Homework 5

Alexandre Olivé Pellicer
aolivepe@purdue.edu

1 Theory Question

How do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

In RANSAC, the key to differentiating between inliers and outliers for solving the homography estimation problem lies in calculating the estimation error based on a homography matrix computed from a randomly selected subset of keypoint pairs using the Least-Squares method. The RANSAC algorithm repeats N times the following steps in order to differentiate inliers and outliers:

1 **Sampling Interest Points**: First, a small subset of keypoint pairs of size n between the two images is randomly selected. These points are used to compute a homography matrix.

2 **Transformation and Distance Calculation**: Using the computed homography, all keypoints from the first image are transformed to the coordinate space of the second image. This is done using the homogeneous coordinates of the keypoints. For each transformed keypoint, the distance between it and its corresponding matched keypoint in the second image is calculated.

3 **Threshold-Based Classification**: If the distance between the transformed keypoint (estimated point) and its corresponding keypoint in the second image is below a defined threshold (denoted as δ), the point is classified as an inlier. If the distance is above the threshold, it is classified as an outlier.

4 **Stopping Criterion**: RANSAC repeats this process over multiple iterations. When a set of inliers that satisfies the desired conditions (e.g., a minimum number of inliers, denoted as M) is found, the process stops. The final homography is then computed using the entire inlier set.

Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

These three methods are presented to find an optimal homography. Given an initial homography computed using the inlier set of the RANSAC algorithm, the goal is to refine it looking for the optimal homography that minimizes following cost function:

$$C(p) = ||X - f(p)||^2 \tag{1}$$

where p is the vector containing the coefficients of the homography, $f(p)$ represents the projections of the keypoints in the image 1 to the image 2 (estimations) and X contains the actual keypoints in image 2. Therefore, $C(p)$ is the square of the norm of the estimation error vector.

The Levenberg-Marquardt (LM) algorithm combines the strengths of both Gradient Descent (GD) and Gauss-Newton (GN) to offer a fast and stable optimization method. GD is stable when far from the solution but becomes slow as it approaches the global minimum. This happens because the update steps become smaller since the closer to the minimum, the smaller the slope of the cost function. GN is faster near the minimum but unstable when far from it. Gauss-Newton can reach the minimum of the cost function with one step but it is dangerous since it will not work if the initial computed homography is not close to the optimal solution (the mimimum of the cost function). LM is presented as the method that combines the strengths of GD and GN in order to offer the best approach: fast and stable. LM uses a damping factor (μ) to switch between GD and GN. When the initial computed homography is far from the minimum, LM behaves like GD by increasing μ, ensuring stability. When

close to the minimum, LM behaves like GN by reducing μ, speeding up convergence. This balance provides both speed and numerical stability throughout the optimization process. More details about how the LM algorithm works are provided in Section 2.

2 Description of Implementations

2.1 RANSAC Algorithm

We implement the RANSAC algorithm following the steps explained in the first answer of the theory questions. The RANSAC algorithm is implemented in the *ransac* function of the code. Given a pair of images, the input of the *ransac* function are the set of keypoints corresponding to each image as well as their correspondences that have been computed using the SIFT method. The brute force matching method has been used to find the correspondences between the keypoints of the 2 images.

These are the parameters that have been used in the implementation:

- n_{total}: number of keypoints found by the SIFT method
- $n = 20$: number of randomly selected matched keypoints to compute an homography.
- $\delta = 3\sigma$ with $\sigma = 2$: minimum estimation error that needs to be satisfied in order to consider that a keypoint is part of the inlier set.
- $e = 0.4$: probability that a matching of keypoints is an outlier.
- $p = 0.99$: probability that at least one trial in N has no outliers.
- $N = \frac{\ln(1-p)}{\ln(1-(1-\epsilon)^n)}$: number of iterations of the RANSAC algorithm.
- M: minimum number of keypoints that need to have an estimation error below δ in order to consider it the inlier set.

First we select n random keypoints to compute the homography. We compute it using the Least-Squares solution used in previous homeworks. Then, the keypoints from the first image are projected to the second image. If the distance between the projected keypoints from image 1 to image 2 and their actual keypoint correspondences from image 2 is below δ, that correspondence is considered an inlier. Otherwise, it is considered an outlier. This proces is repeated N times randomly selecting every time n different keypoints to compute the homography. If the number of correspondences with distance below δ is higher than M, this is considered the inlier set. At the end of the N iterations, if any set has more correspondences with distance below δ than M, the set that had more correspondences with distance below δ is considered the inlier set.

2.2 Least-Squares method

Since only the ratios of the elements of the homography are important (h_{33}=1), given a pair of correspondences **x**=(x, y, w) and **x'**=(x', y', w') we can write

$$\begin{bmatrix} -y'w \\ x'w \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -w'x & -w'y & -w'w & y'x & y'y \\ w'x & w'y & w'w & 0 & 0 & 0 & -x'x & -x'y \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \tag{2}$$

This equation can be written as:

$$b = Ah, \tag{3}$$

where the dimensions of A are 2×8, h is 8×1 and b is 2×1. This equation can be extrapolated to n pairs of correspondences of keypoints by concatenating the vectors b and matrices A corresponding to

each correspondence of keypoints. Then, A would be of shape $2n \times 8$ and b of shape $2n \times 1$. Now, we can compute the first eight elements of the homography h considering the n pairs of correspondences as:

$$h = (A^T A)^{-1} A^T b \tag{4}$$

The 9-th element of the homography is set to one (h_{33}=1).

2.3 Levenberg-Marquardt (LM) Algorithm Implementation

For the extra credit section, we implemented the Levenberg-Marquardt (LM) algorithm for homography refinement.

The goal of the LM algorithm is finding the coefficients of the homography h (we denote as p the column vector containing the 9 coefficients of h) that minimize the following cost function:

$$C(p) = ||X - f(p)||^2 \tag{5}$$

where p is the vector containing the coefficients of the homography, $f(p)$ represents the projections of the keypoints in the image 1 to the image 2 (estimations) using the homography h (coefficients of p) and X contains the actual keypoints in image 2. $C(p)$ is the square of the norm of the estimation error vector.

2.3.1 Initialization of Damping Parameter (μ_0)

First we compute the Jacobian matrix as follows:

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial p_1} & \frac{\partial f_1}{\partial p_2} & \cdots & \frac{\partial f_1}{\partial p_n} \\ \frac{\partial f_2}{\partial p_1} & \frac{\partial f_2}{\partial p_2} & \cdots & \frac{\partial f_2}{\partial p_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial p_1} & \frac{\partial f_m}{\partial p_2} & \cdots & \frac{\partial f_m}{\partial p_n} \end{pmatrix} \tag{6}$$

The Jacobian matrix J_f contains the derivatives of the reprojection function f with respect to the homography parameters.

Then, we initialize the damping factor μ_0 as:

$$\mu_0 = \tau \cdot \max\left\{\text{diag}(J_f^T J_f)\right\} \tag{7}$$

where:

- J_f is the Jacobian matrix from Equation 6.
- $\tau = 0.5$.

This value is used to start the iterative process.

2.3.2 Iteration and Update Process

The iteration process begins with $\mu_k = \mu_0$ and runs for a total of K_{total} iterations (we set it to 100 iterations). During each iteration, the following steps are taken:

- **Error and Cost Calcultation:** Calculate the error vector e_f as:

  $$e_f = X - f(p_k) \tag{8}$$

  The cost $C(p_k)$ is computed as:

  $$C(p_k) = ||X - f(p_k)||^2 \tag{9}$$

  Note that, initially p_k is the vector with the coefficients of the homography computed using the inlier set.

- **Jacobian Calculation:** Compute Jacobian matrix using p_k as shown in Equation 6.

- **Step Calculation δ_p:** The step δ_p that refines the homography is computed as:

$$\delta_p = \left(J_f^T J_f + \mu_k I\right)^{-1} J_f^T e_f \tag{10}$$

  where I is the identity matrix.

- **Homography Refinement:** The homography parameters are updated using:

$$p_{k+1} = p_k + \delta_p. \tag{11}$$

- **Next Step Cost Calcultation:** $C(p_{k+1})$ is calculated using p_{k+1}.

- **Damping Factor Update (μ):** Update the damping factor ρ using the following formula:

$$\rho = \frac{C(p_k) - C(p_{k+1})}{\delta_p^T (J_f^T J_f \delta_p + \mu \delta_p)} \tag{12}$$

  Based on ρ, update μ for the next iteration:

$$\mu_{k+1} = \mu_k \cdot \max\left\{\frac{1}{3}, 1 - (2\rho - 1)^3\right\} \tag{13}$$

Once the K_{total} iterations have been completed, we would have reached the minimum of the cost function $C(p)$. Therefore, the elements of the last updated vector $p_{K_{total}}$ correspond to the 9 coefficients of the final refined homography.

2.4 Generate Panoramic View

To create the panoramic image, we begin by identifying the central image, image 3. Each of the other images is then transformed relative to this central image by calculating homographies that map their coordinates onto the coordinates of image 3. Then, we calculate the boundaries of the panoramic image by determining the minimum and maximum values from the corners of all images.

Next, we create a large mask to define the dimensions of the panoramic image and apply the corresponding transformations to each image. Each image is warped according to its homography, and the mask is used to accumulate and blend the images onto the panoramic canvas. Finally, we normalize the result to ensure smooth blending and avoid overly bright areas, producing the final panoramic image.

3 Obtained Results

3.1 Task 1

Figure 1 shows the input images that I have used given in the instructions.

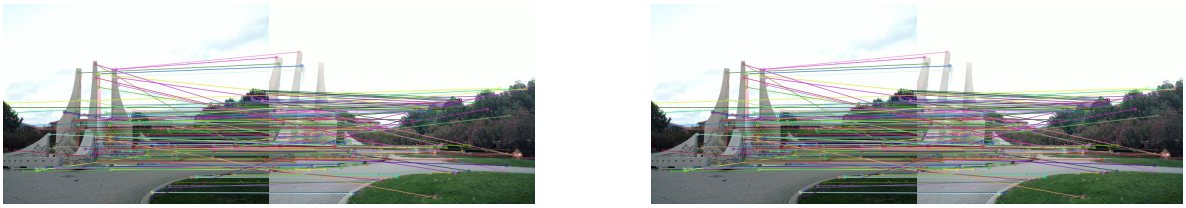
(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4 (e) Image 5

Figure 1: 5 Input images provided with the homework.

Figures 2 and 3 show the keypoints found using the SIFT algorithm in each pair of consecutive images as well as their correspondences.

(a) Image 1 and 2 (b) Image 2 and 3

Figure 2: Keypoints and matchings between 2 images of the set of images provided in the instructions

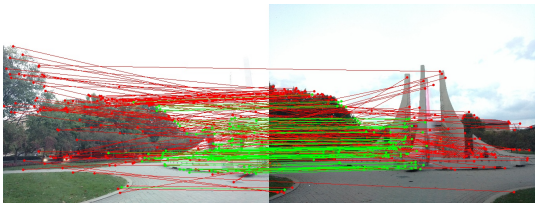


(a) Image 3 and 4 (b) Image 4 and 5

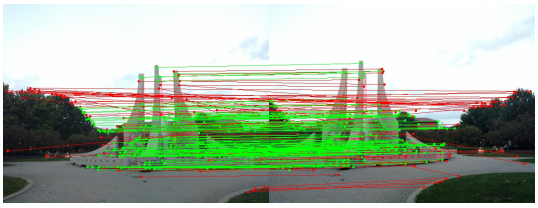
Figure 3: Keypoints and matchings between 2 images of the set of images provided in the instructions

Figures 4 and 5 show the inliers as well as the outliers found using the RANSAC algorithm in each pair of consecutive images. Inliers are painted in green while outliers are painted in red.

Figure 6 shows the panorama built using the homographies computed with the inliers resulting from the RANSAC algorithm.

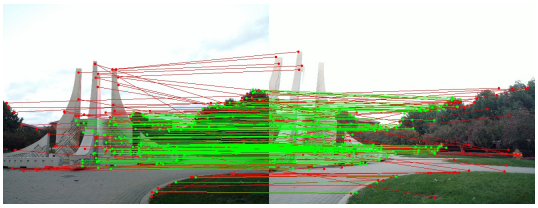


(a) Image 1 and 2

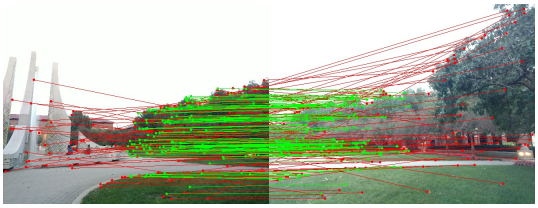


(b) Image 2 and 3

Figure 4: Inliers (green) and outliers (red) between 2 images of the set of images provided in the instructions



(a) Image 3 and 4



(b) Image 4 and 5

Figure 5: Inliers (green) and outliers (red) between 2 images of the set of images provided in the instructions

Figure 7 shows the panorama built using the refined homographies after using our own implementation of the Levenberg-Marquardt (LM) algorithm.

Figure 8 shows the evolution of the geometric error across the different iterations of the Levenberg-Marquardt algorithm for each of the homographies. In Table 1 we show the initial geometric error when just using the RANSAC algorithm and the final geometric error after using the LM algorithm to refine homographies.

Pairs of images	GE after RANSAC	GE after LM
Image 1 and 2	122.47	119.75
Image 2 and 3	187.59	186.75
Image 3 and 4	152.84	151.27
Image 4 and 5	153.57	151.22

Table 1: Comparisson between the geometric error (GE) using the homographies using the inliers of the RANSAC algorithm and after refining the homographies using the LM algorithm.

3.1.1 Observations

Comparing the panoramas built before and after refining the homographies, we can conclude that the inlier finding task of the RAMSAC algorithm is very good since there is almost no difference between the two panoramas. This qualitative evaluation is supported by the quantitave metrics of the geometric error. After refining the homographies we see that there is a slight reduction of the geometric error for each homography. In Figure 8 we can clearly see how the geometric error is smaller in each iteration until it jumps to the global minimum of the cost function. Once the global minimum is reached, there is no more improvement.

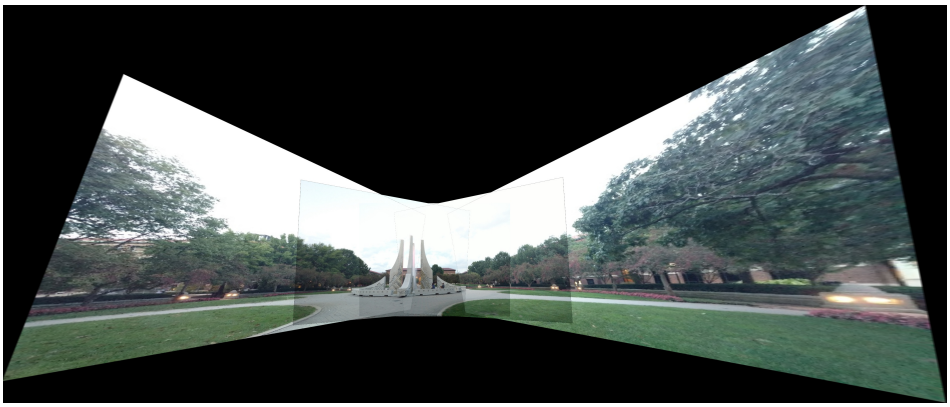


Figure 6: Panorama using the homographies computed with the inliers from the RANSAC algorithm.

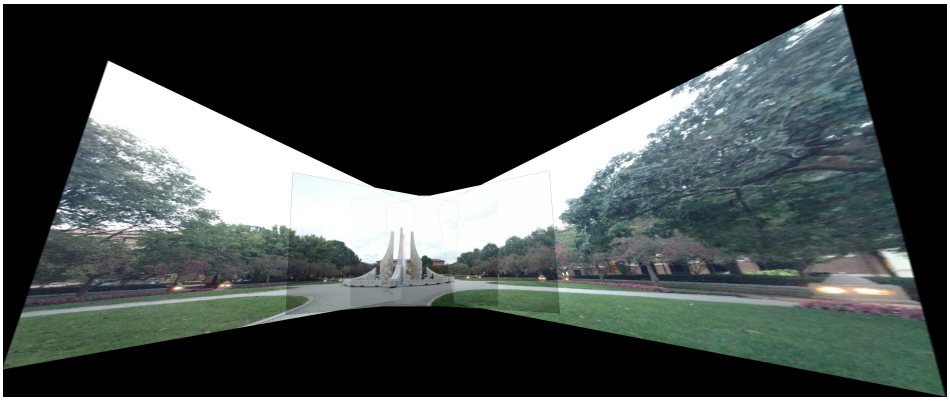


Figure 7: Panorama using the homographies refined with the Levenberg-Marquardt (LM) algorithm.

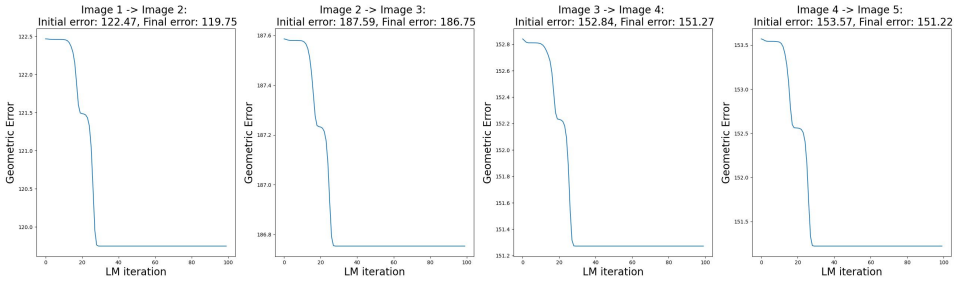


Figure 8: Evolution of the geometric error across the different iterations of the Levenberg-Marquardt algorithm for each of the homographies.

3.2 Task 2

Figure 9 shows the input images that I have taken.

(a) Image 1 (b) Image 2 (c) Image 3 (d) Image 4 (e) Image 5

Figure 9: 5 Input images taken by me.

Figures 10 and 11 show the keypoints found using the SIFT algorithm in each pair of consecutive images as well as their correspondences.

(a) Image 1 and 2 (b) Image 2 and 3

Figure 10: Keypoints and matchings between 2 images of the set of images that I have taken.

Figures 12 and 13 show the inliers as well as the outliers found using the RANSAC algorithm in each pair of consecutive images. Inliers are painted in green while outliers are painted in red.

Figure 14 shows the panorama built using the homographies computed with the inliers resulting from the RANSAC algorithm.

(a) Image 3 and 4

(b) Image 4 and 5

Figure 11: Keypoints and matchings between 2 images of the set of images that I have taken.

(a) Image 1 and 2

(b) Image 2 and 3

Figure 12: Inliers (green) and outliers (red) between 2 images of the set of images that I have taken.

Figure 15 shows the panorama built using the refined homographies after using our own implementation of the Levenberg-Marquardt (LM) algorithm.

Figure 16 shows the evolution of the geometric error across the different iterations of the Levenberg-Marquardt algorithm for each of the homographies. In Table 2 we show the initial geometric error when just using the RANSAC algorithm and the final geometric error after using the LM algorithm to refine homographies.

Pairs of images	GE after RANSAC	GE after LM
Image 1 and 2	12237.72	236.33
Image 2 and 3	6201.89	277.44
Image 3 and 4	247.16	247.04
Image 4 and 5	6856.90	278.84

Table 2: Comparisson between the geometric error (GE) using the homographies using the inliers of the RANSAC algorithm and after refining the homographies using the LM algorithm.

3.2.1 Observations

Comparing the panoramas built before and after refining the homographies, we can conclude that the task of the LM algorithm for this set of images is very important since we can see a clear difference between the panorama built with the non-refined homographies and the panorama built with the refined homographies. From the panorama built before refining homographies, we see that there are some blurry elements. In the panorama built after refining the homographies most of the blurry elements disappear. In the final panorama, we see that the black car closer to the point where the images have been taken looks a bit blur while all the elements that are more far look neat. From

(a) Image 3 and 4

(b) Image 4 and 5

Figure 13: Inliers (green) and outliers (red) between 2 images of the set of images that I have taken.

Figure 14: Panorama using the homographies computed with the inliers from the RANSAC algorithm.

this observation we can say that the overall approach struggles a bit with objects that are closer to the point where the image has been taken while performs very well. This qualitative evaluation is supported by the quantitative metrics of the geometric error. After refining the homographies we see that there is an important reduction of the geometric error for homographies 1 (image 1 $\rightarrow$ image 2), 2 (image 2 $\rightarrow$ image 3) and 4 (image 4 $\rightarrow$ image 5). In Figure 16 we can clearly see how the geometric error is smaller in each iteration until it jumps to the global minimum of the cost function. Once the global minimum is reached, there is no more improvement.

Figure 15: Panorama using the homographies refined with the Levenberg-Marquardt (LM) algorithm.

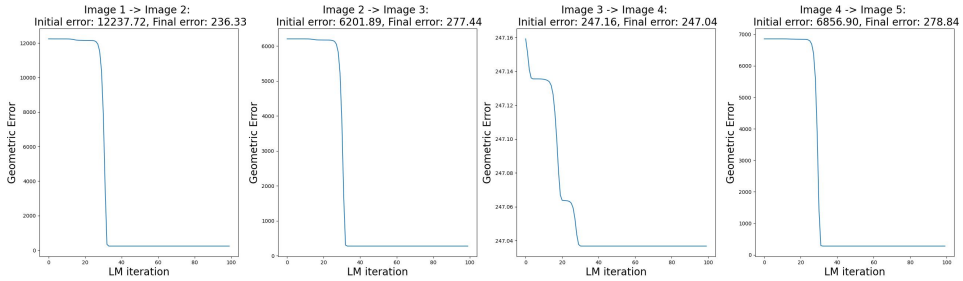


Figure 16: Evolution of the geometric error across the different iterations of the Levenberg-Marquardt algorithm for each of the homographies.

CODE:

```
import matplotlib.pyplot as plt
import random
import numpy as np
import cv2
from scipy.ndimage import maximum_filter
import math
from einops import rearrange
from tqdm import trange , tqdm
import os
from scipy.optimize import least_squares, approx_fprime

np.random.seed(42)

def ImageStitch(images, homographies):
    # homographies:
    # [0] img1->img2
    # [1] img2->img3
    # [2] img3->img4
    # [3] img4->img5

    center_index = len(images) // 2
    updated_homographies = [None] * (len(homographies) + 1)

    # Initialize the transformation for the center image
    updated_homographies[center_index] = np.eye(3)

    # Calculate transformations to the left of the center image
    for i in range(center_index - 1, -1, -1):
        updated_homographies[i] = updated_homographies[i + 1] @
homographies[i]

    # Calculate transformations to the right of the center image
    for i in range(center_index, len(homographies)):
        updated_homographies[i + 1] = updated_homographies[i] @
np.linalg.inv(homographies[i])

    homographies = updated_homographies

    # Find the final dimensions of the image following the method from TA
in Piazza
    full_corners = []
    # Get corners of each image
    for img, H in zip(images, homographies):
        h, w = img.shape[:2]
        corners = np.array([[0, 0, 1],
                            [w, 0, 1],
                            [w, h, 1],
```

```
                            [0, h, 1]
                            ])
        corners_hc = corners.T
        corners_transformed_hc = np.matmul(H, corners_hc)
        corners_homogeneous = corners_transformed_hc /
(corners_transformed_hc[-1, :])
        full_corners.append(corners_homogeneous.T)
    full_corners = np.concatenate(full_corners, axis=0)

    # Find the maximum and minimums corners
    x_min = int(min(full_corners[:, 0]))
    x_max = int(max(full_corners[:, 0]))
    y_min = int(min(full_corners[:, 1]))
    y_max = int(max(full_corners[:, 1]))

    # Final dimensions of the image
    final_size = (y_max - y_min, x_max - x_min)

    # Translation to put all the images inside the found limits
    trans = np.array([[1, 0, -x_min],
                      [0, 1, -y_min],
                      [0, 0, 1]])

    # Adjust homographies by applying the offset translation
    adjusted_homographies = [trans @ h for h in homographies]

    # Initialize panoramic and mask accumulator for blending
    panoramic = np.zeros((final_size[0], final_size[1], 3),
dtype=np.float32)
    full_mask = np.zeros(final_size, dtype=np.float32)

    for img, h in zip(images, adjusted_homographies):
        # Apply the adjusted homography to each image
        warped = cv2.warpPerspective(img, h, final_size[::-1])

        # Create mask to identify visible areas in the warped image
        mask = (cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY) >
0).astype(np.float32)

        # Blend each warped image onto the panoramic
        panoramic += warped * np.expand_dims(mask, axis=2)
        full_mask += mask

    # Normalize the panoramic to avoid bright spots
    panoramic /= (np.expand_dims(full_mask, axis=2) + 1e-8)
    panoramic = np.clip(panoramic, 0, 255).astype(np.uint8)

    return panoramic
```

```
# Method to get homography h by least squares method
def obtain_h(vector_x, vector_x_prima):
    # Build matrix A and vector b
    A = []
    b = []
    for x, x_prima in zip(vector_x, vector_x_prima):
        A += [[0, 0, 0, -x_prima[2]*x[0], -x_prima[2]*x[1], -
x_prima[2]*x[2], x_prima[1]*x[0], x_prima[1]*x[1]],
              [x_prima[2]*x[0], x_prima[2]*x[1], x_prima[2]*x[2], 0, 0,
0, -x_prima[0]*x[0], -x_prima[0]*x[1]]]

        b += [[-x_prima[1]*x[2]],
              [x_prima[0]*x[2]]]

    # Compute homography and reshape to 3x3
    A = np.array(A)
    b = np.array(b)
    h = np.linalg.pinv(A.T @ A) @ A.T @ b
    h = np.append(h, 1)
    h = h.reshape(3, 3)
    return h

def get_inliers(hc_kp1, hc_kp2, h, delta):
    # Get projected points in img1 to img2 (estimations)
    projected_hc_kp2 = h @ hc_kp1.T
    projected_kp2 = projected_hc_kp2/projected_hc_kp2[2]
    # Compute distance between actual point and estimated one
    dist_x_y = (hc_kp2[:, :2] - projected_kp2.T[:, :2])**2
    dist = np.sum(dist_x_y, axis=1)
    # Inlier points are the points with distance below delta
    inliers_pos =  np.where(dist <= delta)
    return inliers_pos[0]

def ransac(hc_kp1, hc_kp2, n = 20, sigma = 2, p = 0.99, e = 0.4):
    # increase n to increase N
    # increase p to increase N
    # increase e to increase N and decrease M
    # Set parameters of the ransac algorithm
    delta = 3* sigma
    N = int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n)))
    print("N: ", N)
    n_total = len(hc_kp1)
    M = (1-e)*n_total
    previous_len = 0

    # Iterate over N
    for i in range(N):
        # Select n random keypoints and compute homography with them
        random_idx = np.random.randint(n_total, size = n)
```

```
        random_kp1 = hc_kp1[random_idx]
        random_kp2 = hc_kp2[random_idx]
        h = obtain_h(random_kp1, random_kp2)

        # Use the computed homography to determine the inlier set
        inliers_pos = get_inliers(hc_kp1, hc_kp2, h, delta)

        # Save inlier set if it was bigger than previous one and stop
iterating if inlier set is bigger than M
        if len(inliers_pos) > previous_len:
            previous_len = len(inliers_pos)
            final_inliers_pos = inliers_pos
            if len(inliers_pos) > M:
                break
    print("iterartion: ", i, "Num inliers: ", len(final_inliers_pos))
    return final_inliers_pos

def putPointsAndLines(img1, img2, hc_inliers_kp1, hc_inliers_kp2,
hc_outliers_kp1, hc_outliers_kp2):
    # Paint keypoints and matches in the concatenation of the 2 images
    concatenated_image = np.hstack((img1, img2))
    img1_width = img1.shape[1]

    # Paint outliers
    for pt1, pt2 in zip(hc_outliers_kp1, hc_outliers_kp2):
        cv2.circle(concatenated_image, (int(pt1[0]), int(pt1[1])),
radius=3, color=(0, 0, 255), thickness=-1)
        pt2_adjusted = (int(pt2[0] + img1_width), int(pt2[1]))
        cv2.circle(concatenated_image, pt2_adjusted, radius=3, color=(0,
0, 255), thickness=-1)
        cv2.line(concatenated_image, (int(pt1[0]), int(pt1[1])),
pt2_adjusted, color=(0, 0, 255), thickness=1)

    # Paint inliers
    for pt1, pt2 in zip(hc_inliers_kp1, hc_inliers_kp2):
        cv2.circle(concatenated_image, (int(pt1[0]), int(pt1[1])),
radius=3, color=(0, 255, 0), thickness=-1)
        pt2_adjusted = (int(pt2[0] + img1_width), int(pt2[1]))
        cv2.circle(concatenated_image, pt2_adjusted, radius=3, color=(0,
255, 0), thickness=-1)
        cv2.line(concatenated_image, (int(pt1[0]), int(pt1[1])),
pt2_adjusted, color=(0, 255, 0), thickness=1)
    return concatenated_image

def error_function(p, kp1_hc, kp2_hc):
    h = p.reshape(3, 3)
    # Get projected points in img1 to img2 (estimations)
    project_kp2_hc = h @ kp1_hc.T
    project_kp2_hc = project_kp2_hc / project_kp2_hc[2]
```

```
    project_kp2_hc = project_kp2_hc.T
    # Get only (x, y) coordinates of the estimation and the actual points
    kp2 = kp2_hc[:, :-1]
    project_kp2 = project_kp2_hc[:, :-1]
    # Compute error between the estimation and the actual points
    X = kp2.reshape(kp2.shape[0]*2)
    f = project_kp2.reshape(project_kp2.shape[0]*2)
    e = X - f
    return e

def jacobian(kp1_hc, h):
    num_points = len(kp1_hc)
    J = np.zeros((num_points * 2, 9))  # Initialize the Jacobian matrix

    for i, pt in enumerate(kp1_hc):
        # Apply homography transformation
        transformed_pt = h @ pt

        # Build Jacobian matrix
        x, y, _ = pt
        fx, fy, fw = transformed_pt
        inv_fw = 1 / fw
        inv_fw2 = inv_fw ** 2
        J[2 * i, :3] = np.array([x * inv_fw, y * inv_fw, inv_fw])
        J[2 * i, 6:] = np.array([-x * fx * inv_fw2, -y * fx * inv_fw2, -
fx * inv_fw2])
        J[2 * i + 1, 3:6] = np.array([x * inv_fw, y * inv_fw, inv_fw])
        J[2 * i + 1, 6:] = np.array([-x * fy * inv_fw2, -y * fy *
inv_fw2, -fy * inv_fw2])

    return J

def levemberg_marquardt(p, hc_kp1, hc_kp2, r = 0.5, K_total = 100, th =
1e-20):
    # r = 0.5, K_total = 50000, th = 1e-13
    # Build Jacobian
    J = jacobian(hc_kp1, p.reshape(3, 3))
    # Estimate initial mu
    mu = r*np.max(np.diagonal(J.T @ J))
    # Vector to save how the geometric error is reduced
    C_p_vector = []
    # Refine for 100 iterations
    for i in range(K_total):
        # Get error
        e = error_function(p, hc_kp1, hc_kp2)
        # Get cost
        C_p = np.linalg.norm(e)**2
        # Update Jacobian with new p
        J = jacobian(hc_kp1, p.reshape(3, 3))
```

```
        # Get delta_p
        delta_p = np.linalg.inv(J.T @ J + mu * np.eye(9)) @ J.T @ e
        # Update p
        p = p + delta_p
        # Compute phi
        C_p_plus_1 = np.linalg.norm(error_function(p, hc_kp1, hc_kp2))**2
        phi = (C_p - C_p_plus_1) / (delta_p.T @ J.T @ e + delta_p.T @ (mu
* np.eye(9)) @ delta_p)
        # Update mu
        mu = mu * max(1/3, 1-(2*phi-1)**3)
        # Append geometric error
        C_p_vector.append(C_p_plus_1)
    return p, C_p_vector

if __name__ == '__main__':
    # Read images
    img1 = cv2.imread(f"/home/aolivepe/HW5/my_HW5_images_2/1.jpg")
    img2 = cv2.imread(f"/home/aolivepe/HW5/my_HW5_images_2/2.jpg")
    img3 = cv2.imread(f"/home/aolivepe/HW5/my_HW5_images_2/3.jpg")
    img4 = cv2.imread(f"/home/aolivepe/HW5/my_HW5_images_2/4.jpg")
    img5 = cv2.imread(f"/home/aolivepe/HW5/my_HW5_images_2/5.jpg")
    images = [img1, img2, img3, img4, img5]

    output_folder = "./mine_2/" # instructions or mine
    num_keypoints = 250
    refined_homographies = []
    homographies = []
    fig, axes = plt.subplots(1, 4, figsize=(8*4, 8))

    # Iterate over the given images
    for k in range(len(images)-1):
        print(f"IMAGES {k+1} AND {k+2}")
        image1 = images[k]
        image2 = images[k+1]

        # Use sift method to find keypoints and matches
        sift_detector = cv2.SIFT_create()
        kp1, vector1 =
sift_detector.detectAndCompute(cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY),
None)
        kp2, vector2 =
sift_detector.detectAndCompute(cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY),
None)
        raw_matches = cv2.BFMatcher(cv2.NORM_L1,
crossCheck=True).match(vector1, vector2)
        sorted_matches = []
        for match in raw_matches:
            sorted_matches.append(match)
            sorted_matches.sort(key=lambda m: m.distance)
```

```
        # Get homogeneous coordinates of the keypoints that have matched
        hc_kp1 = []
        hc_kp2 = []
        for i in range(len(sorted_matches)):
            hc_kp1.append((int(kp1[sorted_matches[i].queryIdx].pt[0]),
int(kp1[sorted_matches[i].queryIdx].pt[1]), 1))
            hc_kp2.append((int(kp2[sorted_matches[i].trainIdx].pt[0]),
int(kp2[sorted_matches[i].trainIdx].pt[1]), 1))
        hc_kp1 = np.array(hc_kp1)
        hc_kp2 = np.array(hc_kp2)

        # Generate image with the 2 images concatenated showing the
matchings
        final_image = cv2.drawMatches(image1, kp1, image2, kp2,
sorted_matches[:num_keypoints], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        cv2.imwrite(f"{output_folder}correspondences_{k}.jpg",
final_image)

        # Use ransac algorithm to determine which matchings correspond to
inliers and which ones correspond to outliers
        final_inliers_pos = ransac(hc_kp1[:num_keypoints],
hc_kp2[:num_keypoints])

        # Get the inliers and outliers
        kp1 = np.array(kp1)
        kp2 = np.array(kp2)
        sorted_matches = np.array(sorted_matches)
        inliers_matches =
sorted_matches[:num_keypoints][final_inliers_pos]
        outliers_matches = np.delete(sorted_matches[:num_keypoints],
final_inliers_pos)

        # These vectors will contain the tuples of the coordinates of the
inliers and the outliers
        hc_inliers_kp1 = []
        hc_inliers_kp2 = []
        hc_outliers_kp1 = []
        hc_outliers_kp2 = []

        for match in inliers_matches:
            # Get the index of the matched inliers
            img1_idx = match.queryIdx  # Index in keypoints1
            img2_idx = match.trainIdx  # Index in keypoints2

            # Get the coordinates of the matched inliers
            (x1, y1) = kp1[img1_idx].pt  # Coordinates in image 1
            (x2, y2) = kp2[img2_idx].pt  # Coordinates in image 2
```

```
            # Add tuple of coordinates in the vector
            hc_inliers_kp1.append((int(x1), int(y1), 1))
            hc_inliers_kp2.append((int(x2), int(y2), 1))

        hc_inliers_kp1 = np.array(hc_inliers_kp1)
        hc_inliers_kp2 = np.array(hc_inliers_kp2)

        for match in outliers_matches:
            # Get the index of the matched outliers
            img1_idx = match.queryIdx  # Index in keypoints1
            img2_idx = match.trainIdx  # Index in keypoints2

            # Get the coordinates of the matched outliers
            (x1, y1) = kp1[img1_idx].pt  # Coordinates in image 1
            (x2, y2) = kp2[img2_idx].pt  # Coordinates in image 2

            # Add tuple of coordinates in the vector
            hc_outliers_kp1.append((int(x1), int(y1), 1))
            hc_outliers_kp2.append((int(x2), int(y2), 1))

        hc_outliers_kp1 = np.array(hc_outliers_kp1)
        hc_outliers_kp2 = np.array(hc_outliers_kp2)

        # Generate image with the 2 images concatenated showing the
matchings between inliers and outliers
        final_image = putPointsAndLines(image1, image2, hc_inliers_kp1,
hc_inliers_kp2, hc_outliers_kp1, hc_outliers_kp2)
        cv2.imwrite(f"{output_folder}inliers_outliers_{k}.jpg",
final_image)

        # Use inliers to get the final homography
        final_h = obtain_h(hc_inliers_kp1 , hc_inliers_kp2)
        homographies.append(final_h)
        final_h = final_h.reshape(9)

        # Refine the final homography using our implementation of the
Levemberg Marquardt (LM) algorithm
        final_h, C_p_vector = levemberg_marquardt(final_h,
hc_inliers_kp1, hc_inliers_kp2)

        # Save homography to build panoramic later
        final_h = final_h.reshape(3, 3)
        refined_homographies.append(final_h)

        # Plot geometric error across iterations of the LM algorithm
        axes[k].plot(C_p_vector)
        axes[k].set_xlabel("LM iteration", fontsize=20)
        axes[k].set_ylabel("Geometric Error", fontsize=20)
```

```
        axes[k].set_title(f"Image {k+1} -> Image {k+2}: \n Initial error:
{C_p_vector[0]:.2f}, Final error: {C_p_vector[-1]:.2f}", fontsize=20)
        print(f"Image {k+1} -> Image {k+2}: \n Initial error:
{C_p_vector[0]:.2f}, Final error: {C_p_vector[-1]:.2f}")

    # Create panoramics and save them
    plt.savefig(f"{output_folder}error.jpg")
    panoramic_ransac = ImageStitch(images, homographies)
    panoramic_after_LM = ImageStitch(images, refined_homographies)
    cv2.imwrite(f"{output_folder}panoramic_ransac.jpg", panoramic_ransac)
    cv2.imwrite(f"{output_folder}panoramic_after_LM.jpg",
panoramic_after_LM)
```