Homework 5

Sravani Ramishetty(sramishe@purdue.edu)

Theory Question 1

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?

We differentiate between the inliers and the outliers using the decision threshold. For each data point, we compute its projection using the estimated homography matrix, if the distance between the actual data point and the projected point is less than the threshold, we call it an inlier else an outlier.

Mathematically, let $\mathbf{x}_i$ and $\mathbf{x}'_i$ be the corresponding points in the two images, and $\mathbf{H}$ be the estimated homography matrix. We project $\mathbf{x}_i$ using $\mathbf{H}$ as:

$$\mathbf{x}'_i \sim \mathbf{H}\mathbf{x}_i$$

We then calculate the distance d_i between the actual point $\mathbf{x}'_i$ and the projected point $\mathbf{H}\mathbf{x}_i$ as:

$$d_i = \left\|\mathbf{x}'_i - \mathbf{H}\mathbf{x}_i\right\|$$

If d_i is less than the threshold δ, the point is classified as an inlier:

$$d_i < \delta \implies \text{inlier}$$

Otherwise, it is classified as an outlier:

$$d_i \geq \delta \implies \text{outlier}$$

Theory Question 2

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.

LM can switch from GN to GD using the damping coefficient(μ). If the parameter is close to local minima, LM switches to GN and when it is far from local minima, it switches to GD.

- When the solution is far from the local minimum, the damping parameter μ is large, and the LM algorithm behaves like GD. In this regime, it takes smaller steps, focusing on stabilizing the optimization process.
- As the solution approaches the local minimum, μ becomes smaller, causing the LM algorithm to behave more like GN. In this case, it takes larger, more efficient steps that converge faster.

Task 1: Automated Homography Estimation using RANSAC and Nonlinear Least-Squares Refinement

The objective of this homework is to implement a fully automated approach for robust homography estimation between two images using RANSAC and refine it using a Nonlinear Least-Squares minimization approach, specifically the Levenberg-Marquardt (LM) algorithm.

The following steps outline the automation of homography estimation between two images:

Step 1: Interest Point Detection and Descriptor Extraction Interest points and their corresponding descriptors are extracted from both images and compared using a similarity criterion. The following techniques can be used for interest point extraction and descriptor computation:

- **Harris Corner Detector:** Detect interest points using the Harris corner detector. Then, for each interest point, compute a gray-level descriptor using a $W \times W$ window around the point. The similarity between corresponding points can be calculated using either:
  - Sum of Squared Differences (SSD)
  - Normalized Cross-Correlation (NCC) (preferred for improved robustness)
- **SIFT, SURF, or SuperPoint+SuperGlue:** Alternatively, we can use more advanced techniques such as SIFT, SURF, or SuperPoint+SuperGlue to extract both interest points and their descriptors. In these cases, the similarity between interest points is calculated using the Euclidean distance between the descriptor vectors(SuperPoint+SuperGlue gave best results in this homework).

Step 2: Linear least square minimization to obtain the initial homography

Given a point X in a planar scene and its corresponding pixel X' in the image plane, for most cameras, we can express the relationship between them as:

$$X' = HX$$

where both X and X' are expressed in homogeneous coordinates:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad X' = \begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix}$$

and H is the 3×3 homography matrix, given by:

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}$$

Expanding the matrix multiplication for $X' = HX$, we obtain the following equations for the components of X':

$$x'_1 = h_{11}x_1 + h_{12}x_2 + h_{13}x_3$$

$$x_2' = h_{21}x_1 + h_{22}x_2 + h_{23}x_3$$
$$x_3' = h_{31}x_1 + h_{32}x_2 + h_{33}x_3$$

Denoting the physical scene coordinates by (x, y) and the physical pixel coordinates by (x', y'), we have the following relationships:

$$x = \frac{x_1}{x_3}, \quad y = \frac{x_2}{x_3}$$
$$x' = \frac{x_1'}{x_3'}, \quad y' = \frac{x_2'}{x_3'}$$

Thus, we can express the physical coordinates of the image pixel as:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

These expressions for the physical pixel coordinates can be rewritten in the homogeneous equation form:

$$xh_{11} + yh_{12} + h_{13} - xx'h_{31} - yx'h_{32} - x'h_{33} = 0$$
$$xh_{21} + yh_{22} + h_{23} - xy'h_{31} - yy'h_{32} - y'h_{33} = 0$$

Converting this into a matrix format and in-homogeneous form with $h_{33} = 1$ with 1 point gives below system of in-homogeneous equations:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \end{bmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} = \begin{pmatrix} x_1' \\ y_1' \end{pmatrix}$$

This system can be represented in the general form as $Ah = b$, where A is a 2×8 matrix of known values based on the coordinates of the point (x_1, y_1) and the corresponding transformed point (x_1', y_1'), h is the vector of unknown homography parameters, and b is a 2×1 vector representing the coordinates of the transformed point.

System of Equations for Multiple Correspondences

For N point correspondences $(x_i, y_i) \leftrightarrow (x_i', y_i')$, where $i = 1, 2, \ldots, N$, we can write a system of equations for each correspondence (X_i, X_i'), represented as $A_ih = b_i$ for the i-th correspondence.

By stacking the equations for all N correspondences, we obtain a system of $2N$ inhomogeneous equations that can be expressed as:

$$Ah = b$$

where A is a $2N \times 8$ matrix of known values, b is a $2N \times 1$ vector of known values, and h is an 8×1 vector of unknown homography parameters.

Solving the Overdetermined System

In general, for $N \geq 4$, the system is overdetermined (i.e., more equations than unknowns), and we solve it using the pseudo-inverse of A. The solution is given by:

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

This provides the least-squares solution for the homography parameters, minimizing the error between the projected and actual points in the target image.

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

Figure 1: Given Input Images

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

Figure 2: Collected Input Images

Step 3: RANSAC Algorithm for Homography Estimation and outlier rejection(Pseudocode)

Algorithm 1: RANSAC for Homography Estimation

Input: Matched keypoints between two images, δ, ϵ, N, M
Output: Best homography matrix H_{best}, list of inliers

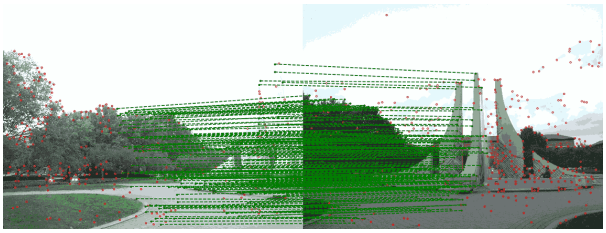
Initialize:
$H_{\text{best}} \leftarrow$ None
best_inliers $\leftarrow 0$
best_inliers_list $\leftarrow$ None
$n_{\text{total}} \leftarrow$ number of matched keypoints
$p \leftarrow 0.99$
$n \leftarrow$ minimum number of points to estimate homography
max_iters $\leftarrow \frac{\log(1-p)}{\log(1-(1-\epsilon)^n)}$
$M \leftarrow (1-\epsilon) \times n_{\text{total}}$
max_iters_all $\leftarrow \frac{n_{\text{total}}!}{n!(n_{\text{total}}-n)!}$

while *iterations* $<$ *max_iters_all* **do**
  Step 1: Randomly select n correspondences
  Step 2: Compute homography matrix H from the selected points
  Step 3: Compute inliers by calculating error for each correspondence
  for *each correspondence* j **do**
    Compute $p_1 \leftarrow (x_j, y_j, 1)^T$
    Compute $p_2 \leftarrow (x'_j, y'_j, 1)^T$
    error $\leftarrow ||p_2 - H \cdot p_1||_2$
    if *error* $< \delta$ **then**
      Increment the inliers count
      Add correspondence j to inliers list
    end
  end
  if *inliers count* $> M$ **then**
    if *inliers count* $>$ *best_inliers* **then**
      Update $H_{\text{best}} \leftarrow H$
      Update best_inliers_list $\leftarrow$ current inliers list
    end
    Break
  end
end

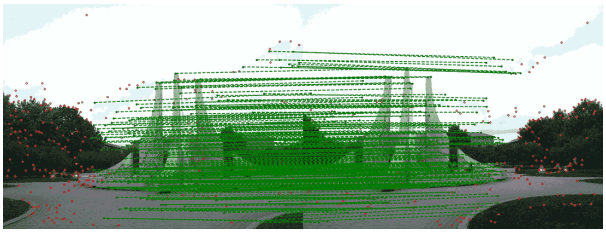
Return: H_{best}, best_inliers_list

In this implementation, the number of iterations exceeded the theoretical value $\frac{\log(1-p)}{\log(1-(1-\epsilon)^n)}$, as the keypoints detected by SIFT contained a significant number of outliers, which prevented the algorithm from finding the best homography matrix. In contrast, the iterations required for SuperPoint+SuperGlue were considerably fewer compared to SIFT. Additionally, the threshold δ used for SuperPoint+SuperGlue (25) was smaller than that for SIFT (75) in order to achieve the desired number of inliers. Once the best homography matrix H_{best} and the corresponding list of inliers were identified, the homography was refined using all the inliers found. The homography was then further optimized using the `'lm' from least_squares` method from `scipy.optimize`.

Extracted correspondences using SuperPoint+SuperGlue between given input images



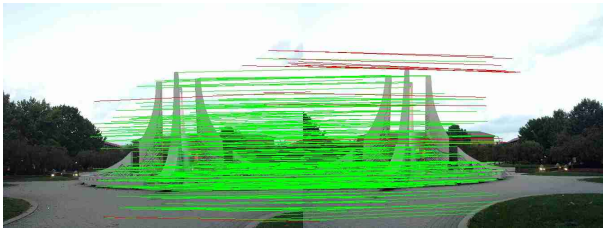
(a) Image 1 and 2 correspondences



(b) Image 2 and 3 correspondences

Figure 3: Correspondences using SuperPoint+SuperGlue

Sets of outliers(red) and inliers(green) for given input images

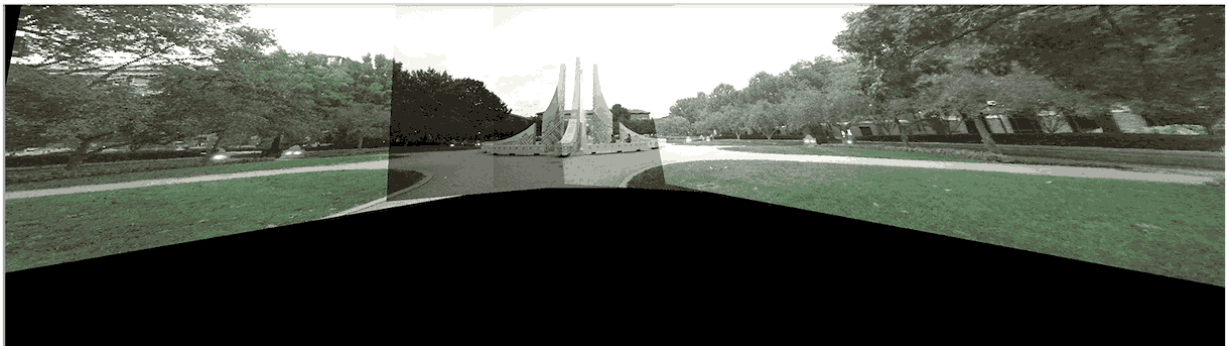


(a) outliers-inliers for Image 1 and 2

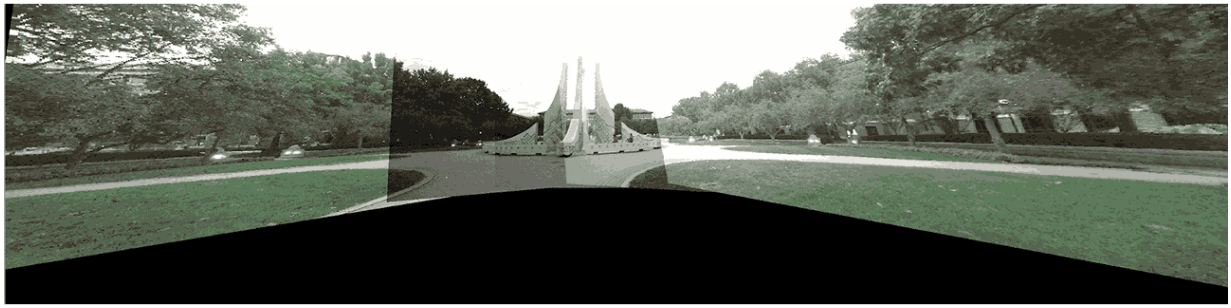
(b) outliers-inliers for Image 2 and 3

Figure 4: outliers-inliers after applying RANSAC

Panoramic view obtained after stitching given input images

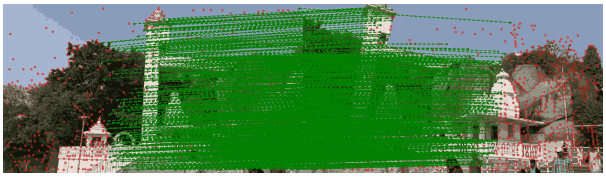


(a) Panoramic view after applying RANSAC

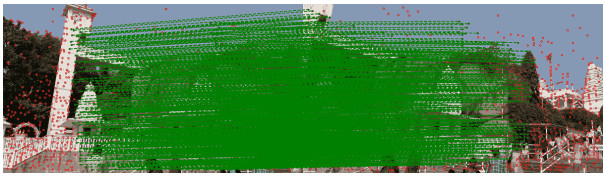


(b) Panoramic view after applying RANSAC and LM

Extracted correspondences using SuperPoint+SuperGlue between collected input images



(a) Image 1 and 2 correspondences



(b) Image 2 and 3 correspondences

Figure 6: Correspondences using SuperPoint+SuperGlue

Sets of outliers (red) and inliers (green) for collected input images

(a) outliers-inliers for Image 1 and 2

(b) outliers-inliers for Image 2 and 3

Figure 7: outliers-inliers after applying RANSAC

Panoramic view obtained after stitching collected input images

(a) Panoramic view after applying RANSAC

(b) Panoramic view after applying RANSAC and LM

Step 4: Extra Credit (Pseudocode)

Algorithm 1: Levenberg-Marquardt for Homography Refinement

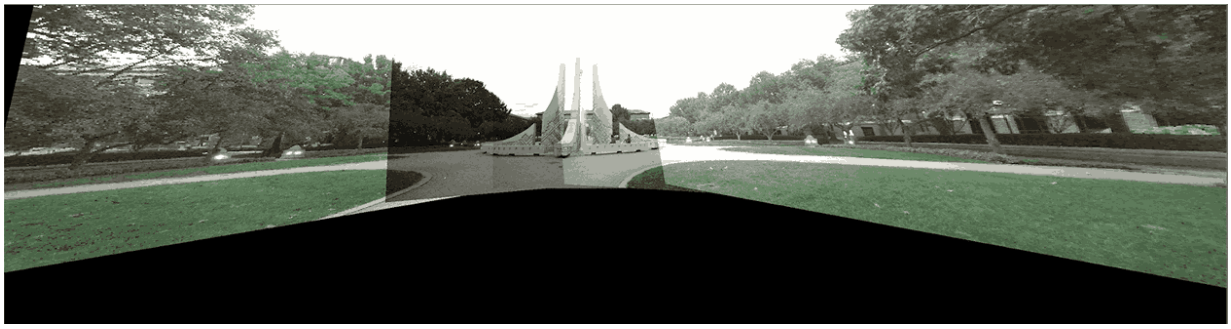
Input: Initial homography matrix H, set of inliers
Output: Refined homography matrix H_{refined}

Initialize:
$\epsilon_1, \epsilon_2, \epsilon_3 \leftarrow 10^{-15}$, max_iterations $\leftarrow 100$
$\nu \leftarrow 2, \tau \leftarrow 0.5$
$h \leftarrow H$ flattened into a 9D vector
iterations $\leftarrow 0$

while *iterations* $<$ *max_iterations* **do**
  Step 1: Compute residuals and Jacobian matrix $J \leftarrow \text{Jacobian}(H, \text{inliers})$
  Compute cost function $e \leftarrow \text{cost\_function}(h, \text{inliers})$
  Step 2: Compute gradient and approximate Hessian
  $g \leftarrow -J^T \cdot e$
  $A \leftarrow J^T \cdot J$
  Step 3: Check for convergence based on gradient
  if $||g||_\infty < \epsilon_1$ **then**
    break
  end
  Step 4: Compute λ and update step
  $I \leftarrow$ identity matrix of size 9
  $\lambda \leftarrow \tau \times \max(\text{diag}(A))$
  while *True* **do**
    Solve for Δh:
    $\Delta h \leftarrow (A + \lambda \cdot I)^{-1} \cdot g$
    if $||\Delta h|| \leq \epsilon_2 \cdot (||h|| + \epsilon_2)$ **then**
      break
    end
    $h_{\text{new}} \leftarrow h + \Delta h$
    $e_{\text{new}} \leftarrow \text{cost\_function}(h_{\text{new}}, \text{inliers})$
    Compute ρ:
    $\rho \leftarrow \frac{||e||^2 - ||e_{\text{new}}||^2}{\Delta h^T \cdot (\lambda \cdot \Delta h + g)}$
    if $\rho > 0$ **then**
      $h \leftarrow h_{\text{new}}$
      $\lambda \leftarrow \lambda \cdot \max\left(\frac{1}{3}, 1 - (2\rho - 1)^3\right)$
      $\nu \leftarrow 2$
      break
    end
    else
      $\lambda \leftarrow \lambda \cdot \nu$
      $\nu \leftarrow 2 \cdot \nu$
    end
  end
  if $||e||^2 \leq \epsilon_3$ **then**
    break
  end
  iterations $\leftarrow$ iterations $+ 1$
end
return h_{new}

Images	Inliers	M	Iterations	Best Inliers
0 1	224	202	1086	208
1 2	318	286	5627	292
2 3	258	232	4012	251
3 4	128	115	4844	117

Table 1: Comparison of Inliers, Minimum Inliers to Accept(M), Iterations, and Best Inliers between Image Pairs



(a) Panoramic view after applying RANSAC and Own LM

Images	RANSAC Cost	Own LM Cost
0 1	2443	1128
1 2	1530	753
2 3	1776	871
3 4	846	390

Table 2: Comparison of RANSAC and LM Cost between given input Image Pairs

Images	Inliers	M	Iterations	Best Inliers
0 1	768	692	529	721
1 2	875	788	513	789
2 3	687	619	770	626
3 4	871	784	37	802

Table 3: Comparison of Inliers, Minimum Inliers to Accept(M), Iterations, and Best Inliers between Collected Pairs

(a) Panoramic view after applying RANSAC and Own LM

Images	RANSAC Cost	Own LM Cost
0 1	56765	27667
1 2	47086	23287
2 3	78634	37355
3 4	58491	29108

Table 4: Comparison of RANSAC and LM Cost between collected input Image Pairs

Programming

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.feature import corner_harris, corner_peaks
from scipy.optimize import least_squares
import random
import os
import math

#SIFT to get the keypoints and descriptors
def sift_feature_matching(img1, img2):
    # Convert images to grayscale
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Initialize the SIFT detector
    sift = cv2.SIFT_create()

    # Detect keypoints and compute descriptors for both images
    keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

    # Use BFMatcher with default params (L2 norm, as it's good for SIFT)
    bf = cv2.BFMatcher()

    # Match descriptors using KNN (k=2 for ratio test)
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)

    # Apply the ratio test to retain good matches (Lowe's ratio test)
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)

    # Draw the matches
    img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2,
       good_matches, None, flags=cv2.
       DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    #save the output image

    cv2.imwrite('sift_sample.jpg', img_matches)
```

```
    print("done!")

    return img_matches, keypoints1, keypoints2, good_matches


#Code to find solution using Linear Least-Squares Ah = b
#How to construct A? Construct A from the correspondences

def homography_eq(p1 , p2):
    A_eq = np.zeros((2,8))
    A_eq[0] = [p1[0], p1[1], 1, 0, 0, 0, -p1[0]*p2[0], -p1[1]*p2[0]]
    A_eq[1] = [0, 0, 0, p1[0], p1[1], 1, -p1[0]*p2[1], -p1[1]*p2[1]]
    return A_eq

#create a function to find the homography matrix H using the
   matched_keypoints where the matched points can be more than 4
def find_homography(matched_keypoints):
    A = []
    b = []
    for i in range(len(matched_keypoints)):
        A.append(homography_eq(matched_keypoints[i][0], matched_keypoints[
           i][1]))
        b.append(matched_keypoints[i][1][0])
        b.append(matched_keypoints[i][1][1])
    A = np.array(A)
    b = np.array(b)
    #Change A to 2n x 8
    A = A.reshape(-1, 8)
    #Change b to column vector
    b = b.reshape(-1, 1)
    #Solve for h using pseudo-inverse
    h = np.linalg.pinv(A) @ b
    H = np.reshape(np.append(h,1),(3,3))
    return H

#find_homography(matched_keypoints)

#Implement RANSAC to find the best homography matrix
#parameters are delta, epsilon, N(number of trails to conduct) and M(
   Minimum number of inliers to accept a model)

def ransac_homography(matched_keypoints, delta= 12, epsilon= 0.1 , n= 20):
    # Initialize the best homography matrix
    best_H = None
    best_inliers = 0
    best_inliers_list = None

    # Number of correspondences
    n_total = len(matched_keypoints)
    #print("n_total:", n_total)

    # Probability that atleast one of the max_iters will be free of
       outliers is p
    p = 0.99
```

```
91
92      # The minimum set of correspondences used to estimate the homography
            matrix be n
93
94      # The maximum number of trails to conduct
95      max_iters = int((np.log(1-p))/(np.log(1-(1-epsilon)**n)))
96      #max_iters = 30
97      #print("max_iters:", max_iters)
98      # The minimum number of inliers to accept a model
99      M = int((1-epsilon)*n_total)
100     print("Minimum inliers to accept:", M)
101
102     max_iters_all = (math.factorial(n_total)/(math.factorial(n)*math.
            factorial(n_total-n)))
103
104     # RANSAC
105     #for i in range(max_iters):
106     interations = 0
107     while interations < max_iters_all:
108         # Randomly select n correspondences
109         random_indices = random.sample(range(n_total), n)
110         #print("random_indices:", random_indices)
111         random_points = [matched_keypoints[i] for i in random_indices]
112         #print("random_points:", random_points)
113
114         # Compute the homography matrix
115         H = find_homography(random_points)
116
117         # Compute the inliers
118         inliers = 0
119         inliers_list = []
120         for j in range(n_total):
121             # Compute the error
122             p1 = np.append(matched_keypoints[j][0], 1)
123             p2 = np.append(matched_keypoints[j][1], 1)
124             error = np.linalg.norm(p2 - H @ p1)
125             #print("error:", error)
126             # Check if the error is less than the delta
127             if error < delta:
128                 inliers += 1
129                 inliers_list.append(j)
130
131         # *************** check if the number of inliers are greater than
                minimum set M ***************
132         if inliers > M:
133             if inliers > best_inliers:
134                 best_inliers = inliers
135                 best_inliers_list = inliers_list
136             print("interations:", interations)
137             break
138         else:
139             interations += 1
140             continue
141
```

```
142
143     print("best_inliers:", best_inliers)
144     #print("best_inliers_list:", best_inliers_list)
145
146     best_points = [matched_keypoints[i] for i in best_inliers_list]
147     best_H = find_homography(best_points)
148     #print("best_H:", best_H)
149
150     return best_H , best_inliers_list
151
152 # Define the cost function (updated)
153 def cost_function(h, points):
154     h = np.reshape(h, (3, 3))
155     p1 = np.array([point[0] + [1] for point in points])  # Homogeneous
        coordinates
156     p2 = np.array([point[1] for point in points])  # Actual 2D points
157
158     # Apply the homography transformation
159     f = (h @ p1.T).T
160     f = f[:, :2] / f[:, 2][:, np.newaxis]  # Normalize homogeneous
        coordinates to get (x, y)
161
162     # Compute error for both x and y coordinates (2N length)
163     error = (p2 - f).flatten()  # Flatten to make it a 1D array of length
        2N
164     return error
165
166 #Implement Jacobian from best_points and the homography matrix
167 def Jacobian(h, points):
168     h = np.reshape(h, (3, 3))
169     J = []
170     # Calculate the error without using a for loop
171     p1 = np.array([point[0] + [1] for point in points])
172     p2 = np.array([point[1] + [1] for point in points])
173     X = p2
174     f = (h @ p1.T).T
175     for i in range(len(points)):
176         x = p1[i][0]
177         y = p1[i][1]
178         J.append([x, y, 1, 0, 0, 0, -x*f[i][0], -y*f[i][0], -f[i][0]])
179         J.append([0, 0, 0, x, y, 1, -x*f[i][1], -y*f[i][1], -f[i][1]])
180     J = np.array(J)
181     return J
182
183 def Leven_Marqua(H, inliers):
184     epsilon1        =   1e-15
185     epsilon2        =   1e-15
186     epsilon3        =   1e-15
187     max_iterations  =   100
188     nu              =   2
189     tau             =   0.5
190     h               =   H.flatten()
191     iterations      =   0
192     while iterations < max_iterations:
```

```
193
194         # Calculate residuals and Jacobian
195         J          = Jacobian(H, inliers)
196         error      = cost_function(h, inliers)
197
198         # Calculate gradient and approximate Hessian
199         g          = J.T @ error
200         A          = J.T @ J
201
202         # Check for convergence
203         if np.linalg.norm(g, ord=np.inf) < epsilon1:
204             break
205
206         # Compute lambda and update step
207         I           = np.eye(9)
208         lambda_     = tau * np.max(np.diag(A))
209
210         while True:
211             # Solve the augmented normal equations
212             delta_h = np.linalg.solve(A + lambda_ * I, g)
213
214             if np.linalg.norm(delta_h) <= epsilon2 * (np.linalg.norm(h) +
                    epsilon2):
215                 break
216
217             h_new = h + delta_h
218             error_new = cost_function(h_new, inliers)
219
220             rho = (np.linalg.norm(error)**2 - np.linalg.norm(error_new)
                    **2) / (delta_h.T @ (lambda_ * delta_h + g))
221
222             if rho > 0:
223                 h = h_new
224                 lambda_ *= max(1/3, 1 - (2*rho - 1)**3)
225                 nu = 2
226                 break
227             else:
228                 lambda_ *= nu
229                 nu *= 2
230
231         if np.linalg.norm(error)**2 <= epsilon3:
232             break
233
234         iterations += 1
235
236     return h_new
237
238 # Write code to read cosequtive images of form 1.jpg, 2.jpg,...,n.jpg
239 # Extract the keypoints and descriptors for each pair of images using SIFT
      function sift_feature_matching above
240 # Use the matched_keypoints to find the homography matrix using
     ransac_homography function
241 # Refine the homography using levenberg-marquardt optimization
242 #project all the views onto common reference frame (middle image) using
```

```
    the homography matrix
243
244 # Read the images
245 images = []
246 for i in range(1, 6):
247     img = cv2.imread(f'pics/{i}.jpg')
248     images.append(img)
249
250 # Initialize the feature extractor
251 feature_extractor = "sift"
252 #feature_extractor = "SuperPoint_SuperGlue"
253
254 Homographies = []
255 ransac_error_list = []
256 error_list = []
257 Leven_Marqua_error_list = []
258 for i in range(len(images)-1):
259     print("images:", i , i+1)
260     img1 = images[i]
261     img2 = images[i+1]
262     if feature_extractor == "sift":
263         img_matches, keypoints1, keypoints2, good_matches =
              sift_feature_matching(img1, img2)
264
265         # Extract the good_matches for each pair of images
266         good_matches_list = []
267         # Initialize the list of points
268         matched_keypoints = []
269         for match in good_matches:
270             # Get the matching keypoints for each of the images
271             img1_idx = match.queryIdx
272             img2_idx = match.trainIdx
273
274             # x - columns
275             # y - rows
276             (x1, y1) = keypoints1[img1_idx].pt
277             (x2, y2) = keypoints2[img2_idx].pt
278
279             matched_keypoints.append([[x1, y1], [x2, y2]])
280     if feature_extractor == "SuperPoint_SuperGlue":
281         #Write code to read matched_keypoints from .npy file for
              SuperPoint+SuperGlue
282         matched_keypoints_np = np.load(f'pics/{i+1}_and_{i+2}
              _matched_points.npy', allow_pickle=True)
283         print(matched_keypoints_np.size)
284
285         #edit matched_keypoints_np to match the format of
              matched_keypoints
286         # Initialize the list of points
287         matched_keypoints = []
288         for match in matched_keypoints_np:
289             # Get the matching keypoints for each of the images
290             x1, y1 = match[0]
291             x2, y2 = match[1]
```

```
            matched_keypoints.append([[x1, y1], [x2, y2]])

    #Apply RANSAC to find the best homography matrix
    best_H , best_inliers_list  = ransac_homography(matched_keypoints,
       delta= 25, epsilon= 0.1 , n= 8)

    best_inliers = [matched_keypoints[i] for i in best_inliers_list]
    outliers     = [matched_keypoints[i] for i in range(len(
       matched_keypoints)) if i not in best_inliers_list]

    #Draw the lines connecting inliers as green  and outliers as red in
       the same image
    img_matches = np.concatenate((img1, img2), axis=1)
    for inlier in best_inliers:
        cv2.line(img_matches, (int(inlier[0][0]), int(inlier[0][1])), (int
           (inlier[1][0] + img1.shape[1]), int(inlier[1][1])), (0, 255, 0)
           , 1)
    for outlier in outliers:
        cv2.line(img_matches, (int(outlier[0][0]), int(outlier[0][1])), (
           int(outlier[1][0] + img1.shape[1]), int(outlier[1][1])), (0, 0,
            255), 1)
    cv2.imwrite(f'pics/{i}_and_{i+1}_ransac.jpg', img_matches)

    #RANSAC error
    error = cost_function(best_H.flatten() , best_inliers)
    #print("error: ", error)
    ransac_error = np.sum(error**2)
    #print("ransac_error: ",np.sum(error)/len(error))
    ransac_error_list.append(ransac_error)
    #Refine the homography using levenberg-marquardt optimization
    res = least_squares(cost_function, best_H.flatten(), method= 'lm',
       args=(best_inliers,))
    H = np.reshape(res.x, (3, 3))
    Homographies.append(H)
    #print("H:", H)
    #print("lm_error:", res.cost)
    error_list.append(res.cost)
    #Use defined LavMer
    Leven_Marqua_H = Leven_Marqua(best_H, best_inliers)
    Leven_Marqua_error = cost_function(Leven_Marqua_H, best_inliers)
    Leven_Marqua_error_list.append(np.sum(Leven_Marqua_error**2))

print("error_list:", error_list)
print("ransac_error_list:", ransac_error_list)
print("Leven_Marqua_error_list:", Leven_Marqua_error_list)




# Calculate the index of the middle image
middle_image_index = len(images)//2

# Initialize an identity matrix to accumulate transformations for images
```

```
    after the middle image
transform_to_middle = np.eye(3)

# Loop through homographies of images after the middle image
# The goal is to adjust these homographies to the middle image's reference
    frame
for i in range(middle_image_index, len(Homographies)):
    # Get the homography matrix
    H = Homographies[i]
    # Compute the transformation to the middle image's reference frame
    transform_to_middle = transform_to_middle @ np.linalg.inv(H)
    Homographies[i] = transform_to_middle

# Initialize an identity matrix to accumulate transformations for images
    before the middle image
transform_to_middle = np.eye(3)

# Loop through homographies of images before the middle image
# The goal is to adjust these homographies to the middle image's reference
    frame
for i in range(middle_image_index-1, -1, -1):
    # Get the homography matrix
    H = Homographies[i]
    # Compute the transformation to the middle image's reference frame
    transform_to_middle = transform_to_middle @ H
    Homographies[i] = transform_to_middle

# Insert an identity matrix at the index of the middle image, as it doesn'
    t need any transformation
# The middle image is the reference frame, so its homography is the
    identity matrix
Homographies = np.insert(Homographies, middle_image_index, np.eye(3), axis
    =0)
print("Homographies:", Homographies.shape)

# Now, all homographies are adjusted relative to the middle image,
    ensuring all images are aligned to a common reference frame
#define a function to find the max and min of the transformed image
def maxmin(h, w, H):
    # Define the four corners of the image
    corners = np.array([[0, 0, 1], [w, 0, 1], [0, h, 1], [w, h, 1]])

    # Apply the homography matrix to the corners
    transformed_corners = H @ corners.T

    # Normalize the transformed corners
    transformed_corners = transformed_corners / transformed_corners[-1]

    # Find the maximum and minimum x and y coordinates
    max_x = np.max(transformed_corners[0])
    min_x = np.min(transformed_corners[0])
    max_y = np.max(transformed_corners[1])
    min_y = np.min(transformed_corners[1])

```

```
    return max_x, min_x, max_y, min_y

#define a create_pano(panorama_image, image, homography_matrix) function to create the panorama image
def create_pano(curr_img, new_img, H):
    # Get dimensions of the current and new images
    h_curr, w_curr, _ = curr_img.shape
    h_new, w_new, _ = new_img.shape

    # Invert the homography matrix
    H_inv = np.linalg.inv(H)

    # Create grid of pixel coordinates (x, y) for the current image
    x_coords, y_coords = np.meshgrid(np.arange(w_curr), np.arange(h_curr))
    # Flatten the grids and create homogeneous coordinates (x, y, 1)
    homogeneous_coords = np.stack([x_coords.ravel(), y_coords.ravel(), np.ones_like(x_coords).ravel()], axis=1)

    # Apply the inverse homography transformation to the pixel coordinates
    transformed_coords = H_inv @ homogeneous_coords.T
    transformed_coords /= transformed_coords[2, :]  # Normalize by the last row

    # Extract the transformed x and y coordinates
    x_transformed = transformed_coords[0, :].astype(int)
    y_transformed = transformed_coords[1, :].astype(int)

    # Filter out points that are outside the bounds of the new image
    valid_idx = (x_transformed >= 0) & (x_transformed < w_new) & (y_transformed >= 0) & (y_transformed < h_new)

    # Get the valid pixel locations in both the current and new images
    x_curr_valid = homogeneous_coords[valid_idx, 0].astype(int)
    y_curr_valid = homogeneous_coords[valid_idx, 1].astype(int)
    x_new_valid = x_transformed[valid_idx]
    y_new_valid = y_transformed[valid_idx]

    # Map the valid pixels from new_img to curr_img
    curr_img[y_curr_valid, x_curr_valid] = new_img[y_new_valid, x_new_valid]

    return curr_img



# Initialize translation parameters to calculate total width and height offsets for images
total_translation_x = 0
total_translation_y = 0

for i in range(len(images)//2):
    image = images[i]
    total_translation_y += image.shape[0]  # Accumulate height for vertical alignment
```

```
    total_translation_x += image.shape[1]  # Accumulate width for horizontal alignment
print("total_translation_x:", total_translation_x)
print("total_translation_y:", total_translation_y)

# Initial transformation matrix set as an identity matrix with x-translation offset
initial_homography = np.eye(3)
#print("initial_homography:", initial_homography)
initial_homography[0, 2] = total_translation_x  # Apply translation in x direction
#print("initial_homography:", initial_homography)


# Calculate the final dimensions of the combined panorama image
final_height = 0
final_width = 0
for idx in range(len(images)):
    image = images[idx]
    h, w, _ = image.shape
    # Get the max and min of the transformed image
    max_x, min_x, max_y, min_y = maxmin(h, w, Homographies[idx])
    # Update the final height and width
    final_height = max(final_height, int(np.ceil(max_y) - np.floor(min_y)))
    final_width += w
print("final_height:", final_height)
print("final_width:", final_width)

panorama_image = np.zeros((int(final_height), int(final_width), 3), dtype=np.uint8)
#print("panorama_image:", panorama_image.shape)
# Apply transformations to each image and blend them onto the blank canvas
for idx in range(len(images)):
    image = images[idx]

    # Compute the final homography matrix by combining translation and homography for the image
    homography_matrix = initial_homography @ Homographies[idx]
    #print("homography_matrix:", homography_matrix)
    print("panorama_image:", panorama_image.shape)
    # `create_panorama()` is a custom function that handles warping and blending of images using the homography
    panorama_image = create_pano(panorama_image, image, homography_matrix)

# Convert the final panorama to RGB format (from BGR) for correct color representation
final_panorama_rgb = cv2.cvtColor(panorama_image, cv2.COLOR_BGR2RGB)

# Write the final stitched panorama image
cv2.imwrite('panorama.jpg', final_panorama_rgb)
```

Listing 1: Python Code