ECE 66100 Computer Vision
Homework 5
10/12/2022

Name: *Nikita Ravi (ravi30)*

1 Theory Question

1. Conceptually speaking, 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?

   Answer: The RANSAC algorithm involves taking N trials to estimate the best set of inlier points gathered from a randomly collected sample of n points. This is accomplished by computing a homography using the randomly collected sample of points. This homography will then be used to check if each correspondence lies within an acceptable range δ from the actual coordinate. If the correspondence lies within the acceptable range, it is labelled as an inlier. Otherwise, it is labelled as an outlier. With N trials, the algorithm will find enough inliers that will help compute the closest estimate of the true pairwise homography where the inliers are the true correspondences.

2. As you will see in Lecture 13, the Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method but it can be numerically unstable. 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 reliably fast and numerically stable at the same time

   Answer: The Gradient Descent (GD) method is the optimal method when the estimation obtained from the linear least squares method is farthest from the actual local minimum of the convex function. The Gauss-Newton method is the quickest way to the local minimum but it is not always along the direction of the steepest descent thus making it unstable. However, it is an optimal method to use when the estimation is close to the local minimum. The Levenberg-Marquardt algorithm combines the best of both methods by adding a heuristic damping coefficient μ which when manipulated can steer between the Gradient Descent approach or the Gauss Newton approach. At the beginning, μ is set to an extremely large value so that the steps taken along the convex function mimics Gradient Descent but when the estimation gets closer to the local minimum, μ decreases thus allowing for the steps to mimic Gauss-Newton.

2 Programming Section

2.1 Theoretical Background

2.1.1 RANSAC

As explained in the theoretical question, The RANSAC algorithm involves taking N trials to estimate the best set of inlier points gathered from a randomly collected sample of n points. This

is accomplished by computing a homography using the randomly collected sample of points. This homography will then be used to check if each correspondence lies within an acceptable range δ from the actual coordinate. If the correspondence lies within the acceptable range, it is labelled as an inlier. Otherwise, it is labelled as an outlier. With N trials, the algorithm will find enough inliers that will help compute the closest estimate of the true pairwise homography where the inliers are the true correspondences.

The parameters chosen for the RANSAC algorithm are as follows:

- $n = 5$ since the recommendation is for n to be between $4 < n < 10$
- $\epsilon = 0.85$ where ϵ is the probability that a randomly chosen (x, x') is an outlier
- $p = 0.99$ where p is the probability that at least one of the N trials will be free of outliers in the calculation of the homography matrix
- N trials is calculated as

$$N = \frac{\ln 1 - p}{\ln 1 - (1 - \epsilon)^n} \tag{1}$$

- n_{total} is the number of correspondences
- $M = n_{total}(1 - \epsilon)$ where M is minimum value for the size of the inlier set for it to be acceptable
- $\sigma = 2$ where σ is a constant in the noise-induced displacement of the true location of a pixel which is modelled by the Gaussian function

$$g(\Delta x, \Delta y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(\Delta x)^2 + (\Delta y)^2}{2\sigma^2}} \tag{2}$$

  and σ is usually set between $0.5 < \sigma < 2$
- $\delta = 3\sigma$ which is commonly used

2.1.2 Least Square Estimation

Linear Least Square Estimation is used to estimate a starting homography consisting of all the inlier points. The homography H represents a linear mapping from one homogeneous coordinate to another in different planes of the same dimension. $\vec{x'}$ is computed by multiplying H with $\vec{x}$. The matrix H can be expressed as

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

Therefore, using the relation mentioned above, the homogeneous point in the range is given by:

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

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

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

To obtain the coordinates in the range on the real plane, we have to divide the first two coordinates of the homogeneous coordinate by the third coordinate, thus giving

$$x' = \frac{x_1'}{x_3'} = \frac{h_{11}x_1+h_{12}x_2+h_{13}x_3}{h_{31}x_1+h_{32}x_2+h_{33}x_3}$$

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

We can simplify the x' and y' coordinate to also include the actual coordinates in the domain on the real plane by dividing the RHS by x_3 as shown below

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

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

We make the above x' and y' equations into a system of linear equations:

$$h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx' - h_{33}x' = 0$$

$$h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy' - h_{33}y' = 0$$

Therefore, for each correspondence we get the following matrix equation to solve for H

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

In an over-determined system when we have more than four correspondences, the linear least square estimation can be calculated by using the pseudo-inverse

$$\vec{h} = (A^T A)^{-1} A^T b \tag{3}$$

2.1.3 Levenberg-Marquardt Algorithm

As already mentioned in the theory question, the Levenberg-Marquardt algorithm combines the best of Gradient Descent and Gauss-Newton by adding a heuristic damping coefficient μ which when manipulated can steer between the Gradient Descent approach or the Gauss Newton approach. At the beginning, μ is set to an extremely large value so that the steps taken along the convex function mimics Gradient Descent but when the estimation gets closer to the local minimum, μ decreases thus allowing for the steps to mimic Gauss-Newton.

For this homework assignment, the Levenberg-Marquardt algorithm is implemented by using the built-in SciPy function. The cost function is given by the error between the actual coordinates in the range space and the computed coordinates using the homography matrix H.

2.2 Source Code

```
# Name: Nikita Ravi
# Class: ECE 66100
# Homework #5
# Deadline: 09/28/2022

# Import Modules
import cv2
import math
import numpy as np
import os
from scipy.optimize import leastsquares

def getimages(path, task=1):
    path = path + "/Task" + str(task)
    images = [cv2.imread(path + "/" + img) for img in sorted(os.listdir(path)
    [::-1])]
    return images

def displayimages(images, idx=None):
    if(idx):
        cv2.imshow("Window", images[idx])
    else:
        cv2.imshow("Window", images)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    quit()

def computehomography(domaincoord, rangecoord, over=False):
    # From homework #3
    n = domaincoord.shape[1]
    A = np.zeros((2*n, 8))
    b = np.zeros((2*n, 1))
    H = np.zeros((3,3))

    for idx in range(n):
        A[2*idx] = [domaincoord[0][idx], domaincoord[1][idx], 1, 0, 0, 0, (-
    domaincoord[0][idx] * rangecoord[0][idx]), (-domaincoord[1][idx] *
```

```
    rangecoord[0][idx])]
36          A[2*idx + 1] = [0, 0, 0, domaincoord[0][idx], domaincoord[1][idx],
    1, (-domaincoord[0][idx] * rangecoord[1][idx]), (-domaincoord[1][idx] *
     rangecoord[1][idx])]
37          b[2*idx] = rangecoord[0][idx]
38          b[2*idx + 1] = rangecoord[1][idx]
39
40      h = np.array([])
41      if(over):
42          Aplus = np.linalg.pinv(np.matmul(A.T, A)) #pseudo-inverse of A
43          h = np.dot(np.matmul(Aplus, A.T), b)
44      else:
45          h = np.dot(np.linalg.pinv(A), b)
46
47      row = 0
48      for idx in range(0, len(h), 3):
49          spliced = h[idx:idx+3]
50          if(len(spliced) == 3):
51              H[row] = spliced.T
52          else:
53              H[row] = np.append(spliced, [1])
54          row += 1
55
56      return H
57
58  def establishcorrespondences(image1, descriptor1, keyPoint1, image2,
    descriptor2, keyPoint2, path):
59      # Citation: https://docs.opencv.org/4.x/dc/dc3/tutorialpymatcher.html
60      bruteForceMatcher = cv2.BFMatcher() # It takes the descriptor of one
    feature in first set and matches it with all other features in second set
    using some distance calculation - the closest distance is returned
61      matches = bruteForceMatcher.match(descriptor1, descriptor2) # returns
    matches where k is specified by the user
62      matches = sorted(matches, key=lambda x: x.distance)
63      domaincoord, rangecoord = [], []
64
65      for match in matches:
66          domainpoint = list(map(int, keyPoint1[match.queryIdx].pt))
67          domainpoint.append(1)
68          domaincoord.append(domainpoint)
69          rangepoint = list(map(int, keyPoint2[match.trainIdx].pt))
70          rangepoint.append(1)
71          rangecoord.append(rangepoint)
72
73      combined = np.hstack((image1, image2))
74      result = cv2.drawMatches(image1, keyPoint1, image2, keyPoint2, matches
    [:100], combined, flags=cv2.DrawMatchesFlagsNOTDRAWSINGLEPOINTS)
75      cv2.imwrite(path, result)
76
77      domaincoord, rangecoord = domaincoord, rangecoord
78      return domaincoord, rangecoord
79
80  def sift(image1, image2, idx1, idx2):
81      siftdetector = cv2.xfeatures2d.SIFTcreate() # Create SIFT Detector
```

```
    keyPoint1, descriptor1 = siftdetector.detectAndCompute(image1, None)
    keyPoint2, descriptor2 = siftdetector.detectAndCompute(image2, None)

    return establishcorrespondences(image1, descriptor1, keyPoint1, image2,
    descriptor2, keyPoint2, "hw05/siftcorrespondences/Task1/image" + str(idx1
    ) + "image" + str(idx2) +".jpg")

def findinliers(H, domaincoord, rangecoord, delta):
    rangehat = np.matmul(H, domaincoord.T).T
    rangehat = rangehat.T / rangehat.T[2,:]
    rangehat = rangehat.T
    error = np.sum(np.abs(rangehat - rangecoord)**2, axis=1)
    indices = np.where(error ¡= delta)[0]
    return indices

def ransac(images, task=1, LM=True):
    """
    Constants
    n = 5 (4 ¡ n ¡ 10)
    ntotal = number of correspondences
    epsilon = 0.85
    p = 0.99
    N = ln(1-p)/(ln[1-(1-epsilon) n])
    M = (1-epsilon) * ntotal
    delta = 3 * sigma - typical p.11.3
    sigma = 2 - typical p11.3
    """
    # Constants
    n = 5
    epsilon = 0.85
    p = 0.99
    N = int(math.log(1-p) / (math.log(1-(1-epsilon)**n)))
    sigma = 2
    delta = 3 * sigma

    # Inputs
    images = [img.copy() for img in images]
    gray = [cv2.cvtColor(img, cv2.COLORBGR2GRAY) if(len(img.shape) == 3) else
     img for img in images]
    Hlist = []
    bestinliers = []

    for idx in range(len(gray) - 1):
        domaingray, domainimage = gray[idx], images[idx]
        rangegray, rangeimage = gray[idx + 1], images[idx + 1]
        domaincoord, rangecoord = sift(domaingray, rangegray, idx, idx +
    1)
        domaincoord, rangecoord = np.array(domaincoord), np.array(
    rangecoord)

        ntotal = domaincoord.shape[0]
        M = int((1 - epsilon) * ntotal)
        inlierpercentage = 0
        indices = list(range(ntotal))
```

```
131
132         bestinlieridx = None
133         for trial in range(N): # Conduct N Trials
134             randomsampleindices = np.random.choice(indices, n) # Randomly
    select a sample of indices of n correspondences
135             randomdomains = domaincoord[randomsampleindices]
136             randomranges = rangecoord[randomsampleindices]
137             Hwithnoise = computehomography(randomdomains.T, randomranges.
    T, over=False)
138             inlieridx = findinliers(Hwithnoise, domaincoord, rangecoord,
     delta)
139
140             numinlierindices = len(inlieridx)
141             if(numinlierindices / ntotal ¿= inlierpercentage):
142                 inlierpercentage = numinlierindices / ntotal
143                 bestinlieridx = inlieridx
144
145         if(len(bestinlieridx) ¡ M):
146             print("WARNING: Number of inlier points is less than the minimum
    requirement")
147
148         bestinliers.append(bestinlieridx)
149         H = computehomography(domaincoord[bestinlieridx].T, rangecoord[
    bestinlieridx].T, over=True)
150         if(LM):
151             hvector = H.flatten()
152             result = leastsquares(costfunction, hvector, args=(domaincoord
    [bestinlieridx], rangecoord[bestinlieridx]), method="lm")
153             hprime = result.x
154             H = hprime.reshape((3,3))
155         Hlist.append(H)
156         displayinliersoutliers(bestinliers, domaincoord, rangecoord,
    domainimage, rangeimage, "hw05/inlieroutlier/Task" + str(task) + "/
    image" + str(idx) + "image" + str(idx + 1))
157     return Hlist, bestinliers
158
159 def costfunction(h, domaincoord, rangecoord):
160     X = []
161     F = []
162     #[(x,y,1),...
163     for idx in range(len(domaincoord)):
164         xd = domaincoord[idx][0]
165         yd = domaincoord[idx][1]
166         xr = rangecoord[idx][0]
167         yr = rangecoord[idx][1]
168         X.append(xr)
169         X.append(yr)
170
171         F.append((h[0] * xd + h[1] * yd + h[2]) / (h[6] * xd + h[7] * yd + h
    [8]))
172         F.append((h[3] * xd + h[4] * yd + h[5]) / (h[6] * xd + h[7] * yd + h
    [8]))
173
174     return np.array(X) - np.array(F)
```

```
175
176 def displayinliersoutliers(inlierindices, domaincoord, rangecoord, image1
    , image2, path):
177     combinedinliers, combinedoutliers = np.hstack((image1, image2)), np.
    hstack((image1, image2))
178     height, width, channel = image1.shape
179     inlierindices = inlierindices[0].tolist()
180
181     for idx in range(domaincoord.shape[0]):
182         # Inliers
183         if(idx in inlierindices):
184             pt1 = tuple(domaincoord[idx])
185             pt2 = rangecoord[idx][0] + width, rangecoord[idx][1]
186
187             cv2.circle(combinedinliers, pt1[0:2], 3, (255, 0, 0), 1)
188             cv2.circle(combinedinliers, pt2[0:2], 3, (0, 255, 0), 1)
189             cv2.line(combinedinliers, pt1[0:2], pt2[0:2], (0, 0, 255), 1)
190
191         else:
192             # Outliers
193             pt1 = tuple(domaincoord[idx])
194             pt2 = (rangecoord[idx][0] + width, rangecoord[idx][1])
195
196             cv2.circle(combinedoutliers, pt1[0:2], 3, (255, 0, 0), 1)
197             cv2.circle(combinedoutliers, pt2[0:2], 3, (0, 255, 0), 1)
198             cv2.line(combinedoutliers, pt1[0:2], pt2[0:2], (0, 0, 255), 1)
199
200     cv2.imwrite(path + "inliers.jpg", combinedinliers)
201     cv2.imwrite(path + "outliers.jpg", combinedoutliers)
202
203 def generatepanorama(H, images, path):
204     images = [img.copy() for img in images]
205     N = len(images)
206     mid = N // 2
207
208     midH = np.eye(3)
209     for idx in range(mid, len(H)):
210         midH = np.matmul(midH, np.linalg.inv(H[idx])) #H23, H34
211         H[idx] = midH
212
213     midH = np.eye(3)
214     for idx in range(mid - 1, -1, -1):
215         midH = np.matmul(midH, H[idx]) #H12, H01
216         H[idx] = midH
217
218     H.insert(mid, np.eye(3))
219     tx = 0
220     for idx in range(mid):
221         tx += images[idx].shape[1]
222     translation = np.array([[1,0,tx], [0,1,0], [0,0,1]], dtype=float)
223
224     height, width = 0, 0
225     for idx in range(N):
226         height = max(height, images[idx].shape[0])
```

```
        width += images[idx].shape[1]

    combined = np.zeros((height, width, 3), np.uint8)
    for idx in range(N):
        Hcurr = np.matmul(translation, H[idx])
        combined = pixelmapping(combined, images[idx], Hcurr)

    cv2.imwrite(path + "panorama.jpg", combined)

def pixelmapping(combined, image, H):
    height, width,   = image.shape
    combinedheight, combinedwidth,   = combined.shape
    H = np.linalg.inv(H)

    Xarray, Yarray = np.meshgrid(np.arange(0, combinedwidth, 1), np.arange(0,
     combinedheight, 1))
    pixels = np.vstack((Xarray.ravel(), Yarray.ravel())).T
    pixels = np.hstack((pixels[:, 0:2], pixels[:, 0:1]*0+1))

    transformedpixel = np.dot(H, pixels.T)
    transformedpixel = transformedpixel / transformedpixel[2, :]
    transformedpixel = transformedpixel.T[:, 0:2].astype('int')

    validpixels, validtransformed = findvalidpixels(pixels,
   transformedpixel, width - 1, height - 1)
    for idx in range(validpixels.shape[0]):
        if((combined[validpixels[idx, 1], validpixels[idx, 0]] != 0).all()
   == False):
            combined[validpixels[idx, 1], validpixels[idx, 0]] = image[
   validtransformed[idx, 1], validtransformed[idx, 0]]

    return combined

def findvalidpixels(pixels, transformed, width, height):
    xMin = transformed[:,0] ¿= 0
    transformed = transformed[xMin, :]
    pixels = pixels[xMin, :]

    xMax = transformed[:,0] ¡= width
    transformed = transformed[xMax, :]
    pixels = pixels[xMax, :]

    yMin = transformed[:,1] ¿= 0
    transformed = transformed[yMin, :]
    pixels = pixels[yMin, :]

    yMax = transformed[:,1] ¡= height
    transformed = transformed[yMax, :]
    pixels = pixels[yMax, :]

    return pixels, transformed


if  name ==" main ":
```

```
    # Task 1
    images1 = getimages(r"hw05/Images", task=1)
    optimumH, bestinliers = ransac(images1, task=1, LM=False)
    levenmarq = "withoutLM"
    generatepanorama(optimumH, images1, path="hw05/panaroma/Task"+str(1)+"/"
    + levenmarq)

    optimumH, bestinliers = ransac(images1, task=1, LM=True)
    levenmarq = "LM"
    generatepanorama(optimumH, images1, path="hw05/panaroma/Task"+str(1)+"/"
    + levenmarq)

    # Task 2
    images2 = getimages(r"hw05/Images", task=2)
    optimumH, bestinliers = ransac(images2, task=2, LM=False)
    levenmarq = "withoutLM"
    generatepanorama(optimumH, images2, path="hw05/panaroma/Task"+str(2)+"/"
    + levenmarq)

    optimumH, bestinliers = ransac(images2, task=2, LM=True)
    levenmarq = "LM"
    generatepanorama(optimumH, images2, path="hw05/panaroma/Task"+str(2)+"/"
    + levenmarq)
```

Listing 1: The Source Code

2.3 Inputs and Outputs

Figure 1: First Set of Inputs

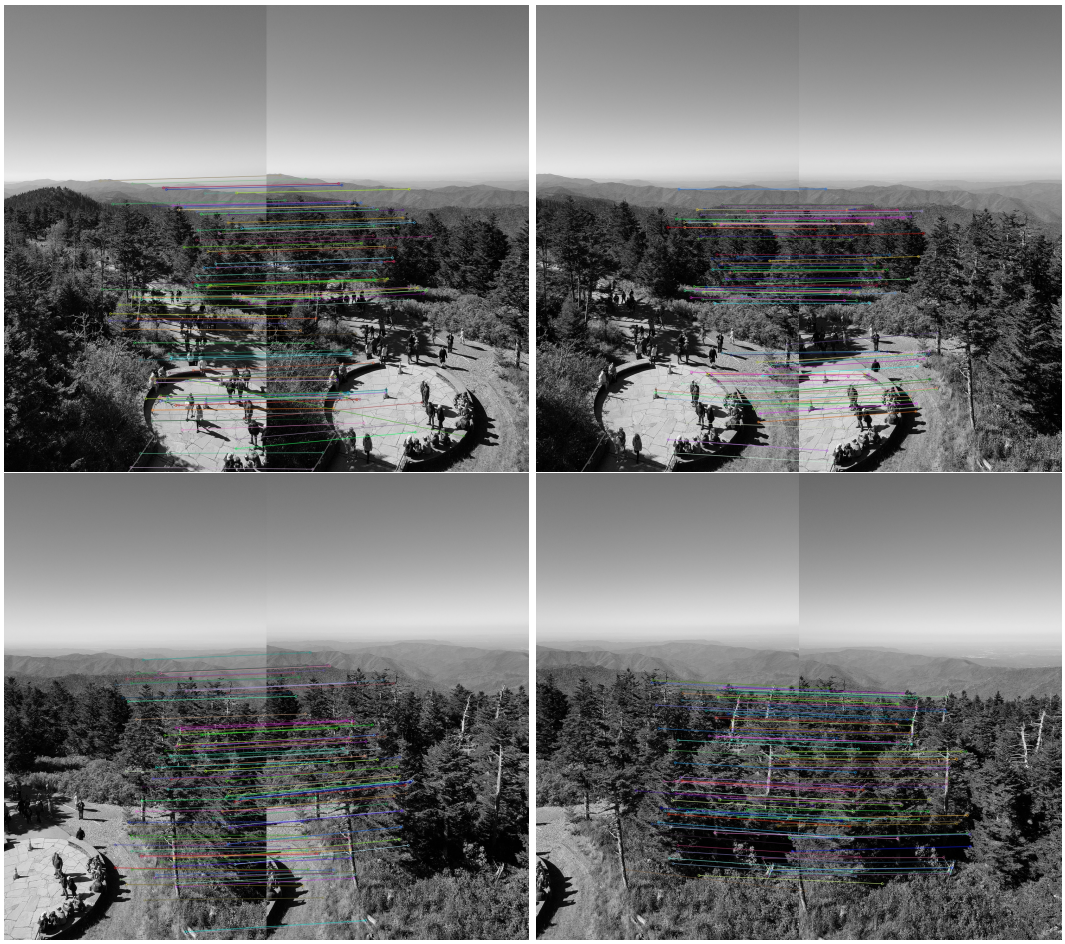

Figure 2: SIFT Correspondences for Input Set 1

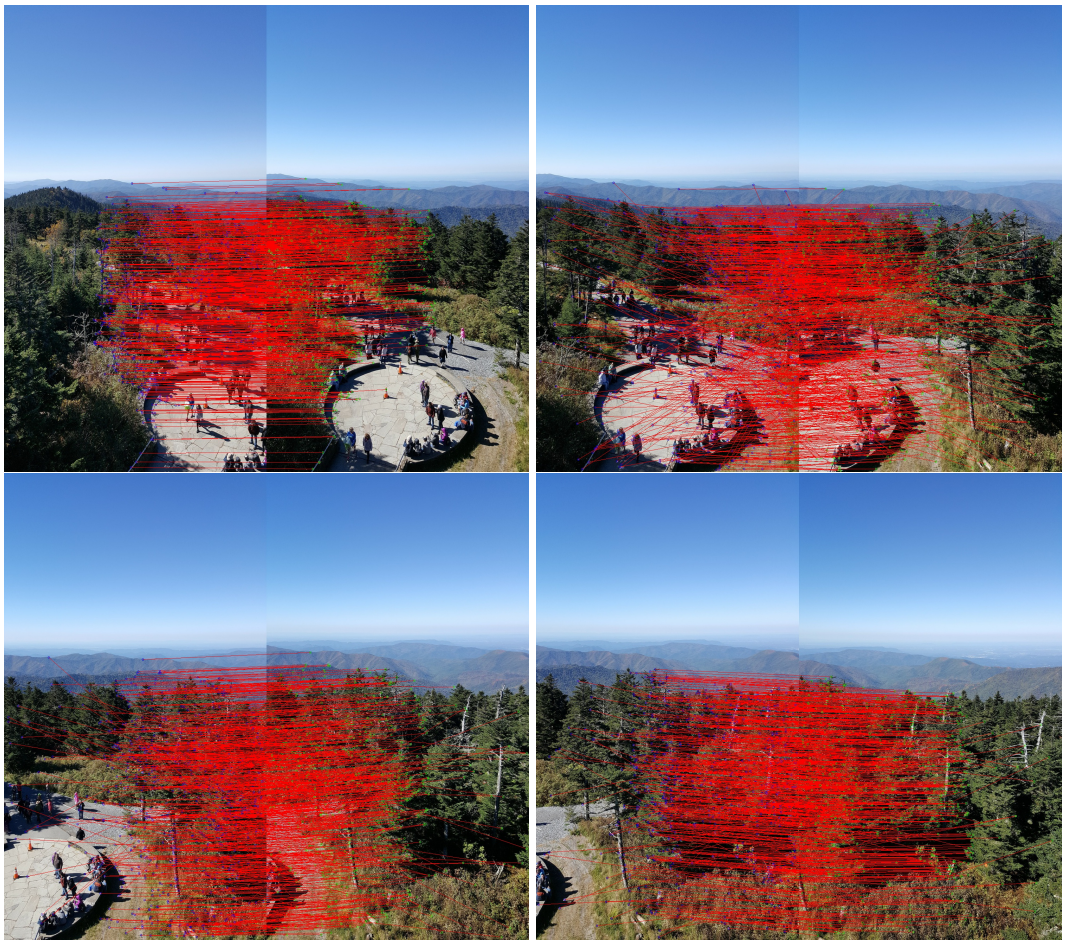

Figure 3: Inliers for Input Set 1

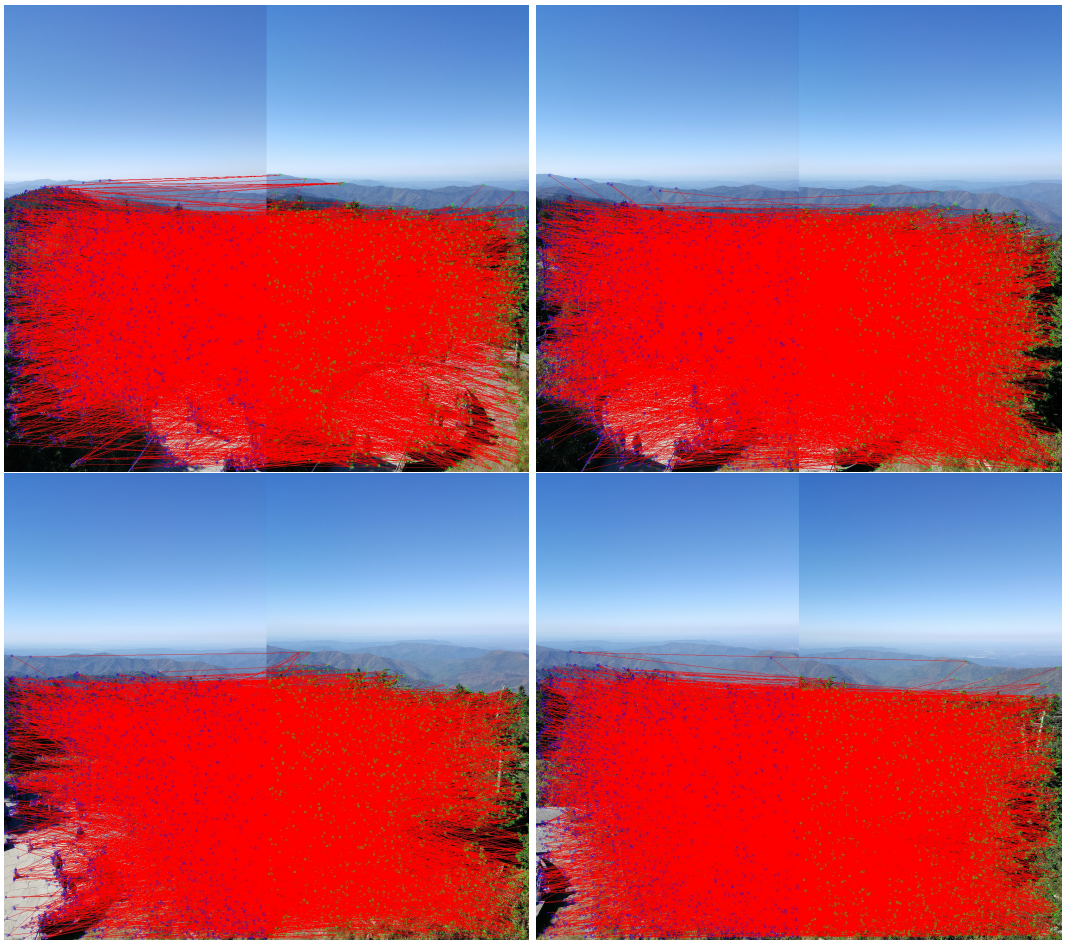

Figure 4: Outliers for Input Set 1

(a) With Levenberg-Marquadt

(b) Without Levenberg-Marquadt

Figure 5: Panoramas for Input Set 1

Figure 6: Second Set of Inputs

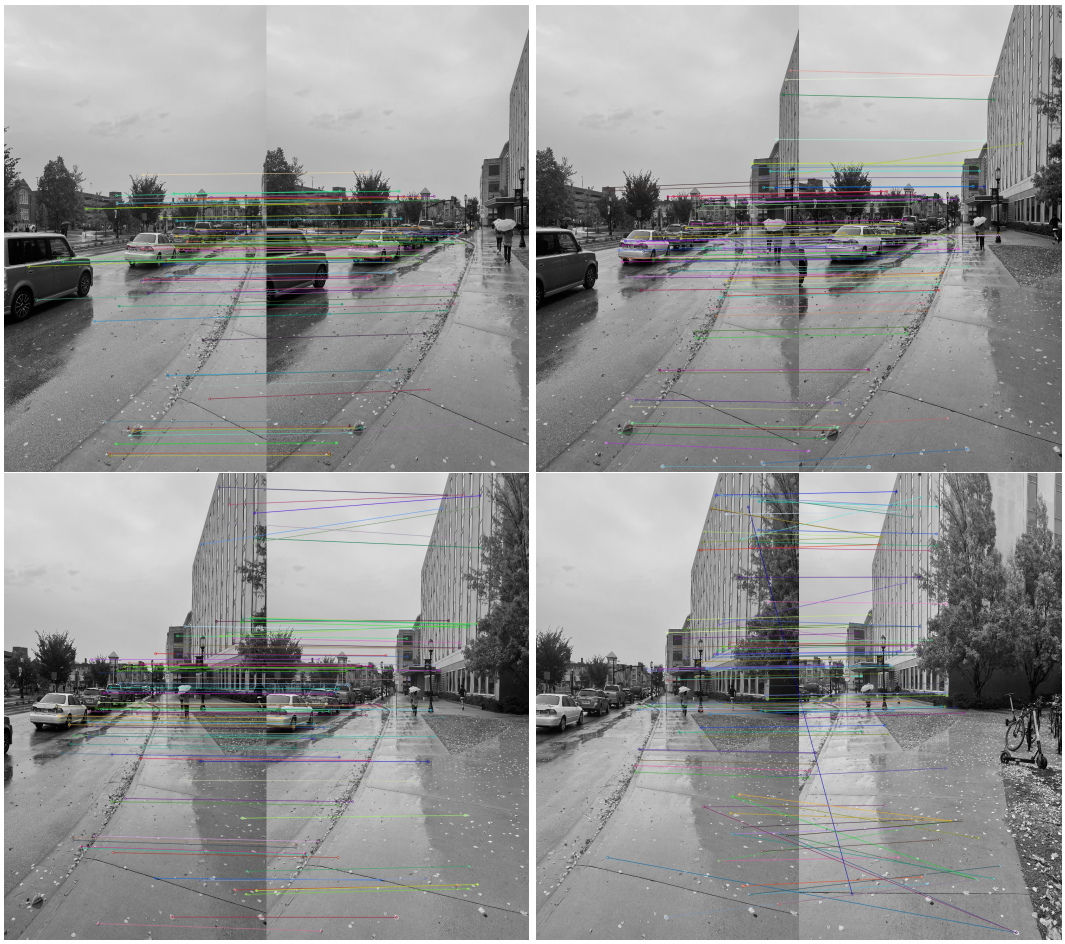

Figure 7: SIFT Correspondences for Input Set 2

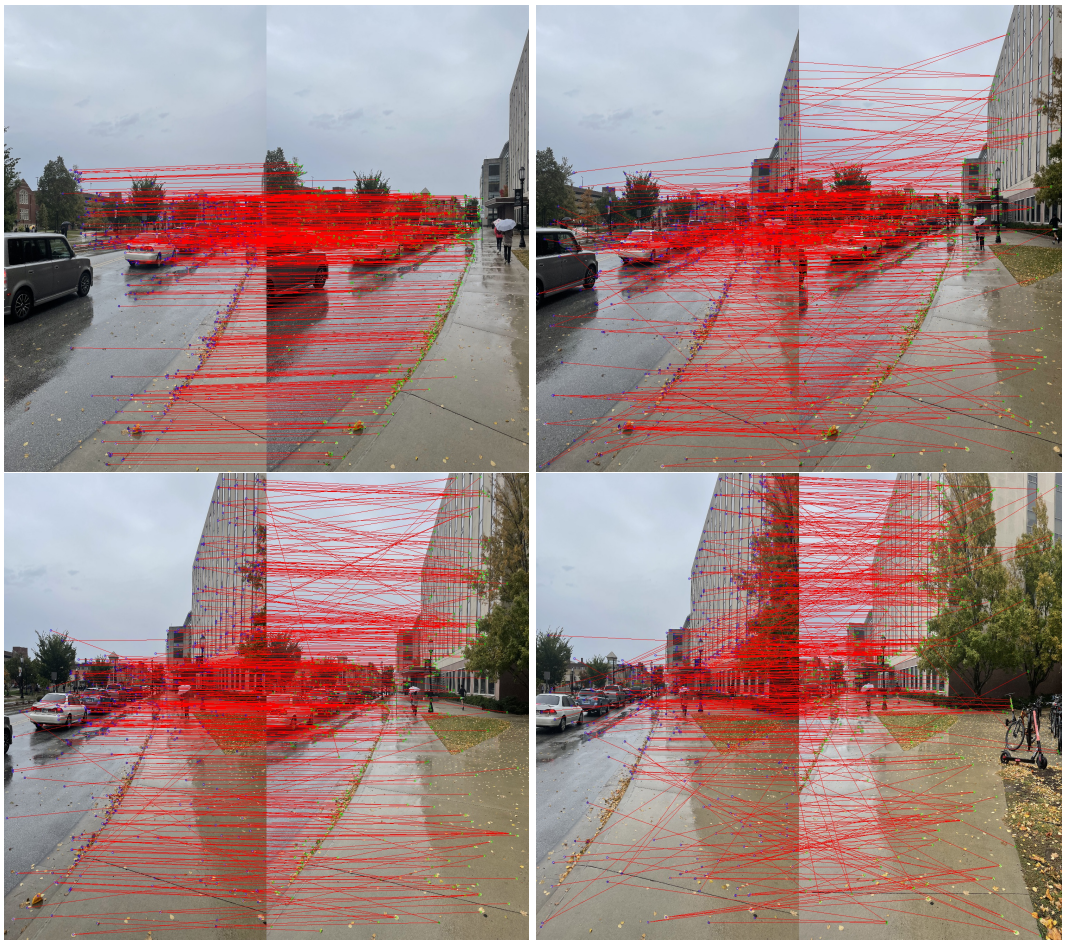

Figure 8: Inliers for Input Set 2

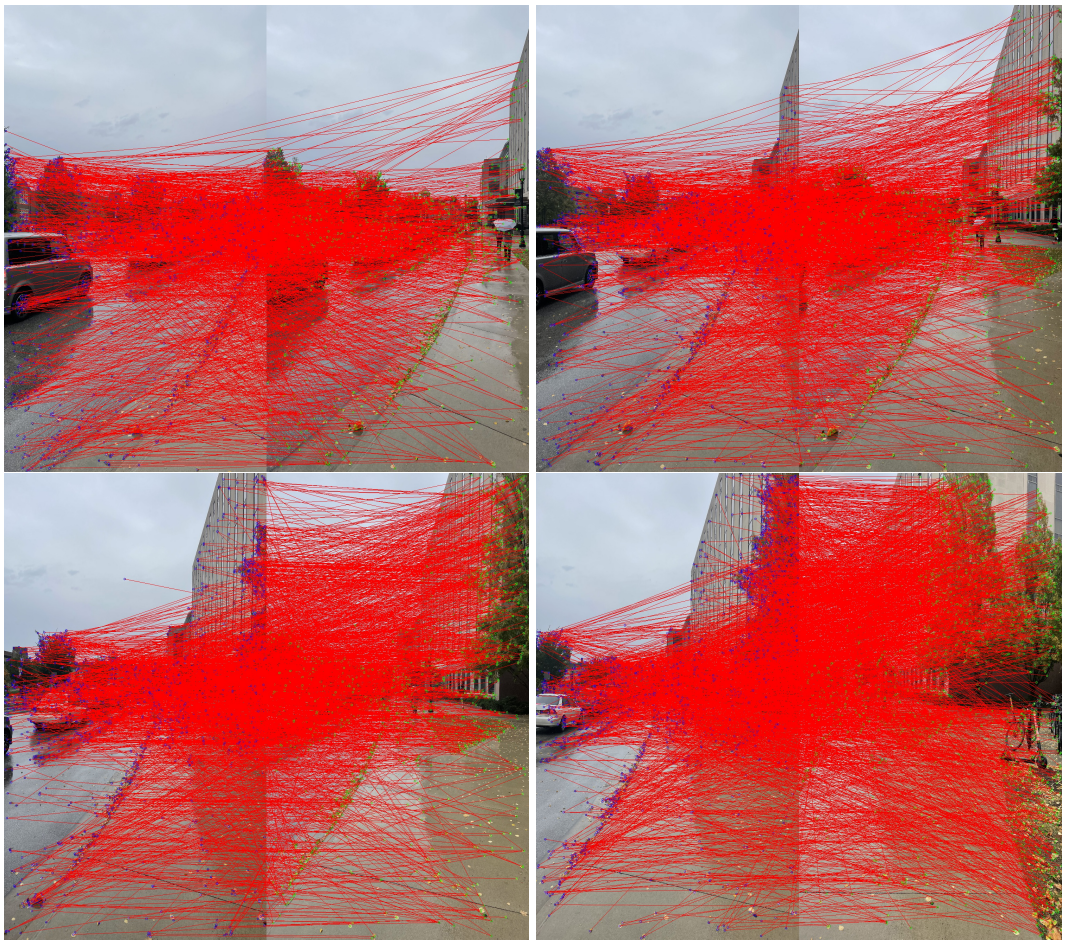

Figure 9: Outliers for Input Set 2

(a) With Levenberg-Marquadt

(b) Without Levenberg-Marquadt

Figure 10: Panoramas for Input Set 2