Purdue University

ECE 661 Computer Vision

Homework 5

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Theory Questions

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?

To differentiate between inliers an outliers, we take a probabilistic approach. From our correspondence/matching algorithm, we know that most of the points in the first image were matched accurately to their corresponding points in the second image. But some of them did not (these are outliers). But the basic understanding is that **majority** of the points were matched correctly. This is important for us to proceed with the next steps. Let us pick a sample of **n** number of matching pairs of points and calculate the homography **H**. Using this homography **H**, we can estimate the corresponding point in Image 2 given a point in Image 1. When we estimate the point, we can see either of the two cases:

1. CASE 1 : The estimated point and actual point lie very close to each other (let us say <x pixels apart).
2. CASE 2: The estimated point and the actual point lie at a distance more than x pixels apart.

Supposing we agree upon a value for **x**. Then we can decide whether point is an inlier or an outlier. In this case, the point detected in CASE 1 would be an inlier and the point detected in CASE 2 would be an outlier. The explanation for that:
If P_2 is the point in Image 2 which is mapped to P_1 in Image 1, we can justify that for the given homography **H** P_2 is said to be a correct match if:

$$P_2 = P_1 H$$

To do this, we first calculate the value and assign it a variable **X**. Where X is:

$$X = P_1 H$$

We then say P_2 is an inlier if and only if:

$$P_2 = X$$

P_2 is an outlier if:

$$P_2 \neq X$$

But in reality, rarely is the case where P_2 and X are exactly equal. Therefore we set a cutoff value (δ). The value is a measure of distance between the estimated point and the true point. Therefore, the modified logic becomes:

- Inlier if:
$$distance(P_2, X) <= \delta$$
- Outlier if:
$$distance(P_2, X) > \delta$$

We saw how, for a given homography **H** we can build a list of inlier points and outlier points. But this is based on the assumption that our **H** is the true homography that maps all the points in the Image 1 to the points in Image2. But this is not the case. There is no way of knowing (yet) whether the homography is the right hoomography to decide inliers and outliers. To solve this problem, we use a statistical method. Recall that we assume that the majority of the matches are inlier matches. Our goal is to find the set of all inlier matches.
Suppose we repeat the process of finding homography and getting list of inliers (as explained above) **N** number of times. We then introduce ϵ as the probability that a picked sample is an outlier. $(1-\epsilon)$ would be the probability that a sample is an inlier. Let us say that we pick n number of samples to compute the homography **H**, then the probability that all the picked samples (in the n number of samples) are inlier is given by:

$$(1-\epsilon)^n$$

Extending this, the probability that **at least one** sample in the set of n samples was an outlier is given by:

$$1-(1-\epsilon)^n$$

If we were to conduct the same process of picking n random samples from the set of all correspondences and calculating homographies **N** number of times, then the probability that **all** the trials involve **at least one** sample which was an outlier is given by:

$$[1-(1-\epsilon)^n]^N$$

Extending this, the probability that **at least one trial** is free of outliers is given by:

$$1-[1-(1-\epsilon)^n]^N$$

This is exactly what we want. We need that **one** trial which had **no** outlier samples. The question then becomes, how many times should we do the trials (N times). The answer is simple, if we take the probability that at least one trial is free of outliers as **p** then we know that:

$$p = 1-[1-(1-\epsilon)^n]^N$$

Therefore, we have finally arrived at:

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

In summary,

- A trial is where we pick n number of samples and estimate homography.
- Using the homography, we decide which match is a good match and which is a bad match (inlier and outlier). We do this by the using the logic we discussed about in the above paragraphs.
- We repeat the trial N number of time. We calculate N by setting the value of p as 0.99.
- After N number of trials, we pick the list which has the highest number of inliers. This is based on our initial assumption that majority of the mathces in the images are good matches (inliers).

- The homography associated with this list of inliers is the true homography that correctly maps the points in the image pair.

This is the RANSAC method of differentiating between inliers and outliers by homography estimation.

As you will see in Lecture 12, 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 reasonably fast and numerically stable at the same time.

In all three of the methods, our goal is to minimize the cost function which is given by:

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

Unlike the linear least square method of estimation, we attempt to minimise the geometric distance between the estimated homography and the true homography as opposed to minimizing the linear distance between them. To minimise the cost, we begin with a guessed value for $\vec{p}$ and we work our way to the minimum value by descending along the cost surface to the bottom point. In order to get a good and fast result, we have to make sure our first guess value for $\vec{p}$ is a good guess. A good guess would be a value which is as close to the true value as possible. Since, the least squares estimated homography of the biggest inlier set using RANSAC is a close enough value, we take the first guess as that. So $\vec{p_0}$ would be the homography estimated for the biggest inlier set. By working with the first guess, we descend down into the bottom of the cost surface. We have the following three options which dictate how fast and how effectively we reach the bottom:

1. Gradient-Descent Method (GD)
2. Gauss-Newton Method (GN)
3. Levenberg-Marquardt Method (LM)

Let us take a look at the first method.

Gradient-Descent Method (GD)

In the GD method, we begin with the initial guess value of $\vec{p_0}$ which has a cost value of $c(\vec{p_0})$ and then we take a step towards the lower cost value by calculating $\vec{p_1}$ where:

$$c(\vec{p_1}) < c(\vec{p_0})$$

The next step is given by the equation:

$$p_{k+1} = p_k - \gamma_k \nabla C|_{\vec{p}=\vec{p_k}} \tag{2}$$

Where:

- p_k is the homography estimation for the Kth step.
- γ_k is the step size controller
- ∇C is the gradient of the cost at p_k

GD, although, very effective in producing a result is a very slow process. Why is this? Take a look at equation 2. We calculate the value of the next step by first calculating the gradient at the current position. At first, the gradient is very large and hence the steps will be big. But as we near the bottom of the surface, the gradient becomes lesser and lesser in magnitude. This makes the step size smaller. Therefore, there is an direct relation between the step size and the distance from the true minimum (bottom of the surface). The closer the point, the smaller the step size. We can now see why this would slow down the GD process. The analogy of this would be to take the following case: Imagine we are at one end (A) of a line AB. We need to reach the other end B. Let us take the length of the AB as 1 unit. We are allowed to take only steps which are exactly half the value of the remaining distance. So, the first step would be 1/2 units. The second step would be 1/4 units. The third step would be 1/8 units. Once can see that, theoretically, we would never reach the end B as the distance between the current position and the end B would never become 0. We would only be able to 'slowly' approach 0. Each step smaller than the previous step. Hence, GD is painfully slow.

Gauss-Newton Method (GN)

In GD we saw that the next step would always be in the direction of the steepest descent. So, starting from p_0, we would descend to the next value p_1 which would be a straight drop down into the valley of the surface. But, what if the fastest way to the bottom is not by taking the steepest descent always? In the GN method, we leverage this fact. We explore the possibility that our best path may not be along the direction of steepest descent. In the GN mehtod, we always assume that the next step will directly take us to the bottom! This means that in our very first step, we are already very near the bottom. We can already begin to see why GN is faster than GD. If the next step is given by:

$$\vec{p_{k+1}} = \vec{p_k} + \vec{\delta_{p_k}}$$

We assume that $\vec{p_{k+1}}$ is the point at the very bottom of the valley. We can represent that in equation in the form:

$$\vec{X} \approx \vec{f}(\vec{p} + \vec{\delta_p}) = \vec{f}(\vec{p}) + J_{\vec{f}}.\vec{\delta_p} \tag{3}$$

This means that $\vec{\delta_p}$ must be a solution to the equation:

$$J_{\vec{f}}.\vec{\delta_p} = \vec{\epsilon}(\vec{p}) \tag{4}$$

Where:

$$\vec{\epsilon}(\vec{p}) = \vec{X} - \vec{f}(\vec{p}) \tag{5}$$

From Lecture 10, we know that the best solution, thus, for $\vec{\delta_p}$ would be give by the pseudoinverse representation:

$$\vec{\delta_p} = (J_{\vec{f}}^T J_{\vec{f}})^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{6}$$

This is how we reach the best $\vec{p}$ value using the GN method. The trade off here would be the heavy need to ensure that the beginning value ($\vec{p_0}$) is already a value very close to the bottom of the cost surface. We can see how GN is much faster but less reliable than GD.

Levenberg-Marquardt Method (LM)

Now, let us see how we combine the GD and GN methodologies into a single LM methodology to achieve fast **and** reliable results. In essence, we will have to come up with a system of equation(s) which makes sure that the approach:

- Behaves like GD when the solution is far from the bottom of the cost surface.
- Behaves like GN when the solution is near the minimum.

How do we do this? Let us examine the equation 6 and rewrite it as:

$$(J_{\vec{f}}^T J_{\vec{f}} + \mu I)\vec{\delta_p} = J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{7}$$

We introduce a variable μI as a damping coefficient. If the damping coefficient is high enough, the equation returns a solution which resembles a solution that would have been returned by a GD equation. If the damping equation is 0 then the solution would resemble that of GN. Just like earlier, we start with an initial $\vec{p_0}$ value and take steps towards the bottom. Where:

$$\vec{p_1} = \vec{p_0} + \vec{\delta_p} \tag{8}$$

Where:

$$\vec{\delta_p} = (J_{\vec{f}}^T J_{\vec{f}} + \mu I)^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{9}$$

The question then becomes: How do we set the damping value? The answer is that we begin with an assumed value for the damping coefficient and initial $\vec{p_0}$. We use these guess values to calculate the δ_p value by using equation 9. Then we calculate the next p value using equation 8. Next, we run a 'quality-test' to see if the damping values are acceptable. Remember that begin the LM method like a GD and end it like a GN. The test we do is by comparing the ratio of the actual change in cost to the change in cost as predicted by a particular damping coefficient value. The testing coefficient is given by:

$$\rho_{k+1}^{LM} = \frac{c(\vec{p_k}) - c(\vec{p_{k+1}})}{\vec{\delta_p}^T J_{\vec{f}}^T \vec{\epsilon}(\vec{p_k}) + \vec{\delta_p}^T \mu I \vec{\delta_p}} \tag{10}$$

$$\mu_{k+1} = \mu_k . max(\frac{1}{3}, 1 - (2\rho_{k+1}^{LM} - 1)^3) \tag{11}$$

Where we begin with μ_0 value we get from:

$$\mu_0 = \tau . max(diag(J_f^T J_f)) \tag{12}$$

Where τ is some value between 0 and 1. Based on the value obtained from equation 11 and 12, we calculate ρ_{k+1}^{LM} value. If this is negative, then we get a μ value with strong steer towards GD. If this value is positive then we get a μ value with strong steer towards GN. So this is how we combine the powers of both GD and GN into the LM method to get fast and reliable results.

Programming tasks for this homework

Objective of the homework is build an algorithm which can mosaic images and create a panorama image. For this task, we will use 5 images. The broad tasks in order to achieve the objective of image mosaicing:

- Select image pairs in the order 1-2, 2-3, 3-4, 4-5.
- Use SIFT library in OpenCV to detect interest points in the image pair.

- Use BfMatcher or Eucledian method to find correspondence between the interest pairs in the image pair.
- Employ RANSAC algorithm to filter out the inlier matches (good correspondence) from the list of matches.
- Additionally, refine the homographies to get the stitched image. We use the Levenberg-Marquardt algorithm to refine the homographies.
- Build the image mosaic to get the final panoramic image using Image 3 as the middle image.

Using SIFT to detect interest points

SIFT stands for Scale Invariant Feature Transform. In this homework assignment, we are required to only implement the openCV SIFT library to find the SIFT corner points. Therefore, we will not be writing functions to calculate and estimate the corner points. Hence, I will not be writing in detail about the SIFT algorithm here. Instead I will point out the highlights of what the SIFT library in openCV does and how we finally match the points.

- Using the values of difference of Gaussian across scale and space we select a maxima. A maxima value might be a potential keypoint in that particular scale and space.
- We eliminate low contrast keypoints and edge points from the list of potential keypoints. This is done using a 2 X 2 Hessian Matrix. What we have remaining will thus be the accurate corners in the image.
- We then find the prominent orientation of the keypoint. We do this by calculating the gradient in magnitude and orientation in the image. We use these values as weights to build a histogram. The place where the histogram peaks is the required orientation.
- We then create a 128-dimensional vector at each of the maxima points. By normalising we then get a unit vector of the same form. This vector acts as a descriptor for the keypoint or the corner.
- Lastly, for feature matching between the two images, we can use eucledian distances between the corresponding windows. We can also use the principle of K-Nearest Neighbor (KNN) where k=2. Additionally we also employ a eucledian based matching algorithm to compare and use the better results of the two.

RANSAC - Random Sample Consensus

Random Sample Consensus is a very elegant algorithm used to filter for inlier values in a given data set. The data set in this case is the list of all the matched points. Our job, using RANSAC, is to make sure there are no outlier matched pairs. An outlier matched pair is any pair which is a wrong match in the image pairs. We cannot afford to have wrong matches if we plan to stitch the images into a panoramic image. For a good image mosaic, we will need to make sure the images align in the exact way. To do this, refer to the explanation to the first theory question explained above in this document. In this section, let us see how we implement the RANSAC algorithm in practice. We

know from the same discussion that we need to conduct N number of trials to get an inlier set of suitable size. N is given by the equation:

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

Where all the notations are same as explained in the initial discussion on Theory Question 1. Let us set the following parameters:

- p = 0.99. That is 99 percent is the probability that at least one of the trials is free of outliers.
- $\epsilon = 0.2$. That is 20 percent is the probability that a chosen pair of points is an outlier.
- n = 6. That is we take 6 random pairs of matched points to compute the homograhy for each trial.
- $\delta = 3$. That is we say a point is an inlier if the distance to the estimated point is less than 3, outlier if it is greater than 3.
- M is the minimum number of inliers in a set for the set to be acceptable. M is give by:

$$M = (1-\epsilon) * Totalnumberofcorrespondences.$$

Thus, in the code, we first calculate the value of N. We then pick n ($n = 6$) random pairs of corresponding points. We then compute the homography. Using this homography, we then decide if each correspondence is an inlier or an outlier. Extending the discussion from my answer to Theory Question 1, we first calculate the value of X. Where $X = P_1 H$. We then calculate the distance between P_2 and X. Where P_1 and P_2 are correspondences.
Distance measurement
If X is at (x,y) and if P_2 is at (u,v) then distance **d** is given by:

$$d = \sqrt{(x-u)^2 + (y-v)^2}$$

We then use the following logic:

- Inlier if $d <= \delta$
- Outlier if $d > \delta$

For each trial we store the set of inliers in a dictionary along with the corresponding homography that was estimated to find decide those inliers. Once we have finished doing N trials, we check the dictionary for the set with the **largest** number of inliers. This is the final list of inliers we need to proceed to the next step of refining the homography. The homography which is associated with this set of inliers in the dictionary, is the final homography needed to be refined and then used to stitch the image.

Linear Least Squares Minimization - Effective homography estimation

Let H be the homography we need to estimate to map the points of the first image to the second image. By now, we know that:

$$X' = HX$$

Where X' is a point on Image 2 and X is the corresponding point on Image 1. Extending the same we get the relation:

$$(X')\mathbf{X}(HX) = 0$$

If X' and HX are the same, then their cross product will be equal to 0. If H is 3X3 matrix, we can alternatively represent it as:

$$H = \begin{bmatrix} h_1^T \\ h_2^T \\ h_3^T \end{bmatrix}$$

Where h_1, h_2 and h_3 are the column vectors of the rows 1,2 and 3 respectively. Let us represent the point X' by its homogeneous coordinates (x',y',w') . Using this, we now have:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} X \begin{bmatrix} (h_1^T)X \\ (h_2^T)X \\ (h_3^T)X \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The above equation yields three linear equation that we will have to solve to get the values for H matrix.

$$\vec{0}^T h' - w' X^T h^2 + y' X^T h^3 = 0$$
$$w' X^T h' + \vec{0}^T h^2 - x' X^T h^3 = 0$$
$$-y' X^T h' + x' X^T h^2 - \vec{o}^T X^T h^3 = 0$$

The above two equation are not linearly independent equations and hence we can solve for the three values h_1, h_2 and h_3 by using only the first two equations.If we have n pairs of points, then we will have 2n such pairs of equations. Let us define a matrix A which then gives us the relation:

$$A\vec{h} = 0$$

Where is a 2n X 9 matrix made by stacking up the coefficients of the first two equations for a given pair of points. By computing a SVD for this matrix. The last column of the vector V in the decomposition that has the lowest eigenvalue is the solution we need.

Homography refinement using LM method

The theory behind the Levenberg-Marquardt method has been explained in detailed in my answer to the second theory question which has been given in the same document. We use this refined homography to project the 5 images onto a common plane.

Stitching the images together

We use the concept of product homography to find the homographies with respoect to the middle image which in this case is the third image. So if H_{ij} is the homography from image i to image j then we have the following product homographies that we will be using to stitch the images:

- $H_{13} = H_{23}$ X H_{12}
- $H_{43} = H_{34}^{-1}$
- $H_{53} = H_{34}^{-1}$ X H_{54}^{-1}

Input Images

Figure 1: **Image 1**

Figure 2: **Image 2**

Figure 3: **Image 3**

Figure 4: **Image 4**

Figure 5: **Image 5**

Result

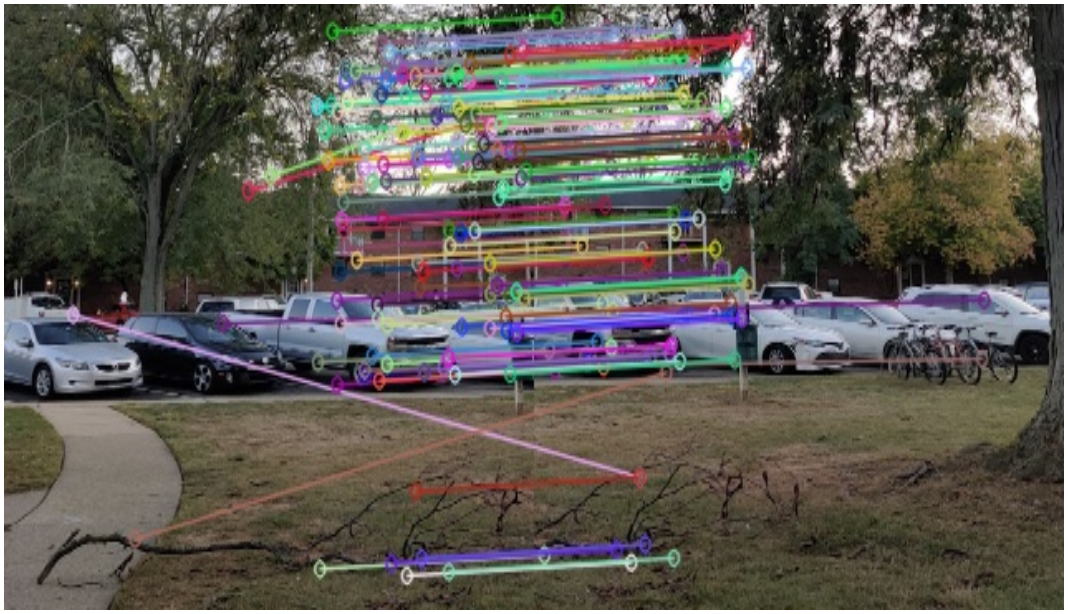

Figure 6: **First pair - All matches**

Figure 7: **Second pair - All matches**

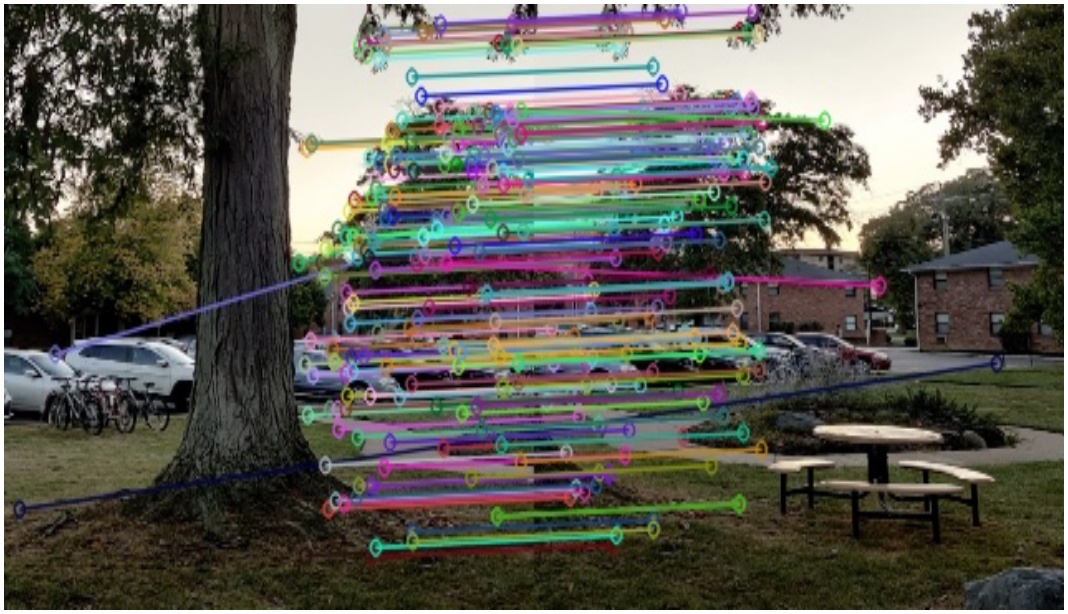

Figure 8: **Third pair - All matches**

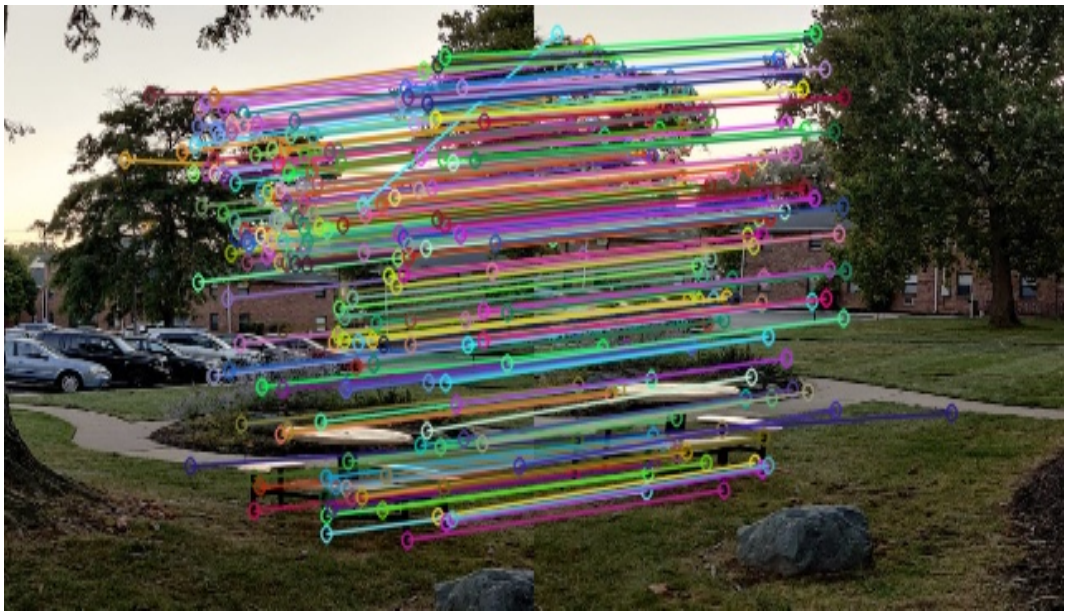

Figure 9: **Fourth pair - All matches**

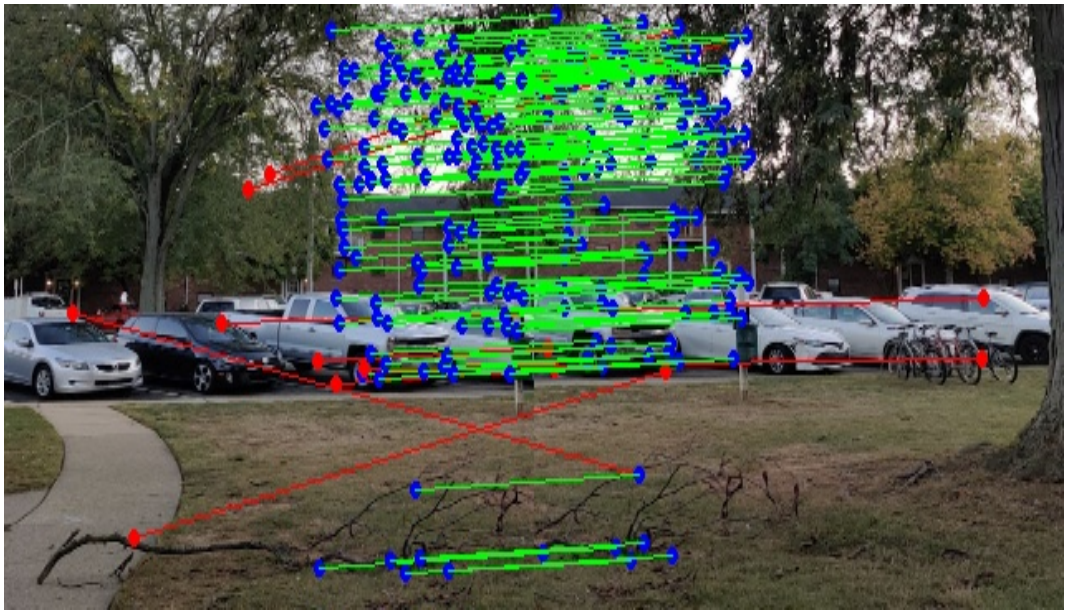

Figure 10: **First pair - Inliers and outliers**

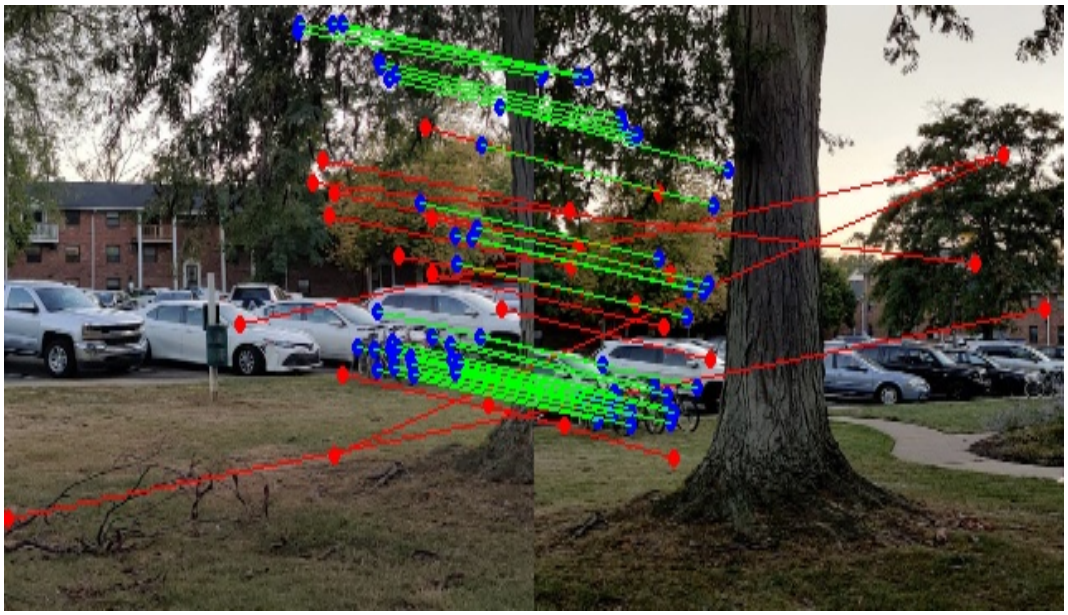

Figure 11: **Second pair - Inliers and outliers**

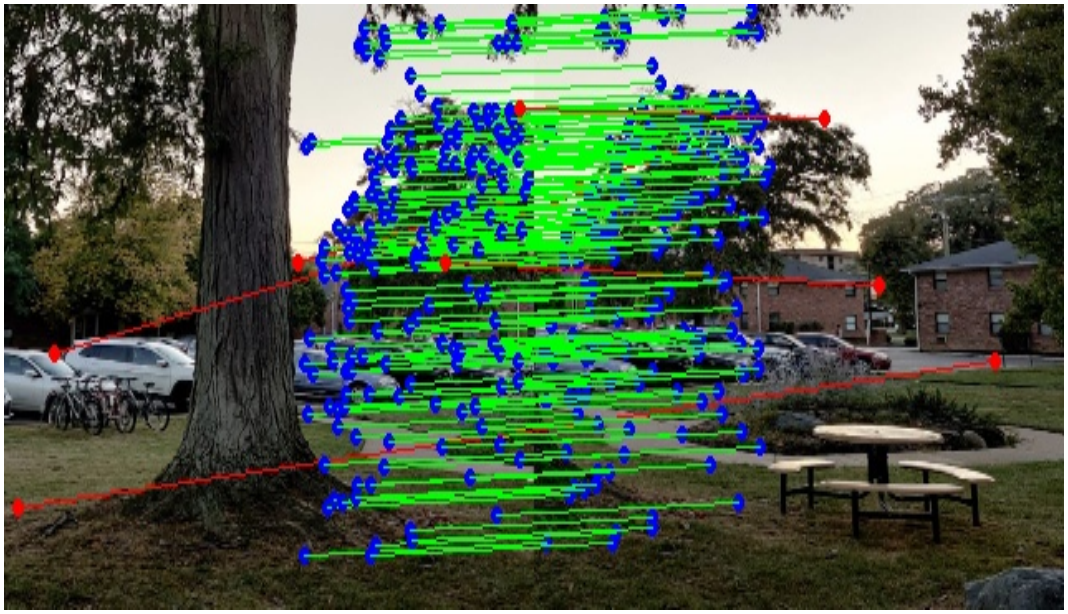

Figure 12: **Third pair - Inliers and outliers**

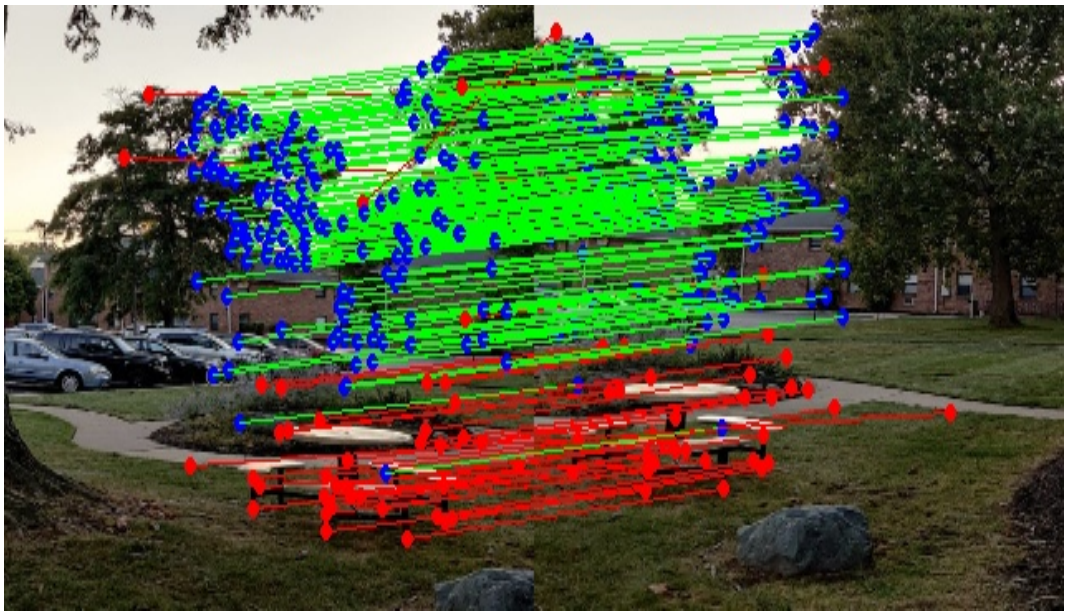

Figure 13: **Fourth pair - Inliers and outliers**

Figure 14: **Panorama without LM refinement**

Figure 15: **Panorama with LM refinement**

Source code

Please note that for two of the functions in my code (draw inliers outliers and calculate lls homography) I referred the implementation from this link:
Link
After referring the code from that link, I implemented it on my own in my code which is given below

```
1
"""
Computer Vision - Purdue University - Homework 5

Author : Arjun Kramadhati Gopi, MS-Computer & Information
   Technology, Purdue University.
Date: Oct 5, 2020


[TO RUN CODE]: python3 imagemosaic.py
Output:
    [jpg]: Panoramic image stitched from 5 input images.
"""

import cv2 as cv
import math
import numpy as np
import random
import time
from scipy import signal as sg
import tqdm
import copy
import threading
from scipy.optimize import least_squares
from scipy.optimize import minimize

class Panorama:
    def __init__(self, image_addresses, scale, kvalue=0.04):
```

```
27          self.image_addresses = image_addresses
28          self.scale = scale
29          self.originalImages = []
30          self.grayscaleImages = []
31          self.filters = {}
32          self.cornerpointdict = {}
33          self.slidingwindowdict = {}
34          self.correspondence = {}
35          self.homographydict = {}
36          self.kvalue = kvalue
37          for i in range(len(self.image_addresses)):
38              self.originalImages.append(cv.resize(cv.imread(self.
                image_addresses[i]), (320, 240)))
39              self.grayscaleImages.append(cv.resize(cv.cvtColor(cv.
                imread(self.image_addresses[i]), cv.COLOR_BGR2GRAY
                ), (320, 240)))
40          self.siftobject = cv.SIFT_create()
41
42      def weightedPixelValue(self, rangecoordinates, objectQueue):
43          """
44          [This function calculates the weighted pixel value at the
                given coordinate in the target image]
45
46          Args:
47              rangecoordinates ([list]): [This is the coordinate of
                    the pixel in the target image]
48              objectQueue ([int]): [This is the index number of the
                    list which has the coordinates of the roI for the
                    Object picture]
49
50          Returns:
51              [list]: [Weighted pixel value - RGB value]
52          """
53
54          pointOne = (int(np.floor(rangecoordinates[1])), int(np.
                floor(rangecoordinates[0])))
55          pointTwo = (int(np.floor(rangecoordinates[1])), int(np.
                ceil(rangecoordinates[0])))
56          pointThree = (int(np.ceil(rangecoordinates[1])), int(np.
                ceil(rangecoordinates[0])))
57          pointFour = (int(np.ceil(rangecoordinates[1])), int(np.
                floor(rangecoordinates[0])))
58
59          pixelValueAtOne = self.originalImages[objectQueue][
                pointOne[0]][pointOne[1]]
60          pixelValueAtTwo = self.originalImages[objectQueue][
                pointTwo[0]][pointTwo[1]]
61          pixelValueAtThree = self.originalImages[objectQueue][
                pointThree[0]][pointThree[1]]
62          pixelValueAtFour = self.originalImages[objectQueue][
                pointFour[0]][pointFour[1]]
63
64          weightAtOne = 1 / np.linalg.norm(pixelValueAtOne -
```

```
            rangecoordinates)
65          weightAtTwo = 1 / np.linalg.norm(pixelValueAtTwo -
               rangecoordinates)
66          weightAtThree = 1 / np.linalg.norm(pixelValueAtThree -
               rangecoordinates)
67          weightAtFour = 1 / np.linalg.norm(pixelValueAtFour -
               rangecoordinates)
68
69          return ((weightAtOne * pixelValueAtOne) + (weightAtTwo *
               pixelValueAtTwo) + (
70                      weightAtThree * pixelValueAtThree) + (
                           weightAtFour * pixelValueAtFour)) / (
71                              weightAtFour + weightAtThree +
                                   weightAtTwo + weightAtOne)
72
73      def get_panorama_done(self):
74          """
75          This function calls the necessary functions to get the
               final panorama image output
76          :return: None
77          """
78          self.calculate_ransac_parameters()
79          for i in range(0,4,1):
80              self.perform_ransac((str(i),str(i+1),str(i)+str(i+1))
                    )
81          self.get_product_homography()
82          self.get_panorama_image(('02','12','22','32','42'))
83
84      def get_panorama_image(self,tags):
85          """
86          This function first computes the size of the final
               panorama image. This it stitches the 5 images into
87          a panoramic image
88          :param tags: Tags for key values where the respective
               homography matrices are stored in the dictionary
89          :return: Stores the final image
90          """
91          cornerlist = []
92          for i in range(len(tags)):
93              endpoints = np.zeros((3,4))
94              endpoints[:,0] = [0,0,1]
95              endpoints[:,1] = [0, self.originalImages[i].shape[1],
                     1]
96              endpoints[:,2] = [self.originalImages[i].shape[0],0,
                    1]
97              endpoints[:,3] = [self.originalImages[i].shape[0],
                    self.originalImages[i].shape[1], 1]
98              corners = np.matmul(self.homographydict[tags[i]],
                    endpoints)
99              for i in range(corners.shape[1]):
100                 corners[:, i] = corners[:, i] / corners[-1, i]
101             cornerlist.append(corners[0:2, :])
102         minvalue =np.amin(np.amin(cornerlist,2),0)
```

```
103         maxvalue = np.amax(np.amax(cornerlist, 2), 0)
104         imagesize = maxvalue - minvalue
105         pan_img = np.zeros((int(imagesize[1]), int(imagesize[0]),
                3))
106         for i in range(len(tags)):
107             print(i)
108             H = np.linalg.inv(self.homographydict[tags[i]])
109             for column in range(0,pan_img.shape[0]):
110                 for row in range(0,pan_img.shape[1]):
111                     print(str(column)+ " out of " + str(pan_img.
                            shape[0]))
112                     sourcecoord = np.array([row+minvalue[0],
                            column+minvalue[1], 1])
113                     destcoord = np.array(np.matmul(H,sourcecoord)
                            )
114                     destcoord = destcoord/destcoord[-1]
115                     if (destcoord[0]>0 and destcoord[1]>0 and
                            destcoord[0]<self.originalImages[i].shape
                            [1]-1 and destcoord[1]<self.originalImages
                            [i].shape[0]-1):
116                         pan_img[column][row] = self.
                                weightedPixelValue(destcoord,i)
117
118         cv.imwrite("panorama.jpg",pan_img)
119
120     def get_product_homography(self):
121         """
122         Calculates the correct homography needed to get the final
                image. Since, we are taking the
123         3rd image as the center image, we need all the
                homographies with respect to the 3rd image.
124         :return: None. Stores the homographies in a dictionary
125         """
126         H02 = np.matmul(self.homographydict['01'], self.
                homographydict['12'])
127         H02 = H02/H02[-1,-1]
128         self.homographydict['02']=H02
129         H12 = self.homographydict['12']/self.homographydict
                ['12'][-1,-1]
130         self.homographydict['12']=H12
131         H32 = np.linalg.inv(self.homographydict['23'])
132         H32 = H32/H32[-1,-1]
133         self.homographydict['32']=H32
134         H42=np.linalg.inv(np.matmul(self.homographydict['23'],
                self.homographydict['34']))
135         H42=H42/H42[-1,-1]
136         self.homographydict['42']=H42
137         H22 = np.identity(3)
138         self.homographydict['22']=H22
139
140     def calculate_ransac_parameters(self, pvalue=0.999,
            epsilonvalue=0.40,samplesize=6 ):
141         """
```

```
142         Calculates the ransac parameters.
143         :param pvalue: p value is taken as 99.9%
144         :param epsilonvalue: We assume 40% of the correspondences
                are outliers
145         :param samplesize: we take 6 random samples to calculate
               homography
146         :return: None.
147         """
148         self.ransactrials = int((math.log(1-pvalue)/math.log
               (1-(1-epsilonvalue)**samplesize)))
149         # self.ransaccutoffsize = int(math.ceil((1-epsilonvalue)*
               correspondencedatasize))
150
151     def refine_homography_objective_function(self, H, sourcpoints
           , destinationpoints):
152         """
153         Objective function for the scipy optimise least squares
               function.
154         :param H: Homography
155         :param sourcpoints: The points in the correspondences
               which are in the source image
156         :param destinationpoints: The points in the
               correspondences which are in the destination image
157         :return: error between the predicted and the actual point
                in the destination image
158         """
159         H = H.reshape(3, 3)
160         sourcpoints = np.concatenate((sourcpoints, np.ones((
               sourcpoints.shape[0], 1)), np.float)), axis=1)
161         predictedpoints = np.matmul(H, sourcpoints.T).T
162         predictedpoints = predictedpoints // predictedpoints[:,
               2].reshape(-1,1)
163         error = (predictedpoints[:, :2] - destinationpoints) ** 2
164         error = np.sqrt(np.sum(error, axis=1))
165
166         return error
167
168     def refine_homography(self, H, sourcepoints,
           destinationpoints):
169         """
170         Refines the homography using the scipy library
171         :param H: Homography
172         :param sourcpoints: The points in the correspondences
               which are in the source image
173         :param destinationpoints: The points in the
               correspondences which are in the destination image
174         :return: Refined homography matrix in 3X3 shape
175         """
176         refinedH = least_squares(self.
               refine_homography_objective_function, np.squeeze(H.
               reshape(-1, 1)), method='lm',
177                                                 args=(sourcepoints
                                                        ,
```

```
                                                    destinationpoints
                                                    ))
178         refinedH = refinedH.x.reshape(3, 3)
179         return refinedH
180
181     def calculate_lls_homography(self, image1points, image2points
            ):
182         """
183         Function to calculate the homography using linear least
                squares mehtod
184         :param image1points: The points in the correspondences
                which are in the source image
185         :param image2points: The points in the correspondences
                which are in the destination image
186         :return: Homography matrix in 3X3 shape
187         """
188         H = np.zeros((3, 3))
189         # Setup the A Matrix
190         A = np.zeros((len(image1points) * 2, 9))
191         for i in range(len(image1points)):
192             A[i * 2] = [0, 0, 0, -image1points[i, 0], -
                    image1points[i, 1], -1, image2points[i, 1] *
                    image1points[i, 0],
193                         image2points[i, 1] * image1points[i, 1],
                            image2points[i, 1]]
194             A[i * 2 + 1] = [image1points[i, 0], image1points[i,
                    1], 1, 0, 0, 0, -image2points[i, 0] * image1points
                    [i, 0],
195                             -image2points[i, 0] * image1points[i,
                                1], -image2points[i, 0]]
196
197         U, D, V = np.linalg.svd(A)
198         V_T = np.transpose(V)
199         H_elements = V_T[:, -1]
200         H[0] = H_elements[0:3] / H_elements[-1]
201         H[1] = H_elements[3:6] / H_elements[-1]
202         H[2] = H_elements[6:9] / H_elements[-1]
203
204         return H
205
206
207     def perform_ransac(self,tags, samplesize=6, cutoff=3, refine
            =True):
208         """
209         Function to perform RANSAC to filter out inliers and
                outliers in the correspondences.
210         :param tags: String values for the keys in the
                dictionaries being used to retrieve relevant data
211         :param samplesize: 6 samples per trial
212         :param cutoff: cut off value to decide inlier vs outlier
213         :param refine: True if we need to refine homography,
                False if we do not need refinement
214         :return: We call the draw function to draw the inliers
```

```
            and the outliers.
        """
        correspondence = self.correspondence[tags[2]]
        image1points = np.zeros((len(correspondence), 2))
        image2points = np.zeros((len(correspondence), 2))
        image1points = correspondence[:, 0:2]
        image2points = correspondence[:, 2:]
        count = 0
        listofinliersfinal =[]
        listofoutliersfinal = []
        homographyfinal =np.zeros((3,3))

        for iteration in range(self.ransactrials):
            print(str(iteration) + " of " + str(self.ransactrials
                ))
            print(len(image1points))
            ip_index = np.random.randint(0, len(image1points),
                samplesize)
            image1sample = image1points[ip_index, :]
            image2sample = image2points[ip_index, :]
            H = self.calculate_lls_homography(image1sample,
                image2sample)
            dest_pts_estimate = np.zeros((image2points.shape),
                dtype='int')
            for index in range(len(image1points)):
                dest_pts_nonNorm = np.matmul(H, ([image1points[
                    index, 0], image1points[index, 1], 1]))
                dest_pts_estimate[index, 0] = dest_pts_nonNorm[0]
                    / dest_pts_nonNorm[-1]
                dest_pts_estimate[index, 1] = dest_pts_nonNorm[1]
                    / dest_pts_nonNorm[-1]

            estimationerror = dest_pts_estimate - image2points
            errorsqaure = np.square(estimationerror)
            dist = np.sqrt(errorsqaure[:, 0] + errorsqaure[:, 1])
            validpointidx = np.where(dist <= cutoff)
            invalidpointidx = np.where(dist > cutoff)
            innlierlist=[]
            outlierlist =[]
            for i,element in enumerate(dist):
                if element <=cutoff:
                    innlierlist.append([image1points[i][1],
                        image1points[i][0],dest_pts_estimate[i
                        ][1],dest_pts_estimate[i][0] ])
                else:
                    outlierlist.append([image1points[i][0],
                        image1points[i][1], image2points[i][0],
                        image2points[i][1]])

            Inliers = [1 for val in dist if (val < 3)]
            if len(Inliers) > count:
                count = len(Inliers)
                listofinliersfinal =innlierlist
```

```
256                 listofoutliersfinal =outlierlist
257                 homographyfinal = H
258
259         if refine == True:
260             print("Refining...")
261             self.homographydict[tags[2]] = self.refine_homography
                    (homographyfinal, image1points, image2points)
262         else:
263             self.homographydict[tags[2]]=homographyfinal
264         print(len(listofinliersfinal))
265         print(len(listofoutliersfinal))
266         self.draw_inliers_outliers(tags, correspondence,
                homographyfinal, 3)
267
268     def draw_inliers_outliers(self, tags, correspondences,
            homography, cutoffvalue):
269         """
270         We use this function to draw the inliers and the outliers
                on the image.
271         :param tags: Values for the keys in the relevant
                dictionary
272         :param correspondences: matched points
273         :param homography: H matrix
274         :param cutoffvalue: decision value to decide inliers and
                outliers
275         :return: Writes the image with inliers and outliers
276         """
277         firstimage = self.originalImages[int(tags[0])]
278         secondimage = self.originalImages[int(tags[1])]
279         nrows = max(firstimage.shape[0], secondimage.shape[0])
280         ncol = firstimage.shape[1] + secondimage.shape[1]
281         resultimage = np.zeros((nrows, ncol, 3))
282         resultimage[:firstimage.shape[0], :firstimage.shape[1]] =
                firstimage
283         resultimage[:secondimage.shape[0], firstimage.shape[1]:
                firstimage.shape[1] + secondimage.shape[1]] =
                secondimage
284         image1points = correspondences[:, 0:2]
285         image2points = correspondences[:, 2:]
286         inliersimage1 = []
287         inliersimage2 = []
288         for src_pt in range(len(image1points)):
289             estimate = np.matmul(homography, [image1points[src_pt
                    , 0], image1points[src_pt, 1], 1])
290             estimate = estimate / estimate[-1]
291             diff = estimate[0:2] - image2points[src_pt, :]
292             errorinestimation = np.sqrt(np.sum(diff ** 2))
293             if errorinestimation < cutoffvalue:
294                 inliersimage1.append(image1points[src_pt, :])
295                 inliersimage2.append(image2points[src_pt, :])
296                 cv.circle(resultimage, (int(image1points[src_pt,
                        0]), int(image1points[src_pt, 1])), 2, (255,
                        0, 0), 2)
```

```
297                 cv.circle(resultimage, (firstimage.shape[1] + int
                        (image2points[src_pt, 0]), int(image2points[
                        src_pt, 1])), 2,
298                           (255, 0, 0), 2)
299                 cv.line(resultimage, (int(image1points[src_pt,
                        0]), int(image1points[src_pt, 1])),
300                         (firstimage.shape[1] + int(image2points[
                            src_pt, 0]), int(image2points[src_pt,
                             1])), (0, 255, 0))
301             else:
302                 cv.circle(resultimage, (int(image1points[src_pt,
                        0]), int(image1points[src_pt, 1])), 2, (0, 0,
                        255), 2)
303                 cv.circle(resultimage, (firstimage.shape[1] + int
                        (image2points[src_pt, 0]), int(image2points[
                        src_pt, 1])), 2,
304                           (0, 0, 255), 2)
305                 cv.line(resultimage, (int(image1points[src_pt,
                        0]), int(image1points[src_pt, 1])),
306                         (firstimage.shape[1] + int(image2points[
                            src_pt, 0]), int(image2points[src_pt,
                             1])), (0, 0, 255))
307
308         cv.imwrite("results/"+str(tags[2])+"inlieroutlier.jpg",
                resultimage)
309
310     def draw_correspondence_inlier_outlier(self, inlierlist,
            outlierlist):
311         """
312         This function draws the correspondence between the corner
                points in the pair of images. We denote each
313         corner point by a small circle around it. The
                correspondence is denoted by a line connecting the two
                points.
314         Before drawing the points, we first filter the
                correspondences based on a cutoff value so that we
                retain only
315         fairly accurate matches and not completely-off matches.
316         :param tags: Values to access and store values by key in
                the dictionaries
317         :param cutoffvalue: Value used to filter the list of
                matched corner points
318         :param style: Either filter values above the cutoff value
                or filter the values below it.
319         :return: Returns the resultant stitched image with the
                denoted correspondence lines.
320         """
321         resultImage = np.hstack((self.originalImages[0], self.
                originalImages[1]))
322         for element in inlierlist:
323             print(element)
324             p1x = int(element[1])
325             p1y = int(element[0])
```

```
326             p2x = int(element[1]) + 640
327             p2y = int(element[0])
328             cv.line(resultImage, (p1x,p1y), (p2x,p2y), [0, 255,
                    0], 1 )
329             # cv.circle(resultImage,(columnvalueone, rowvalueone)
                    , 2, [0, 0, 0], 2)
330             # cv.circle(resultImage, (columnvaluetwo, rowvaluetwo
                    ), 2, [0, 0, 0], 2)
331         cv.imwrite("sdfdsfsdf.jpg",resultImage)
332
333     def update_dict_values(self,tags):
334         """
335         Convert the matched type variables into the form that we
                need
336         :param tags: Values for the keys in the dictionary
337         :return: Stores the values in a dictionary
338         """
339         tempdict = dict()
340         ip1=[]
341         ip2=[]
342         matchedpoints = self.correspondence[tags[2]]
343         (keypoint1, descriptor1) = self.cornerpointdict[tags[0]]
344         (keypoint2, descriptor2) = self.cornerpointdict[tags[1]]
345         for matchedpoint in matchedpoints:
346             imageoneindex = matchedpoint[0].queryIdx
347             imagetwoindex = matchedpoint[0].trainIdx
348             (x1, y1) = keypoint1[imageoneindex].pt
349             (x2, y2) = keypoint2[imagetwoindex].pt
350             # tempdict[(x1,y1)]=(x2,y2)
351             ip1.append((x1,y1))
352             ip2.append((x2,y2))
353         ip1=np.array(ip1)
354         ip2=np.array(ip2)
355         x = np.concatenate((ip1,ip2),axis=1)
356         self.correspondence[tags[2]] = x
357
358
359     def draw_correspondence(self, tags, cutoffvalue, style):
360         """
361         This function draws the correspondence between the corner
                points in the pair of images. We denote each
362         corner point by a small circle around it. The
                correspondence is denoted by a line connecting the two
                points.
363         Before drawing the points, we first filter the
                correspondences based on a cutoff value so that we
                retain only
364         fairly accurate matches and not completely-off matches.
365         :param tags: Values to access and store values by key in
                the dictionaries
366         :param cutoffvalue: Value used to filter the list of
                matched corner points
367         :param style: Either filter values above the cutoff value
```

```
            or filter the values below it.
368         :return: Returns the resultant stitched image with the
               denoted correspondence lines.
369         """
370         copydict = copy.deepcopy(self.correspondence[tags[0]])
371         print(copydict)
372         for (key,value) in self.correspondence[tags[0]].items():
373             if style == 'greaterthan':
374                 if value[1]> cutoffvalue:
375                     copydict.pop(key)
376             elif style == 'lesserthan':
377                 if value[1]< cutoffvalue:
378                     copydict.pop(key)
379         resultImage = np.hstack((self.originalImages[0], self.
               originalImages[1]))
380         horizontaloffset = 640
381         print(copydict)
382         for (key,value) in copydict.items():
383             # print((key,value))
384             columnvalueone = key[1]
385             rowvalueone = key[0]
386             columnvaluetwo = value[0][1] + horizontaloffset
387             rowvaluetwo = value[0][0]
388             cv.line(resultImage, (columnvalueone,rowvalueone), (
                   columnvaluetwo,rowvaluetwo), [0, 255, 0], 1 )
389             cv.circle(resultImage,(columnvalueone, rowvalueone),
                   2, [0, 0, 0], 2)
390             cv.circle(resultImage, (columnvaluetwo, rowvaluetwo),
                    2, [0, 0, 0], 2)
391         return resultImage
392
393     def sift_corner_detect(self, queueImage, tag):
394         """
395         This function detected and computes the sift keypoints
               and the descriptors. We use the generated keypoints
396         to draw them on the picture. These are the detected
               corners.
397         :param queueImage: Index of the location at which the
               image under consideration is stored in the list
398         :param tag: Values to access and store values by key in
               the dictionaries
399         :return: None. Stores the image.
400         """
401         keypoint, descriptor = self.siftobject.detectAndCompute(
               self.grayscaleImages[queueImage], None)
402         self.cornerpointdict[tag] = (keypoint, descriptor)
403
404     def sift_correpondence(self, queueImages, tags, method):
405         """
406         This function estimates the correspondences between the
               sift corners detected in the pair of images.
407         We have two options to estimate this: 1) Using BFMatcher
               function of OpenCV to get K-Nearest
```

```
408             Neighbors or 2) Use a custom built
409             eucledian distance based estimator
410             :param queueImages: Index of the location at which the
                    image under consideration is stored in the list
411             :param tags: Values to access and store values by key in
                    the dictionaries
412             :param method: Use BFMatcher or custom built eucledian
                    matcher.
413             :return: None. Stores the matched keypoints in a global
                    dictionary self.correspondence
414             """
415             (keypoint1, descriptor1) = self.cornerpointdict[tags[0]]
416             (keypoint2, descriptor2) = self.cornerpointdict[tags[1]]
417             if method =='OpenCV':
418                 matchedpoints = cv.BFMatcher().knnMatch(descriptor1,
                        descriptor2, k=2)
419                 filteredmatchedpoints = []
420                 for pointone, pointtwo in matchedpoints:
421                     if pointone.distance < (pointtwo.distance * 0.75)
                            :
422                         filteredmatchedpoints.append([pointone])
423                 self.correspondence[tags[2]]=filteredmatchedpoints
424                 result = cv.drawMatchesKnn(self.originalImages[
                        queueImages[0]], keypoint1, self.originalImages[
                        queueImages[1]],keypoint2, filteredmatchedpoints,
                        None, flags=2)
425                 cv.imwrite("results/"+ str(tags[2]) + ".jpg", result)
426             elif method =='Custom':
427                 tempdict = dict()
428                 for index,element in enumerate(descriptor1):
429                     list = []
430                     list2 = []
431                     for index2, element2 in enumerate(descriptor2):
432                         eucledianddistance = np.sqrt(np.sum(np.square
                                ((element-element2))))
433                         list.append(eucledianddistance)
434                         list2.append(keypoint2[index2])
435                     minimumvalue = min(list)
436                     id = list2[list.index(minimumvalue)]
437                     tempdict[(int(keypoint1[index].pt[1]),int(
                            keypoint1[index].pt[0]))]=((int(id.pt[1]),int(
                            id.pt[0])),minimumvalue)
438                 self.correspondence[tags[2]] = tempdict
439
440 if __name__ =='__main__':
441
442     """
443     Code starts here
444     """
445     tester = Panorama(['input_images/1.jpg','input_images/2.jpg',
            'input_images/3.jpg', 'input_images/4.jpg',
446                        'input_images/5.jpg'], 0.707)
447     for i in range(5):
```

```
448         tester.sift_corner_detect(i, str(i))
449     print("Detected SIFT interest points in 5 images.")
450     for i in range(0,4,1):
451         print(i)
452         tester.sift_correpondence((i,i+1),(str(i),str(i+1),str(i)
               +str(i+1)), 'OpenCV')
453         tester.update_dict_values((str(i),str(i+1),str(i)+str(i
               +1)))
454 
455     tester.get_panorama_done()
```