ECE661: Homework 5
Fall 2018

Gopikrishnan Sasi Kumar
PUID 0030843999
gsasikum@purdue.edu

October 10, 2018

1 Introduction

The goal of the homework is to carry out image mosaicing to create a panorama using 5 overlapping images of a scene. Automatic computation of homographies between pairs of images and their subsequent refinement by non-linear least squares minimization using Levenberg-Marquardt algorithm needs to be carried out for the work. The major steps involved in the work are as follows.

1. Extract interest points from the sequence of overlapping photos of the scene. The SIFT algorithm is used for this purpose.
2. Determine the correspondences between the interest points thus extracted for each of the pairs of successive images of the scene in the sequence.
3. For each of the pairs of images, use RANSAC algorithm for outlier rejection in the correspondences and create an initial estimate of the homography between the pair of images using linear least squares minimization.
4. Refine the homography thus obtained for each of the image pairs using nonlinear least squares minimization using Levenberg-Marquardt algorithm. This algorithm is implemented by me, and no open-source implementations are used for this step. The scipy.optimize.root function is used just for comparison with the homography generated by my implementation by displaying the homographies determined by both of them.
5. Stitch together the sequence of overlapping photos of the scene using the homographies computed to create a single panoramic photo.

These tasks are carried out for 5 photos of a scene captured by me by standing in a fixed location and turning through an angle, ensuring that there is sufficient overlap between successive photos. These images shall be named Image 1 to 5. Details regarding each of the steps are discussed in the following sections.

2 Finding SIFT correspondences

For each of the images in the sequence, 2000 interest points are extracted using the SIFT algorithm (using opencv library in python). The correspondences between the interest points in each of the pairs of images is carried out using the euclidean distance between the feature vectors extracted from the images. Using just the euclidean distance can lead to erroneous correspondences in images with repetitive patterns, as features around repetitive patterns will show match with more than one feature in the other image. Such features need to be eliminated. Hence the ratio of the euclidean distance of a feature with its closest and second closest match in the other image (γ) is used as a metric to threshold. All matched feature

pairs between the first and the second images with this ratio lesser than or equal to 0.6 are considered as valid correspondences between the pair of images.

3 RANSAC

RANSAC stands for "Random Sample Consensus". This algorithm is used to reject false pairings (if any) in the correspondences obtained as in the earlier section. The steps in the implementation of RANSAC are as follows.

- The decision threshold δ to construct the inlier set is set as 3 pixels.
- The number of trials to conduct N is determined as $N = \frac{\ln(1-p)}{\ln[1-(1-\epsilon)^n]}$, where $p = 0.99$ is the probability that at least one of the N trials will be free of outliers in the calculation of homography, $\epsilon = 0.2$ is the probability that a randomly chosen correspondence pair is an outlier (i.e., we assume that 20% of correspondence pairs are false), and $n = 6$ is the number of correspondences used in each trial to compute the homography.
- The minimum value for the size of the inlier set for it to be acceptable is determined as $M = (1-\epsilon)n_{total}$, where n_{total} is the total number of correspondences.
- N trials are carried out by randomly sampling n pairs of correspondences in each trial, computing the homography (H) using the n correspondences (using linear least-squares minimization as in Section 4), and finding the inliers in the the whole set of correspondences for the computed homography H.
- For determining which of the correspondences are inliers, the following logic is carried out. Suppose the correspondences are of the form (X, X').
  1. HX is computed for all the points X.
  2. The difference between HX and X' is computed as $(\Delta x, \Delta y)$.
  3. Squared distance between the measured X' and expected HX is computed as $d^2 = (\Delta x)^2 + (\Delta y)^2$.
  4. All the correspondences with $d^2 \leq \delta^2$ are determined as the inliers.
- The largest such inlier set obtained in the N trials is used to compute the final homography (H_{RANSAC}) by carrying out linear least squares minimization as in Section 4.
- The number of inliers in the largest such set is compared with M and a warning message is displayed if M is larger than the length of the obtained inlier set.
- The largest inlier set thus obtained is passed on to the step for nonlinear least squares minimization using Levenberg-Marquardt algorithm.

4 Estimating Homography with Linear Least Squares Minimization

This section talks about estimating homography using a set of correspondences. This method is used in the RANSAC algorithm discussed in the previous section to carry out linear least squares minimization to determine H_{RANSAC}.

Based on the exercise carried out in earlier homeworks (Home Work 2 and 3), for a homography H that transforms a point X as $X' = HX$, the following observations hold.

$$X' = HX \tag{1}$$

Considering the fact that H is a homogeneous representation, and that we can safely take the $(3,3)^{th}$ element of H as 1 for our application,

$$H = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \tag{2}$$

X and X' are the points (x, y) and (x', y') in the domain and range of H respectively, in the homogeneous coordinate representation, given by:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{3}$$

$$X' = \begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} \tag{4}$$

The coordinates of the transformed point are given by:

$$x' = a_{11}x + a_{12}y + a_{13} - a_{31}xx' - a_{32}yx' \tag{5}$$

$$y' = a_{21}x + a_{22}y + a_{23} - a_{31}xy' - a_{32}yy' \tag{6}$$

If the correspondences (x_i, y_i) and (x'_i, y'_i) for $i = 1, 2, 3, ...n$ are known, we have $2n$ equations and 8 unknowns as follows:

$$\underbrace{\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_n \\ y'_n \end{pmatrix}}_{\text{Xdash}} = \underbrace{\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_nx'_n & -y_nx'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -x_ny'_n & -y_ny'_n \end{pmatrix}}_{\text{A}} \underbrace{\begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{pmatrix}}_{\widehat{h}} \tag{7}$$

For a least squares approximation of $\widehat{h}$ for $n \geq 4$ points of correspondence between two images, the following equation holds.

$$\widehat{h} = (A^T A)^{-1} A^T Xdash \tag{8}$$

where A is of order $2n \times 8$, (n being the number of points of correspondence). Now, once we find out $\widehat{h}$, the homography H is obtained by adding 1 as the last element of $\widehat{h}$, and reshaping it into a 3×3 matrix.

5 Refining Homographies with Levenberg-Marquardt Algorithm

Before going into the details of how Levenberg-Marquardt algorithm (hereafter called LM algorithm for brevity) is used on the output of RANSAC, the LM algorithm and its implementation shall be discussed.

5.1 LM algorithm

The general nonlinear optimization problem can be stated as follows.

$$\min_{\vec{p}} ||\vec{E}||^2 \tag{9}$$

where

$$\vec{E}(\vec{p}) = \begin{pmatrix} e_1(\vec{p}) \\ e_2(\vec{p}) \\ \vdots \end{pmatrix} \tag{10}$$

The LM algorithm implemented for the general nonlinear optimization problem is as follows.

1. Start with an initial guess $\vec{p_0}$ for $\vec{p_k}$
2. Determine the Jacobian of $\vec{E}$, $J_{\vec{E}}$ at $\vec{p_0}$. Initialize μ_k to $T \times \max(diag(J_{\vec{E}}^T J_{\vec{E}}))$, where $0 < T \leq 1$ (chosen to be 0.5)
3. Determine $\vec{E}$ and $J_{\vec{E}}$ at $\vec{p_k}$, where $J_{\vec{E}}$ is the Jacobian of $\vec{E}$ given by

$$J_{\vec{E}} = \begin{pmatrix} \frac{\partial e_1}{\partial p_1} & \frac{\partial e_1}{\partial p_2} & \cdots & \frac{\partial e_1}{\partial p_n} \\ \frac{\partial e_2}{\partial p_1} & \frac{\partial e_2}{\partial p_2} & \cdots & \frac{\partial e_2}{\partial p_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_m}{\partial p_1} & \frac{\partial e_m}{\partial p_2} & \cdots & \frac{\partial e_m}{\partial p_n} \end{pmatrix} \tag{11}$$

4. Compute the step increment in $\vec{p_k}$ given by

$$\vec{\delta_p} = -(J_{\vec{E}}^T J_{\vec{E}} + \mu_k I)^{-1} J_{\vec{E}}^T \vec{E}(\vec{p_k}) \tag{12}$$

5. Compute $\vec{p_{k+1}}$ as

$$\vec{p_{k+1}} = \vec{p_k} + \vec{\delta_p} \tag{13}$$

6. Compute ρ as

$$\rho = \frac{C(\vec{p}) - C(\vec{p_{k+1}})}{-\vec{\delta_p}^T J_{\vec{E}}^T \vec{E}(\vec{p_k}) + \vec{\delta_p}^T \mu_k \vec{\delta_p}} \tag{14}$$

   where

$$C(\vec{p}) = [\vec{E}(\vec{p_k})]^T \vec{E}(\vec{p_k}) \tag{15}$$

7. Compute μ_{k+1} for the next step of iteration

$$\mu_{k+1} = \mu_k \times \max(\frac{1}{3}, 1 - (2\rho - 1)^3) \tag{16}$$

8. Make μ_k equal to μ_{k+1}
9. If $C(\vec{p}) - C(\vec{p_{k+1}}) \geq 0$, make $\vec{p_k}$ equal to $\vec{p_{k+1}}$
10. Repeat from step 3 until $||\vec{\delta_p}|| < \xi$, where ξ is the threshold for stopping iteration.

The algorithm for the generalized problem statement of nonlinear optimization was implemented by me and this implementation was used for this work. The code is provided in the function LM_algo in hw5_library.py.

5.2 Homography Refinement

The homography computed by RANSAC algorithm (H_{RANSAC}) for each of the image pairs needs to be refined further based on nonlinear least-squares minimization. This is carried out using the LM algorithm. The inlier set determined in RANSAC is used in this step.
Suppose the number of inliers is N (not to be confused with the N in RANSAC). Hence we have N point-to-point correspondences in the inlier set. If the correspondences are from the set of points (x_i, y_i) to (x'_i, y'_i) for $i = 1, 2, \ldots N$, define the 2N-vectors given by:

$$\vec{X} = \begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_N \\ y'_N \end{pmatrix} \tag{17}$$

$$\vec{f} = \begin{pmatrix} f_1^1 \\ f_2^1 \\ f_1^2 \\ f_2^2 \\ \vdots \\ f_1^N \\ f_2^N \end{pmatrix} \tag{18}$$

where (f_1^i, f_2^i) is the mapping of the point (x_i, y_i) using the homography $H_{optimized}$, and

$$f_1^i = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{19}$$

$$f_2^i = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{20}$$

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

This optimization problem can be stated as

$$\min_{\vec{h}} ||\vec{X} - \vec{f}(\vec{h})||^2 \tag{22}$$

where

$$\vec{h} = \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} \tag{23}$$

This is a nonlinear optimization problem, and is solved using the LM algorithm with flattened H_{RANSAC} (numpy.ravel(H_{RANSAC})) as the initial guess for $\vec{h}$. LM optimization is carried out on:

$$\vec{E} = \vec{X} - \vec{f}(\vec{h}) \tag{24}$$

Also, the Jacobian of $\vec{E}$ is given by:

$$J_{\vec{E}} = -J_{\vec{f}} \tag{25}$$

The partial derivatives of the first 2 elements of $\vec{f}$ are given as follows. These values are used for computing the Jacobian used for LM optimization.

$$\frac{\partial f_1^i}{\partial h_{11}} = \frac{x_i}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{26}$$

$$\frac{\partial f_1^i}{\partial h_{12}} = \frac{y_i}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{27}$$

$$\frac{\partial f_1^i}{\partial h_{13}} = \frac{1}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{28}$$

$$\frac{\partial f_1^i}{\partial h_{21}} = 0 \tag{29}$$

$$\frac{\partial f_1^i}{\partial h_{22}} = 0 \tag{30}$$

$$\frac{\partial f_1^i}{\partial h_{23}} = 0 \tag{31}$$

$$\frac{\partial f_1^i}{\partial h_{31}} = \frac{-x_i(h_{11}x_i + h_{12}y_i + h_{13})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{32}$$

$$\frac{\partial f_1^i}{\partial h_{32}} = \frac{-y_i(h_{11}x_i + h_{12}y_i + h_{13})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{33}$$

$$\frac{\partial f_1^i}{\partial h_{33}} = \frac{-(h_{11}x_i + h_{12}y_i + h_{13})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{34}$$

$$\tag{35}$$

$$\frac{\partial f_2^i}{\partial h_{11}} = 0 \tag{36}$$

$$\frac{\partial f_2^i}{\partial h_{12}} = 0 \tag{37}$$

$$\frac{\partial f_2^i}{\partial h_{13}} = 0 \tag{38}$$

$$\frac{\partial f_2^i}{\partial h_{21}} = \frac{x_i}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{39}$$

$$\frac{\partial f_2^i}{\partial h_{22}} = \frac{y_i}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{40}$$

$$\frac{\partial f_2^i}{\partial h_{23}} = \frac{1}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{41}$$

$$\frac{\partial f_2^i}{\partial h_{31}} = \frac{-x_i(h_{21}x_i + h_{22}y_i + h_{23})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{42}$$

$$\frac{\partial f_2^i}{\partial h_{32}} = \frac{-y_i(h_{21}x_i + h_{22}y_i + h_{23})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{43}$$

$$\frac{\partial f_2^i}{\partial h_{33}} = \frac{-(h_{21}x_i + h_{22}y_i + h_{23})}{(h_{31}x_i + h_{32}y_i + h_{33})^2} \tag{44}$$

The refined homographies between the pairs of images are used for generating the panorama.

6 Image Mosaicing

The last step in the panorama generation is image-mosaicing in which the different images are all projected onto a common reference plane, and homographies are applied on them to get the panorama. Even though the program is coded such that it will work for any number of images, we will limit the discussion

to the case where the number of images is 5. This pretty much generalizes the modus-operandi for any number of images. The steps involved are sumamrized below.

- We project all the images onto the plane of the middle image so as to get a symmetric clipping.
- If H_{ij} is the homography from image i to image j, refinement of the homographies using LM algorithm provides us with H_{12}, H_{23}, H_{34} and H_{45}.
- The homographies from all the images to the middle image (Image 3) are obtained as follows.

$$H_{13} = H_{23} \times H_{12} \tag{45}$$
$$H_{43} = H_{34}^{-1} \tag{46}$$
$$H_{53} = H_{34}^{-1} \times H_{45}^{-1} \tag{47}$$

- Now, all the homographies H_{13}, H_{23}, H_{43} and H_{53} need to be left-multiplied by a translation homography so as to translate the third image (middle image) to the centre of the canvas. Without this translation, the third image will start at the origin of the canvas and the first 2 images will be on the negative x region. A translation along x of twice the width of the individual images is applied to all the afore-mentioned homographies, and all the 5 images are projected onto a canvas of width 5 times the width of the individual images, and height same as them.
- In order to project the image onto the canvas, the same function that was used in earlier homeworks is reused.

7 Results

The 5 images used for the exercise are shown in Figure 1. The extracted correspondences between adjacent image-pairs are shown in Figures 2, 5, 8 and 11. The inliers and outliers for each of the image pairs are also shown in the figures that follow. The panorama generated without refinement of H_{RANSAC} using LM algorithm is shown in Figure 14). The final panorama with refinement of H_{RANSAC} using LM algorithm is shown in Figure 15. It is to be noted that even without refinement, the panorama obtained is extremely good. This shows that the correspondence matching logic using the ratios discussed in section 2 along with RANSAC algorithm are robust to begin with. However, marginal improvement is observed after refinement of the homographies using LM algorithm.
Even though my own implementation of LM algorithm is used, the scipy.optimize.root function is also used to compare with the homography obtained in the former by displaying both the results. They were observed to be equal.
The source code for the exercise is given in section 8.

Parameter	Description	Value
γ	Ratio threshold for SIFT matching	0.6
δ	Decision threshold to generate inlier set	3
p	Probability value used in RANSAC	0.99
ϵ	Percentage of outliers assumed in RANSAC	0.2
n	Number of correspondences used to compute homography	6
T	Parameter used for initializing μ_k in LM	0.5
ξ	Threshold for $\|\|\vec{\delta_p}\|\|$ below which LM iteration is stopped	10^{-23}

Table 1: Parameters used in the exercise

Figure 1: Images 1 to 5

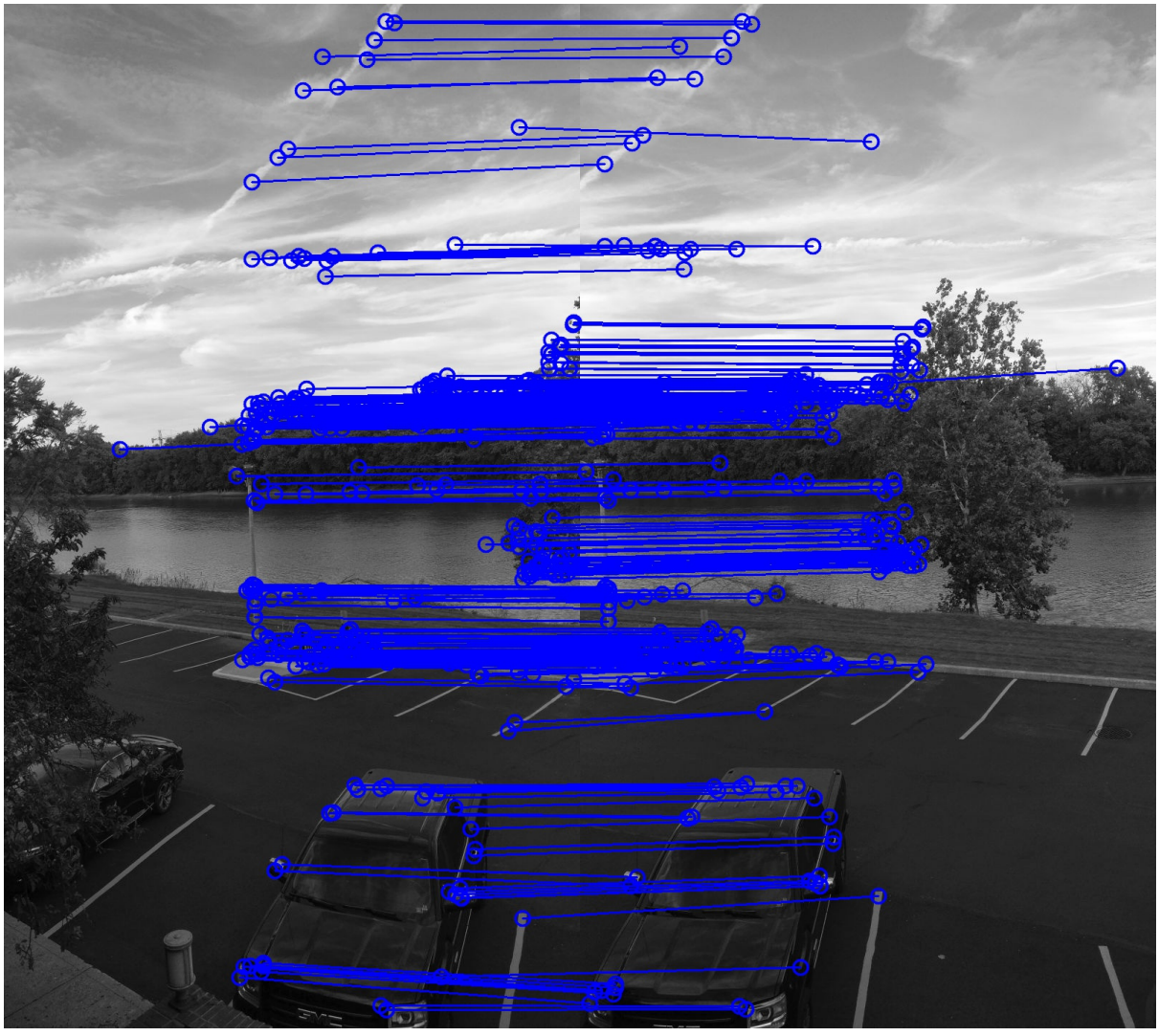

Figure 2: Extracted correspondences between images 1 and 2

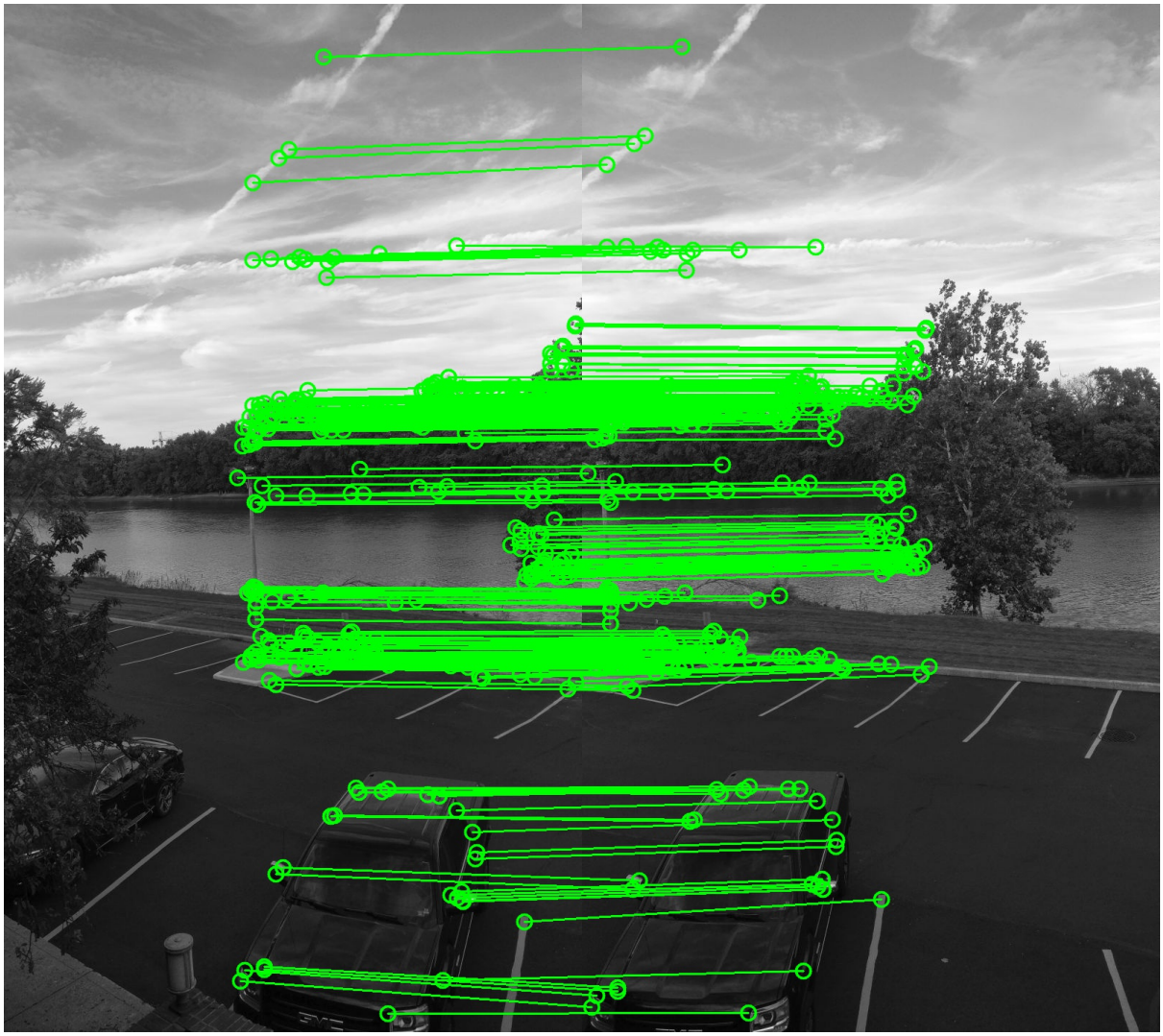

Figure 3: Inlier correspondences between images 1 and 2

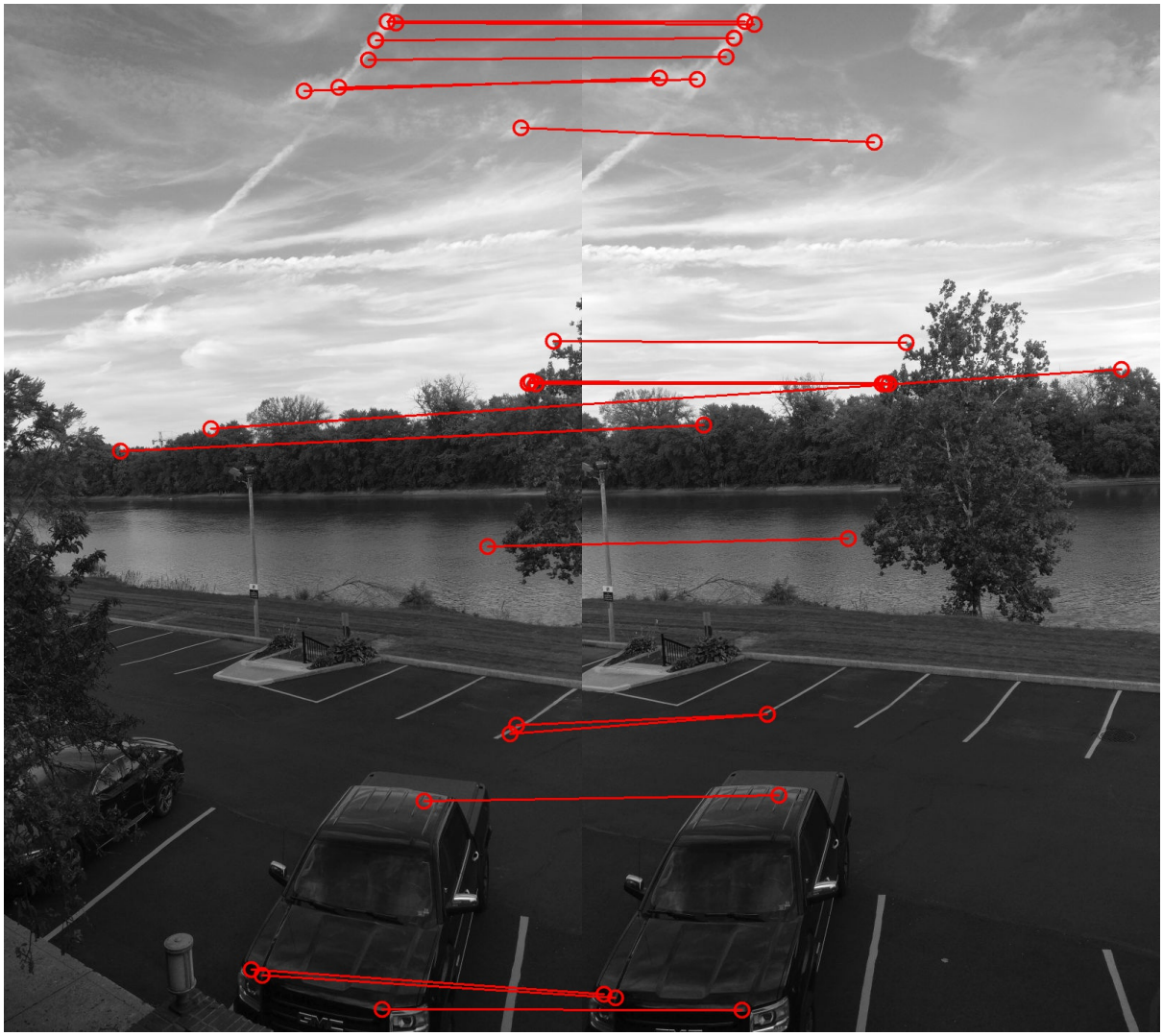

Figure 4: Outlier correspondences between images 1 and 2

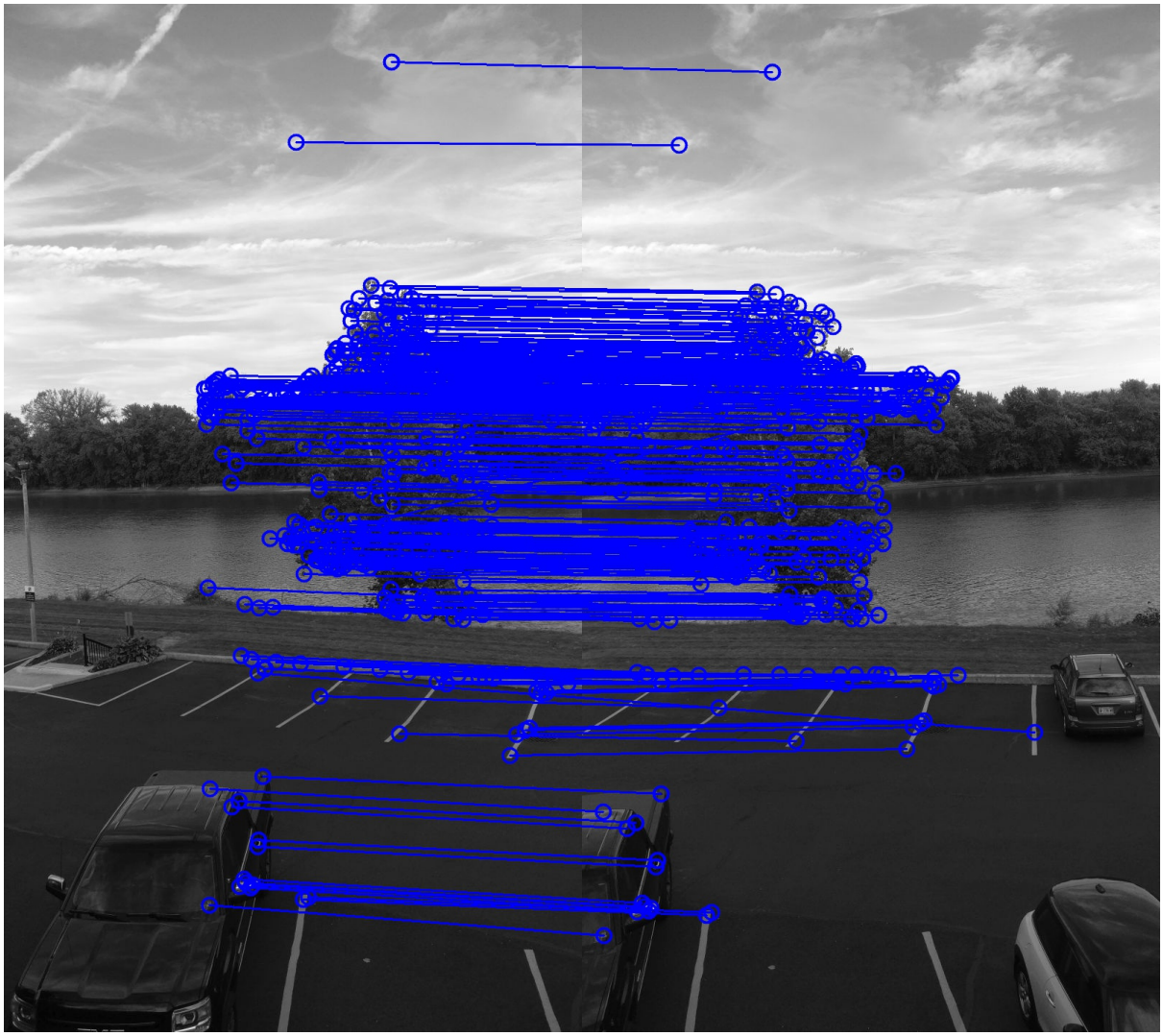

Figure 5: Extracted correspondences between images 2 and 3

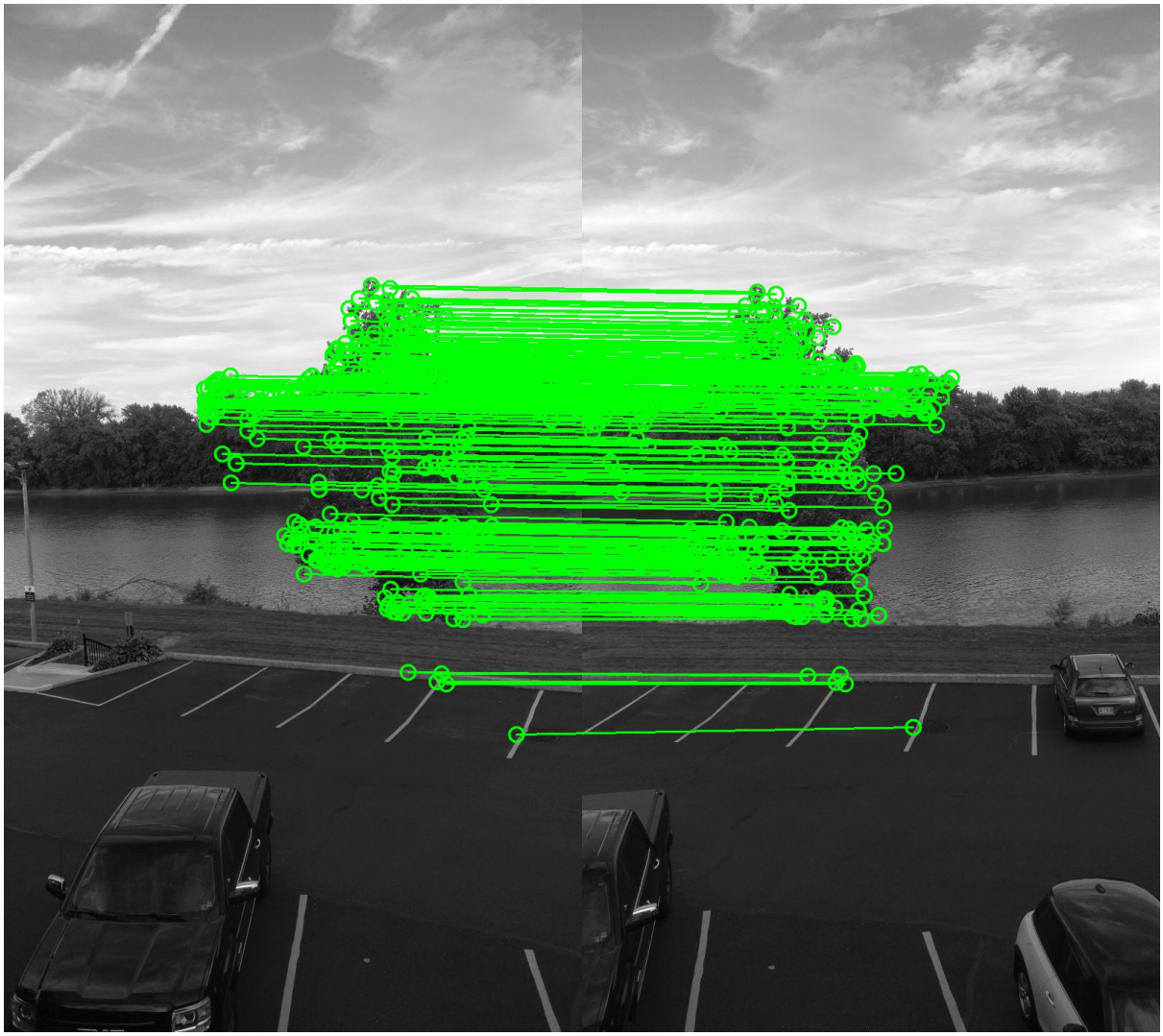

Figure 6: Inlier correspondences between images 2 and 3

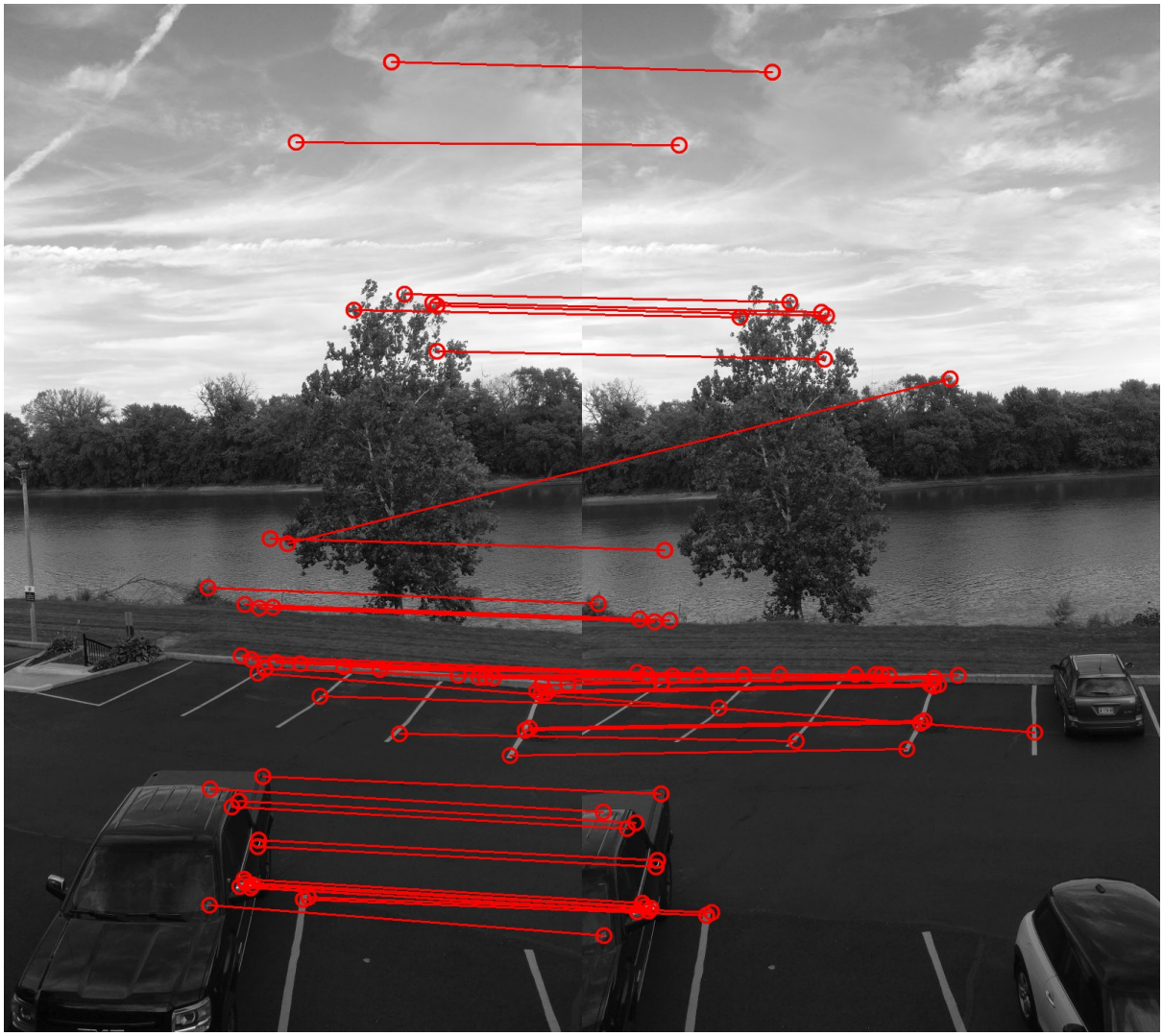

Figure 7: Outlier correspondences between images 2 and 3

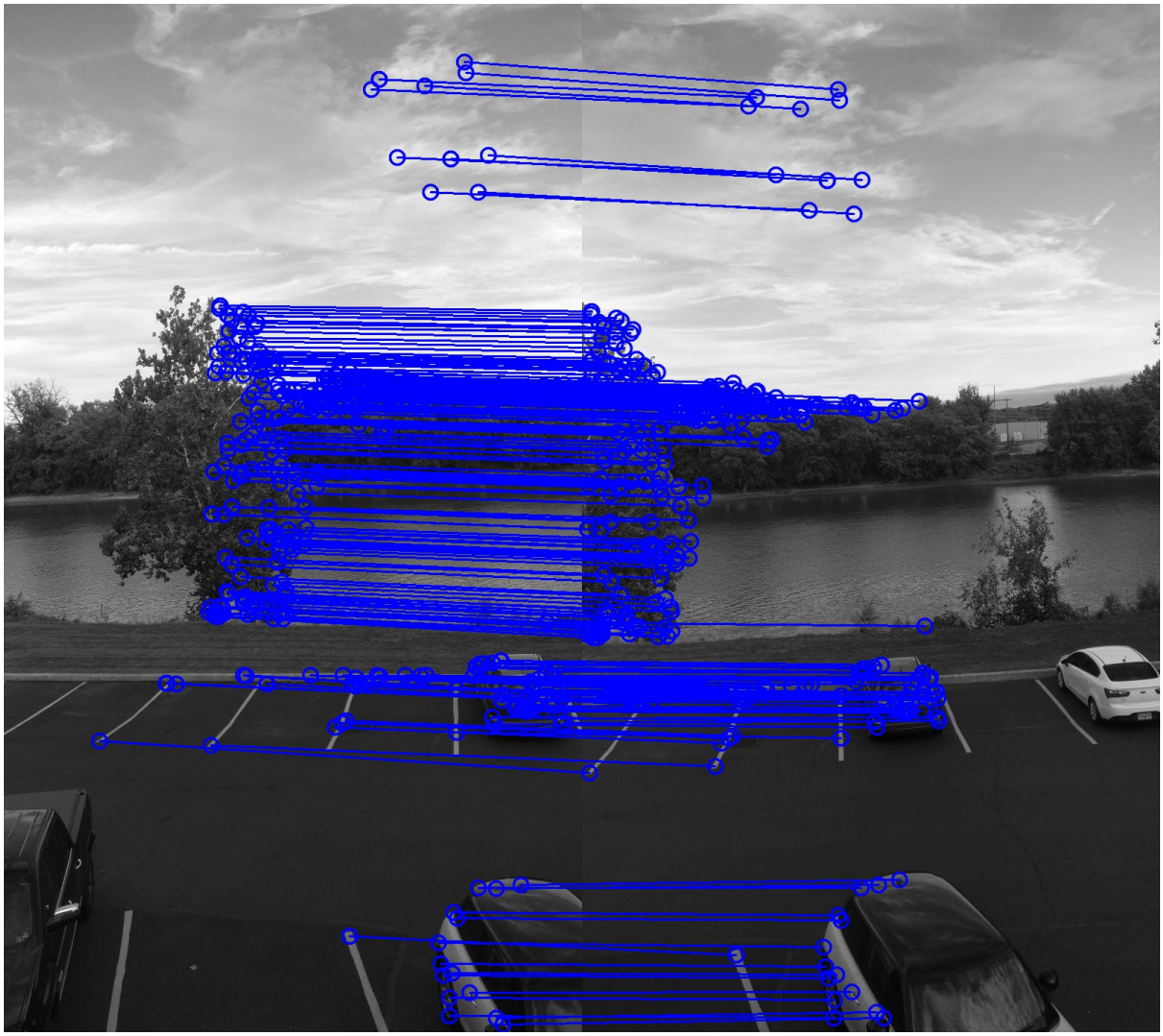

Figure 8: Extracted correspondences between images 3 and 4

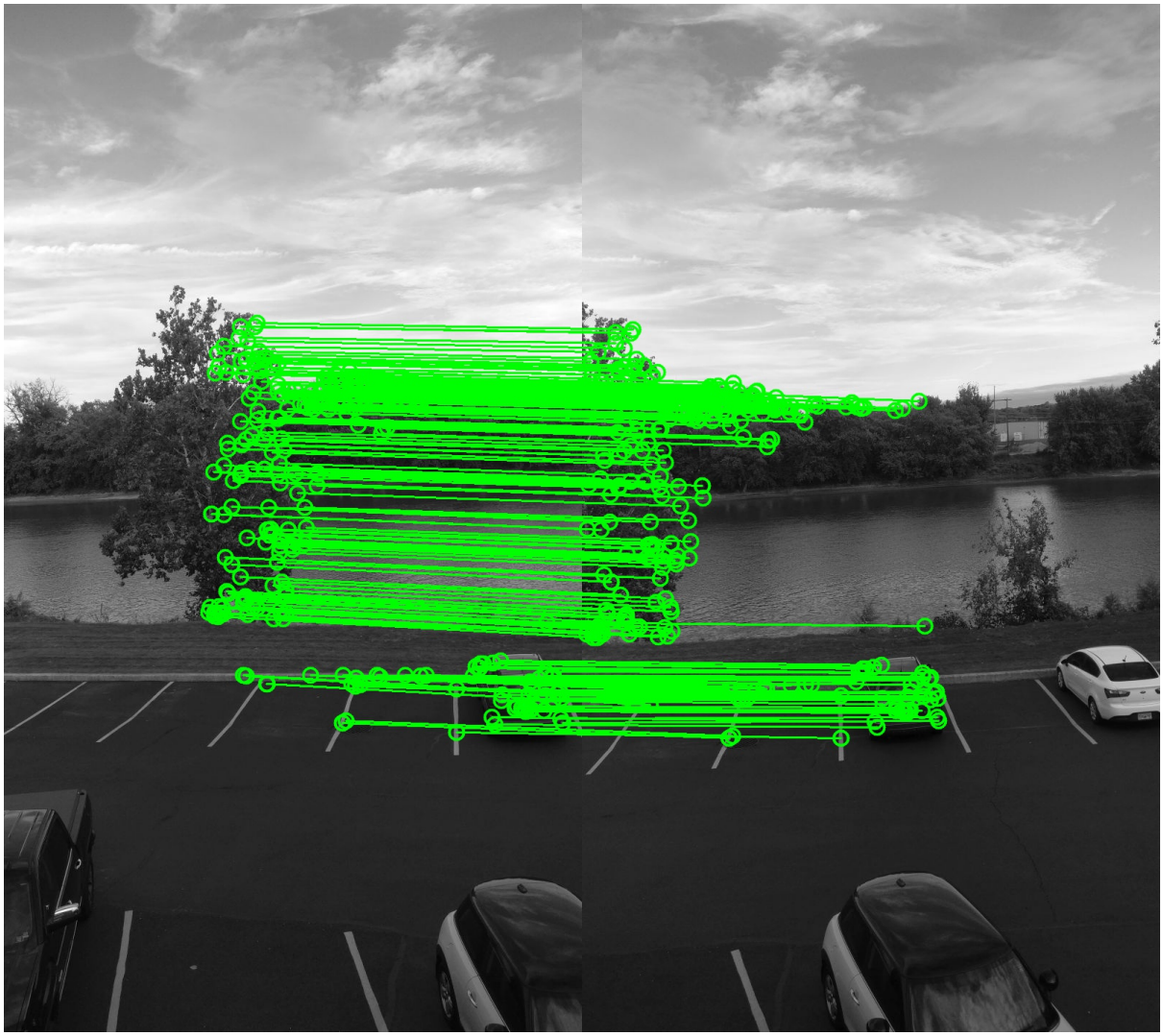

Figure 9: Inlier correspondences between images 3 and 4

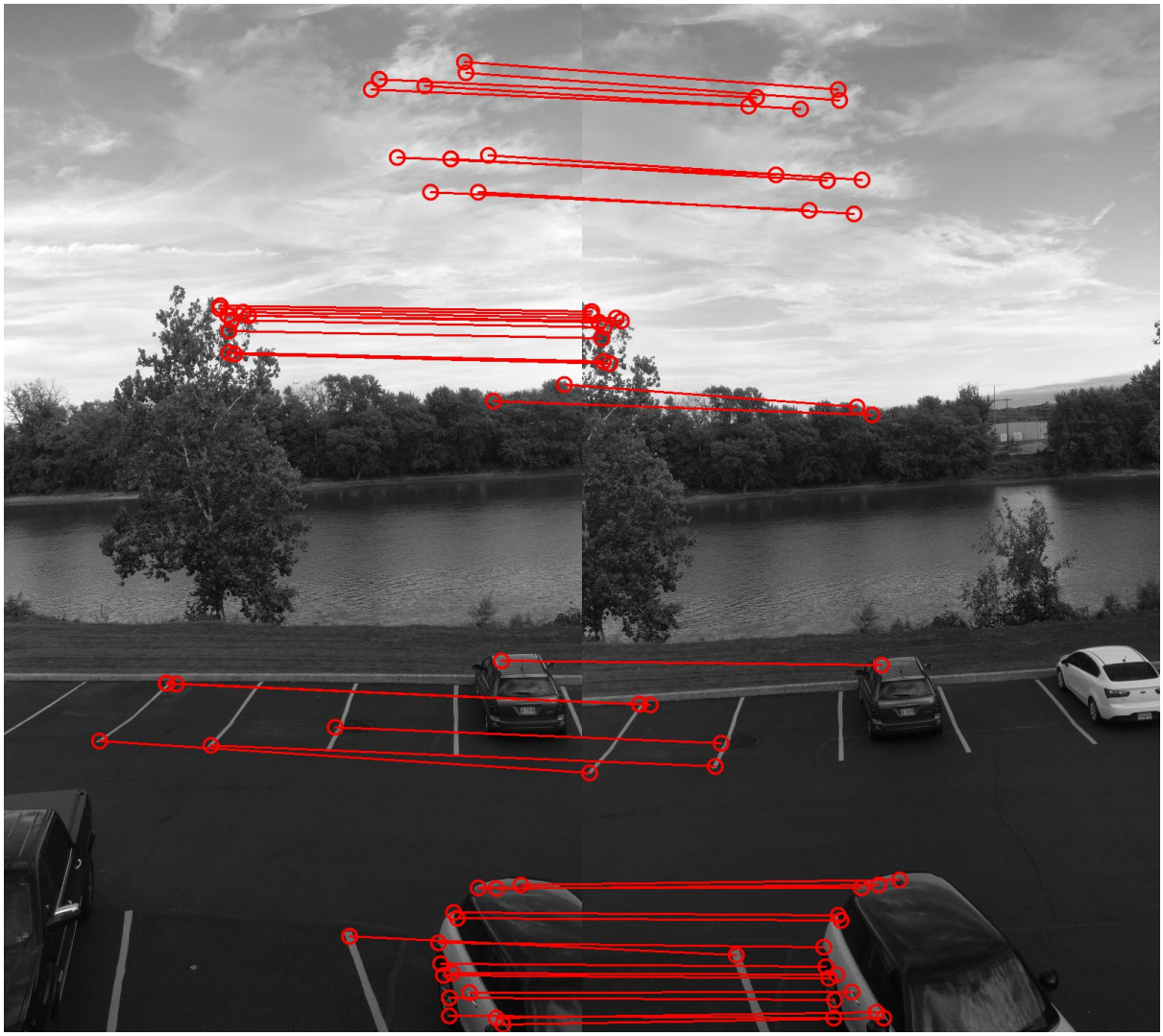

Figure 10: Outlier correspondences between images 3 and 4

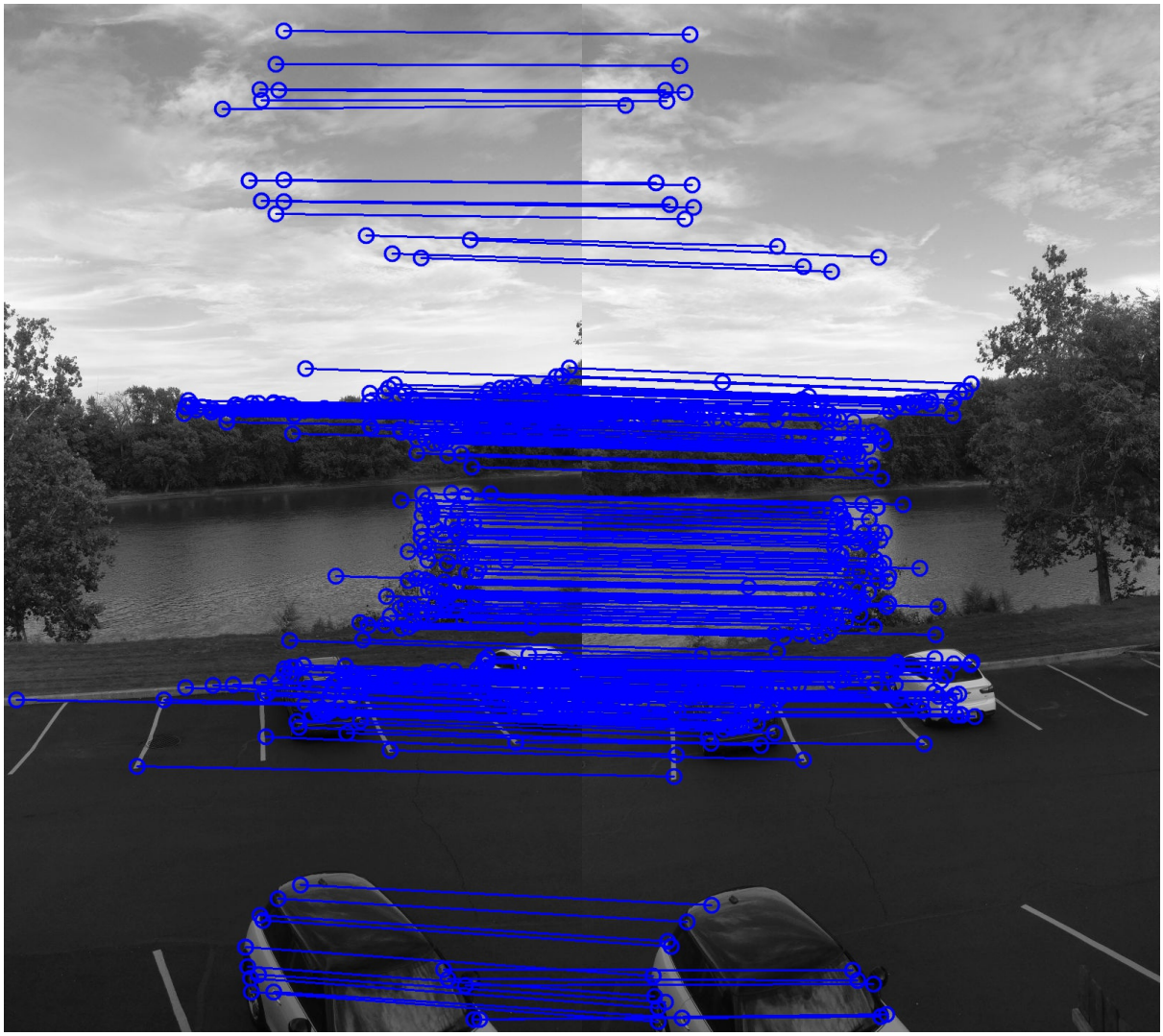

Figure 11: Extracted correspondences between images 4 and 5

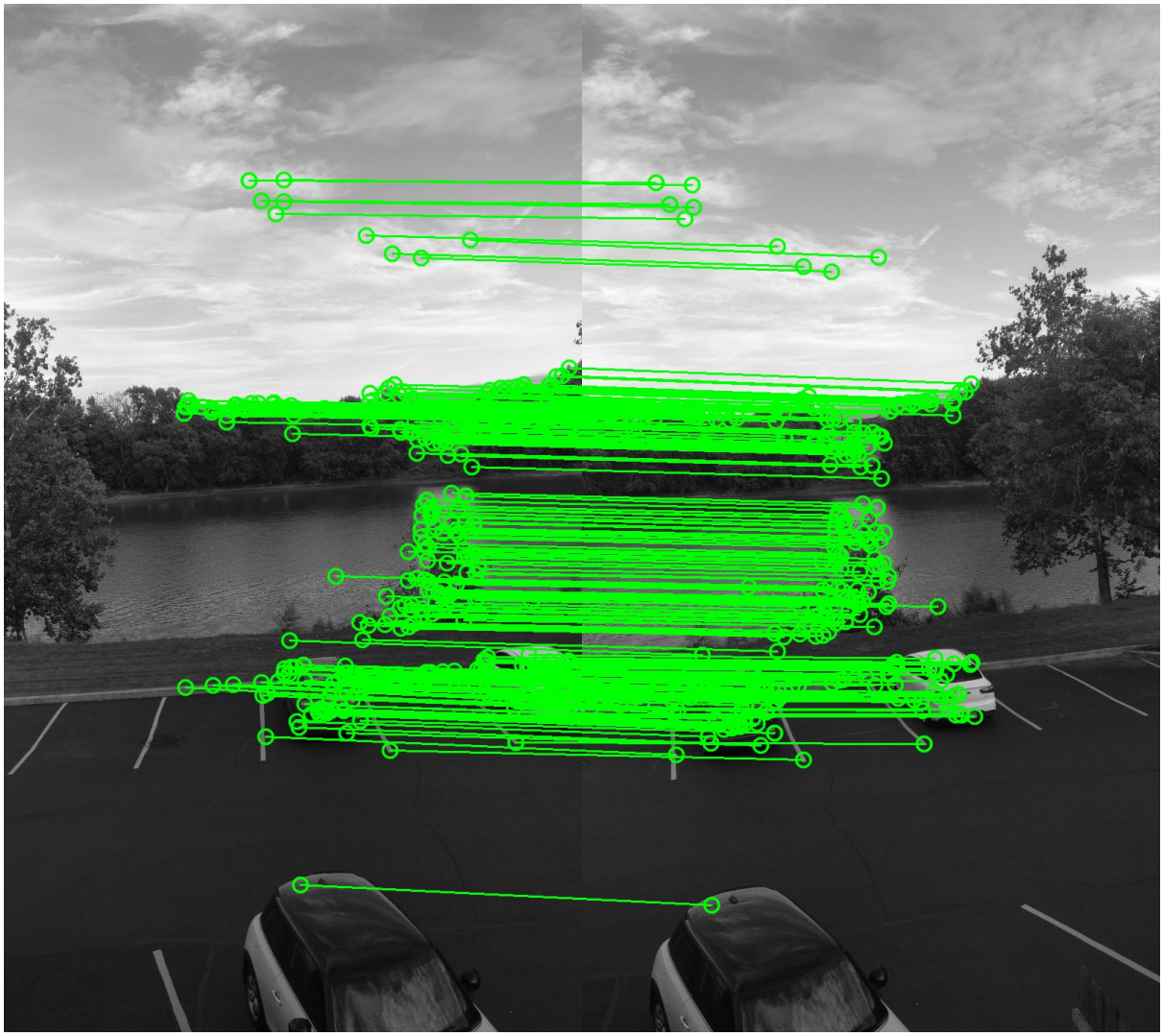

Figure 12: Inlier correspondences between images 4 and 5

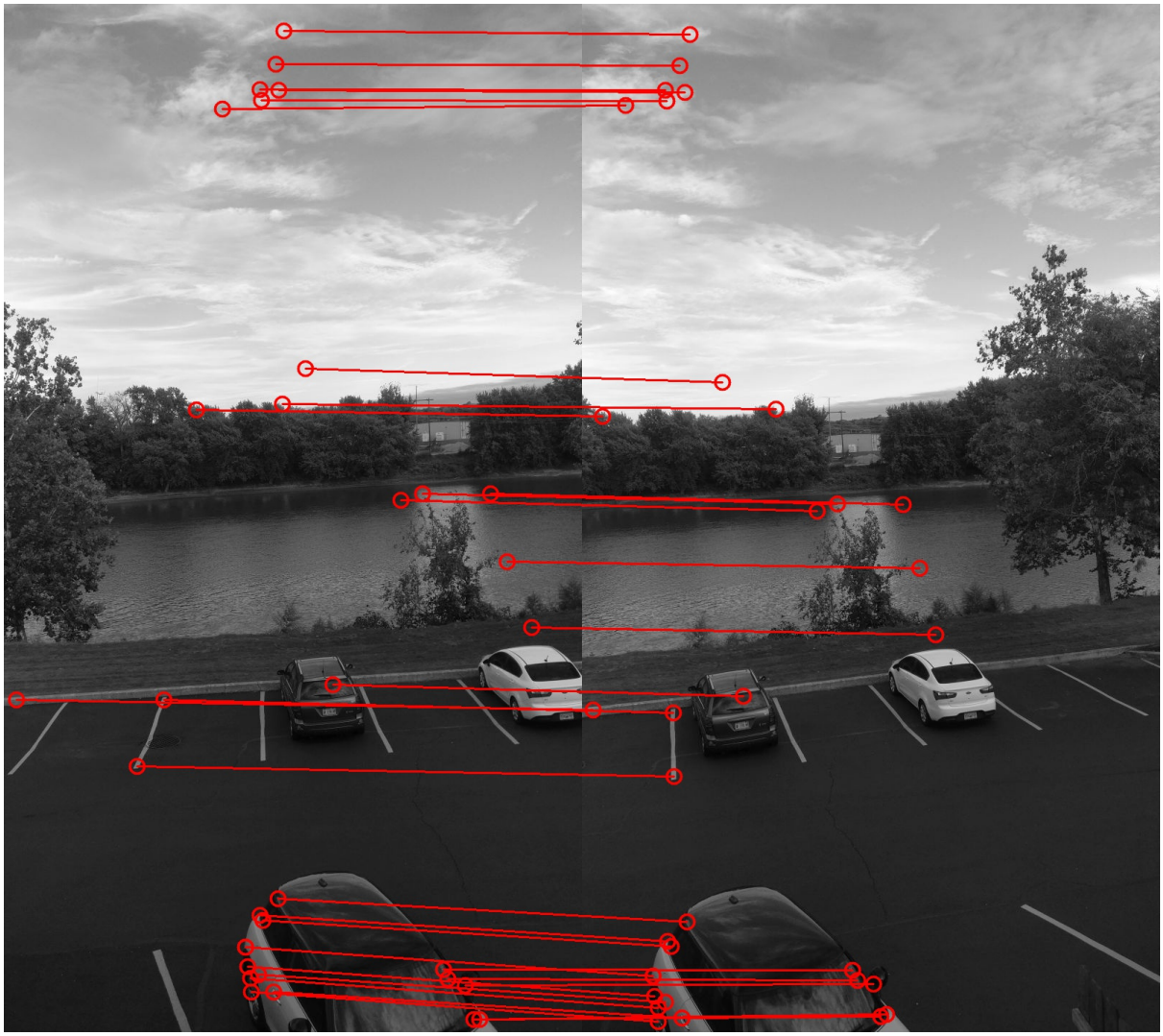

Figure 13: Outlier correspondences between images 4 and 5

Figure 14: Panorama without refinement with LM algorithm

Figure 15: Final panorama after refinement with LM algorithm

8 Source code

The code is in python 2.7, with hw5_library.py having the common functions used, and panorama_with_LM.py and panorama_without_LM.py calling the required functions from hw5_library.py. panorama_with_LM.py generates panorama of the 5 images using LM refinement and panorama_without_LM.py generates the same without LM refinement. The names are self-explanatory.

The whole source code is provided in the pages that follow.

8.1 hw5_library.py

```
# -*- coding: utf-8 -*-

# Author: Gopikrishnan Sasi Kumar (Krish)

# This library contains the functions used for HW5 of ECE661


import numpy as np
import cv2
import math
import random
import time
from scipy.optimize import root

PINK = (147, 20, 255)
BLUE = (255, 0, 0)
RED = (0, 0, 255)
CYAN = (255, 255, 0)
GREEN = (0, 255, 0)

featureRadius = 10
featureThickness = 2
line_thickness = 2

# Number of SIFT features per image
no_sift_features = 2000  # 7000

# Threshold for ratio between 2 best correspondences
sift_threshold = 0.6  # 300

# Variables for RANSAC
n_ransac = 6  # number of correspondences for random sample [4<n<10]
e_ransac = 0.2  # fraction of false correspondences
p_ransac = 0.99  # prob. of atleast one trial being free of outliers
del_ransac = 3.0  # decision threshold to construct the inlier set

# Variables for Levenberg-Marquardt Optimization
LM_threshold = 0.00000000000000000001


def get_matching_image(image1, image2, correspondences, color):
    """
    Function that returns the image matching the correspondences
    :param image1: 1st image
```

```
    :param image2: 2nd image
    :param correspondences: pairs of pixels with correspondences
    between image1 and image2
    :param color: Color for matching line
    :return: image with correspondences matched
    """
    shape1 = image1.shape
    shape2 = image2.shape
    height = max(shape1[0], shape2[0])
    img1 = np.zeros((height, shape1[1], 3), dtype='uint8')
    img1[0:height, 0:shape1[1]] = image1

    img2 = np.zeros((height, shape2[1], 3), dtype='uint8')
    img2[0:height, 0:shape2[1]] = image2

    # generate an image with image 1 and 2 on left and right
    image = np.concatenate((img1, img2), axis=1)

    index = 0

    for point_pair in correspondences:
        point1 = point_pair[0]
        point2 = point_pair[1]
        point1 = (int(round(point1[0])), int(round(point1[1])))
        point2 = (int(round(point2[0])), int(round(point2[1])))
        cv2.circle(image, tuple(point1), featureRadius,
                   color, featureThickness, cv2.LINE_AA)
        cv2.circle(image, tuple([point2[0] + shape1[1], point2[1]]), featureRadius,
                   color, featureThickness, cv2.LINE_AA)
        cv2.line(image, tuple(point1), tuple([point2[0] + shape1[1], point2[1]]),
                 color, line_thickness)
        index = index + 1

    return image


def open_named_window(image, windowname, switch=False):
    """
    Function to open cv2 named window with image and wait for keystroke
    :param image:
    :param windowname:
    :param switch: window required or not? Optional input.
    :return:
    """
    if switch == False:
        return 1
    cv2.namedWindow(windowname, cv2.WINDOW_NORMAL)
    cv2.imshow(windowname, image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return 1


def task_sift(img1, img2, input_folder, outfile, output_folder):
    """
    Function that carries out SIFT feature detection and
```

```
    correspondence matching for img1 and img2 in input_folder
    :param img1: image1
    :param img2: image2
    :param input_folder: input folder
    :param output_folder: output folder into which results are saved
    :return: correspondences. Also saving of files into result folder.
    """
    # Extract features from first image
    image1 = cv2.imread(input_folder + img1)
    gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    sift1 = cv2.xfeatures2d.SIFT_create(no_sift_features)
    print("Features determined in Image 1")
    kp1, des1 = sift1.detectAndCompute(gray1, None)

    # Extract features from second image
    image2 = cv2.imread(input_folder + img2)
    gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    sift2 = cv2.xfeatures2d.SIFT_create(no_sift_features)
    print("Features determined in Image 2")
    kp2, des2 = sift2.detectAndCompute(gray2, None)

    # Find correspondences between features in image1 and image2
    correspondences = find_sift_correspondences(kp1, des1, kp2, des2)
    image = get_matching_image(cv2.cvtColor(gray1, cv2.COLOR_GRAY2BGR),
                               cv2.cvtColor(gray2, cv2.COLOR_GRAY2BGR),
                               correspondences, BLUE)
    cv2.imwrite(output_folder + outfile, image)
    print("Matching images saved.")
    return correspondences


def find_sift_correspondences(kp1, des1, kp2, des2):
    """
    Function to find the correspondences between feature points in image1
    to those in image2
    :param kp1: keypoints in image 1
    :param des1: feature descriptors for keypoints in kp1
    :param kp2: keypoints in image 2
    :param des2: feature descriptors for keypoints in kp2
    """
    print("Determining correspondences")

    class data:  # class structure used for sorting point pairs
        def __init__(self, point_pair, value):
            self.point_pair = point_pair
            self.value = value

    des1_order = des1.shape
    des2_order = des2.shape

    correspondences = []
    corr_new = []

    # Find best 2 matches for each feature descriptor of image 1 with
    # those of image 2
    for i in range(des1_order[0]):
```

```
            pairs = [data([], np.Inf), data([], np.Inf)]
            point1 = kp1[i].pt
            # point1 = (int(round(point1[0])), int(round(point1[1])))
            for j in range(des2_order[0]):
                point2 = kp2[j].pt
                # point2 = (int(round(point2[0])), int(round(point2[1])))
                dist = np.linalg.norm(des1[i, :] - des2[j, :])
                new_pair = [point1, point2]
                pairs.append(data(new_pair, dist))
                pairs = sorted(pairs, key=lambda x: x.value)
                pairs.pop()  # maintain the number of elements as 2

            # Find ratio of distances to best and second best matches
            thisvalue = pairs[0].value / pairs[1].value

            # Sort the correspondences based on this ratio
            correspondences.append(data(pairs[0].point_pair, thisvalue))

        # Sort point pairs based on ratio of euclidean
        # distance between best and second best matches
        correspondences = sorted(correspondences, key=lambda x: x.value)

        # Choose only the best point pairs as output, i.e. small value of ratio
        # fid = open("test.txt", 'w')
        for i in range(len(correspondences)):
            # fid.write(str(correspondences[i].value) + "\n")
            if correspondences[i].value > sift_threshold:
                break  # exit loop when the correspondence strength is low
            corr_new.append(correspondences[i].point_pair)
        # fid.close()
        return corr_new


    def ransac(correspondences):
        """
        Function that carries out RANSAC algorithm for the input correspondences
        :param correspondences: list of correspondences between images.
        :return Homography from first set to second set of points.
        Also return the inliers.
        """

        print("RANSAC Algorithm Started.")

        # Number of trials
        N_ransac = math.log(1 - p_ransac) / math.log(1 - (1 - e_ransac) ** n_ransac)
        N_ransac = int(math.ceil(N_ransac))
        print("N_ransac = {}".format(N_ransac))

        # Total number of correspondences
        n_total = len(correspondences)
        print("No. of correspondences = {}".format(n_total))

        # Minimum size of inlier set
        M_ransac = (1 - e_ransac) * n_total
        M_ransac = int(math.ceil(M_ransac))
        print("Minimum required size of inlier set = {}".format(M_ransac))
```

```
    num_inliers = -1  # number of inliers
    H_final = []  # final homography to be used

    # Strip the correspondences into X and Xdash, so that these can be used
    # in other functions readily.
    X = []
    Xdash = []
    inliers = []
    for point_pair in correspondences:
        X.append(point_pair[0][0])
        X.append(point_pair[0][1])
        X.append(1)
        Xdash.append(point_pair[1][0])
        Xdash.append(point_pair[1][1])
        Xdash.append(1)
    X = np.array(X, dtype='float64')
    Xdash = np.array(Xdash, dtype='float64')
    X = X.reshape(-1, 3)
    X = X.T
    Xdash = Xdash.reshape(-1, 3)
    Xdash = Xdash.T

    # Run RANSAC loop for N_ransac iterations
    for trial in range(N_ransac):
        # choose n random samples
        this_sample = random.sample(correspondences, n_ransac)
        H = compute_homography(this_sample)  # compute homography.

        # Find out the inliers in correspondences with del_ransac as
        # decision threshold
        inliers_temp = find_inliers(H, X, Xdash, del_ransac, correspondences)

        if len(inliers_temp) > num_inliers:
            num_inliers = len(inliers_temp)
            inliers = inliers_temp
            # H_final = H

    print("Maximum inlier size obtained = {}".format(num_inliers))
    if num_inliers < M_ransac:
        print("Warning: Minimum size of inlier set not obtained!")

    # Now compute homography using the inlier set
    H_final = compute_homography(inliers)

    return [H_final, inliers]


def compute_homography(sample):
    """
    Function that calls gethomography_multi after
    arranging data as X and Xdash from sample.
    Sample contains correspondences as pairs of points.
    It also returns the X and Xdash values used.
    """
    points1 = []
```

```
    points2 = []
    for point_pair in sample:
        points1.append(point_pair[0][0])
        points1.append(point_pair[0][1])
        points2.append(point_pair[1][0])
        points2.append(point_pair[1][1])
    points1 = np.array(points1).reshape(-1, 1)
    points2 = np.array(points2).reshape(-1, 1)
    return gethomography_multi(points1, points2)


def gethomography_multi(X, Xdash):
    """Function that returns the homography from X to Xdash (given Xdash = HX)
    for more than 4 points of correspondence
    X is the coordinates of the points in the domain of the mapping in the form
    [x1,y1,x2,y2,x3,y3,x4,y4,...].T
    Xdash is the coordinates of the points in the range of the mapping in the
    form [x1',y1',x2',y2',x3',y3',x4',y4',...].T"""

    # defining A
    try:
        assert (X.shape == Xdash.shape)
    except:
        print('Error in dimensions of inputs to the function >>GETHOMOGRAPHY')
    A = np.array([])  # define A as a vector and reshape later
    for i in range(X.shape[0] // 2):
        A = np.append(A, [X[2 * i][0], X[2 * i + 1][0], 1, 0, 0, 0,
                          -X[2 * i][0] * Xdash[2 * i][0],
                          -X[2 * i + 1][0] * Xdash[2 * i][0], 0, 0, 0,
                          X[2 * i][0], X[2 * i + 1][0],
                          1, -X[2 * i][0] * Xdash[2 * i + 1][0],
                          -X[2 * i + 1][0] * Xdash[2 * i + 1][0]])

    A = A.reshape(X.shape[0], 8)
    try:
        ATAinv = np.linalg.inv(np.dot(A.T, A))
    except:
        print('A is not invertible!!! >>GETHOMOGRAPHY')
    h = np.dot(ATAinv, np.dot(A.T, Xdash))  # h still doesn't have (3,3)th element.
    assert (h.shape == (8, 1))
    h = np.append(h, [[1]], axis=0)
    return h.reshape(3, 3)  # return the 3x3 H matrix


def find_inliers(H, X, Xdash, delta, correspondences):
    """
    The function to find the inliers for a given H, X, Xdash set, and delta.
    The list of correspondences is also input so as to reduce regeneration
    of point pairs list.
    Returns the list of inliers.
    """
    Y = np.matmul(H, X)  # Actual mapping with H
    Y = Y / Y[2, :]  # Normalising wrt x3 in HC repr.
    # But Xdash is the observed mapping
    error = Y - Xdash  # 3rd element in each point's HC becomes 0.
    # So, #peace during sum.
```

```
    squared_error = error ** 2
    d_squared = np.sum(squared_error, axis=0)
    ind = np.where(d_squared <= delta ** 2)
    return [correspondences[i] for i in ind[0]]


def find_H_between_a_pair(img_file1, img_file2, infolder, outfile, outfolder, LM_on=True):
    """
    Function to find the homography between a pair of images based on
    the feature descriptors in the pair of images
    outfile is the file to save the image showing correpondence match
    between the pair of input images
    :param LM_on: Nonlinear Optim with LM alg on/off.
    :return Homography from img1 to img2
    """
    # Determine the correspondences between the pair of images
    # based on SIFT descriptors
    corr = task_sift(img_file1, img_file2, infolder, outfile, outfolder)

    # Find the homography based on RANSAC algorithm
    [H, inliers] = ransac(corr)

    # Generate image showing inliers and outliers
    image_inlier_outlier(img_file1, img_file2, infolder, outfile, outfolder,
                         inliers, corr, [GREEN, RED])

    # If the LM_on switch is True, refine H with LM algorithm
    if LM_on:
        H = LM_algorithm(H, inliers)

    return H


def image_inlier_outlier(img1file, img2file, infolder, outfile, outfolder, inliers, corr, colors):
    """
    Function to save and return images showing inliers and outliers in
    correspondences between 2 images
    :param img1file: file name 1
    :param img2file: file name 2
    :param infolder: input files' folder
    :param outfile: output filename's first part
    :param outfolder: output folder
    :param inliers: set of inliers
    :param corr: set of all correspondences, of which inliers is a part
    :param colors: list of 2 colors, in the order [inlier-color, outlier-color]
    :return [inlier image, outlier image]
    """
    # Generate outlier set
    outliers = []
    for point_pair in corr:
        if point_pair not in inliers:
            outliers.append(point_pair)

    image1 = cv2.imread(infolder + img1file, cv2.IMREAD_GRAYSCALE)
    image2 = cv2.imread(infolder + img2file, cv2.IMREAD_GRAYSCALE)
```

```
    image_inliers = get_matching_image(cv2.cvtColor(image1, cv2.COLOR_GRAY2BGR),
                                       cv2.cvtColor(image2, cv2.COLOR_GRAY2BGR),
                                       inliers, colors[0])
    cv2.imwrite(outfolder + outfile + "_inliers.jpg", image_inliers)
    image_outliers = get_matching_image(cv2.cvtColor(image1, cv2.COLOR_GRAY2BGR),
                                        cv2.cvtColor(image2, cv2.COLOR_GRAY2BGR),
                                        outliers, colors[1])
    cv2.imwrite(outfolder + outfile + "_outliers.jpg", image_outliers)
    print("Inliers and outliers saved to images")
    return [image_inliers, image_outliers]


def generate_panorama(imagefiles, infolder, panorama_out, outfolder, LM_on):
    """
    Function to generate the panorama
    :parameter imagefiles: list of filenames of images
    :parameter panorama_out: output file name for panorama
    :parameter LM_on: Non linear optimn with LM on/off
    """
    # Determine the homographies between each of the adjacent pairs
    # of images
    H_all = []
    image_count = len(imagefiles)
    for i in range(image_count - 1):
        print("\nProcessing Image Pair {}\n----------------------------".format(i + 1))
        H = find_H_between_a_pair(imagefiles[i], imagefiles[i + 1], infolder,
                                  "corresp_pair" + str(i) + ".jpg", outfolder, LM_on)
        H_all.append(H)

    # Compute all homographies with respect to the central image
    middle = int((image_count + 1) / 2) - 1

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(middle, len(H_all)):
        H_wrt_ref = np.matmul(H_wrt_ref, np.linalg.inv(H_all[i]))
        H_all[i] = H_wrt_ref

    H_wrt_ref = np.eye(3, dtype=np.float64)
    for i in range(middle - 1, -1, -1):
        H_wrt_ref = np.matmul(H_wrt_ref, H_all[i])
        H_all[i] = H_wrt_ref

    # homography for the middle image
    H_all.insert(middle, np.eye(3, dtype=np.float64))

    # Now, the middle image needs to be placed somewhere in the
    # middle of the panorama. Hence, we need to multiply each of
    # homographies with the translation matrix too.

    tx = 0
    ty = 0

    for i in range(middle):
        image = cv2.imread(infolder + imagefiles[i])
        imshape = image.shape
        tx = tx + imshape[1]
```

```
    H_translate = np.array([1, 0, tx, 0, 1, ty, 0, 0, 1], dtype=float)
    H_translate = H_translate.reshape(3, 3)

    for i in range(len(H_all)):
        H_all[i] = np.matmul(H_translate, H_all[i])

    # Stitch together all the images
    image = stitch_images(imagefiles, infolder, H_all)
    print("Panorama generated")
    cv2.imwrite(outfolder + panorama_out, image)
    print("Panorama saved to file")
    return image


def stitch_images(imagefiles, infolder, H_all):
    """
    Function to stitch together all the images based on H_all
    """
    print("\nStitching images\n----------------------")
    height = 0
    width = 0
    image_all = []
    for image_name in imagefiles:
        image = cv2.imread(infolder + image_name)
        imshape = image.shape
        height = max(height, imshape[0])
        width = width + imshape[1]
        image_all.append(image)

    # Generate canvas
    canvas = np.zeros((height, width, 3), np.uint8)

    print("Length of image_all: {}".format(len(image_all)))
    print("Length of H_all: {}".format(len(H_all)))

    for count, image in enumerate(image_all):
        canvas = applyhomography(image, canvas, H_all[count])

    return canvas


def applyhomography(image, canvas, H):
    """ Function to project image onto canvas
    :param image: Input image to transform
    :param canvas: Canvas
    :param H: Homography from image to canvas
    :return: canvas with image projected
    """
    tic = time.time()
    print('Projecting to canvas \nTimer started')
    temp = image.shape  # Get dimensions of image
    ipimageH = temp[0]
    ipimageL = temp[1]

    H = np.linalg.inv(H)  # Invert H so that H is the homography
```

```
    # from outputimage to image

    worldH = canvas.shape[0]
    worldL = canvas.shape[1]

    # Define a matrix for the canvas pixel coordinates
    worldCoordinates = np.array([[i, j, 1] for i in range(worldL) for j in range(worldH)])
    worldCoordinates = worldCoordinates.T  # create a matrix with all the pixels in the
    # canvas. These are surely integers.

    transformedCoord = np.dot(H, worldCoordinates)  # apply homography
    transformedCoord = transformedCoord / transformedCoord[2, :]  # normalize the points
    # with x3 so as to get the points in the form [x,y,1].T. Note that the coordinates
    # will be float values.
    toc = time.time()
    print('Transformed coordinates computed at: ' + str(toc - tic) + ' seconds')

    [validWorld, validTransWorld] = getValidPixels(worldCoordinates, transformedCoord,
                                                   0, ipimageL - 2, 0, ipimageH - 2)
    # Get the pixel coordinates of the canvas that need to be changed, and their
    # corresponding transformations.
    toc = time.time()
    print('Pixels that need to be updated determined at: ' + str(toc - tic) + ' seconds')

    numbPts = validWorld.shape[1]
    for i in range(numbPts):  # Loop through each of the valid canvas pixels and replace
        # with ipimage pixels
        Pointnew = validTransWorld[:, i]  # Pointnew is the transformed point in the ipimage.
        # It is of type float. It is of shape (3,)
        canvas[validWorld[1][i]][validWorld[0][i]][:] = getpixel(image, Pointnew)
        # Get the ipimage-pixel from ipimage

    toc = time.time()
    print('Image Projection finished at: ' + str(toc - tic) + ' seconds')
    return canvas


def getpixel(image, floatPoint):
    """Function to find the pixel value at a point on the image. As the point
    is not an integer, weighted average based on L2norm is used"""
    x = int(np.floor(floatPoint[0]))
    y = int(np.floor(floatPoint[1]))
    Point = np.array([floatPoint[0], floatPoint[1]])
    d00 = np.linalg.norm(Point - np.array([x, y]))
    d01 = np.linalg.norm(Point - np.array([x, y + 1]))
    d10 = np.linalg.norm(Point - np.array([x + 1, y]))
    d11 = np.linalg.norm(Point - np.array([x + 1, y + 1]))

    return (image[y][x][:] * d00 + image[y + 1][x][:] * d01 + image[y][x + 1][:] * d10
            + image[y + 1][x + 1][:] * d11) / (d00 + d01 + d10 + d11)


def getValidPixels(worldCoordinates, transformedCoord, xLow, xUp, yLow, yUp):
    """
    :param worldCoordinates: All the coordinates in the world for which
    corresponding transformation is trans..Coord
```

```
    :param transformedCoord: Transformed values for world coordinates
    :xLow, xUp, yLow, yUp: allowed range of x and y in transformed output
    :return: [validWorld, validTransWorld] Valid pixels in the world and their
    coreesponding pixels in the transformed world
    """
    temp = transformedCoord[0, :] >= xLow
    worldCoordinates = worldCoordinates[:, temp]
    transformedCoord = transformedCoord[:, temp]

    temp = transformedCoord[0, :] <= xUp
    worldCoordinates = worldCoordinates[:, temp]
    transformedCoord = transformedCoord[:, temp]

    temp = transformedCoord[1, :] >= yLow
    worldCoordinates = worldCoordinates[:, temp]
    transformedCoord = transformedCoord[:, temp]

    temp = transformedCoord[1, :] <= yUp
    worldCoordinates = worldCoordinates[:, temp]
    transformedCoord = transformedCoord[:, temp]

    return [worldCoordinates, transformedCoord]


def LM_algorithm(H, inliers):
    """
    Function to find the optimized homography using nonlinear
    least squares algorithm using the inliers.
    """
    print("Refining homography with Levenberg-Marquardt algorithm")
    h = H.ravel()
    # h = h.reshape(1, 9)
    sol = root(cost_jac_func, h, args=(inliers), jac=True, method='lm')
    H = sol.x
    H = H / H[-1]
    print(H)

    sol = LM_algo(cost_jac_func, h, args=inliers, del_p_thresh=LM_threshold)
    H = sol.x
    H = H / H[-1]
    print(H)

    H = H.reshape(3, 3)
    print("Homography refined with LM algorithm")
    return H


def cost_jac_func(h, corr):
    """
    Function that returns the (Xi-fi) and its jacobian for LM optimization
    """
    X = []
    F = []
    J = []
    # Computing X, f and Jf
    for point_pair in corr:
```

```
            point1 = point_pair[0]
            x = point1[0]
            y = point1[1]
            point2 = point_pair[1]
            X.extend([point2[0], point2[1]])

            f1numt = f1num(h, point1)
            fdent = fden(h, point1)
            f2numt = f2num(h, point1)
            F.extend([f1numt / fdent, f2numt / fdent])

            k1 = -f1numt / fdent ** 2
            k2 = -f2numt / fdent ** 2
            J.extend([x / fdent, y / fdent, 1 / fdent, 0, 0, 0, k1 * x, k1 * y, k1])
            J.extend([0, 0, 0, x / fdent, y / fdent, 1 / fdent, k2 * x, k2 * y, k2])

        X = np.array(X)

        F = np.array(F)

        J = np.array(J)
        J = J.reshape(-1, 9)

        E = X - F  # errors.
        C = E  # Xi - Fi
        E = E.reshape(-1, 1)
        Jacobian = -J  # Jacobian of Xi-Fi
        return C, Jacobian


    def f1num(h, x):
        return h[0] * x[0] + h[1] * x[1] + h[2]


    def fden(h, x):
        return h[6] * x[0] + h[7] * x[1] + h[8]


    def f2num(h, x):
        return h[3] * x[0] + h[4] * x[1] + h[5]


    def LM_algo(func_ip, p0, args=None, del_p_thresh=0.0001):
        """
        Function to carry out Levenberg-Marquardt nonlinear optimization.
        It optimizes Sum_i({e(p)_i}^2)
        :param func: A function that returns e_i and Jacobian of e_i
        :param p0: initial guess
        :param args: arguments to func
        :param del_p_thresh: Threshold for ||del(p)|| below which algorithm
        is terminated
        """

        def cost(ep):
            return np.matmul(ep.T, ep)
```

```
    class data:
        # class of data that contains the solution and final cost, used
        # for returning data
        def __init__(self, x, cost):
            self.x = x
            self.cost = cost

    def func(p, args):
        if args == None:
            return func_ip(p)
        else:
            return func_ip(p, args)

    T = 0.5  # value used for initialisation of u_k

    pk = p0
    [e, je] = func(p0, args)
    JtJ = np.matmul(je.T, je)
    I = np.eye(JtJ.shape[0])
    u0 = T * np.diag(JtJ).max()
    uk = u0

    while (True):
        [e, je] = func(pk, args)  # get e_i and its jacobian
        # Compute del(p) and new p_k
        JtJ = np.matmul(je.T, je)
        JfTE = -np.matmul(je.T, e)
        del_p = np.matmul(np.linalg.inv(JtJ + uk * I), JfTE)
        pk_plus1 = pk + del_p
        # Determine quality of computed p_k and compute next u_k
        [e_next, je_next] = func(pk_plus1, args)
        cost_new = cost(e_next)
        cost_diff = cost(e) - cost_new
        rho = (cost_diff) / (np.matmul(del_p.T, JfTE) + np.matmul(del_p.T, uk * del_p))
        uk_plus1 = uk * max(1.0 / 3, 1 - (2 * rho - 1) ** 3)
        uk = uk_plus1
        # Retain p_k if new p_k leads to jumping over minimum
        if cost_diff >= 0:
            pk = pk_plus1
            # Break loop if norm of del_p below threshold
            if np.linalg.norm(del_p) < del_p_thresh:
                break

    return data(pk, cost_new)
```

8.2 panorama_with_LM.py

```
# -*- coding: utf-8 -*-

# Author: Gopikrishnan Sasi Kumar (Krish)

# This script finds the panorama using Levenberg-Marquardt nonlinear
# optimization

from hw5_library import *
```

```
img_files = ["1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg"]
image = generate_panorama(img_files, "images/", "panorama_with_my_LM.jpg",
                          "results/", True)
```

8.3 panorama_without_LM.py

```
# -*- coding: utf-8 -*-

# Author: Gopikrishnan Sasi Kumar (Krish)

# This script finds the panorama without using Levenberg-Marquardt nonlinear
# optimization

from hw5_library import *

img_files = ["1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg"]
image = generate_panorama(img_files, "images/", "panorama_without_LM.jpg",
                          "results/", False)
```