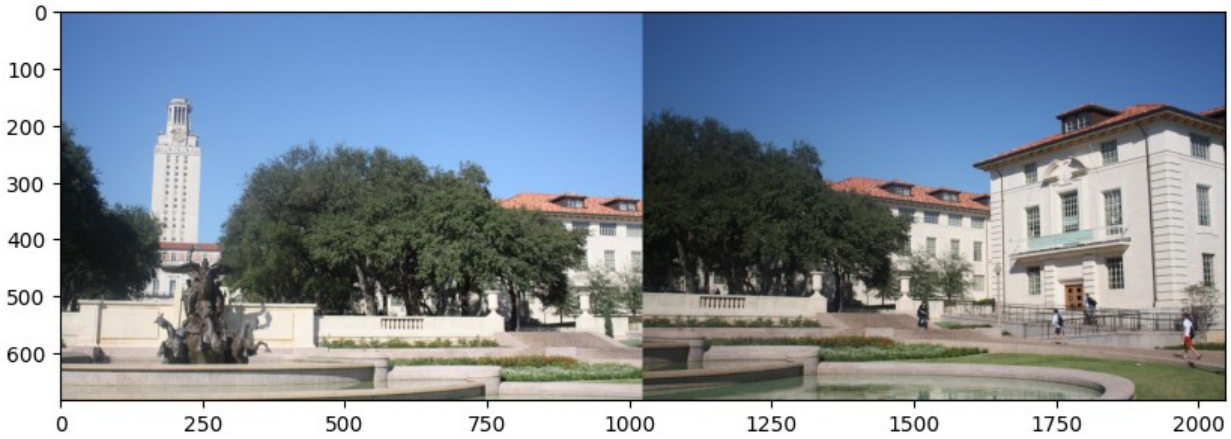
1. Extract SIFT keypoints and descriptors from each image and propose possible correspondences by matching SIFT descriptors between the two images. Note that this step outputs some false correspondences, which will be pruned in the next step.
2. Estimate the homography between the two images using the following RANSAC loop:

```
For N iterations:
    i. Get random subset of correspondences.
    ii. Compute the homography H using homogeneous direct linear
transform (DLT) applied to the random subset of correspondences.
    iii. Count the number of inliers, where inliers are the
correspondences (in the whole set of correspondences) that the
homography
    fits well (using Euclidean distance as the error metric).
    iv. Keep the homography H with the largest number of inliers and
H's corresponding set of inliers.
```

1. Recompute the homography H using the set of inliers from step 2.
2. Use the homographies obtained from step 3 to stitch together the images to form a panorama.

Load the two images to be stitched together.

```
image1 = np.asarray(Image.open('Data/Problem_3/uttower_left.jpg'))
image2 = np.asarray(Image.open('Data/Problem_3/uttower_right.jpg'))
plot = display_color(np.concatenate((image1, image2), axis=1))
```



Answer 3.1

The first step of image stitching is to obtain SIFT keypoints and descriptors for each image. Complete the function `run_sift(image, num_features)` that returns the SIFT keypoints and descriptors for a color image `image`; `num_features` is a parameter that limits the number of SIFT keypoints and descriptors the function should return. Note that you will need to convert the image to grayscale before running SIFT. *Hint*: OpenCV has functions to convert an RGB image to grayscale and to compute SIFT keypoints and descriptors given a grayscale image.

Copy paste your solution in the cell below on Overleaf for Question 3.1.

```
# Write your code in this cell.

def run_sift(image, num_features):
    grayimg = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    sift = cv2.SIFT_create(nfeatures=num_features)
    keypoints, descriptors = sift.detectAndCompute(grayimg, None)
    return keypoints, descriptors
```

Answer 3.2

After computing the SIFT keypoints and descriptors, you will then need to match SIFT descriptors between the two images to obtain a set of possible correspondences. To obtain these correspondences, you should follow the steps below as proposed in the SIFT paper (Lowe 2004):

```
For every keypoint kp in the first image,

1. Find the two keypoints kp1, kp2 in the second image with
descriptors des1, des2 that are closest and second closest,
respectively,
```

```
to the descriptor des of the first image's keypoint. The distance
metric is Euclidean distance d(x, y).
2. Add (kp, kp1) as a possible correspondence if d(des, des1) < ratio
* d(des, des2), where ratio is a parameter of the algorithm.
The idea here is that you want the descriptor des1 in the second image
to be much closer to the descriptor des in the first image
than any other descriptor in the second image.
```

Complete the function `find_sift_correspondences(kp1, des1, kp2, des2, ratio)` that returns a list of possible correspondences given SIFT keypoints `kp1, kp2` and descriptors `des1, des2` from the two images and the parameter `ratio`.

Copy paste your solution in the cell below on Overleaf for Question 3.2.

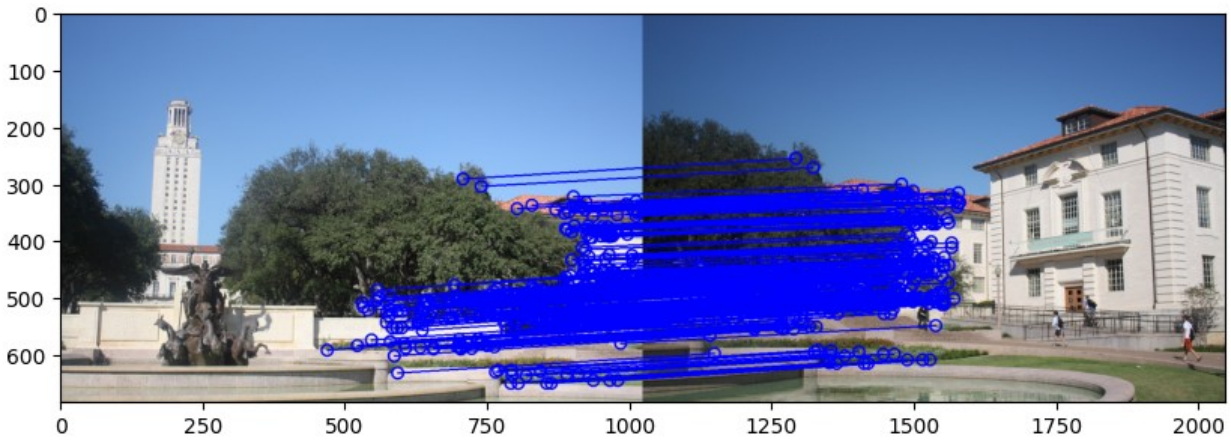
```
# Write your code in this cell.

def find_sift_correspondences(kp1, des1, kp2, des2, ratio):
    correspondences = []

    for idx, kp in enumerate(kp1):
        des = des1[idx]
        distances = [np.linalg.norm(des - des2[i]) for i in
range(len(kp2))]
        idx_closest = np.argsort(distances)
        idx_kp_1 = idx_closest[0]
        idx_kp_2 = idx_closest[1]
        if np.linalg.norm(des - des2[idx_kp_1]) < ratio *
np.linalg.norm(des - des2[idx_kp_2]):
            correspondences.append((kp.pt, kp2[idx_kp_1].pt))

    return correspondences
```

Answer 3.3

Now that you have computed the possible correspondences, you can now visualize them. You should observe that there are many correspondences, some of which are quite noisy (you will observe this more clearly after implementing RANSAC).

Upload the saved image on Overleaf for Question 3.3.

```
kp1, des1 = run_sift(image1, 2000)
kp2, des2 = run_sift(image2, 2000)
correspondences = find_sift_correspondences(kp1, des1, kp2, des2, 0.6)
plot = plot_correspondences(image1, image2, correspondences, (0, 0,
255))
os.makedirs('Data/Solutions', exist_ok=True)
plot.savefig('Data/Solutions/question_3_3.pdf', format='pdf',
bbox_inches='tight')
```



Answer 3.4

After obtaining possible correspondences, you will need to use the RANSAC loop to prune the outlier correspondences. You will implement the RANSAC loop in parts. The first part is to compute the homography given a list of correspondences. Complete the function `compute_homography(correspondences)` that computes a homography matrix H using homogeneous direct linear transform (DLT) given a list of correspondences (in heterogeneous coordinates).

Copy paste your solution in the cell below on Overleaf for Question 3.4.

```
# Write your code in this cell.

def compute_homography(correspondences):
    A = []
    for i in range(len(correspondences)):
        x1, y1 = correspondences[i][0]
        x2, y2 = correspondences[i][1]
        A.append([x1, y1, 1, 0, 0, 0, -x2 * x1, -x2 * y1, -x2])
        A.append([0, 0, 0, x1, y1, 1, -y2 * x1, -y2 * y1, -y2])
    A = np.array(A)
    U, S, V = np.linalg.svd(A)
    H = V[-1].reshape(3, 3)
    return H
```

Answer 3.5

In addition to computing a homography matrix H given a list of correspondences, it is also useful to apply a homography to warp a list of 2D points. Complete the function `apply_homography(points, homography)` that applies a homography to warp a list of 2D points given in heterogeneous coordinates. The function should return a list of 2D points, also in heterogeneous coordinates.

Hint: You should convert the input 2D points into homogeneous coordinates before applying the homography. After applying the homography, you get the output 2D points in homogeneous coordinates, so you have to convert the output points back into heterogeneous coordinates.

Copy paste your solution in the cell below on Overleaf for Question 3.5.

```
# Write your code in this cell.

def apply_homography(points, homography):
    out = []
    for out_point in points:
        out_point = np.array([out_point[0], out_point[1], 1])
        out_point = np.dot(homography, out_point)
        out_point = out_point / out_point[2]
        out.append(out_point[:2])
    return np.array(out)
```

Answer 3.6

Another part of the RANSAC loop is determining the inliers, where inliers are the correspondences that the homography fits within a Euclidean distance threshold. Complete the function `compute_inliers(homography, correspondences, threshold)` that returns the list of inliers and the list of outliers given a homography matrix, a set of potential correspondences, and a distance threshold.

Copy paste your solution in the cell below on Overleaf for Question 3.6.

```
# Write your code in this cell.

def compute_inliers(homography, correspondences, threshold):
    inliers, outliers = [], []

    for i in range(len(correspondences)):
        point1 = np.array(correspondences[i][0])
        point2 = np.array(correspondences[i][1])
        point1_transformed = apply_homography(point2[None, :],
np.linalg.inv(homography)).squeeze()
        if np.linalg.norm(point1 - point1_transformed) < threshold:
            inliers.append(correspondences[i])
        else:
            outliers.append(correspondences[i])
    return inliers, outliers
```

Answer 3.7

Now that you have implemented functions to compute a homography matrix and its inliers, you can now implement the RANSAC loop. Complete the function `ransac(correspondences, num_iterations, num_sampled_points, threshold)`, which runs the RANSAC loop to return a homography matrix with its corresponding sets of inliers and outliers. The parameters to the function are the set of possible correspondences, the number of iterations, the number of correspondences to sample for each iteration, and the inliers distance threshold.

Copy paste your solution in the cell below on Overleaf for Question 3.7.

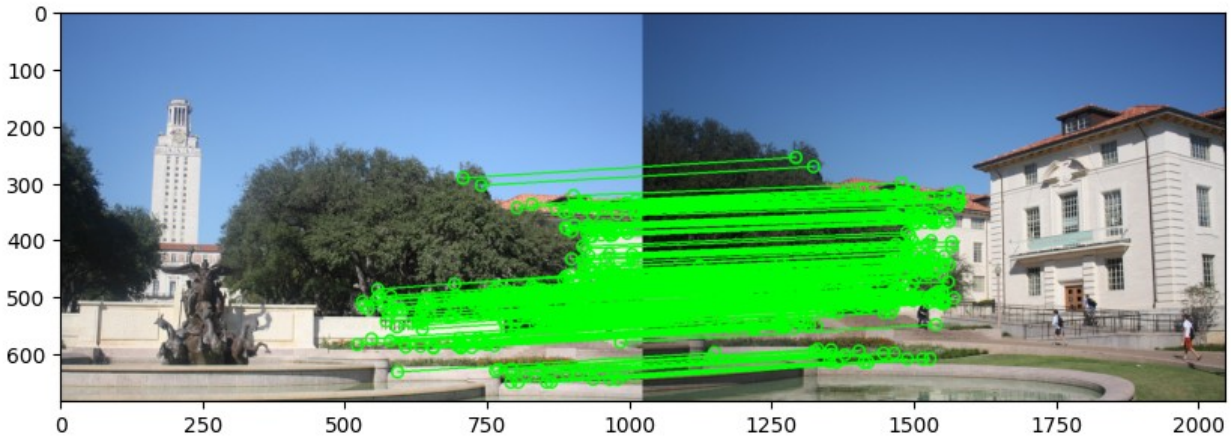
```
def ransac (correspondences, num_iterations, num_sampled_points,
threshold):
    count = 0
    max_inlier_count = 0
    while(count < num_iterations):
        sample = random.sample(correspondences, num_sampled_points)
        inliers, outliers =
compute_inliers(compute_homography(sample), correspondences,
threshold)
        if len(inliers) > max_inlier_count:
            max_inlier_count = len(inliers)
            best_inliers = inliers
            best_outliers = outliers
        count += 1
    return compute_homography(sample), best_inliers,best_outliers
```

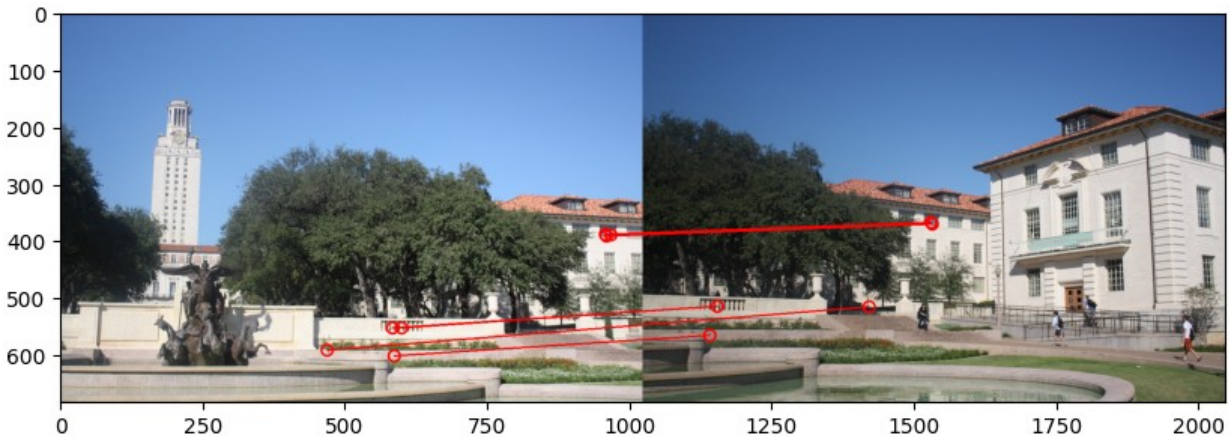
Answer 3.8

You will now visualize the inlier (first image in green) and outlier (second image in red) correspondences returned by RANSAC. Note that some of the outliers are quite noisy.

Execute the cell below and copy the saved images on Overleaf for Question 3.8.

```
_, inliers, outliers = ransac(correspondences, 50, 6, 3)
inliers_plot = plot_correspondences(image1, image2, inliers, (0, 255,
0))
inliers_plot.savefig('Data/Solutions/question_3_8_inliers.pdf',
format='pdf', bbox_inches='tight')
outliers_plot = plot_correspondences(image1, image2, outliers, (255,
0, 0))
outliers_plot.savefig('Data/Solutions/question_3_8_outliers.pdf',
format='pdf', bbox_inches='tight')
```





Answer 3.9

The final portion that needs to be implemented is the actual stitching of the two images given the homography H from the first image to the second image. For the stitching process, you should apply inverse warping to warp the second image to the viewpoint of the first image. The inverse warping works as follows:

1. Find where pixels `p1` in the first image (output image) are warped to in the second image (input image) when applying the homography H. Note that the homography H itself is the inverse transform as you are currently trying to warp the second image to the first image, which would use homography H^{-1}. Let `p2` be the corresponding pixel location in the second image.
2. Assuming that `p2` is in the second image, the pixel locations `p2` may not be integer locations. Apply bilinear interpolation to compute the pixel value of the second image at location `p2`.

As bilinear interpolation is needed for the inverse warping, complete the function `interpolate(image, loc)` that implements bilinear interpolation to compute the pixel value in the image corresponding to a pixel location.

Bilinear interpolation works as follows; for a given location $(x+\Delta x, y+\Delta y)$, where x, y are integers and $0 \leq \Delta x, \Delta y < 1$, we consider the four neighboring integer pixel locations $(x, y), (x+1, y), (x, y+1), (x+1, y+1)$ and then perform multiple linear interpolations (it may help to take a look at lecture 8 slide 68 for some diagrams):

1. Compute pixel value $I(x+\Delta x, y)$: $I(x + \Delta x, y) = I(x, y) (1 - \Delta x) + I(x + 1, y) \Delta x $.
2. Compute pixel value $I(x+\Delta x, y+1)$: $I(x + \Delta x, y + 1) = I(x, y + 1) (1 - \Delta x) + I(x + 1, y + 1) \Delta x $.
3. Compute pixel value $I(x+\Delta x, y+\Delta y)$:
   $I(x+\Delta x, y+\Delta y) = I(x+\Delta x, y)(1-\Delta y) + I(x+\Delta x, y+1)\Delta y.$

Substituting the first two steps into the last step, the following formula for bilinear interpolation is obtained:

$I(x+\Delta x, y+\Delta y) = I(x,y)(1-\Delta x)(1-\Delta y) + I(x+1,y)\Delta x(1-\Delta y) + I(x,y+1)(1-\Delta x)\Delta y + I(x+1,y+1)\Delta x \Delta y$
.

Copy paste your solution in the cell below on Overleaf for Question 3.9.

```
def interpolate(image, loc):
    x0 = int(loc[0])
    y0 = int(loc[1])
    loc_x = loc[0]
    loc_y = loc[1]
    if x0<image.shape[1]-1 and y0<image.shape[0]-1 and y0>=0 and
x0>=0:
        dx = loc_x - x0
        dy = loc_y - y0
        pixel = (image[y0,x0]*(1-dx)*(1-dy)+image[y0,x0+1]*dx*(1-dy)
+image[y0+1,x0]*(1-dx)*dy+image[y0+1,x0+1]*dx*dy)
    else:
        return 0 #out of bounds
    return pixel
```

Answer 3.10

Now that you have implemented bilinear interpolation, you can implement a function to actually stitch the two images together. Complete the function `stitch_image_given_H(image1, image2, homography)` that stitches `image1` and `image2` together and returns the stitched image given a homography matrix from `image1` to `image2`. This function should apply inverse warping as described above. For this function, you can assume that `image1` is to the left of `image2`; if a pixel location in the stitched image belongs to both images, you can average the corresponding pixel values from both images.

Copy paste your solution in the cell below on Overleaf for Question 3.10.

```
def stitch_image_given_H(front_img, back_img, homography_matrix):
    front_height, front_width = front_img.shape[:2]
    back_height, back_width = back_img.shape[:2]
    warp_canvas = np.zeros((back_height, 2*back_width,
front_img.shape[2]))
    empty_space = np.zeros((back_height, back_width,
front_img.shape[2]))

    inverse_homography = np.linalg.inv(homography_matrix)
    expanded_back_img = np.hstack((empty_space, back_img))
    for vert_idx in range(back_height):
        for horiz_idx in range(-back_width, back_width):
            transformed_coords = np.dot(inverse_homography,
np.array([horiz_idx, vert_idx, 1]))
            mapped_col = transformed_coords[0] / transformed_coords[2]
            mapped_row = transformed_coords[1] / transformed_coords[2]
            if 0 <= mapped_col < front_width and 0 <= mapped_row <
front_height:
```

```
                interpolated_pixel = interpolate(front_img,
(mapped_col, mapped_row))
                warp_canvas[vert_idx, horiz_idx + back_width] =
interpolated_pixel

    stitched_result = np.hstack((empty_space, empty_space))
    for i in range(stitched_result.shape[0]):
        for j in range(stitched_result.shape[1]):
            for channel in range(stitched_result.shape[2]):
                if expanded_back_img[i, j, channel] != 0 and
warp_canvas[i, j, channel] != 0:
                    stitched_result[i, j, channel] =
(expanded_back_img[i, j, channel] + warp_canvas[i, j, channel]) / 2
                elif expanded_back_img[i, j, channel] == 0:
                    stitched_result[i, j, channel] = warp_canvas[i, j,
channel]
                elif warp_canvas[i, j, channel] == 0:
                    stitched_result[i, j, channel] =
expanded_back_img[i, j, channel]
    return stitched_result
```

Answer 3.11

Now, you have implemented all the parts of the image stitching pipeline. Put it all together and complete the function `stitch_image(image1, image2, num_features, sift_ratio, ransac_iter, ransac_sampled_points, inlier_threshold, use_ransac)` that stitches two images together given various parameters of the algorithm (number of SIFT features, ratio for obtaining initial SIFT correspondences, RANSAC parameters, whether to use RANSAC). *Reminder*: after you get the list of inliers with RANSAC, you should recompute the homography using this set of inliers.

Copy paste your solution in the cell below on Overleaf for Question 3.11.

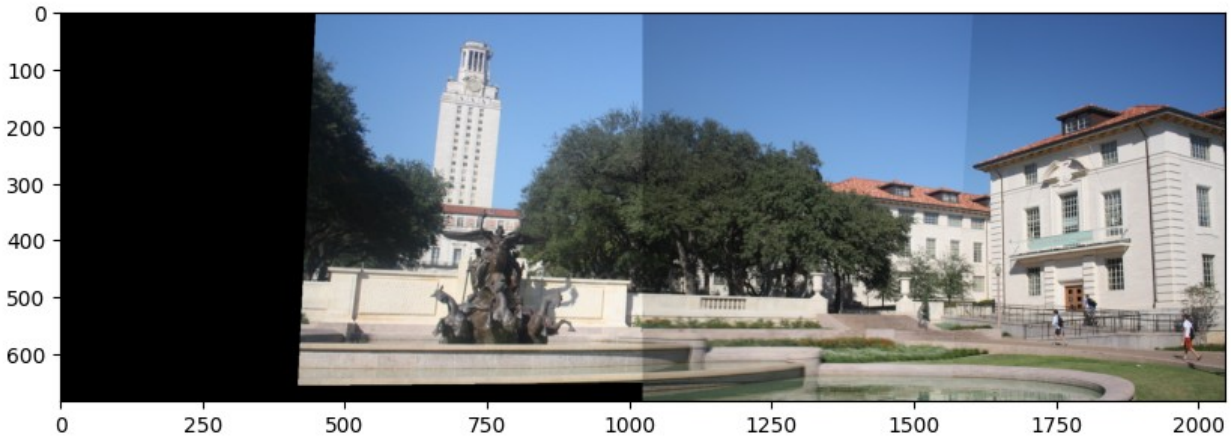
```
# Write your code in this cell.

def stitch_image(image1, image2, num_features, sift_ratio,
ransac_iter, ransac_sampled_points, inlier_threshold,
use_ransac=True):
    kp1, des1 = run_sift(image1, num_features)
    kp2, des2 = run_sift(image2, num_features)
    correspondences = find_sift_correspondences(kp1, des1, kp2, des2,
sift_ratio)
    if use_ransac:
        homography, _, __ =
ransac(correspondences,ransac_iter,ransac_sampled_points,inlier_thresh
old)
    else:
        homography = compute_homography(correspondences)
```

```
    stitched_image = stitch_image_given_H(image1, image2, homography)
    return stitched_image
```

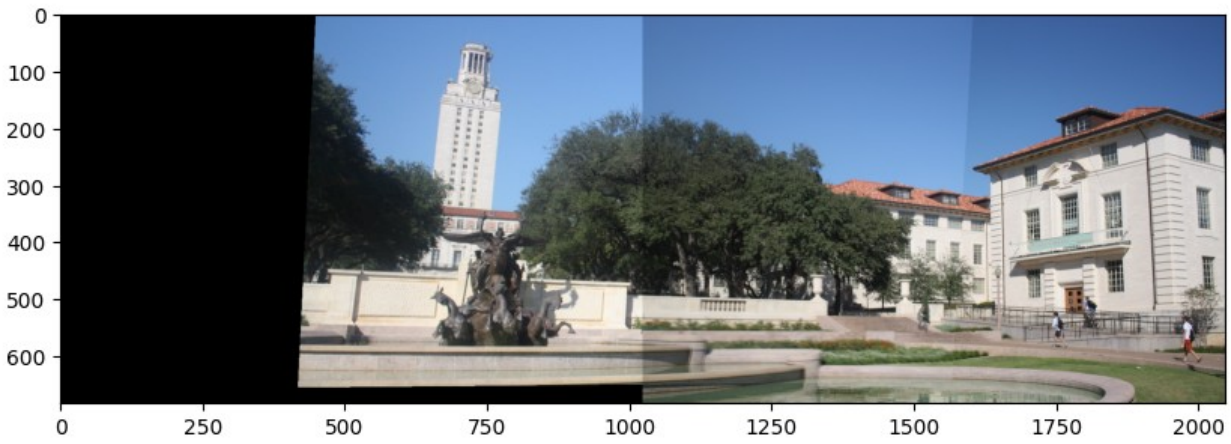
Answer 3.12

Execute the cell below and copy the saved image on Overleaf for Question 3.12. The cell runs the image stitching function to stitch the two sample images together (assuming that they are in left to right order). It is expected that there will be a small part of the image that is black. *Note:* This cell may take up to a few minutes to execute.

```
# The black part is a region of the stitched image that does not come
from the two original images.
stitched_image = stitch_image(image1, image2, 2000, 0.6, 50, 6, 3)
stitched_plot = display_color(stitched_image / 255.0)
stitched_plot.savefig('Data/Solutions/question_3_12.pdf',
format='pdf', bbox_inches='tight')
```




```
stitched_plot = display_color(stitched_image / 255.0)
stitched_plot.savefig('Data/Solutions/question_3_12.pdf',
format='pdf', bbox_inches='tight')
```

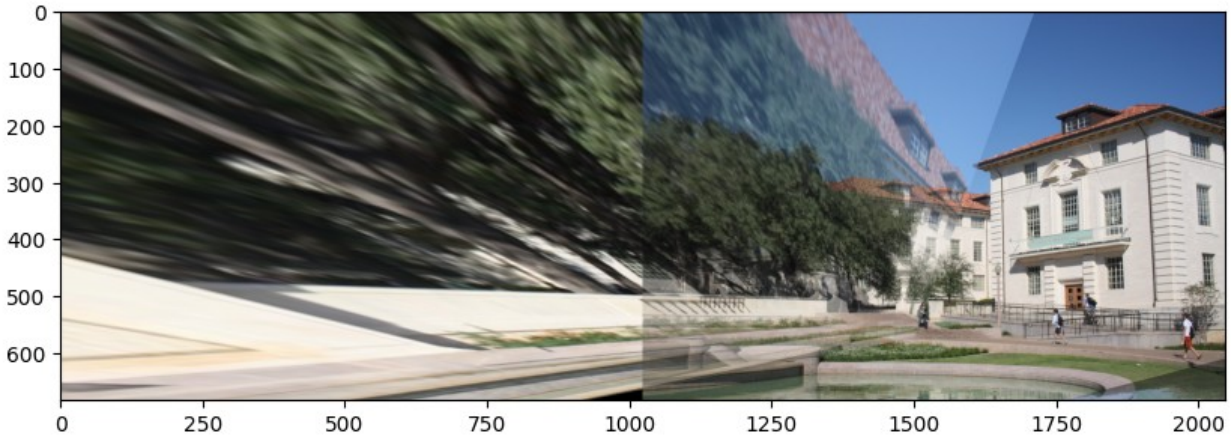


Answer 3.13

Execute the cell below and copy the saved image on Overleaf for Question 3.13. The cell runs the image stitching function to stitch the two sample images together (assuming that they are in left to right order) without filtering out outlier correspondences using RANSAC. You should observe that without RANSAC, the stitching does not work properly, showing the importance of RANSAC in the image stitching pipeline. *Note:* This cell may take up to a few minutes to execute.

```
# The black part is a region of the stitched image that does not come
from the two original images.
stitched_image = stitch_image(image1, image2, 2000, 0.6, 50, 6, 3,
False)
stitched_plot = display_color(stitched_image / 255.0)
stitched_plot.savefig('Data/Solutions/question_3_13.pdf',
format='pdf', bbox_inches='tight')
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```



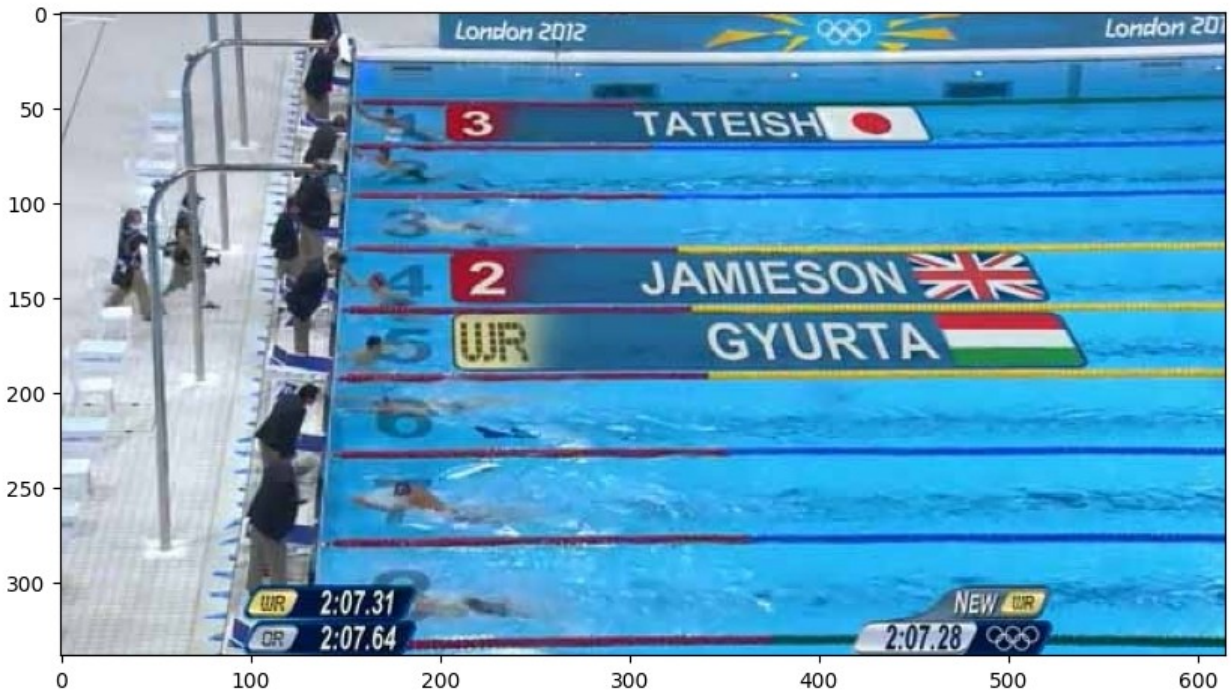
Question 4

Olympic Champion Using Homography

In this question, you will be making yourself the new World Swimming Champion using homography.

You are given the following image from the London 2012 Olympics.

```
img = np.asarray(Image.open('Data/Problem_4/pool-vfx.jpg'))
_ = display_color(img/255.0)
```



You have to use homography to make yourself the new World Champion:

```
img_final = Image.open('Data/Problem_4/question_4.png')
display(img_final)
```

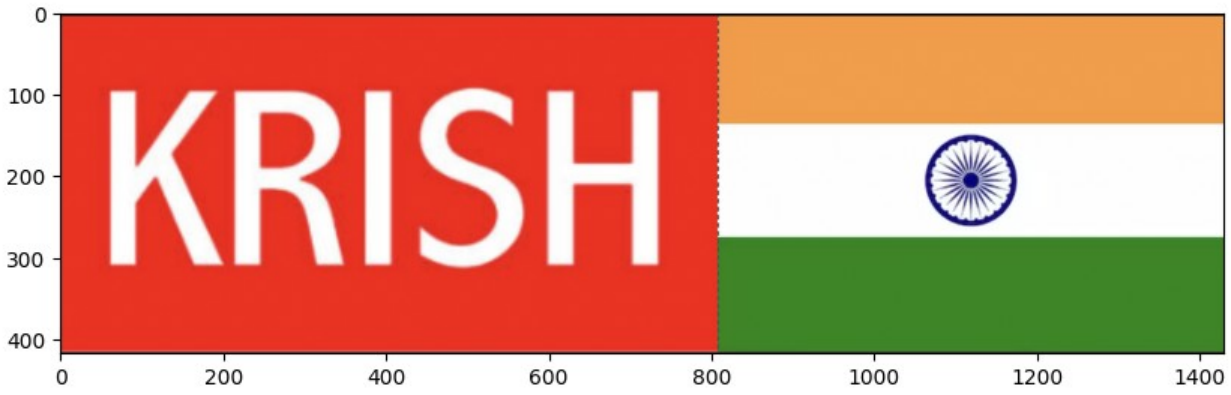
In the previous question, you used homography to stitch two images. You will use many functions from the previous question for this question. In the previous question, you used SIFT

to compute the correspondences between the two images. Here you will estimate the corresponding points manually to construct the homography matrix and create the new image.

You will need two images for this question: (1) the pool image which we have provided, (2) an image with your name and country flag. To get an image with your name and flag, you are free to choose any method you want. In the example above, we used Keynote (on Mac) + Screenshot to get that image.

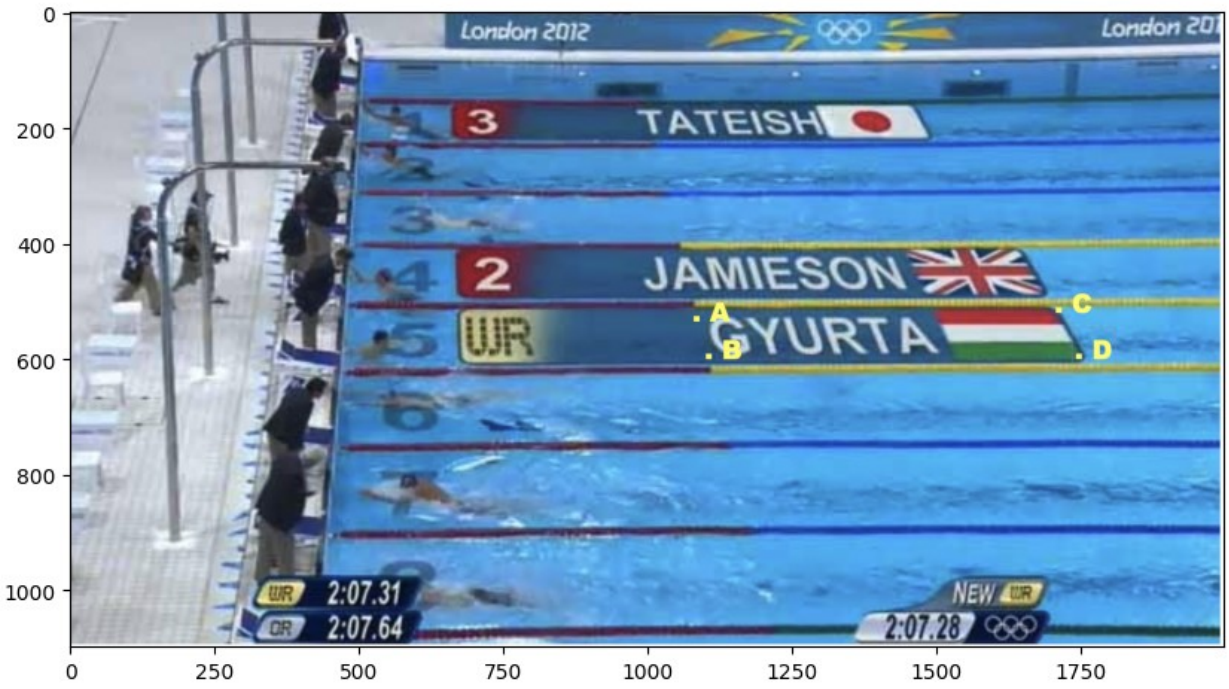
```
# Load your name+flag image here. You are free to choose any logo you want; we just provided a flag as an example.

name_img = np.asarray(Image.open('Data/Problem_4/krish.png').convert('RGB'))
display_color(name_img/255.0)
```

```
<module 'matplotlib.pyplot' from '/opt/homebrew/lib/python3.11/site-packages/matplotlib/pyplot.py'>
```



In the previous question, you used SIFT to compute the corresponding points. For this question, we have provided you with 4 points which should correspond to the 4 corners of your name+flag image.

```
cp = np.asarray(Image.open('Data/Problem_4/Corresponding_points.png'))
_ = display_color(cp/255.0)
```



The (x, y) coordinates for points A, B, C, D in the above image, which will correspond to the 4 corners of your name+flag image, are provided below. For these (x, y) coordinates, the positive x direction is along the right, the positive y direction is along the bottom, and the origin is the top left pixel of the image.

(x, y) coordinates of the 4 points: $A = (334, 158)$ $B = (340, 190)$ $C = (528, 157)$ $D = (545, 187)$

Answer 4.1

Fill in the values for the points corresponding to A, B, C, D in your name+flag image. Copy the below cell on Overleaf for Question 4.1.

```
# Fill in the values for the corresponding points for your name+flag image.
A_1 = [0,0]
B_1 = [0,410]
C_1 = [1450,0]
D_1 = [1450,410]
correspondence = [
                  ([334,158], A_1),
                  ([340,190], B_1),
                  ([528,157], C_1),
                  ([545,187], D_1),
]
```

Using the 4 correspondences, you can construct the homography matrix.

```
homography = compute_homography(correspondence)
```

Now, you will need to stitch the name+flag image into the original Olympic pool image. Complete the function `stitch_image_given_H_new(pool_image, name_flag_image)` for this task, which stitches the name+flag image into the original Olympic pool image. Copy paste your solution in the cell below on Overleaf for Question 4.2.

This function should be similar to the stitching function you wrote previously with minor differences: (1) Previously, since you had to stitch two images side by side, the output image had twice the number of columns as the original image. For this question, since you will be stitching the name+flag image inside the pool image, the output image will have the same number of columns as the input pool image. In other words the output will have the same dimension as `pool_image`. (2) If a pixel location in the stitched image is valid in `pool_image` and has a valid inverse-warped pixel location in `name_flag_image`, then you will use the pixel value from `name_flag_image` instead of averaging both images' pixel locations.

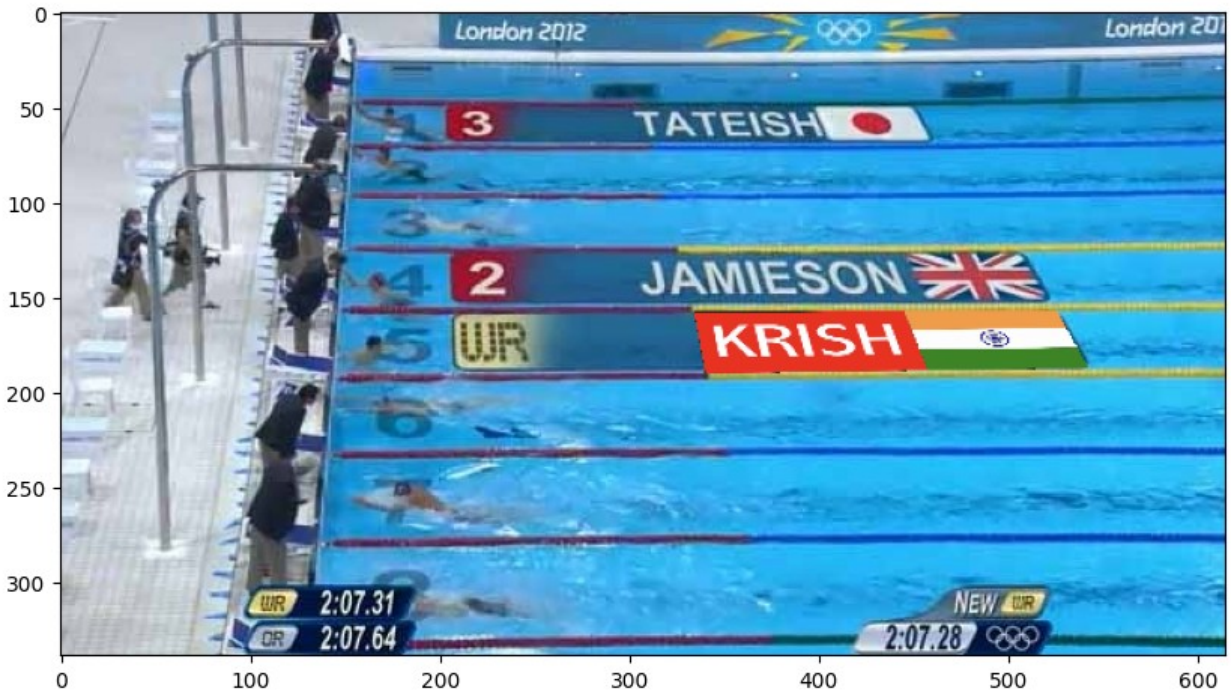
Answer 4.2

```
def stitch_image_given_H_new(image1, image2, homography):
    primary_height, primary_width = image1.shape[:2]
    secondary_height, secondary_width = image2.shape[:2]
    transformed_canvas = np.zeros((secondary_height, secondary_width,
image1.shape[2]))
    for row in range(secondary_height):
        for col in range(secondary_width):
            homogeneous_coord = np.dot(homography, np.array([col, row,
1]))
            trans_col = homogeneous_coord[0] / homogeneous_coord[2]
            trans_row = homogeneous_coord[1] / homogeneous_coord[2]
            if 0 <= trans_col < primary_width and 0 <= trans_row <
primary_height:
                pixel_val = interpolate(image1, (trans_col,
trans_row))
                transformed_canvas[row, col] = pixel_val
    combined_image = np.where(transformed_canvas != 0,
transformed_canvas, image2)

    return combined_image
```

Answer 4.3

Execute the cell below and copy the saved image on Overleaf for Question 4.3. The cell runs the new stitching function to stitch your name+flag image into the original Olympic pool image.

```
new_olympic_champion = stitch_image_given_H_new(name_img, img,
homography)
plot = display_color(new_olympic_champion,True)
os.makedirs('Data/Solutions', exist_ok=True)
plot.savefig('Data/Solutions/question_4.png', format='png',
bbox_inches='tight')
```



Question 5

Eight-Point Algorithm

In this question, you will use the eight-point algorithm to reconstruct 3D points associated with some correspondences between two images of the same scene. For this task, you will implement a pipeline with three broad steps:

1. Implement the eight-point algorithm to estimate the essential matrix.
2. Compute the translation and rotation between the cameras' coordinate frames using the essential matrix.
3. Reconstruct the 3D points by solving for their depths. Combining the depths with the 2D points will yield the reconstructed 3D points.

Load the correspondences. The correspondences are given as a list of tuples $\left(\left(x_1, y_1\right), \left(x_2, y_2\right)\right)$, where $\left(x_1, y_1\right)$ and $\left(x_2, y_2\right)$ are the corresponding points from the first and second image, respectively.

```
def format_correspondences(correspondences):
    formatted_corr = []
    for correspondence in correspondences:
        point1, point2 = correspondence[0:2], correspondence[2:]
        formatted_corr.append((point1, point2))
    return formatted_corr
```

```
correspondences = np.load('Data/Problem_5/correspondences.npy')
correspondences = format_correspondences(correspondences)
```

Answer 5.1

The first step for reconstructing the 3D points X_1, X_2 is to estimate the essential matrix using the eight-point algorithm. Complete the function `compute_essential_matrix(correspondences)`, which returns the essential matrix given some correspondences using the algorithm in lecture 9. Copy paste your solution in the cell below on Overleaf for Question 5.1.

Assume that the essential matrix corresponds to rotating and translating the first image's camera frame to the second image's camera frame (i.e. $X_2 = R\,X_1 + T$, where R is the rotation matrix and T is the translation vector). Enforce that the essential matrix Q must be rank 2 by running SVD on it ($Q = U\,\Sigma\,V^T$), setting the smallest singular value of Q to 0 to get a matrix Σ' with the new singular values, and outputting the modified essential matrix $Q = U\,\Sigma'\,V^T$.

Hint: This process is similar to computing the homography between two images (question 3.4).

```
# Write your code in this cell.

print("shape of correspondences",len(correspondences))
print("correspondences",correspondences)
\

def compute_essential_matrix(correspondences):

    matrix_A = np.array([])
    for pair in correspondences:
        point1_x, point1_y = pair[0][0], pair[0][1]
        point2_x, point2_y = pair[1][0], pair[1][1]
        new_row = np.array([point2_x * point1_x, point2_x * point1_y,
point2_x,
                            point2_y * point1_x, point2_y * point1_y,
point2_y,
                            point1_x, point1_y, 1])
        if(matrix_A.shape[0] == 0):
            matrix_A = new_row
        else:
            matrix_A = np.vstack((matrix_A, new_row))
    singular_u, singular_s, singular_vh = np.linalg.svd(matrix_A,
full_matrices=True)
    essential_matrix = singular_vh[-1, :].reshape((3, 3))
    u_final, s_final, vh_final = np.linalg.svd(essential_matrix,
full_matrices=True)
    s_final[-1] = 0
    essential_matrix = np.matmul(np.matmul(u_final, np.diag(s_final)),
vh_final)
    return essential_matrix
```

```
shape of correspondences 110
correspondences [(array([-0.04625099, -0.05824104]), array([-
0.05940542, -0.05824104])), (array([-0.01139174,  0.0413723 ]),
array([-0.01468035,  0.04268301])), (array([-0.05019732, -
0.05824104]), array([-0.06466719, -0.05824104])), (array([ 0.01886346,
-0.06348385]), array([ 0.02149434, -0.06348385])),
(array([ 0.00965535, -0.03268235]), array([ 0.0109708, -0.032027 ])),
(array([-0.01862668,  0.03678485]), array([-0.02191529,
0.0387509 ])), (array([ 0.01491713, -0.05693034]), array([ 0.01689029,
-0.05627499])), (array([-0.10807682, -0.05758569]), array([-
0.10610366, -0.05824104])), (array([ 0.02544067, -0.00777902]),
array([ 0.029387  , -0.00581296])), (array([-0.02651934, -
0.00384691]), array([-0.0284925 , -0.00384691])), (array([-0.02059984,
0.00663871]), array([-0.02323073,  0.00729406])), (array([-0.00744541,
-0.0346484 ]), array([-0.01139174, -0.03399305])),
(array([ 0.01820574, -0.03858051]), array([ 0.01952118, -
0.03792516])), (array([-0.05940542, -0.02875025]), array([-0.07321757,
-0.0294056 ])), (array([-0.01139174,  0.00336195]), array([-
0.01402263,  0.0040173 ])), (array([-0.01665351, -0.05627499]),
array([-0.02191529, -0.05627499])), (array([-0.00547224, -
0.01695393]), array([-0.00810313, -0.01629858])), (array([ 0.01820574,
-0.07462481]), array([ 0.01557485, -0.07396946])),
(array([ 0.00242042, -0.05889639]), array([ 0.00044725, -
0.05824104])), (array([0.05438043, 0.14950521]), array([0.05569587,
0.15605872])), (array([-0.08637201,  0.01188151]), array([-0.08637201,
0.01057081])), (array([-0.01862668, -0.0641392 ]), array([-0.02651934,
-0.0641392 ])), (array([0.03662194, 0.0361295 ]), array([0.04583004,
0.03940625])), (array([-0.05348592, -0.06282849]), array([-0.06992897,
-0.06348385])), (array([-0.05019732, -0.06151779]), array([-
0.06729808, -0.06217314])), (array([-0.01204946,  0.00467265]),
array([-0.01599579,  0.00598335])), (array([-0.09360695, -
0.03399305]), array([-0.09689555, -0.0346484 ])), (array([0.06029992,
0.13181073]), array([0.06687714, 0.13836424])), (array([-0.03704288, -
0.05561963]), array([-0.04625099, -0.05561963])), (array([ 0.03333333,
-0.03661446]), array([ 0.03859511, -0.03530376])), (array([-
0.00678769,  0.02760994]), array([-0.0100763 ,  0.02892064])),
(array([0.02412523, 0.09969854]), array([0.03070245, 0.10363064])),
(array([-0.01862668,  0.0747952 ]), array([-0.02257301,
0.0761059 ])), (array([ 0.0260984 , -0.00974507]), array([ 0.03004473,
-0.00777902])), (array([-0.09755328,  0.15999083]), array([-
0.11136543,  0.15605872])), (array([-0.05743225,  0.00598335]),
array([-0.06992897,  0.005328  ])), (array([-0.01402263,
0.01188151]), array([-0.01731123,  0.01253686])), (array([-0.04625099,
0.01319221]), array([-0.05940542,  0.01319221])), (array([-0.13043936,
0.15212661]), array([-0.13109708,  0.14688381])), (array([-0.1074191 ,
-0.05496428]), array([-0.10478821, -0.05561963])), (array([-
0.09755328, -0.03006095]), array([-0.09952644, -0.0307163 ])),
(array([-0.02125756, -0.03399305]), array([-0.0284925 , -
0.03399305])), (array([ 0.01952118, -0.05693034]), array([ 0.02346751,
-0.05693034])), (array([-0.04690871, -0.06610525]), array([-
```

```
0.06203631, -0.06610525])), (array([-0.01204946,  0.08659152]),
array([-0.01599579,  0.08790222])), (array([ 0.00570902, -
0.01302182]), array([ 0.00439358, -0.01171112])), (array([-0.11202315,
-0.06479455]), array([-0.11070771, -0.0654499 ])), (array([-
0.11004999, -0.06086244]), array([-0.10807682, -0.06151779])),
(array([-0.02783478, -0.00581296]), array([-0.02980795, -
0.00581296])), (array([ 0.00570902, -0.01498788]), array([ 0.0050513 ,
-0.01367717])), (array([-0.0436201,  0.1069074]), array([-0.0587477 ,
0.10625205])), (array([-0.0738753 , -0.03268235]), array([-0.08374112,
-0.0333377 ])), (array([-0.04296238,  0.11477161]), array([-
0.05677453,  0.11477161])), (array([-0.07847935, -0.12574218]),
array([-0.0830834 , -0.12508683])), (array([-0.06466719,
0.00598335]), array([-0.07519074,  0.005328  ])), (array([0.04583004,
0.10756275]), array([0.05503815, 0.11215021])), (array([-0.098211  ,
0.16719969]), array([-0.11268087,  0.16261223])), (array([-0.01468035,
-0.00712366]), array([-0.02059984, -0.00646831])), (array([-
0.09689555, -0.12049938]), array([-0.10347277, -0.11984403])),
(array([-0.05743225,  0.00794941]), array([-0.07058669,
0.00729406])), (array([-0.01665351,  0.0413723 ]), array([-0.02059984,
0.04268301])), (array([-0.05348592,  0.04399371]), array([-0.06664036,
0.04333836])), (array([-0.00876085,  0.04858117]), array([-0.01204946,
0.04989187])), (array([-0.00876085,  0.04661511]), array([-0.01204946,
0.04792581])), (array([-0.00810313,  0.08790222]), array([-0.01204946,
0.08921292])), (array([-0.01402263,  0.08855757]), array([-0.01862668,
0.09052363])), (array([-0.08702973,  0.005328  ]), array([-0.08834517,
0.00467265])), (array([-0.00612997,  0.08790222]), array([-0.00941857,
0.08986827])), (array([-0.04756643,  0.07938266]), array([-0.06072086,
0.07938266])), (array([-0.04427782,  0.11215021]), array([-0.05808998,
0.11215021])), (array([-0.00284136,  0.04464906]), array([-0.0041568 ,
0.04595976])), (array([ 0.01557485, -0.04841077]), array([ 0.01820574,
-0.04775542])), (array([-0.09689555, -0.03858051]), array([-
0.09886872, -0.03923586])), (array([-0.04098921, -0.03268235]),
array([-0.05348592, -0.0333377 ])), (array([-0.08702973,
0.00926011]), array([-0.08702973,  0.00860476])), (array([-0.098211  ,
-0.05496428]), array([-0.09886872, -0.05561963])), (array([-
0.06269403,  0.04399371]), array([-0.0738753 ,  0.04333836])),
(array([-0.11860037,  0.12656793]), array([-0.1166272 ,
0.12263582])), (array([-0.08637201,  0.01515827]), array([-0.08637201,
0.01384757])), (array([0.05174954, 0.11608231]), array([0.0596422 ,
0.12198047])), (array([-0.07584846, -0.05627499]), array([-0.08439884,
-0.05627499])), (array([-0.01468035,  0.09838784]), array([-
0.01862668,  0.09969854])), (array([-0.11531176, -0.08052297]),
array([-0.11202315, -0.08052297])), (array([-0.09492239, -
0.00384691]), array([-0.09689555, -0.00450226])), (array([-0.13043936,
0.14884986]), array([-0.12912391,  0.14360705])), (array([-0.04953959,
0.00729406]), array([-0.06269403,  0.00663871])), (array([-0.11070771,
-0.08838718]), array([-0.10873454, -0.08838718])), (array([-
0.03770061, -0.03137165]), array([-0.04822415, -0.03137165])),
(array([-0.00547224,  0.08331477]), array([-0.00876085,
0.08528082])), (array([-0.0528282, -0.0654499]), array([-0.06992897, -
```

```
0.0654499 ])), (array([ 0.01623257, -0.04447867]), array([ 0.01754801,
-0.04316797])), (array([-0.05151276,  0.11739301]), array([-
0.06664036,  0.11673766])), (array([-0.04953959,  0.04661511]),
array([-0.06269403,  0.04595976])), (array([-0.11465404, -
0.0307163 ]), array([-0.10939227, -0.032027  ])), (array([-0.0528282 ,
0.04595976]), array([-0.06664036,  0.04530441])), (array([-0.0771639 ,
-0.01367717]), array([-0.08571429, -0.01433253])), (array([-
0.01665351, -0.00843437]), array([-0.02323073, -0.00777902])),
(array([-0.03967377, -0.0307163 ]), array([-0.05085504, -
0.0307163 ])), (array([-0.11531176, -0.08248902]), array([-0.11202315,
-0.08314437])), (array([-0.11070771, -0.09035323]), array([-
0.10873454, -0.09035323])), (array([-0.04296238,  0.08855757]),
array([-0.05677453,  0.08790222])), (array([-0.01533807,
0.0348188 ]), array([-0.01862668,  0.0361295 ])), (array([-0.10939227,
-0.03858051]), array([-0.10807682, -0.03923586])), (array([-
0.00810313,  0.08200406]), array([-0.01204946,  0.08331477])),
(array([-0.11202315, -0.00450226]), array([-0.10873454, -
0.00581296])), (array([-0.03835833, -0.00646831]), array([-0.05019732,
-0.00646831])), (array([-0.05151276,  0.11411626]), array([-
0.06664036,  0.11346091])), (array([-0.10610366,  0.16588898]),
array([-0.11728493,  0.16195688])), (array([-0.04953959,
0.04202766]), array([-0.06269403,  0.0413723 ])), (array([-0.09360695,
-0.00646831]), array([-0.09623783, -0.00777902]))]
```

Answer 5.2

Now that you have computed the essential matrix, you can then compute the translation and rotation between the coordinate frames of both images' cameras. Complete the function `compute_translation_rotation(essential_matrix)`, which returns the translation vector T, rotation matrix R, and $\hat{T}$ (the matrix associated with T) given the essential matrix Q. Copy paste your solution in the cell below on Overleaf for Question 5.2.

Note: Lecture 9 has a typo with the formula for the rotation matrix. The middle matrix in the formula should be transposed.

```
def compute_translation_rotation(essential_matrix):
    W_matrix = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
    U, singular_values, V_transpose = np.linalg.svd(essential_matrix,
full_matrices=True)
    rotation_matrix = U @ W_matrix @ V_transpose
    translation_matrix_form = U @ np.diag(singular_values) @ W_matrix
@ U.T
    translation_vector = np.array([translation_matrix_form[2, 1],
translation_matrix_form[0, 2], translation_matrix_form[1, 0]])
    return translation_vector, rotation_matrix,
translation_matrix_form
```

Answer 5.3

You can sanity check the translation and rotation by checking that $\hat{T}$ is approximately skew-symmetric ($\hat{T}^T \approx -\hat{T}$, approximate due to numerical issues) and that the rotation matrix R is orthogonal ($R^T = R^{-1}$); note that you don't check that $det\ R = 1$, which may not hold due to numerical issues. Execute the cell below, which prints various useful quantities such as T, R, $\hat{T}$, R^T, R^{-1} and copy the output on Overleaf for question 5.3.

```
essential_matrix = compute_essential_matrix(correspondences)
translation, rotation, t_hat =
compute_translation_rotation(essential_matrix)
print("Translation vector: ", translation)
print("Rotation matrix: \n", rotation)
print("T_hat: \n", t_hat)
print("R^T: \n", np.transpose(rotation))
print("R^-1: \n", np.linalg.inv(rotation))
```

```
Translation vector:  [-0.70244976 -0.01835019  0.14775649]
Rotation matrix:
 [[ 0.96765823 -0.01752971  0.25165504]
 [ 0.01634127  0.99984327  0.00681171]
 [-0.251735   -0.00247905  0.96779303]]
T_hat:
 [[ 1.11753402e-04 -1.53002559e-01 -1.83501945e-02]
 [ 1.47756490e-01  4.94428979e-04  6.78484370e-01]
 [ 1.87296502e-02 -7.02449764e-01 -6.06182381e-04]]
R^T:
 [[ 0.96765823  0.01634127 -0.251735  ]
 [-0.01752971  0.99984327 -0.00247905]
 [ 0.25165504  0.00681171  0.96779303]]
R^-1:
 [[ 0.96765823  0.01634127 -0.251735  ]
 [-0.01752971  0.99984327 -0.00247905]
 [ 0.25165504  0.00681171  0.96779303]]
```

Answer 5.4

With the translation and rotation matrices, you can solve for the depths of the 3D points in the scene. Complete the function `compute_depths(correspondences, translation, rotation)` that computes the depths of the corresponding 3D points in the scene relative to both cameras given a list of 2D correspondences and the translation vector and rotation matrix from the first camera to the second camera. Copy paste your solution in the cell below on Overleaf for Question 5.4. *Hint*: The pseudoinverse may be useful.

```
def compute_depths(correspondences, translation, rotation):
    depths = []
    for correspondence in correspondences:
        yt = np.array([correspondence[1][0], correspondence[1][1],
1]).reshape(-1, 1)
```

```
        y = np.array([correspondence[0][0], correspondence[0][1],
1]).reshape(-1, 1)
        A = np.hstack((-np.matmul(rotation, y), yt))
        depth_solution = np.matmul(np.linalg.pinv(A), translation)
        depths.append(np.array([depth_solution[0],
depth_solution[1]]))
    return depths
```

Answer 5.5

Using the depths of the 3D points, you can scale the 2D points, represented using homogeneous coordinates, to obtain the reconstructed 3D points. Complete the function `reconstruct_3d(correspondences, depths)` that reconstructs the corresponding 3D points relative to both cameras given 2D correspondences and the associated depths relative to the two cameras. The result should be returned as a list of correspondences of 3D points, where each correspondence is a tuple (X_1, X_2) with X_1 and X_2 being the coordinates of the 3D point relative to the first and second images' cameras. Copy paste your solution in the cell below on Overleaf for Question 5.5.

```
# Write your code in this cell.

def reconstruct_3d(correspondences, depths):
    points_3d = []
    for i, pair in enumerate(correspondences):
        pt1, depth1 = pair[0], depths[i][0]
        pt2, depth2 = pair[1], depths[i][1]
        point_3d_1 = depth1 * np.array([pt1[0], pt1[1], 1])
        point_3d_2 = depth2 * np.array([pt2[0], pt2[1], 1])
        points_3d.append((point_3d_1, point_3d_2))
    return points_3d
```

Answer 5.6

Now that you have implemented the 3D reconstruction, you can check that the results are consistent with the input 2D correspondences by reprojecting the 2D points in image 1 into image 2. This cell checks this consistency via the following steps:

1. Reconstruct the 3D points X_1 relative to the first camera
2. Transform X_1 into 3D points X_{warp} relative to the second camera (using the translation vector T and rotation matrix R)
3. Project X_{warp} back into 2D points x_{warp} and compute mean relative error between x_{warp} and the input 2D points x_2 in the second image.

Execute the cell below and report the mean relative error you obtain below for Question 5.6; the mean relative error should be within a few percent.

```
essential_matrix = compute_essential_matrix(correspondences)
translation, rotation, _ =
```

```
compute_translation_rotation(essential_matrix)
depths = compute_depths(correspondences, translation, rotation)
corr_3d = reconstruct_3d(correspondences, depths)
rel_errors = []
print(correspondences)
print ("zip ",zip(correspondences, corr_3d))
for (point1_2d, point2_2d), (point1_3d, point2_3d) in
zip(correspondences, corr_3d):
    warped_point1_3d = np.matmul(rotation, point1_3d) + translation
    warped_point1_2d = warped_point1_3d / warped_point1_3d[2]
    rel_errors.append(np.linalg.norm((warped_point1_2d[:2] -
point2_2d)) / np.linalg.norm(point2_2d))
print(np.mean(rel_errors))
```

```
[(array([-0.04625099, -0.05824104]), array([-0.05940542, -
0.05824104])), (array([-0.01139174,  0.0413723 ]), array([-0.01468035,
0.04268301])), (array([-0.05019732, -0.05824104]), array([-0.06466719,
-0.05824104])), (array([ 0.01886346, -0.06348385]),
array([ 0.02149434, -0.06348385])), (array([ 0.00965535, -
0.03268235]), array([ 0.0109708, -0.032027 ])), (array([-0.01862668,
0.03678485]), array([-0.02191529,  0.0387509 ])), (array([ 0.01491713,
-0.05693034]), array([ 0.01689029, -0.05627499])), (array([-
0.10807682, -0.05758569]), array([-0.10610366, -0.05824104])),
(array([ 0.02544067, -0.00777902]), array([ 0.029387  , -
0.00581296])), (array([-0.02651934, -0.00384691]), array([-0.0284925 ,
-0.00384691])), (array([-0.02059984,  0.00663871]), array([-
0.02323073,  0.00729406])), (array([-0.00744541, -0.0346484 ]),
array([-0.01139174, -0.03399305])), (array([ 0.01820574, -
0.03858051]), array([ 0.01952118, -0.03792516])), (array([-0.05940542,
-0.02875025]), array([-0.07321757, -0.0294056 ])), (array([-
0.01139174,  0.00336195]), array([-0.01402263,  0.0040173 ])),
(array([-0.01665351, -0.05627499]), array([-0.02191529, -
0.05627499])), (array([-0.00547224, -0.01695393]), array([-0.00810313,
-0.01629858])), (array([ 0.01820574, -0.07462481]),
array([ 0.01557485, -0.07396946])), (array([ 0.00242042, -
0.05889639]), array([ 0.00044725, -0.05824104])), (array([0.05438043,
0.14950521]), array([0.05569587, 0.15605872])), (array([-0.08637201,
0.01188151]), array([-0.08637201,  0.01057081])), (array([-0.01862668,
-0.0641392 ]), array([-0.02651934, -0.0641392 ])), (array([0.03662194,
0.0361295 ]), array([0.04583004, 0.03940625])), (array([-0.05348592, -
0.06282849]), array([-0.06992897, -0.06348385])), (array([-0.05019732,
-0.06151779]), array([-0.06729808, -0.06217314])), (array([-
0.01204946,  0.00467265]), array([-0.01599579,  0.00598335])),
(array([-0.09360695, -0.03399305]), array([-0.09689555, -
0.0346484 ])), (array([0.06029992, 0.13181073]), array([0.06687714,
0.13836424])), (array([-0.03704288, -0.05561963]), array([-0.04625099,
-0.05561963])), (array([ 0.03333333, -0.03661446]),
array([ 0.03859511, -0.03530376])), (array([-0.00678769,
0.02760994]), array([-0.0100763 ,  0.02892064])), (array([0.02412523,
0.09969854]), array([0.03070245, 0.10363064])), (array([-0.01862668,
```

0.0747952]), array([-0.02257301, 0.0761059])), (array([0.0260984 ,
-0.00974507]), array([0.03004473, -0.00777902])), (array([-
0.09755328, 0.15999083]), array([-0.11136543, 0.15605872])),
(array([-0.05743225, 0.00598335]), array([-0.06992897,
0.005328])), (array([-0.01402263, 0.01188151]), array([-0.01731123,
0.01253686])), (array([-0.04625099, 0.01319221]), array([-0.05940542,
0.01319221])), (array([-0.13043936, 0.15212661]), array([-0.13109708,
0.14688381])), (array([-0.1074191 , -0.05496428]), array([-0.10478821,
-0.05561963])), (array([-0.09755328, -0.03006095]), array([-
0.09952644, -0.0307163])), (array([-0.02125756, -0.03399305]),
array([-0.0284925 , -0.03399305])), (array([0.01952118, -
0.05693034]), array([0.02346751, -0.05693034])), (array([-0.04690871,
-0.06610525]), array([-0.06203631, -0.06610525])), (array([-
0.01204946, 0.08659152]), array([-0.01599579, 0.08790222])),
(array([0.00570902, -0.01302182]), array([0.00439358, -
0.01171112])), (array([-0.11202315, -0.06479455]), array([-0.11070771,
-0.0654499])), (array([-0.11004999, -0.06086244]), array([-
0.10807682, -0.06151779])), (array([-0.02783478, -0.00581296]),
array([-0.02980795, -0.00581296])), (array([0.00570902, -
0.01498788]), array([0.0050513 , -0.01367717])), (array([-0.0436201,
0.1069074]), array([-0.0587477 , 0.10625205])), (array([-0.0738753 ,
-0.03268235]), array([-0.08374112, -0.0333377])), (array([-
0.04296238, 0.11477161]), array([-0.05677453, 0.11477161])),
(array([-0.07847935, -0.12574218]), array([-0.0830834 , -
0.12508683])), (array([-0.06466719, 0.00598335]), array([-0.07519074,
0.005328])), (array([0.04583004, 0.10756275]), array([0.05503815,
0.11215021])), (array([-0.098211 , 0.16719969]), array([-0.11268087,
0.16261223])), (array([-0.01468035, -0.00712366]), array([-0.02059984,
-0.00646831])), (array([-0.09689555, -0.12049938]), array([-
0.10347277, -0.11984403])), (array([-0.05743225, 0.00794941]),
array([-0.07058669, 0.00729406])), (array([-0.01665351,
0.0413723]), array([-0.02059984, 0.04268301])), (array([-0.05348592,
0.04399371]), array([-0.06664036, 0.04333836])), (array([-0.00876085,
0.04858117]), array([-0.01204946, 0.04989187])), (array([-0.00876085,
0.04661511]), array([-0.01204946, 0.04792581])), (array([-0.00810313,
0.08790222]), array([-0.01204946, 0.08921292])), (array([-0.01402263,
0.08855757]), array([-0.01862668, 0.09052363])), (array([-0.08702973,
0.005328]), array([-0.08834517, 0.00467265])), (array([-0.00612997,
0.08790222]), array([-0.00941857, 0.08986827])), (array([-0.04756643,
0.07938266]), array([-0.06072086, 0.07938266])), (array([-0.04427782,
0.11215021]), array([-0.05808998, 0.11215021])), (array([-0.00284136,
0.04464906]), array([-0.0041568 , 0.04595976])), (array([0.01557485,
-0.04841077]), array([0.01820574, -0.04775542])), (array([-
0.09689555, -0.03858051]), array([-0.09886872, -0.03923586])),
(array([-0.04098921, -0.03268235]), array([-0.05348592, -
0.0333377])), (array([-0.08702973, 0.00926011]), array([-0.08702973,
0.00860476])), (array([-0.098211 , -0.05496428]), array([-0.09886872,
-0.05561963])), (array([-0.06269403, 0.04399371]), array([-
0.0738753 , 0.04333836])), (array([-0.11860037, 0.12656793]),

```
array([-0.1166272 ,  0.12263582])), (array([-0.08637201,
0.01515827]), array([-0.08637201,  0.01384757])), (array([0.05174954,
0.11608231]), array([0.0596422 , 0.12198047])), (array([-0.07584846, -
0.05627499]), array([-0.08439884, -0.05627499])), (array([-0.01468035,
0.09838784]), array([-0.01862668,  0.09969854])), (array([-0.11531176,
-0.08052297]), array([-0.11202315, -0.08052297])), (array([-
0.09492239, -0.00384691]), array([-0.09689555, -0.00450226])),
(array([-0.13043936,  0.14884986]), array([-0.12912391,
0.14360705])), (array([-0.04953959,  0.00729406]), array([-0.06269403,
0.00663871])), (array([-0.11070771, -0.08838718]), array([-0.10873454,
-0.08838718])), (array([-0.03770061, -0.03137165]), array([-
0.04822415, -0.03137165])), (array([-0.00547224,  0.08331477]),
array([-0.00876085,  0.08528082])), (array([-0.0528282, -0.0654499]),
array([-0.06992897, -0.0654499 ])), (array([ 0.01623257, -
0.04447867]), array([ 0.01754801, -0.04316797])), (array([-0.05151276,
0.11739301]), array([-0.06664036,  0.11673766])), (array([-0.04953959,
0.04661511]), array([-0.06269403,  0.04595976])), (array([-0.11465404,
-0.0307163 ]), array([-0.10939227, -0.032027  ])), (array([-
0.0528282 ,  0.04595976]), array([-0.06664036,  0.04530441])),
(array([-0.0771639 , -0.01367717]), array([-0.08571429, -
0.01433253])), (array([-0.01665351, -0.00843437]), array([-0.02323073,
-0.00777902])), (array([-0.03967377, -0.0307163 ]), array([-
0.05085504, -0.0307163 ])), (array([-0.11531176, -0.08248902]),
array([-0.11202315, -0.08314437])), (array([-0.11070771, -
0.09035323]), array([-0.10873454, -0.09035323])), (array([-0.04296238,
0.08855757]), array([-0.05677453,  0.08790222])), (array([-0.01533807,
0.0348188 ]), array([-0.01862668,  0.0361295 ])), (array([-0.10939227,
-0.03858051]), array([-0.10807682, -0.03923586])), (array([-
0.00810313,  0.08200406]), array([-0.01204946,  0.08331477])),
(array([-0.11202315, -0.00450226]), array([-0.10873454, -
0.00581296])), (array([-0.03835833, -0.00646831]), array([-0.05019732,
-0.00646831])), (array([-0.05151276,  0.11411626]), array([-
0.06664036,  0.11346091])), (array([-0.10610366,  0.16588898]),
array([-0.11728493,  0.16195688])), (array([-0.04953959,
0.04202766]), array([-0.06269403,  0.0413723 ])), (array([-0.09360695,
-0.00646831]), array([-0.09623783, -0.00777902]))]
zip  <zip object at 0x15c4906c0>
0.024302595899405602
```