COMPUTER VISION

ECE 661

HOMEWORK 5

Tanzeel U. Rehman

Email: *trehman@purdue.edu*

0. Theory Questions:

Q: 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?

Ans: The interest point extracted from two different photos of same scene have to be used together with normalized cross correlation (NCC) or sum of square differences (SSD) to find the correspondences (see section A for details). Now to identify the correspondences as inliers from a correspondence having form (X, X'), we estimated the range point as HX by projecting the domain point (X) using the homography under consideration. We, then computed the squared residuals between the estimated range points and actual range points as: $res_x, res_y = (HX - X')^2$. Now these residuals were used to find the distance as: $d = \sqrt{res_x + res_y}$. Now all the correspondences which have $d < \delta$ were considered as the inliers and the others were outliers. If the given homography have inliers greater than M, then this will be considered as acceptable homography. Out of the N trials, the homography having the maximum number of inliers was considered as final solution (see section B for more details and the notations).

Q: 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.

Ans: The LM algorithm combines both GD and GN using formulation below:

$$\left(J_{\vec{f}}^T J_{\vec{f}} + \mu I\right) \overrightarrow{\delta_p} = J_{\vec{f}}^T \vec{\epsilon}\,\overrightarrow{(p_k)}$$

Where, μ is damping coefficient. The term on the right side of equation is the GD and rearranging the remaining terms and rewriting will lead to the GN. Now depending on the damping coefficient, the LM will be steered to either GD or GN. The details on finding the damping coefficient can be seen in the section D. In this system if damping coefficient is much larger than the diagonal elements of $J_{\vec{f}}^{T} J_{\vec{f}}$, the solution will be steered to the GD, while if damping coefficient is zero, the solution will purely be GN. When we suspect that LM is not generating the desirable results, its if steered to GD as it results the safer convergence. This steering mechanism controlled by the damping coefficient helps to utilize the power of both GD and GN, thereby leading to the reasonably fast and numerically stable solution.

1. Introduction:

This homework generates panorama using five overlapping images. The brief overview of the method used is as follows:

a) In the first step, SIFT operator from the OpenCV was used to extract the key features of an image. The SIFT features were then used together with normalized cross-correlation (NCC) to establish the correspondences between an image pair.

b) In order to remove the false correspondences, we used the random sample consensus (RANSAC) algorithm to find only the inliers. These inliers were then used to find the homographies using linear least squares approach.

c) Finally, we used Levenberg-Marquardt (LM) algorithm to further refine the homogrpahies in non-linear fashion. The LM method used the homographies estimated in earlier step as initial guess.

1.1. Description of Methods:

A. SIFT Correspondences:

For this homework, we used OpenCV implementation of SIFT feature operator. For each image, we extracted maximum of 5000 interest points. The correspondences between two image pairs was calculated by finding a feature in image 2 which is closest to the feature in image 1. The closeness measure for this homework was NCC, which can be computed using Eq. 1. In order to avoid noisy correspondences, all feature points between the first and second images were considered as valid features with NCC being greater than rejection ratio. For this homework the rejection ratio was manually adjusted to a value of 0.9 for all image pairs.

$$NCC = \frac{\sum_x \sum_y ((f_1(x,y) - m1)(f_2(x,y) - m2)}{\sqrt{\sum_x \sum_y ((f_1(x,y) - m1)^2 (f_2(x,y) - m2)^2}} \quad \text{(Eq.1)}$$

B. RANSAC:

The correspondences detected by using the SIFT algorithm with NCC between an image pair can have some false correspondences or the noisy correspondences. Using these correspondences directly for finding the homography with the help of linear least squares approach (see section C for more details) will yield the inappropriate or noisy results. Therefore, to avoid this, RANSAC was performed to remove the outliers from the correspondences. The inliers found by RANSAC was further used to find the homography.

The RANSAC, randomly selects the minimum number of correspondences needed to estimate the homography using linear least squares approach. Now, we find the points that support or agrees

with the constructed homography and these points will be considered as the inliers for the specific homography. This process of finding the inliers support was repeated for N trials and the homography having the maximum inlier support was considered as the final estimate of the homography. In order to compute the number of trials (N), following parameters need to be defined:

- Decision threshold (δ) of 3 pixels to determine if any correspondence is inlier or not.
- Probability (P) that atleast one of the N trials is free from the outliers. Here we used P = 0.99.
- Total number of correspondences (n_{total}) found by NCC using SIFT features.
- Smallest number of correspondences (n) for computing the homography. Here we used n = 6.
- The probability (ϵ) that a randomly selected correspondence is an outlier. Since, we used the NCC for correspondence estimation, the results were relatively noisy, therefore we used ϵ = 0.8.
- Now, the total number of trials (N) are: $N = \frac{\ln(1-p)}{\ln[1-(1-\epsilon)^n]}$.
- The minimum size (M) of the inlier set that can be accepted is given by: $M = \mathrm{n}(1-\epsilon)$.

In order to identify the correspondences as inliers from a correspondence of the form (X, X'), we estimated the range point as HX by projecting the domain point (X) using the homography under consideration. We, then computed the squared residuals between the estimated range points and

actual range points as: $res_x, res_y = (HX - X')^2$. Now these residuals were used to find the distance as: $d = \sqrt{res_x + res_y}$. Now all the correspondences which have $d < \delta$ were considered as the inliers and the others were outliers. Now, for given homography estimate, if the inliers were greater than *M*, then this will be considered as acceptable homography. Out of the *N* trials, the homography having the maximum number of inliers was considered as final solution and was used for generating the panorama (see section E for more details). The set having maximum number of inliers was further used with LM algorithm to refine the homographies.

C. Homography estimation with linear least squares:

Given a point *X* in a world scene and its corresponding pixel location *X′* in the distorted image plane can be given by Eq. 2.

$$X' = HX \qquad \text{(Eq.2)}$$

Where, *X* and *X′* are homogeneous coordinates of the form $\begin{bmatrix} x_1 \\ y_1 \\ w_1 \end{bmatrix}$ and $\begin{bmatrix} x_2' \\ y_2' \\ w_2' \end{bmatrix}$ for physical points (*x*, *y*) in world plane and (*x′*, *y′*) in distorted image plane respectively. *H* is non-singular homography matrix in homogeneous coordinates which can be represented by Eq. 3. Under the assumption that our correspondences are error free, it must be the case that for a given correspondence (*X*, *X′*), the vector *X′* and *HX* are identical. Therefore, it must follow Eq. 4 (based on lecture 10). The resulting form can be expressed using Eq. 5. The solution to Eq. 5 can be obtained using linear least squares approach by $\min \left|\left|A\vec{h}\right|\right| = 0$ subjected to the constraint $\vec{h} = 0$. This solution can be obtained by using the singular value decomposition (SVD) which is basically the eigenvector of A^TA corresponding to the smallest eigenvalue. The resultant homography is the linear least squared

optimized homography that can be used with RANSAC and can also be used as initial guess with LM algorithm.

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \quad \text{(Eq.3)}$$

$$X'HX = 0 \quad \text{(Eq.4)}$$

$$\begin{bmatrix} 0 & 0 & 0 & -w_i'x_i & -w_i'y_i & -w_i'w_i & y_i'x_i & y_i'y_i & y_i'w_i \\ w_i'x_i & w_i'y_i & w_i'w_i & 0 & 0 & 0 & -x_i'x_i & -x_i'y_i & -x_i'w_i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = 0 \quad \text{(Eq.5)}$$

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

D. Refining Homographies with Levenburg-Marquardt (LM):

The LM algorithm combines the Gauss-Newton and Gradient descent algorithms to achieve the numerical stability and reducing the processing time. The general form of LM algorithm can be given by Eq. 6.

$$\left(J_{\vec{f}}^T J_{\vec{f}} + \mu I\right) \overrightarrow{\delta_p} = J_{\vec{f}}^T \vec{\epsilon}\, \overrightarrow{(p_k)} \quad \text{(Eq.6)}$$

Where, $J_{\vec{f}}$ is the Jacobian of the function $f_{(\vec{p})}$ with respect to the components of $\vec{p}$ and can be represented by Eq. 7. For the correspondences of the form (X, X'), the function $f_{(\vec{p})}$ is used to represent projection of physical points in domain plane (X) to the physical points in the range plane obtained through the homography matrix H (Eq. 8). It should be noted that the output of function $f_{(\vec{p})}$is predicted/estimated range points (X'_{est}). The μ is damping coefficient, I is the identity matrix

and $\vec{\epsilon} = \overrightarrow{X'} - f_{(\overrightarrow{pk})}$ represents the residuals in the estimated range points (X'_{est}) at k[th] iteration and $\overrightarrow{\delta_p}$ is the change in the value of $\overrightarrow{pk}$ after k[th] iteration.

$$J_{\vec{f}} = \begin{pmatrix} \partial f_1/\partial p_1 & \cdots & \partial f_1/\partial p_n \\ \vdots & \ddots & \vdots \\ \partial f_n/\partial p_1 & \cdots & \partial f_m/\partial p_n \end{pmatrix} \quad \text{(Eq.7)}$$

$$f_{(\vec{p})} = estimated_{range\_points} = HX \quad \text{(Eq.8)}$$

The LM algorithm refines the homography matrices as follows:

- Start with an initial guess $\overrightarrow{p0}$. Compute the initial value of the damping coefficient $\mu 0$ as: $\mu 0 = \tau . \max\left(diag(J_{\vec{f}}^T, J_{\vec{f}}\right)$ with τ between 0 and 1.

- Compute the $\overrightarrow{\delta_p}$ from Eq. 6 by inverting $\left(J_{\vec{f}}^T J_{\vec{f}} + \mu I\right)$

- Check the quality of computed $\overrightarrow{\delta_p}$ by using the ratio of actual change in the cost function to the change in the cost function predicted by particular choice (Eq. 9). Where, $C\overrightarrow{(p_k)} = \left|\left|\vec{X} - f_{(\vec{p})}\right|\right|^2$ is the cost at the k[th] iteration.

- The damping coefficient for the next iteration can be given by Eq. 10.

- Repeat the above process, till the number of thresholds reached, or there is no decrease in the $C\overrightarrow{(p_k)}$ thus indicating the convergence.

$$\rho_{k+1}^{LM} = \frac{C\overrightarrow{(p_k)} - C\overrightarrow{(p_{k+1})}}{\overrightarrow{\delta_p}^T J_{\vec{f}}^T \vec{\epsilon}(p_k) + \overrightarrow{\delta_p}^T \mu_k I \overrightarrow{\delta_p}} \quad \text{(Eq.9)}$$

$$\mu_{k+1} = \mu_k . \max\{\frac{1}{3}, 1 - 2(\rho_{k+1}^{LM} - 1)^3\} \quad \text{(Eq.10)}$$

For this homework we used the scipy.optimize.least squares algorithm with LM optimization method to refine the homogrpahies.

E. Panorama generation:

In order to generate the panorama, we need to project all the images onto a common reference plane. As we have 5 images, therefore this common reference plane was considered as image 3 and all other images are projected onto its plane. Let H_{12}, H_{23}, H_{34}, and H_{45} be the homographies between different image pairs. The homographies from all the images to image 3 can be as shown below:

$$H_{13} = H_{12} \times H_{23} \quad \text{(Eq.11)}$$

$$H_{43} = {H_{34}}^{-1} \quad \text{(Eq.12)}$$

$$H_{53} = {H_{34}}^{-1} \times {H_{45}}^{-1} \quad \text{(Eq.13)}$$

Now, all these homographies were used to project all images onto image 3.

1.2. Parameters used for homework:

a) SIFT and NCC:

- Best_features = 5000
- contrastThreshold = 0.03
- edgeThreshold = 10
- sigma = 2.6
- NCC_rejection_threshold = 0.9

b) RANSAC:

- $\delta = 3$ pixels
- $n = 6$
- $P = 0.99$
- $\epsilon = 0.8$ (This was set to high as the NCC leaves some false correspondences)

c) LM :

Parameters used for refining the homogrpahies with scipy.opttimize.least_squares.

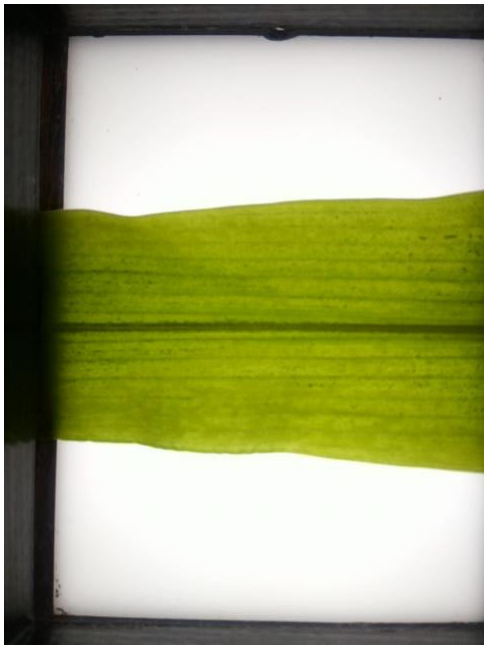
- jac.= cs
- method = ‘lm’
- gtol = 1e-8

2. Task 1:

2.1. Input and resulting images:

(a)

(b)



(c)

(d)

(e)

Fig 1: Set of 5 input images used for this homework. Images were captured with the help of RaspberryPi camera for the corn leaf.

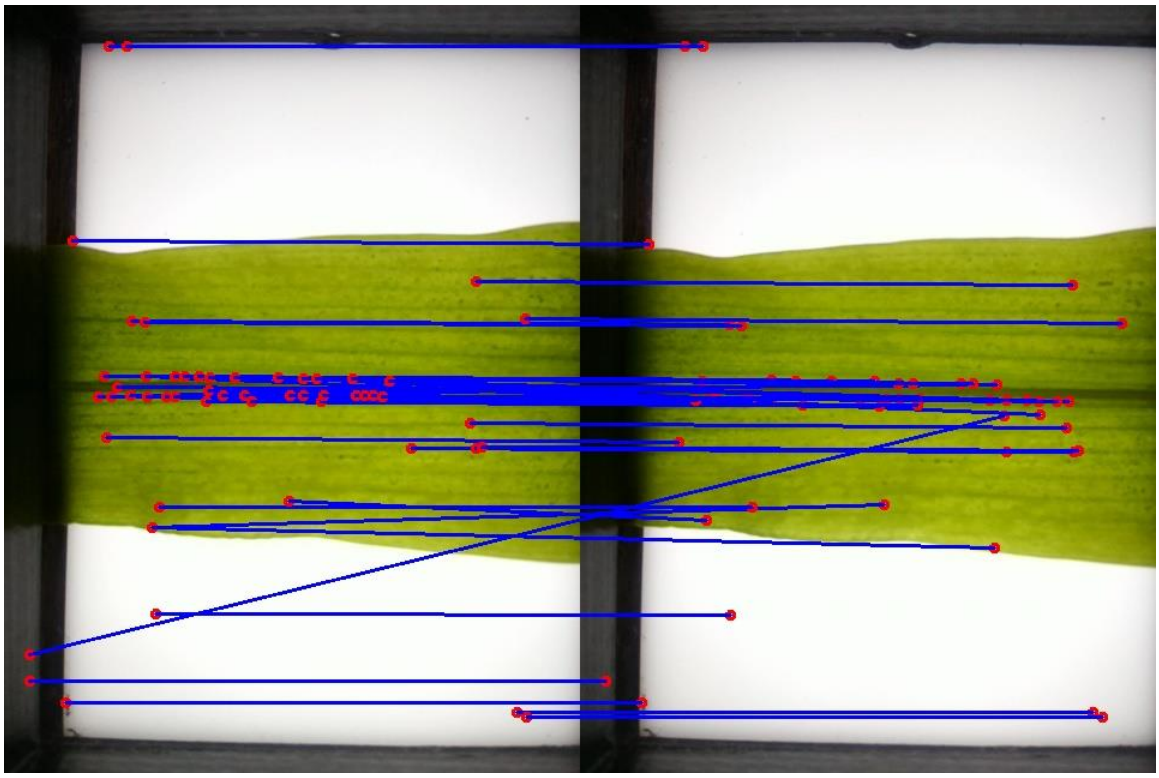


Fig 2: Correspondences detected by SIFT with NCC between image pair a and b.

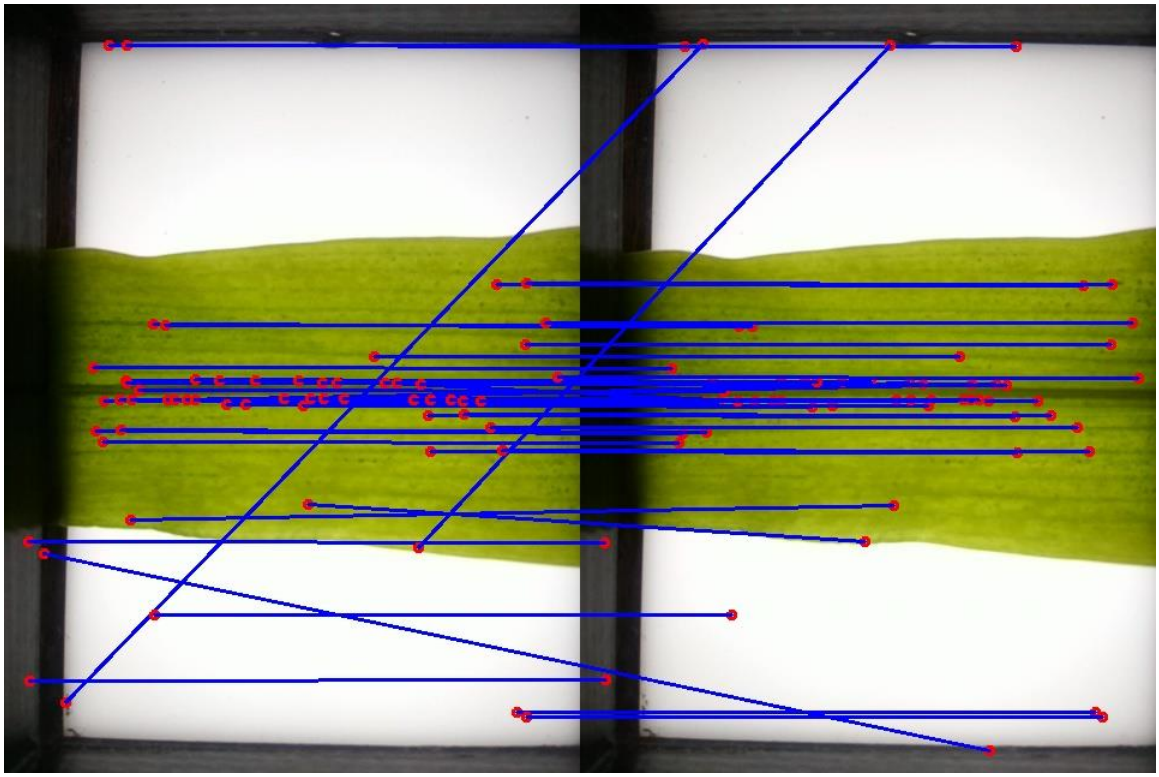


Fig 3: Correspondences detected by SIFT with NCC between image pair b and c.

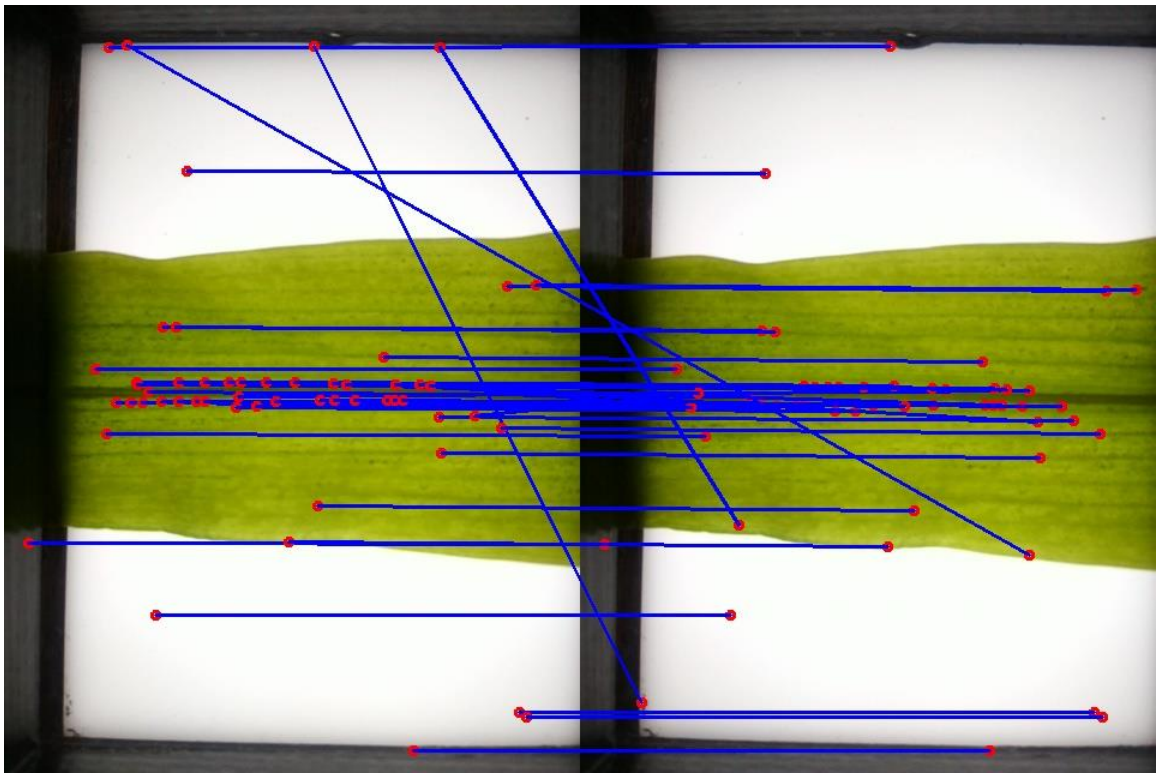


Fig 4: Correspondences detected by SIFT with NCC between image pair c and d.

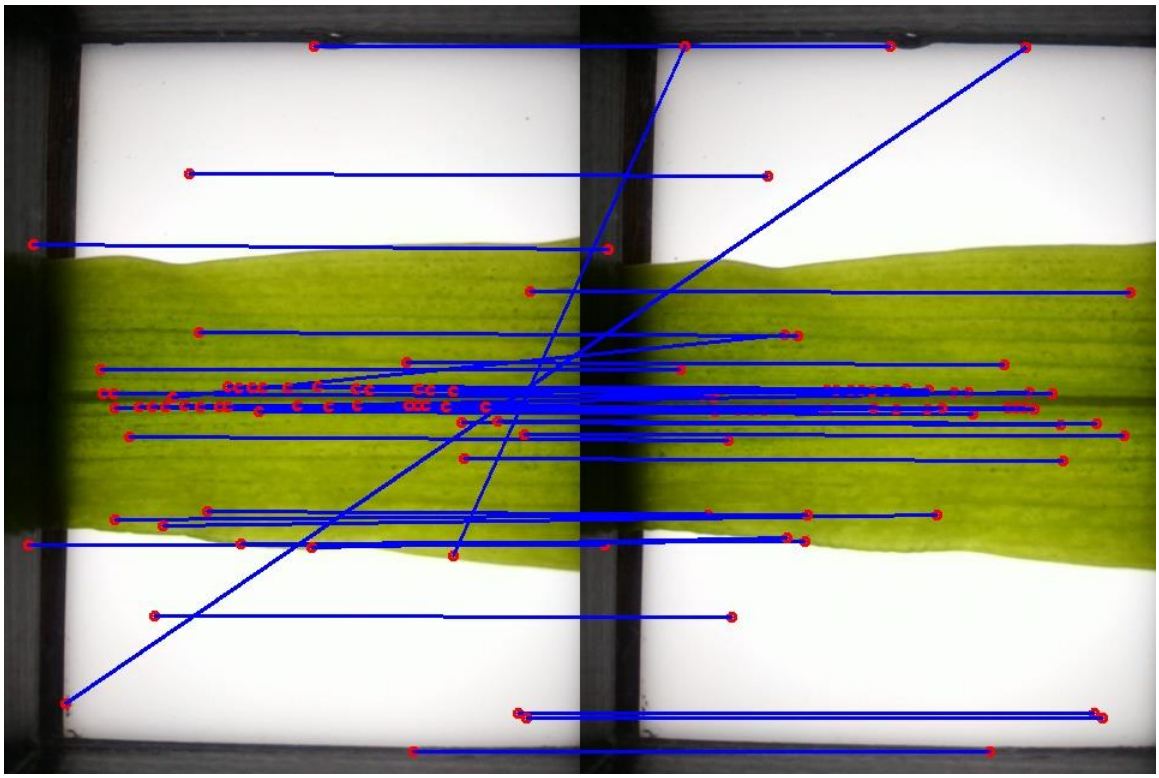


Fig 5: Correspondences detected by SIFT with NCC containing image pair d and e.

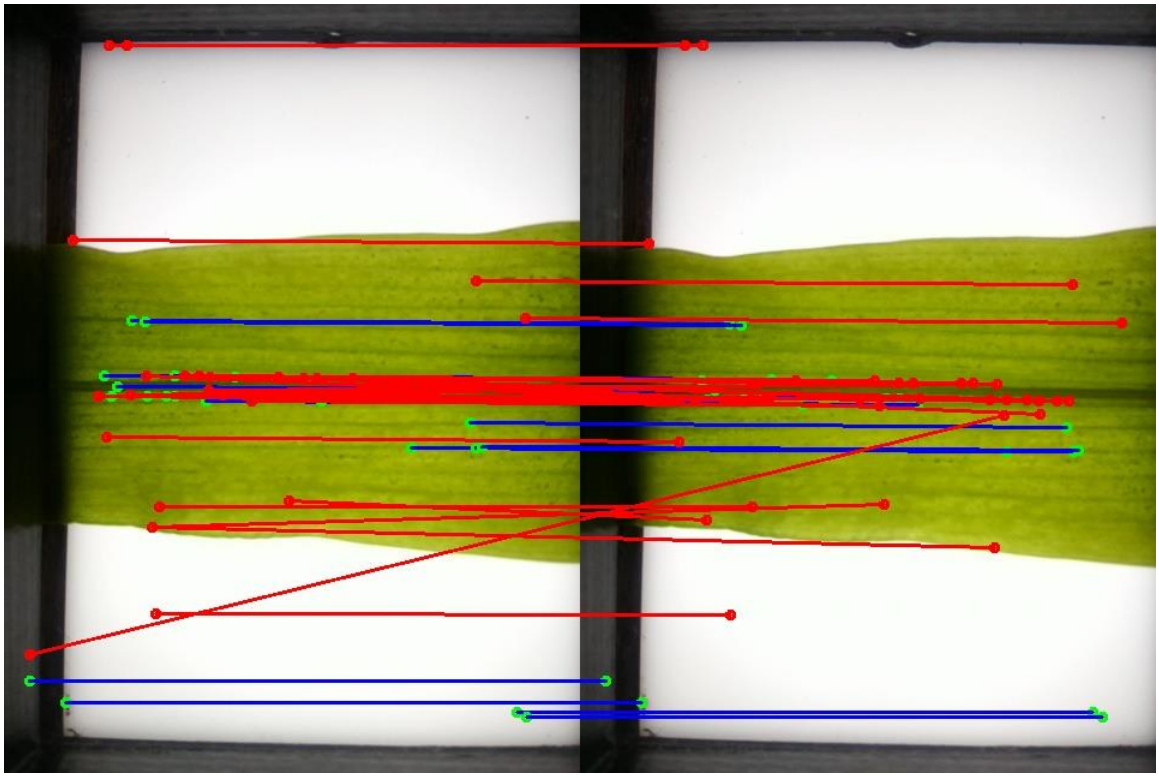


Fig 6: Correspondences rejected by RANSAC algorithm (Red) between image pair a and b.

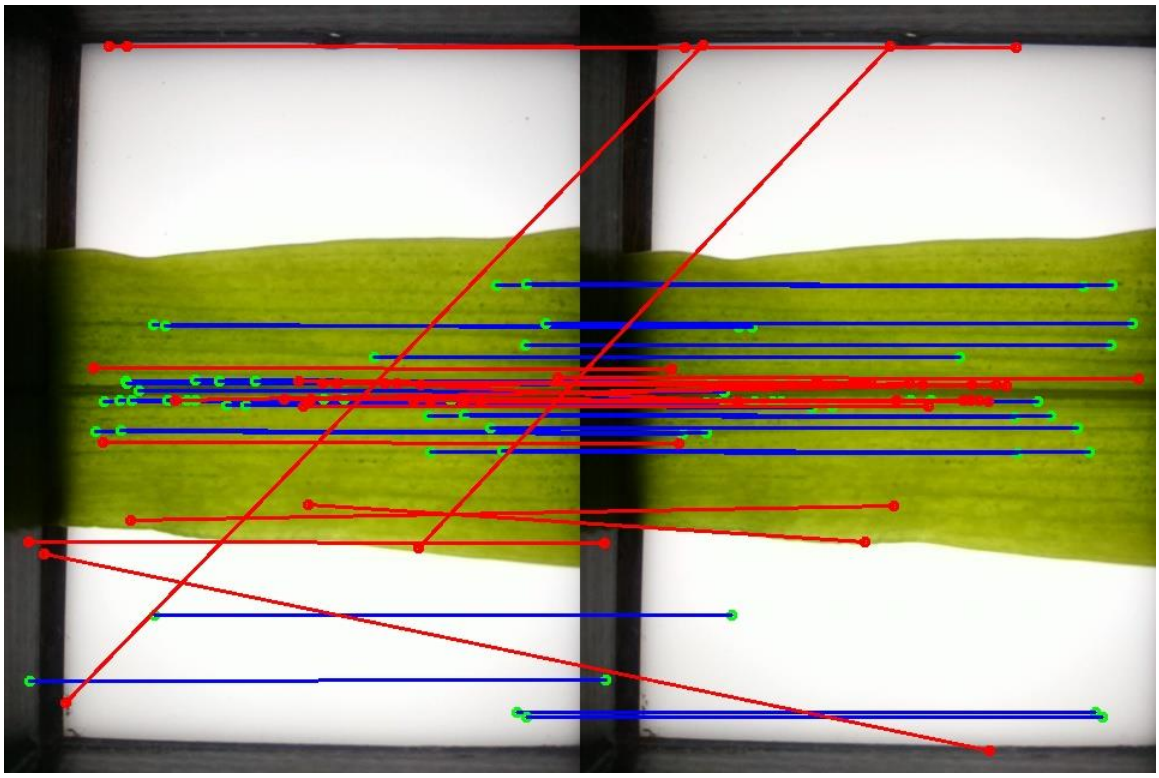


Fig 7: Correspondences rejected by RANSAC algorithm (Red) between image pair b and c.

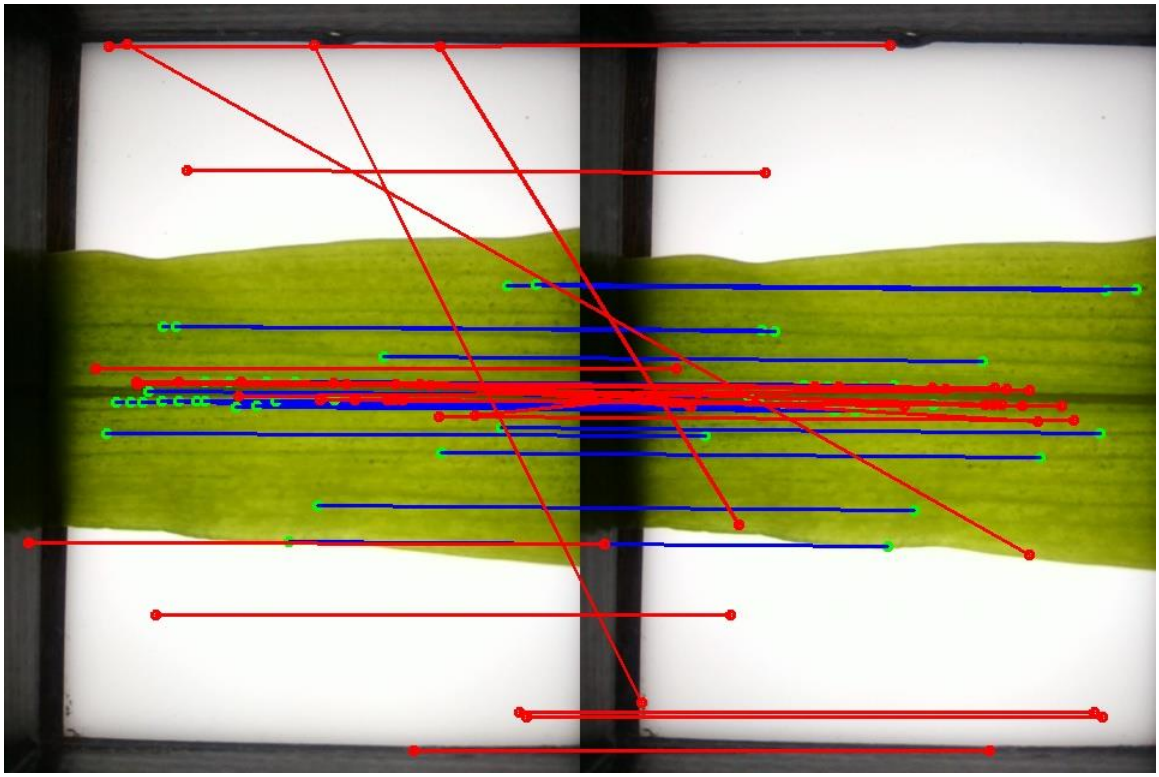


Fig 8: Correspondences rejected by RANSAC algorithm (Red) between image pair c and d.

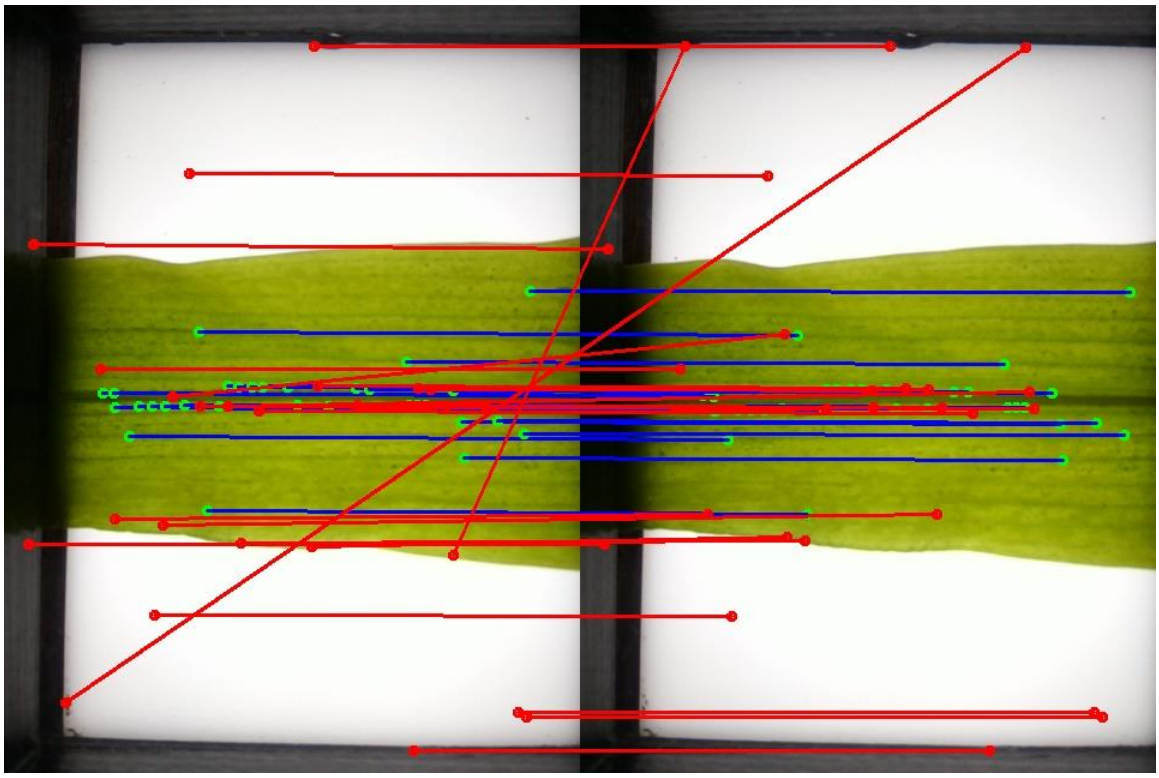


Fig 9: Correspondences rejected by RANSAC algorithm (Red) between image pair d and e.

Fig 10: Panorama generated by using the homographies estimated with the help of RANSAC algorithm.

Fig 11: Panorama generated by using the homographies refined using Levenburg-Marquardt algorithm.

3. Source Code:

3.1.Function calls for Task 1.1:

```
'''----------Main Code for Image Mosacing---------'''
# Read the images of the leaf, Get the SIFT features and Save NCC Correspondences
img_1,img_col_1,img_2,img_col_2,match_12 = Save_Correspondences(57,58)
_,_,img_3,img_col_3,match_23 = Save_Correspondences(58,59)
_,_,img_4,img_col_4,match_34 = Save_Correspondences(59,60)
_,_,img_5,img_col_5,match_45 = Save_Correspondences(60,61)

#Get the Homography matrices using RANSAC and the array of inliers
H_RAN_12,Inliers_12 = Save_Correspondences_In_Out(img_col_1,img_col_2,match_12,3,6,0.99,0.8,filename1=57,filename2=58)
```

```
H_RAN_23,Inliers_23 = Save_Correspondences_In_Out(img_col_2,img_col_3,match_23,3,6,0.99,0.8,filename1=58,filename2=59)
H_RAN_34,Inliers_34 = Save_Correspondences_In_Out(img_col_3,img_col_4,match_34,3,6,0.99,0.8,filename1=59,filename2=60)
H_RAN_45,Inliers_45 = Save_Correspondences_In_Out(img_col_4,img_col_5,match_45,3,6,0.99,0.8,filename1=60,filename2=61)

# Image 3 is the center image, so project every other image on it
H_RAN_13 = np.matmul(H_RAN_12,H_RAN_23) #This is product of 12,23
H_RAN_23 = H_RAN_23    #This will not chnage
# As projecting 3 on 3, therefore the H33 will be identity matrix
H_RAN_33 = np.identity(3)
H_RAN_43 = np.linalg.inv(H_RAN_34)  #This is inv of 34
H_RAN_35 = np.matmul(H_RAN_34,H_RAN_45) #This is product of 34,45
H_RAN_53 = np.linalg.inv(H_RAN_35)  #Invert the matrix as it is inverse of 35


#Get the possible dimensions and offset of the panorama
x13min,y13min,x13max,y13max=Bounds_Undistorted(H_RAN_13,img_1)
x23min,y23min,x23max,y23max=Bounds_Undistorted(H_RAN_23,img_2)
x33min,y33min,x33max,y33max=Bounds_Undistorted(H_RAN_33,img_3)
x43min,y43min,x43max,y43max=Bounds_Undistorted(H_RAN_43,img_4)
x53min,y53min,x53max,y53max=Bounds_Undistorted(H_RAN_53,img_5)
#Overall min and max
min_x = np.minimum (np.minimum (np.minimum(np.minimum(x13min,x23min), x33min),x43min), x53min)
min_y = np.minimum (np.minimum (np.minimum(np.minimum(y13min,y23min), y33min),y43min), y53min)
max_x = np.maximum (np.maximum (np.maximum(np.maximum(x13max,x23max), x33max),x43max), x53max)
max_y = np.maximum (np.maximum (np.maximum(np.maximum(y13max,y23max), y43max),y43max), y53max)
min_xy = np.array([min_x,min_y])
max_xy = np.array([max_x,max_y])
panorama_RANSAC_Dim = max_xy - min_xy
panorama_RANSAC = np.zeros((int(panorama_RANSAC_Dim[1]),int(panorama_RANSAC_Dim[0]),3))
#Create Panorama for RANSAC homograpies
panorama_RANSAC = transform_to_panorama(panorama_RANSAC,img_col_1,H_RAN_13,min_xy)
panorama_RANSAC = transform_to_panorama(panorama_RANSAC,img_col_2,H_RAN_23,min_xy)
```

```
panorama_RANSAC =
transform_to_panorama(panorama_RANSAC,img_col_3,H_RAN_33,min_xy)
panorama_RANSAC =
transform_to_panorama(panorama_RANSAC,img_col_4,H_RAN_43,min_xy)
panorama_RANSAC =
transform_to_panorama(panorama_RANSAC,img_col_5,H_RAN_53,min_xy)
cv2.imwrite("Panorama_RANSAC2.jpg",panorama_RANSAC)

'''----------Generating the panorama using the LM refined Homographies------'''
# Refine the homographies using the LM algorithm
H_LM12 = LM_Homography(H_RAN_12,Inliers_12)
H_LM23 = LM_Homography(H_RAN_23,Inliers_23)
H_LM34 = LM_Homography(H_RAN_34,Inliers_34)
H_LM45 = LM_Homography(H_RAN_45,Inliers_45)

''' Since LM changes the Homographies from the HC representation, therefore we need to convert them back'''

# Image 3 is the center image, so project every other image on it
H_LM13 = np.matmul(H_LM12,H_LM23)#This is product of 12,23
H_LM13 = H_LM13/H_LM13[-1,-1]
H_LM23 = H_LM23/H_LM23[-1,-1] #This will not chnage
# As projecting 3 on 3, therefore the H33 will be identity matrix
H_LM33 = np.identity(3)
H_LM43 = np.linalg.inv(H_LM34) #This is inv of 34
H_LM43 = H_LM43/H_LM43[-1,-1]
H_LM35 = np.matmul(H_LM34,H_LM45)  #This is product of 34,45
H_LM53 = np.linalg.inv(H_LM35) #Invert the matrix as it is inverse of 35
H_LM53 = H_LM53/H_LM53[-1,-1]

#Get the possible dimensions and offset of the panorama
x13min,y13min,x13max,y13max=Bounds_Undistorted(H_RAN_13,img_1)
x23min,y23min,x23max,y23max=Bounds_Undistorted(H_RAN_23,img_2)
x33min,y33min,x33max,y33max=Bounds_Undistorted(H_RAN_33,img_3)
x43min,y43min,x43max,y43max=Bounds_Undistorted(H_RAN_43,img_4)
x53min,y53min,x53max,y53max=Bounds_Undistorted(H_RAN_53,img_5)
#Overall min and max
min_x = np.minimum (np.minimum (np.minimum(np.minimum(x13min,x23min), x33min),x43min), x53min)
min_y = np.minimum (np.minimum (np.minimum(np.minimum(y13min,y23min), y33min),y43min), y53min)
max_x = np.maximum (np.maximum (np.maximum(np.maximum(x13max,x23max), x33max),x43max), x53max)
max_y = np.maximum (np.maximum (np.maximum(np.maximum(y13max,y23max), y43max),y43max), y53max)
min_xy = np.array([min_x,min_y])
```

```
    max_xy = np.array([max_x,max_y])

    panorama_LM_Dim = max_xy - min_xy
    panorama_LM = np.zeros((int(panorama_LM_Dim[1]),int(panorama_LM_Dim[0]),3))

    panorama_LM = transform_to_panorama(panorama_LM,img_col_1,H_LM13,min_xy)
    panorama_LM = transform_to_panorama(panorama_LM,img_col_2,H_LM23,min_xy)
    panorama_LM = transform_to_panorama(panorama_LM,img_col_3,H_LM33,min_xy)
    panorama_LM = transform_to_panorama(panorama_LM,img_col_4,H_LM43,min_xy)
    panorama_LM = transform_to_panorama(panorama_LM,img_col_5,H_LM53,min_xy)
    cv2.imwrite("Panorama_LM2.jpg",panorama_LM)
```

A. Different functions for generating the panorama using RANSAC and LM algorithm:

```
def grayscale(imgname,resize_scale=0.5,resize_falg=True):
    '''
    Read the image and convert it into the grayscale if not already.
    '''
    #Read the color image
    img_color = cv2.imread(imgname)
    # Adjust the width and height by a constant factor, this maintains the aspect ratio
    if resize_falg == True:
        w = int(img_color.shape[1]*resize_scale)
        h = int(img_color.shape[0]*resize_scale)
        img_color=cv2.resize(img_color,(w,h),interpolation = cv2.INTER_LINEAR)
    else:
        img_color = img_color

    #Check if the image has been read
    if img_color is not None:
        #Check if the image is color
        if len(img_color.shape)==3:
            #Convert to gray scale
            img_gray= cv2.cvtColor(img_color,cv2.COLOR_BGR2GRAY)
        elif len(img_color.shape)==1:
            img_gray=img_color
        else:
            print("Image shape not identified")
        return  img_gray,img_color
    else:
        print ("Image not found:"+imgname)
    return None

def Get_SIFT_Features(img_gray,best_features=5000):
    '''
    Function for finding the SIFT features of a grayscale image
```

```
    '''
    #Create a SIFT object
    sift = cv2.SIFT_create(best_features,contrastThreshold=0.03,edgeThreshold=10,sigma=2.6)
    #sift = cv2.SIFT_create(nfeatures=5000,nOctaveLayers=4,contrastThreshold=0.03,edgeThreshold=10,sigma=4)
    #Compute the sift features and descriptors
    kp, des = sift.detectAndCompute(img_gray,None)
    return kp, des

def Get_BruteForce_Correspondences(sift_des1, sift_des2):
    '''
    Use this function or the NCC correspondences
    '''
    #Initialize the Brute force matching
    BF_matcher = cv2.BFMatcher()

    #Find the 2 valid matches for each point, Use Lowe threshold
    two_matches = BF_matcher.knnMatch(sift_des1,sift_des2,k=2)

    #Store all valid matches
    Valid_matches = []

    # Pick the best match based on Lowe's paper
    for m1, m2 in two_matches:
        if m1.distance < 0.75 * m2.distance:
            Valid_matches.append([m1])


    return Valid_matches

def Get_NCC_Correspondences(Cornerslist_1,Cornerslist_2,des1,des2,reject_ratio=0.8):

    #Convert the corner lists to the arrays
    corner_image1 = GetXY_Coordinates_Kp(Cornerslist_1)
    corner_image2 = GetXY_Coordinates_Kp(Cornerslist_2)

    #corner_image1 = np.array(Cornerslist_1)
    #corner_image2 = np.array(Cornerslist_2)

    #win_half = int(window_dim/2)
    #Initialize an empty list for storing corners
    valid_correspondences = []

    #Initialize 2D matrix to store the distances of a specific point in image 1 with everyother point in image 2
```

```
    #Size will be num_corners_1 x num_corners_2
    F = np.zeros((len(corner_image1),len(corner_image2)))

    for y in range(len(corner_image1)):
        for x in range(len(corner_image2)):
            f1 = des1[y,:]
            f2 = des2[x,:]
            mean1 = np.mean(f1)
            mean2 = np.mean(f2)
            numemnator = np.sum((f1- mean1)*(f2- mean2))
            denomenator = np.sqrt((np.sum((f1- mean1)**2))*(np.sum((f2- mean2)**2)))
            F[y,x] = numemnator/denomenator
    #Identify the corresponding corner points in the two images by thresholding
    for y in range(len(corner_image1)):
        x=np.argmax(F[y,:])
        if F[y,x] > reject_ratio:
            F[:,x] = np.NINF   #Mark that this column has been taken to avoid double correspondence
(hard learn't lesson)

valid_correspondences.append([corner_image1[y,0],corner_image1[y,1],corner_image2[x,0],cor
ner_image2[x,1]])

    return np.array(valid_correspondences)

def GetXY_Coordinates_Kp(kp):
    '''
    Function for extracting the coordinates from the list of keypoints extracted from the SIFT
    '''
    pts=[]
    for key in kp:
        pts.append([np.round(key.pt[0],0),np.round(key.pt[1],0)])
    return np.array(pts)

def find_homography (Domain_pts, Range_pts):
    '''
    function for estimating the 3 x 3 Homography matrix---Modified from HW2
     Inputs:
        Domain_pts: An n x 2 array containing coordinates of domian image points(Xi,Yi)
        range_point: An n x 2 array containing coordinates of range image points(Xi',Yi')
    Output: A 3 x 3 Homography matrix
    '''

    # Find num of points provided
    n = Domain_pts.shape[0]
    #Initialize A Design matrix having size of 2n x 8
    A = np.zeros((2*n,9))
```

```
    H = np.zeros((3,3))
    #Loop through all the points provided and stack them vertically, this will result in 2n x 9 Design matrix
    for i in range (n):
        A[i*2:i*2+2]=Get_A_matrix(Domain_pts[i],Range_pts[i])
    '''
    Compute the h vector (9 x 1) by using least sqaures solution.Decompose the A matrix using SVD
    to obtain the eigenvector corresponding to smallest eigen value of A^TA, which is basically h.
    '''
    #Decompose the A matrix and obtain the
    U,D,V = np.linalg.svd(A)
    h = V.T[:,8] #Eigen vector corresponding to the smallest eigen value of D
    # Rearrange the vector h to Homography matrix H

    H[0] = h[0:3] / h[-1]
    H[1] = h[3:6] / h[-1]
    H[2] = h[6:9] / h[-1]
    #h=np.dot(np.linalg.inv(np.dot(A.T,A)),np.dot(A.T,y))
    return H

def Get_A_matrix(domain_point,range_point):
    '''
    function for generating a 2 x 9 design matrix needed to compute Homography
    Inputs:
        domain_point: Coordinates of a point in the domain image (x,y)
        range_point: Coordinates of corresponding point in the range image (x',y')
    Output: A 2 x 9 design matrix
    '''
    # Extract the x and y coordinates from a point pair
    x,y=domain_point[0], domain_point[1]
    xr,yr=range_point[0], range_point[1]
    #  Make A matrix
    A=np.array([[0,0,0,-x,-y,-1,yr*x,yr*y,yr],[x,y,1,0,0,0,-xr*x,-xr*y,-xr]])
    return A

def RANSAC(Combined_Correspondences,delta,n,Probability,E):
    '''
    Function for running the RANSAC algorithm to find the inliers and
    rejecting the outliers.
    '''
    # Total number of trials needed to find the inliers
    N = (np.log(1 - Probability)/np.log(1-(1-E)**n)).astype(np.int)
    ntotal = Combined_Correspondences.shape[0]
    #Min size of inlier set
```

```
    M = int(ntotal* (1-E))
    print("Total Number of trials: ",N)
    print("Minimum size of inlier set: ",M)

    number_inliers = -1

    for trial in range(N):
        #Pick points randomly, compute homography and find inliers
        idx = np.random.randint(0,ntotal,n)
        H_temp                                                                    =
find_homography(Combined_Correspondences[idx,0:2],Combined_Correspondences[idx,2:4])
        #Inlier counts
        Inliers,_ = find_inliers(Combined_Correspondences,H_temp,delta)
        #Condition for finding the solution with max inliers
        if len(Inliers) > number_inliers:
            number_inliers = len(Inliers)
            #Find the solution that also passes the minimum threshold criteria
            if number_inliers > M:
                H_final=H_temp

    return H_final

def func_LM_Homography(H, Domain_x,Domain_y,Range_x,Range_y):
    '''
    Function that need to be supplied to the scipy optimize module. Requires all inputs as 1d array
    The first argyment will be optimized as a result. Optimization will be done based on the
Eucledian distance
    '''
    #Combine the x and y from the domain and range points
    Domain_Inliers = (np.array([Domain_x,Domain_y])).T
    Range_Inliers = (np.array([Range_x,Range_y])).T
    #Reshape the H matrix to be 3 x 3 matrix
    H = H.reshape((3,3))
    # Change domain points to the HC representation to compute the estimated range positions
    Domain_Inliers = np.append(Domain_Inliers,np.ones((len(Domain_Inliers),1)),axis=1)
    Domain_Inliers = Domain_Inliers.T # Make dimensions 3 x n
    #Apply homography to estimate the position in the range image
    Range_estimate = H @ Domain_Inliers
    Range_estimate = Range_estimate/ Range_estimate[2]
    Range_estimate = Range_estimate[0:2].T # Change the dimensions back to n x 2
    # Compute the error
    sq_residuals = (Range_estimate - Range_Inliers)**2
    Euclidean_dist = np.sqrt(np.sum(sq_residuals,axis=1))
    return Euclidean_dist

def LM_Homography(H,Combined_Inliers):
```

```
    '''
    Function for computing the LM refined Homography
    '''
    #Reshape the input H matrix obtained from Linear approach
    H0 = np.reshape(H,9)
    Dm_Inlier_x = Combined_Inliers[:,0]
    Dm_Inlier_y = Combined_Inliers[:,1]
    Rg_Inlier_x = Combined_Inliers[:,2]
    Rg_Inlier_y = Combined_Inliers[:,3]

    res_lsq = least_squares(func_LM_Homography, H0,
args=(Dm_Inlier_x,Dm_Inlier_y,Rg_Inlier_x,Rg_Inlier_y),method = 'lm')
    H_LM = res_lsq.x
    H_LM = H_LM.reshape((3,3))
    return H_LM

def find_inliers(Combined_Correspondences,H,delta):
    '''
    Function for finding the inliers
    '''
    # Divide the combined correspondences to domain and range points
    Domain_pts = Combined_Correspondences[:,0:2]
    Range_pts = Combined_Correspondences[:,2:4]
    # Change domain points to the HC representation to compute the estimated range positions
    Domain_pts = np.append(Domain_pts,np.ones((len(Domain_pts),1)),axis=1)
    Domain_pts = Domain_pts.T # Make dimensions 3 x n
    #Apply homography to estimate the position in the range image
    Range_estimate = H @ Domain_pts
    Range_estimate = Range_estimate/ Range_estimate[2]
    Range_estimate = Range_estimate[0:2].T # Change the dimensions back to n x 2
    # Compute the error
    sq_residuals = (Range_estimate - Range_pts)**2
    Euclidean_dist = np.sqrt(np.sum(sq_residuals,axis=1))
    #Identify the inliers and outliers based on the distance and delta values
    idx = Euclidean_dist < delta
    Inliers = Combined_Correspondences[idx]
    Outliers = Combined_Correspondences[~idx]
    return Inliers,Outliers

def Show_Correspondences(img_1_color,img_2_color,corresponding_corners):
    '''
    Function for plotting the corresponding points on the image
    '''
    #Shape of input images
    h1,w1=img_1_color.shape[0:2]
    h2,w2=img_2_color.shape[0:2]
```

```
    #Find the maximum height from 2 images. This will be the height of output image
    max_height = max(h1,h2)
    #create an empty image having size of max_height x w1+w2
    plot_img = np.zeros((max_height,(w1+w2),3))
    #Fill empty image with image1 and 2, this will leave empty border on the
    #image of least height
    plot_img [0:h1,0:w1,:] = img_1_color
    plot_img [0:h2,w1:,:] = img_2_color

    for point in corresponding_corners:
        #Plot a circle of red color with a radius of 3 to mark the corner points on img1.
        cv2.circle(plot_img,tuple(point[0:2].astype(int)),3,(0,0,255),2)
        #Plot a circle of red color with a radius of 3 to mark the corner points on img2.
        cv2.circle(plot_img,tuple([int(point[2]+w1),int(point[3])]),3,(0,0,255),2)  #shift the pointer
by width of img1
        #Plot the blue line joining corresponding points on images

cv2.line(plot_img,tuple(point[0:2].astype(int)),tuple([int(point[2]+w1),int(point[3])]),(255,0,0),2
)
    return plot_img

def
Show_Correspondences_In_Outliers(img_1_color,img_2_color,corresponding_corners,H,delta):
    '''
    Function for plotting the corresponding points on the image
    '''
    #Shape of input images
    h1,w1=img_1_color.shape[0:2]
    h2,w2=img_2_color.shape[0:2]
    #Find the maximum height from 2 images. This will be the height of output image
    max_height = max(h1,h2)
    #create an empty image having size of max_height x w1+w2
    plot_img = np.zeros((max_height,(w1+w2),3))
    #Fill empty image with image1 and 2, this will leave empty border on the
    #image of least height
    plot_img [0:h1,0:w1,:] = img_1_color
    plot_img [0:h2,w1:,:] = img_2_color
    #Find the inliers and ouyliers
    inliers,outliers = find_inliers(corresponding_corners,H,delta)

    for point in inliers:
        #Plot a circle of green color with a radius of 3 to mark the corner points on img1.
        cv2.circle(plot_img,tuple(point[0:2].astype(int)),3,(0,255,0),2)
        #Plot a circle of green color with a radius of 3 to mark the corner points on img2.
        cv2.circle(plot_img,tuple([int(point[2]+w1),int(point[3])]),3,(0,255,0),2)  #shift the pointer
by width of img1
```

```
        #Plot the blue line joining corresponding points on images

cv2.line(plot_img,tuple(point[0:2].astype(int)),tuple([int(point[2]+w1),int(point[3])]),(255,0,0),2
)

    for point in outliers:
        #Plot a circle of green color with a radius of 3 to mark the corner points on img1.
        cv2.circle(plot_img,tuple(point[0:2].astype(int)),3,(0,0,255),2)
        #Plot a circle of green color with a radius of 3 to mark the corner points on img2.
        cv2.circle(plot_img,tuple([int(point[2]+w1),int(point[3])]),3,(0,0,255),2)  #shift the pointer
by width of img1
        #Plot the blue line joining corresponding points on images

cv2.line(plot_img,tuple(point[0:2].astype(int)),tuple([int(point[2]+w1),int(point[3])]),(0,0,255),2
)
    return plot_img,inliers

def Save_Correspondences(filename1,filename2):
    img_1,img_col_1=grayscale(("{0}.JPG".format(filename1)),0.5,False)      #Resize my own
images as they are big
    img_2,img_col_2=grayscale(("{0}.JPG".format(filename2)),0.5,False)      #Resize my own
images as they are big
    kp1, des1 = Get_SIFT_Features(img_1)
    kp2, des2 = Get_SIFT_Features(img_2)
    match_12 = Get_NCC_Correspondences(kp1, kp2, des1, des2, 0.9)
    plt_img_12 = Show_Correspondences(img_col_1,img_col_2,match_12)
    filename_write = "Pair{0}_{1}.jpg".format(filename1 , filename2)
    cv2.imwrite(filename_write,plt_img_12)
    return img_1,img_col_1,img_2,img_col_2,match_12

def
Save_Correspondences_In_Out(img_col_1,img_col_2,matches,delta,n,Probability,E,filename1,fi
lename2):
    H_RANSAC = RANSAC(matches,delta,n,Probability,E)
    plt_img_In_Out,Inliers                                                              =
Show_Correspondences_In_Outliers(img_col_1,img_col_2,matches,H_RANSAC,delta)
    filename_write = "Pair_Noise_{0}_{1}.jpg".format(filename1 , filename2)
    cv2.imwrite(filename_write,plt_img_In_Out)
    return H_RANSAC,Inliers
#Code curated from the HW2 and HW3
def transform_to_panorama(Range_img,Domain_img,H,xy_min):
    height, width = Range_img.shape[:2]
    xmin = xy_min [0]
    ymin = xy_min [1]
    H_inv = np.linalg.inv(H)
    for i in range(height):
```

```
        for j in range(width):
            k1 = j + xmin
            k2 = i + ymin
            X_domain = [k1,k2]
            X_domain = np.array(X_domain)
            X_domain = np.append(X_domain,1)
            X_range = np.matmul(H_inv, X_domain)
            X_range = X_range/X_range[-1]
            if(X_range[0] > 0 and X_range[1] > 0 and X_range[0] < Domain_img.shape[1]-1 and
X_range[1] < Domain_img.shape[0]-1):
                Range_img[i,j] = RGB_Averaged(Domain_img,X_range)

    return Range_img
def Bounds_Undistorted(Homography,image):
    #Shape of the distorted image
    image_shape = image.shape

    #Distorted Homogeneous Coordinates of image Bounds
    ImgP= np.array([0,0,1])     # Top left corner of image (X,Y,1)
    ImgQ= np.array([image_shape[1],0,1])    # Top right corner
    ImgS = np.array([image_shape[1],image_shape[0],1]) #Bottom right
    ImgR = np.array([0,image_shape[0],1])   #bottom left

    #Apply the homography on the distroted image bounds to obtain the Corrected image bounds
    WorldP = np.dot(Homography,ImgP)
    WorldP = WorldP/WorldP[2]
    WorldQ = np.dot(Homography,ImgQ)
    WorldQ = WorldQ/WorldQ[2]
    WorldS = np.dot(Homography,ImgS)
    WorldS = WorldS/WorldS[2]
    WorldR = np.dot(Homography,ImgR)
    WorldR = WorldR/WorldR[2]

    #Find the extreme points of the corrected image bounds
    max_point = np.maximum(np.maximum(np.maximum(WorldP ,WorldQ ), WorldS), WorldR)
    min_point = np.minimum (np.minimum (np.minimum(WorldP,WorldQ), WorldS), WorldR)
    #Find the coordinates of cextreme points of corrected image bounds
    xmax,ymax = max_point[0],max_point[1]
    xmin,ymin = min_point[0],min_point[1]
    return xmin, ymin,xmax,ymax

def RGB_Averaged(img,Range_point) :
    x= int(math.floor(Range_point[0]))
    xx= int (math.ceil(Range_point[0]))
    y= int (math.floor(Range_point[1]))
    yy= int (math.ceil(Range_point[1]))
```

```
    w1= 1/np.linalg.norm (np.array ([Range_point [0] -x , Range_point [1] -y]))
    w2= 1/np.linalg.norm (np.array ([Range_point [0] -x , Range_point [1] -yy]))
    w3= 1/np.linalg.norm (np.array ([Range_point [0] -xx , Range_point [1] -y]))
    w4= 1/np.linalg.norm (np.array ([Range_point [0] -xx , Range_point [1] -yy]))

    RGBVal = (w1*img [y] [x] + w2*img [yy][x] + w3*img [y] [xx] + w4*img [yy] [xx])/ (w1 +
w2 + w3 + w4)
    return RGBVal
```