ECE 661 Homework 5

Michael Goldberg

October 4 2024

1 Theoretical Questions

1.1 RANSAC Inliers vs. Outliers

The way we differentiate between inliers and outliers when performing RANSAC relies on consensus across the entire dataset. The inputs to RANSAC are the original correspondencies given from an interest point matching algorithm - in my case, I used SIFT. SIFT provides pairs of points that reside in image1 and image2. Some of these correspondencies are outliers, but most are inliers, which is why RANSAC inlier consensus works. We start off by sampling n pairs of points where $4 \leq n$, in my case I chose 8 to strike a balance between computation time and robustness. We can generate a homography from these sampled pairs using linear least squares and performing SVD to solve for the homography matrix terms. We can then apply this homography to every pair of points returned from SIFT. Then, by calculating the reprojection error for each pair and determining if it lies within a predetermined δ (in my case this was set to 3), then this pair is added to a list of inliers for the calculated homography. We can then compare the size of this inlier set to the size of the previous best inlier set, and if it is larger, then we set this to be the new best inlier set. It can be seen that if the homography is generated from points that are outliers, then the size of the calculated inlier set will be smaller than a homography generated from the true inliers of the whole dataset. Performing a significant amount of iterations in this manner will allow us to converge to an inlier set that is indicative of the entire dataset's inlier set, thereby eliminating the dataset's outliers.

1.2 Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm combines the best of Gradient Descent (GD) and Gauss-Newton (GN) through the use of a damping term. The following is the equation used for LM to determine the step size from the current point:

$$\delta_p = (J_f^T J_f + \mu I)^{-1} J_f^T \epsilon(p_k) \tag{1}$$

If the damping coefficient is set high enough where $\mu > J_f^T J_f$, then this equation acts like GD. However, when $\mu < J_f^T J_f$, it acts like GN. Therefore, the

calculation of a single term μ allows the LM equation to switch between acting as GD and GN as needed to balance both computation time and numerical stability given a certain cost function.

2 Programming Tasks

2.1 Panorama with Engineering Fountain

The following task was to create a panoramic image from 5 separate images of the engineering fountain. Here are the 5 images that were used:

(a) Fountain Image 1

(b) Fountain Image 2

(c) Fountain Image 3

(d) Fountain Image 4

(e) Fountain Image 5

This process can be broken down into four main steps:

1. Find interest point pairs between consecutive images

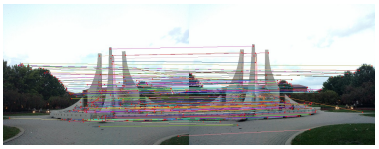
2. Reject correspondency outliers using RANSAC and find initial homography estimation between images

3. Refine homography estimation using nonlinear-least squares

4. Apply homographies between all pairs of consecutive images to form the final panorama

The first step is to find the interest point correspondencies using a matching algorithm. In my case I used SIFT due to its robustness and ease of use. Below are the correspondencies marked between Image 1 and 2, Image 2 and 3, Image 3 and 4, and Image 4 and 5.

(a) SIFT Image 1 to Image 2



(b) SIFT Image 2 to Image 3

(c) SIFT Image 3 to Image 4

(d) SIFT Image 4 to Image 5

It can be clearly seen that there are many outliers across all of the image pair correspondencies. This leads us into RANSAC to remove these outliers. I will first cover my implementation and then show the results.

RANSAC uses some parameters in order to determine sensitivity to outlier rejection as well as computation time. We first start by finding n, which as mentioned above follows the constraint $4 \leq n$. In my case, I chose 8. n signifies how many correspondencies we sample from the set of total matches from SIFT. We also find n_{total}, which is the total number of correspondencies that SIFT gave us. The user-specified parameters are ϵ, , and δ. ϵ is the probability that a randomly chosen correspondence is an outlier. While this is typically set to 0.1, I chose to set this to 0.35 to guarantee that all outliers are caught and removed. is the probability that at least 1 trial is free of outliers - this was set to 0.99. Finally, δ is the allowable tolerance for reprojection error - if the reporjection error is less than δ, then we consider a pair an inlier set. The total number of trials run, N, is calculated through $N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^n)}$. Finally, M represents the size of the inlier set that is acceptable - this allows us to terminate the program early if we have reached a desired number of inliers that we have determined to be sufficient. This is calculated by $M = (1-\epsilon)n_{total}$.

Once these parameters are established, we can understand the main logic behind RANSAC. We perform N trials, where a trial consists of randomly sampling n points from the original matches from SIFT. We then use linear-least

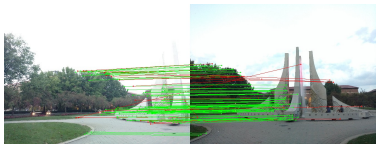
squares to calculate a homography between these n points and apply that homography to **all** point pairs from the original SIFT set. We can then calculate the reprojection error for each point pair and see if it is smaller than δ. If it is, then this point pair is added to the inlier set. Once all matches' reprojection errors are calculated and inliers are assigned, we can compare this inlier set to the previous best inlier set (initialized to an empty set). If the size of the computed inlier set is larger than the previous best inlier set, then the current set is assigned to be the best inlier set. If the size of the best inlier set ever exceeds M, then we end the trials early, as M is a sufficient size for the inliers that we have determined previously.

As part of this process, the homography between the n correspondences needs to be determined through linear-least squares. This is done using homogeneous equations and performing SVD to solve for the homography terms. If we assign $(x1, y1)_a$ and $(x2, y2)_a$ as a single pair of point correspondencies (where $a = 1, 2, ...n$, then the homogeneous equations are set up in this manner:

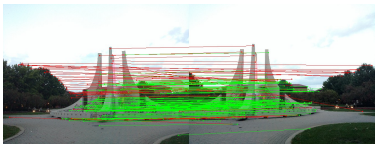
$$Ah = \begin{bmatrix} -x1_a & -y1_a & -1 & 0 & 0 & 0 & x2_a x1_a & x2_a y1_a & x2_a \\ 0 & 0 & 0 & -x1_a & -y1_a & -1 & y2_a x1_a & x2_a y1_a & x2_a \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = 0 \tag{2}$$

It can be seen that A is a matrix of size $2n \times 9$, and in our case that means 18×9. We can use SVD to solve for the h terms, as the h matrix is the singular vector corresponding to the smallest singular value of A.

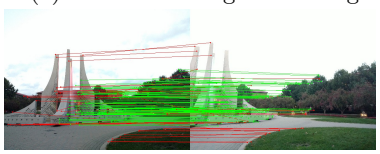
The results of RANSAC can be seen below, with inliers marked in green and outliers marked in red.



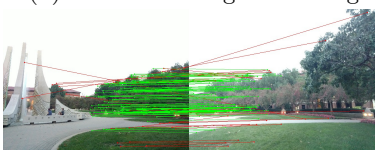
(a) RANSAC Image 1 to Image 2



(b) RANSAC Image 2 to Image 3



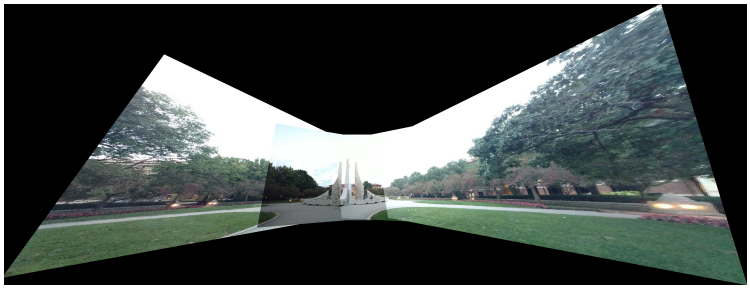
(c) RANSAC Image 3 to Image 4



(d) RANSAC Image 4 to Image 5

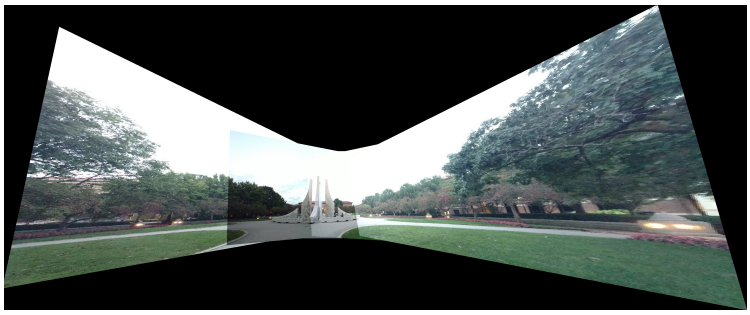
By collecting the best performing linear-least squares homography during

RANSAC, we have a great starting point for LM. Let's see what the panoramic looks like before we implement LM.



(a) Panorama Before LM

The LM implementation I used was the one built into $scipy.optimize.least_squares$ using the "lm" parameter. This refines the initial homography to better align the images. It can clearly be seen that the fountain matches up better after LM is used:



(a) Panorama After LM

2.2 Panorama of Test Images

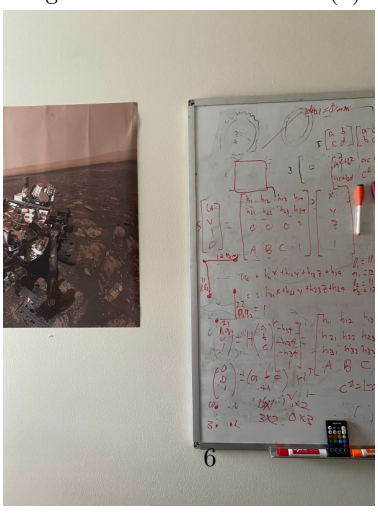
The above process was repeated for my own images, as seen in the following figure.

(a) Wall Image 1

(b) Wall Image 2

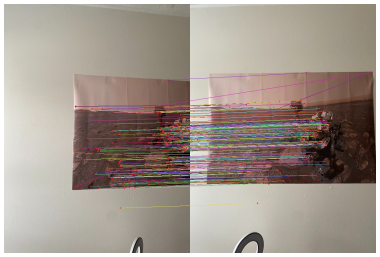
(c) Wall Image 3

(d) Wall Image 4

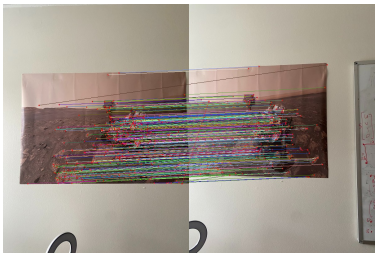


(e) Wall Image 5

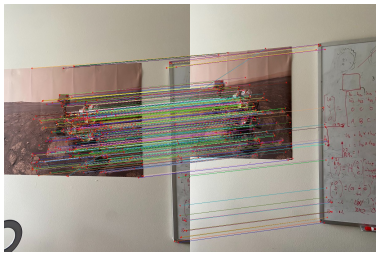
Here are the results of SIFT, RANSAC, and finally LM refinement for the final panorama.



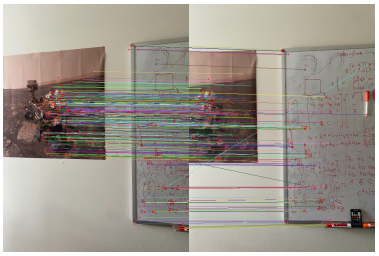
(a) SIFT Image 1 to Image 2



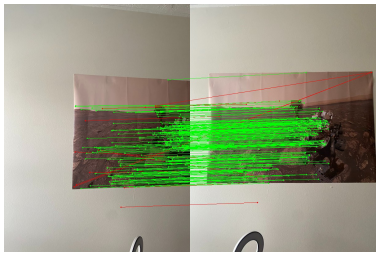
(b) SIFT Image 2 to Image 3



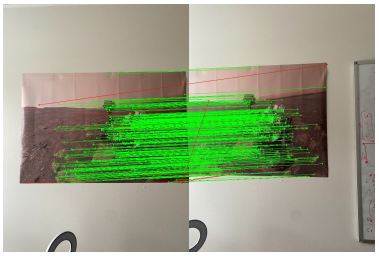
(c) SIFT Image 3 to Image 4



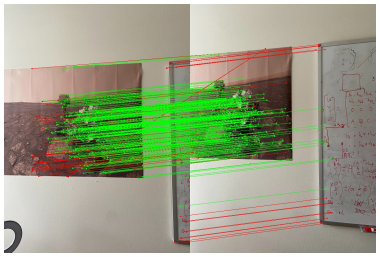
(d) SIFT Image 4 to Image 5



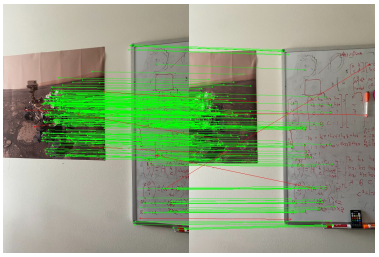
(a) RANSAC Image 1 to Image 2



(b) RANSAC Image 2 to Image 3



(c) RANSAC Image 3 to Image 4



(d) RANSAC Image 4 to Image 5

(a) Panorama Before LM

(a) Panorama After LM

3 Code

```
import cv2
```

```
import random
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

testname = "fountain"

# Loading images
img1 = cv2.imread('pics/' + testname + '/1.jpg')
img2 = cv2.imread('pics/' + testname + '/2.jpg')
img3 = cv2.imread('pics/' + testname + '/3.jpg')
img4 = cv2.imread('pics/' + testname + '/4.jpg')
img5 = cv2.imread('pics/' + testname + '/5.jpg')

save_images = True

# Helper function to convert point to homogeneous coordinates
def HC(point):
 return np.array([point[0], point[1], 1])

def estimate_homography(pairs):
 A = []
 for (x1, y1), (x2, y2) in pairs:
  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)

 _, _, V = np.linalg.svd(A) # Using SVD to solve for homography terms
 H = V[-1].reshape((3, 3))

 return H / H[2, 2]

def calculate_raster(Hleft, Hright, img_left, img_right):
 img_left_height, img_left_width, _ = img_left.shape
 img_right_height, img_right_width, _ = img_right.shape

 # Transforming the leftmost and rightmost images to determined final raster bounds
 corner_top_left = HC([0, 0])
 corner_top_right = HC([img_right_width, 0])
 corner_bottom_right = HC([img_right_width, img_right_height])
 corner_bottom_left = HC([0, img_left_height])

 corners_left_hc = np.array([corner_top_left, corner_bottom_left]).T
 corners_right_hc = np.array([corner_top_right, corner_bottom_right]).T
```

```
 transformed_left = np.matmul(Hleft, corners_left_hc)
 transformed_right = np.matmul(Hright, corners_right_hc)

 transformed_left = transformed_left / (transformed_left[-1, :] + 1e-6)
 transformed_right = transformed_right / (transformed_right[-1, :] + 1e-6)

 # Choosing min_x based on transformed leftmost image corners and max_x based on rightmost t
 min_x = int(min(transformed_left[0,:]))
 max_x = int(max(transformed_right[0,:]))
 min_y = int(min(min(transformed_left[1,:]), min(transformed_right[1,:])))
 max_y = int(max(max(transformed_left[1,:]), max(transformed_right[1,:])))

 w_out = max_x - min_x
 h_out = max_y - min_y

 H_trans = np.array([[1, 0, -min_x], [0, 1, -min_y], [0, 0, 1]])

 return w_out, h_out, H_trans

# Helper function to apply homography to a single point
def apply_homography_point(H, point):
 point_hc = HC(point)

 transformed_point = np.dot(H, point_hc)

 return (transformed_point[0] / transformed_point[2], transformed_point[1] / transformed_poi

def apply_homography_with_inverse(src_image, H, dest_image, Htrans):

 H = np.matmul(Htrans, H)

 H_inv = np.linalg.inv(H)

 H_inv /= H_inv[2, 2]

 domain_height, domain_width, _ = src_image.shape
 range_height, range_width, _ = dest_image.shape

 domain_image = np.zeros((domain_height, domain_width, 3), dtype=np.uint8)

 # Sampling range space to directly place domain image rgb values using the inverse homograp
 for v in range(range_height):
  for u in range(range_width):
   range_point = np.array([u, v, 1])
   domain_point = H_inv @ range_point
```

```
   domain_point /= domain_point[2]

   domain_u, domain_v = int(domain_point[0]), int(domain_point[1])

   if(0 <= domain_u < domain_width and 0 <= domain_v < domain_height):
    dest_image[v, u] = src_image[domain_v, domain_u]

 return dest_image

def sift(img1, img2, namestring):

 gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
 gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

 combined_image = np.concatenate((img1, img2), axis = 1)

 # Instantiating sift object, creating keypoints and descriptors
 sift = cv2.SIFT_create()
 keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
 keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

 # Using k-nearest neighbors for matching, and applying Lowe Ratio test to determine if matc
 matcher = cv2.BFMatcher()

 matches = matcher.knnMatch(descriptors1, descriptors2, k=2)

 good_matches = []

 # Lowe ratio test to determine good matches
 for m, n in matches:
  if m.distance < 0.75 * n.distance:
   pt1 = keypoints1[m.queryIdx].pt
   pt2 = keypoints2[m.trainIdx].pt
   good_matches.append((pt1, pt2))

 # Draw lines between matches
 for match in good_matches:
  (x1, y1) = match[0]
  (x2, y2) = match[1]

  x2 += img1.shape[1]

  cv2.circle(combined_image, (int(x1), int(y1)), 2, color=(0, 0, 255), thickness=-1)
  cv2.circle(combined_image, (int(x2), int(y2)), 2, color=(0, 0, 255), thickness=-1)
```

```
  cv2.line(combined_image, (int(x1), int(y1)), (int(x2), int(y2)), color=(random.randint(0,

 if(save_images):
  cv2.imwrite('pics/' + testname + '/sift_' + namestring + '.jpg', combined_image, [cv2.IMWR

 return good_matches


def ransac(matches, epsilon, p, delta, img1, img2, namestring):

 best_inliers = []
 best_H = None
 n = 8 # Minimum number of correspondences to find a homography
 n_total = len(matches)


 combined_image = np.concatenate((img1, img2), axis = 1) # generation of combined image for

 if len(matches) < 4: # Rejecting sift results with less than the minimum correspondencies f
  print("not enough correspondences to run RANSAC, please adjust SIFT parameters")
  return None, None

 N = math.ceil(math.log(1 - p) / math.log(1 - (1 - epsilon) ** n)) # Finding number of itera
 M = (1 - epsilon) * n_total # calculating acceptable number of outliers for faster computat

 # print("Number of iterations: ", N)

 for i in range(N):
  # Sample random set of n matches from sift and generate homography
  sampled_matches = random.sample(matches, n)

  pairs = [((n[0][0], n[0][1]), (n[1][0], n[1][1])) for n in sampled_matches]

  H = estimate_homography(pairs)

  inliers = []

  # Applying homography from random sample to all matches to determine inlier consensus
  for match in matches:
   (x1, y1) = match[0]
   (x2, y2) = match[1]

   projected_point = apply_homography_point(H, (x1, y1))

   distance = np.linalg.norm(np.array(projected_point) - np.array([x2, y2]))
```

```
   if distance < delta:
    inliers.append(match)

  # Test for if current estimation has better inlier consensus
  if len(inliers) > len(best_inliers):
   best_inliers = inliers
   best_H = H

  if(len(best_inliers)) > M: # if M is reached before max_iterations N, then break as this i
   break


 # Draw lines between matches
 for match in matches:
  (x1, y1) = match[0]
  (x2, y2) = match[1]

  x2 += img1.shape[1]

  if match in best_inliers:
   cv2.circle(combined_image, (int(x1), int(y1)), 2, color=(0, 255, 0), thickness=-1)
   cv2.circle(combined_image, (int(x2), int(y2)), 2, color=(0, 255, 0), thickness=-1)

   cv2.line(combined_image, (int(x1), int(y1)), (int(x2), int(y2)), color=(0, 255, 0), thick
  else:
   cv2.circle(combined_image, (int(x1), int(y1)), 2, color=(0, 0, 255), thickness=-1)
   cv2.circle(combined_image, (int(x2), int(y2)), 2, color=(0, 0, 255), thickness=-1)

   cv2.line(combined_image, (int(x1), int(y1)), (int(x2), int(y2)), color=(0, 0, 255), thick


 if(save_images):
  cv2.imwrite('pics/' + testname + '/ransac_' + namestring + '.jpg', combined_image, [cv2.IM

 return best_H, best_inliers

def nonlinear_least_squares(H_initial, inliers):

 # functions used for LM in scipy
 def homography_transform(H, points):
  H = H.reshape(3, 3)
  points_h = np.hstack([points, np.ones((points.shape[0], 1))]) # Convert to homogeneous coo
  points_transformed_h = np.dot(H, points_h.T).T # Apply homography
  points_transformed_h /= points_transformed_h[:, 2].reshape(-1, 1) # Normalize by the last
  return points_transformed_h[:, :2]
```

```
 def reprojection_error(H, points1, points2):
  points1_proj = homography_transform(H, points1) # Transform points from image 1 using H
  errors = points1_proj - points2  # Calculate the reprojection error
  return errors.ravel()


 points1 = np.array([inlier[0] for inlier in inliers]) # Points from image 1
 points2 = np.array([inlier[1] for inlier in inliers]) # Corresponding points from image 2


 H_initial_flat = H_initial.flatten()

 # Use scipy.optimize.least_squares with the LM argument
 result = least_squares(reprojection_error, H_initial_flat, args=(points1, points2), method=

 # Reshape the optimized homography back into a 3x3 matrix
 H_refined = result.x.reshape(3, 3)

 return H_refined / H_refined[2, 2] # Divide by last term


def create_panorama(img1, img2, img3, img4, img5, H12, H23, H34, H45):

 # Calculating homographies that convert images to middle image (in this case img3)
 H13 = H23 @ H12
 H43 = np.linalg.inv(H34)
 H54 = np.linalg.inv(H45)
 H53 = H43 @ H54

 # Calculate size of output raster for panorama
 panorama_width, panorama_height, Htrans = calculate_raster(H13, H53, img1, img5)

 panorama_img = np.zeros((panorama_height, panorama_width, 3), dtype=np.uint8)

 # Directly apply img3 to panorama img with just translation homography to serve as starting
 for v in range(img3.shape[0]):
  for u in range(img3.shape[1]):
   point_hc = np.array([u, v, 1])

   transformed_point_hc = Htrans @ point_hc
   transformed_point_hc = transformed_point_hc / transformed_point_hc[2]

   panorama_u, panorama_v = int(transformed_point_hc[0]), int(transformed_point_hc[1])

   if(0 <= panorama_v < panorama_height and 0 <= panorama_u < panorama_width):
    panorama_img[panorama_v, panorama_u] = img3[v, u]
```

```
 img3 = panorama_img

 # Apply homography from each image to image 3 using their corresponding homographies, mutat
 img3 = apply_homography_with_inverse(img1, H13, img3, Htrans)
 img3 = apply_homography_with_inverse(img2, H23, img3, Htrans)
 img3 = apply_homography_with_inverse(img4, H43, img3, Htrans)
 img3 = apply_homography_with_inverse(img5, H53, img3, Htrans)

 if(save_images):
  cv2.imwrite('pics/' + testname + '/panorama_before_LM.jpg', img3, [cv2.IMWRITE_JPEG_QUALIT
 else:
  img3 = cv2.cvtColor(img3, cv2.COLOR_BGR2RGB)
  plt.imshow(img3)
  plt.show()

# Step 1, extracting interest points pairwise between adjacent images
matches12 = sift(img1=img1, img2=img2, namestring="12")
matches23 = sift(img1=img2, img2=img3, namestring="23")
matches34 = sift(img1=img3, img2=img4, namestring="34")
matches45 = sift(img1=img4, img2=img5, namestring="45")

# Step 2, performing outlier rejection with RANSAC
H12, inliers12 = ransac(matches=matches12, epsilon=0.35, p=0.99, delta=3, img1=img1, img2=im
H23, inliers23 = ransac(matches=matches23, epsilon=0.35, p=0.99, delta=3, img1=img2, img2=im
H34, inliers34 = ransac(matches=matches34, epsilon=0.35, p=0.99, delta=3, img1=img3, img2=im
H45, inliers45 = ransac(matches=matches45, epsilon=0.35, p=0.99, delta=3, img1=img4, img2=im

# Step 3, Nonlinear Least-Squares Homography Refinement
H12 = nonlinear_least_squares(H12, inliers12)
H23 = nonlinear_least_squares(H23, inliers23)
H34 = nonlinear_least_squares(H34, inliers34)
H45 = nonlinear_least_squares(H45, inliers45)

# Step 4, applying homographies to common reference frame
create_panorama(img1=img1, img2=img2, img3=img3, img4=img4, img5=img5, H12=H12, H23=H23, H34
```