# FALL 22 ECE 661 – COMPUTER VISION

Homework 5

Sahithi Kodali – 34789866
kodali1@purdue.edu

## 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?**

RANSAC (Random Sample Consensus) is a robust estimation algorithm that rejects outliers (false correspondences) among the correspondences determined in pair of images. This process is carried out by the following steps.

i.) Firstly, Homography between pairs of images (say $img_1$ and $img_2$) is computed using a few correspondences (interest points) of the images (say $pt_1$ and $pt_2$) respectively.

ii.) Using the Homography estimated, transformed points ($p_t$) of each interest points in image1 ($img_1$) is determined in the frame of image2 ($img_2$).

ii.) For all the interest points, distance between the transformed point ($p_t$) and point in that frame ($p_2$) is computed, say d.

ii.) A delta value is set as a decision threshold initially and the distance (d) is compared to the threshold to get the outliers and inliers. If distance (d) > threshold (delta), the corresponding/interest points ($p_1$,$p_2$) are said to be outliers and if distance (d) < threshold (delta), the corresponding/interest points ($p_1$,$p_2$) are said to be inliers.

**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 reasonably fast and numerically stable at the same time.**

Gradient-Descent (GD) and Gauss-Newton (GN) algorithms are used in the Nonlinear least-squares minimization for better estimate of the Homography. Though these methods perform well in minimizing the cost functions, they have some disadvantages of their own.

In GD method, gradients are computed to take a step in the steepest descent. As the minimum is approached the gradient becomes smaller and smaller in turn making the step size smaller and smaller, thus requiring more computations and hence excruciatingly slow. In GN method, the constraint is that the initial minimum solution chosen should be close to the true minimum for GN to work, and if the Jacobian computed is rank-deficient GN will fail. Hence, Levenberg-Marquardt (LM) method combines the best of these two methods and computes the minimum cost function.

In LM method, if the starting point or solution is far from the minimum, LM behaves like GD and when approaching the minimum LM behaves like GN. This is because when the product of jacobian matrix and its transpose is a diagonal, a solution that is same in GD and GN is returned.

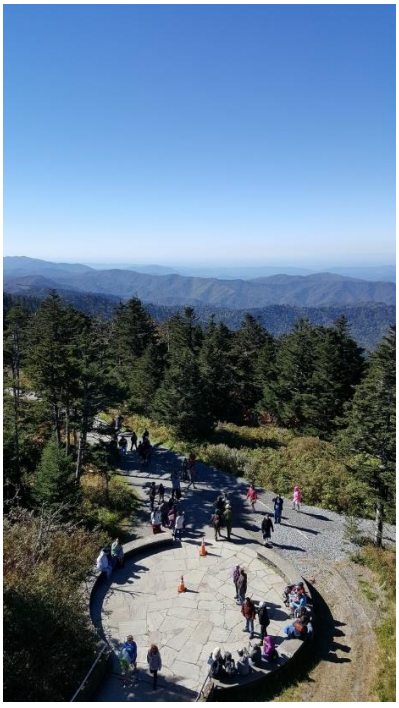
Considering this, a damping factor is added to the jacobian product to combine the best of these two methods. When the damping factor is large, the solution is strongly influenced by GD and when the damping factor is small, the solution is strongly influenced by GN thus combining the speed and stability of the methods at the same time.

The damping factor can be determined by the distance of current solution to the minimum and the cost function of the step taken towards the minimum. If the distance to minimum is large, damping factor is large and the safer convergence of GD is used by LM. However, if the distance to the minimum is smaller and the step can cross the minimum, the faster convergence capability at the minimum of GN is used by LM. In this way, LM method can combine the numerical stability of GD and fastness of GN to obtain an efficient solution.

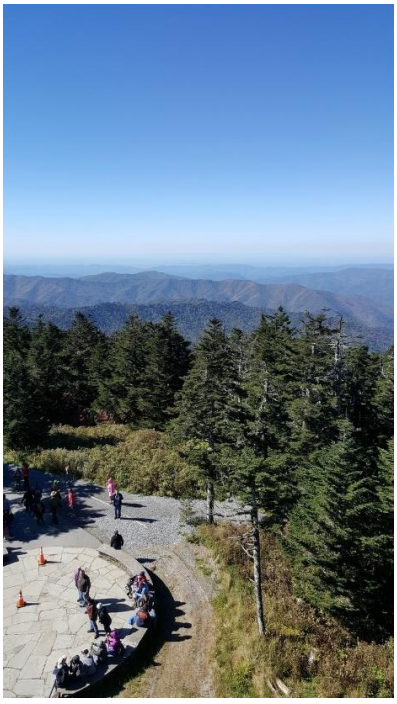
## TASK -1

Five input images shown below are provided to perform a robust homography estimation and refine it using Nonlinear Least-Squares minimization approach like LM algorithm. Further, the images are stitched to get a panorama known as image mosaicing.

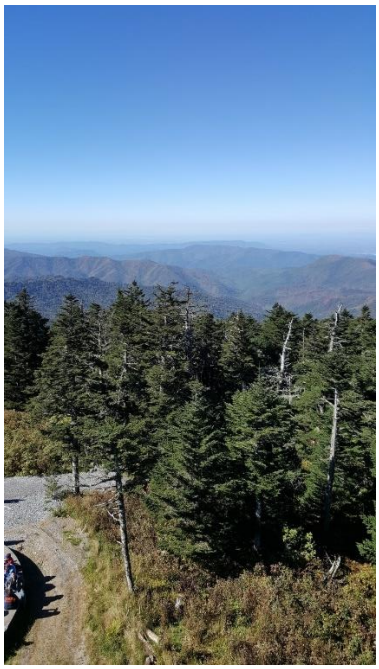
*Input image 1*



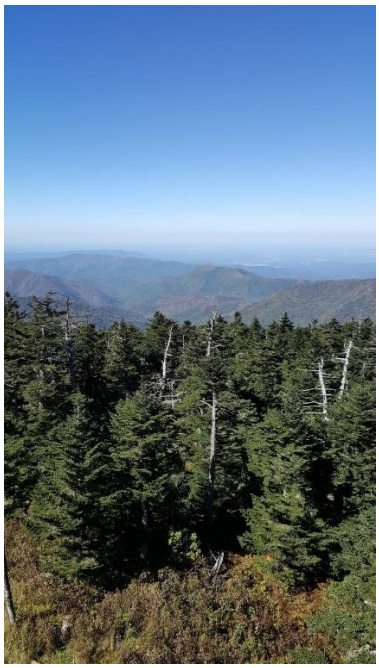
*Input image 2*



*Input image 3*



*Input image 4*



*Input image 5*

For the input images, Scale-Invariant Feature Transform (SIFT) algorithm using OpenCV built in techniques is used to detect corresponding interest points along with their feature descriptors in a pair of images. Based on a distance metric, the corresponding interest points are matched according to the shortest distance and the best match for a point in domain image is obtained as corresponding point. These points calculated using distance metric are used as the correspondences for the RANSAC algorithm.

### 1.1 RANSAC algorithm

In the correspondences determined by SIFT, there will be false correspondences (known as outliers) and RANSAC algorithm is used to determine these outliers and the inliers among these correspondences ($n_{total}$ ). RANSAC algorithm can be implemented by following the steps below.

- Firstly, some initializations need to be performed for obtaining the inliers. A value delta ($\delta = 3$) is given as a decision threshold for obtaining the inlier set. The number of correspondences (n = 6) to be chosen for each trial, the probability of choosing an outlier points (epsilon $\epsilon = 0.3$) and the probability (p = 0.99) that at least one of these trails will give an outlier free Homography computation are initialized.
- The number of trails N and the minimum size of the inlier set that can be acceptable M are calculated using the initialized values as below,

$$N = \frac{\ln(1-p)}{\ln\left(1-(1-\epsilon)^n\right.} \quad and \quad M = (1-\epsilon)n_{total}$$

- In each trail, 'n' correspondences among all the correspondences ($n_{total}$) are selected randomly and their Homography is computed for these using the Linear Least square minimization method discussed in section 1.2 and the inliers obtained among all the correspondences for the computed Homography are calculated. This is carried out for N number of times.
- The inliers are computed by computing HX for each point X in the correspondences of form (X, X`). The difference between the original X` and computed HX is computed as $(\Delta x, \Delta y)$. This is followed by computing the squared distance (d) as $d^2 = \Delta x^2 + \Delta y^2$. The correspondences satisfying $d^2 \leq \delta^2$ are obtained as the inlier set in each trail.
- The correspondences of the largest inlier set obtained among the N trails is compared with the minimum size M and if satisfied is used to compute the final Homography.
- This largest inlier set obtained is further used for Homography refining using Nonlinear least squares minimization using LM method discussed in section 1.3.

### 1.2 Linear Least-Squares method

Linear Least-squares method is used in the RANSAC algorithm to compute Homography as discussed previously. As we know, to project an image onto a world image frame, we must find the relationship between the corresponding points on these image frames. If the homogeneous representation of a point in the domain space/image plane and the projective space/world plane are $\vec{x}\,(x, y, 1)$ and $\vec{x}'(x', y', 1)$ respectively, then the relationship between these two points is represented as,

$$\vec{x}' = H\vec{x}$$

where, H is a non-singular 3x3 matrix known as Homography matrix and can be written as,

$$\mathrm{H} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Since, Homography matrix is a linear mapping of homogeneous coordinates we only consider the ratios of the coordinates i.e., if point ($x_1$, $x_2$, $x_3$) exists then it is same as point ($x_1/x_3$, $x_2/x_3$,1) which is as in the representation of the points $\vec{x}$ and $\vec{x}'$. Hence, the coordinate $H_{(3,3)}$ = i = 1.

From above,

$$\vec{x}' = H\vec{x}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$x' = \mathrm{ax} + \mathrm{by} + \mathrm{c}$$

$$y' = \mathrm{dx} + \mathrm{ey} + \mathrm{f}$$

$$1 = \mathrm{gx} + \mathrm{hy} + 1$$

Thus, the x' and y' coordinates can be written as,

$$x' = \frac{ax+by+c}{gx+hy+1} => gxx' + hyx' + x' = ax + by + c => x' = ax + by + c - gxx' - hyx'$$

$$y' = \frac{dx+ey+f}{gx+hy+1} => gxy' + hyy' + y' = dx + ey + f => y' = dx + ey + f - gxy' - hyy'$$

The equations of $x'$ and $y'$ contains 8 unknowns and the known number of correspondences are n i.e. ($x_i$, $y_i$) for i = 1, 2…., n giving 2n equations.

$$\text{AH} = \text{B} => \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_nx_n' & -y_nx'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -x_ny_n' & -y_ny_n' \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ . \\ . \\ x_n' \\ y_n' \end{bmatrix}$$

Here, A is of order 2n X 8 and for computing a Homography matrix for the above with more than 4 correspondences, the linear least squares method based on pseudo inverse is used as below,

$$\text{H} = (\text{A}^\text{T}\,\text{A})^{-1}\,\text{A}^\text{T}\,X' \quad \text{where pseudoinverse } \text{A}^+ = (\text{A}^\text{T}\,\text{A})^{-1}\,\text{A}^\text{T}$$

Thus, the Homography H is computed and reshaped to a (3 X 3) matrix by adding $h_{33}$ =1.

### 1.3 Levenberg-Marquardt (LM) algorithm

The Homography computed using RANSAC algorithm is refined using the LM algorithm of Nonlinear least-squares minimization.

The inliers obtained by RANSAC (X) and their mapping (f) using H are used for optimization. For N correspondences these can be given as below,

$$X = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ . \\ . \\ x_n' \\ y_n' \end{bmatrix} \quad \text{f} = \begin{bmatrix} f_1^1 \\ f_2^1 \\ f_1^2 \\ f_2^2 \\ . \\ . \\ f_1^N \\ f_2^N \end{bmatrix} \text{ where, } f_1^i = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \quad \text{and} \quad f_2^i = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}}$$

This optimization can be given as,

$$\min_{(h)} ||X - f(h)||^2 \quad \text{where} \quad h = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}$$

Further using the initial H computed using RANSAC algorithm LM method is carried out for optimization on $E = X - f(h)$ and the Jacobian of E is $J_E = - J_f$

The partial derivatives on $f_1^i$ and $f_2^i$ with respect to elements in 'h' are computed (detailed in source code) and used for computing the Jacobian matrix for optimization in LM method.

The refined 'h' computed is used to obtain a panorama of image by performing image mosaicing.

**Implementation:**

LM method is implemented by the steps below ("from scratch").

An initial guess for homography ($p_0$) is taken for pk.

- The Jacobian of E at $p_0$ ($J_E$) is determined using the partial differentiations of function(f) and the damping coefficient $u_k$ is initialized using $u_k = T * \max \{\text{diagonal } (J_E^T J_E)\}$ where $0 < T <= 1$.
- The step increment $\delta_p$ in $p_k$, $p_{k+1}$, the ratio $\rho$ (change in cost function) for determining damping coefficient and $u_{k+1}$ for next step are calculated as below,

$$\delta_p = - (J_E^T J_E + u_k I)^{-1} J_E^T E(p_k)$$

$$p_{k+1} = p_k + \delta_p$$

$$\rho = (E(p_k)^T E(p_k) - E(p_{k+1})^T E(p_{k+1})) / \delta_p J_E^T E(p_k) + \delta_p{}^T u_k \delta_p$$

$$u_{k+1} = u_k * \max (1/3, 1- (2\rho - 1)^3)$$

- Change the damping coefficient to current and if the difference of cost (numerator in $\rho$) is greater than zero, change $p_k$ to $p_{k+1}$

Repeat these steps until the LM threshold initialized for the iterations is greater than the step increments $\delta_p$ .

### => Image mosaicing

The final step of the task is to generate a panorama using image stitching known as image mosaicing. To produce a panorama, all the images are projected on a common centre frame and the image pair homographies are used to produce the stitched image. In this task, we have 5 images and the following steps are followed to get a panorama (works for any number of images).

- All the images coordinates are shifted to be projected onto a centre frame of reference. This is done by reducing left and right side of the images separately to project onto centre frame coordinates.
- The homographies of the image pairs is computed namely $H_{12}$, $H_{23}$, $H_{34}$, $H_{45}$ and the homographies from these to the middle image 3 is obtained by the product of homographies as below,

$$H_{13} = H_{23} * H_{12}$$
$$H_{43} = H_{34}^{-1}$$
$$H_{53} = H_{34}^{-1} * H_{45}^{-1}$$

- The homographies are translated by multiplying with a translation matrix to bring all the images with reference to the centre image frame and project onto the canvas.
- The translated images are now stitched by finding the maximum height of the images and the width is extended to the size of sum of all the images widths which works as a canvas to project the images.
- Further, all the images are projected onto this canvas using the pixel coordinates and the corresponding weighted pixel values giving a panorama.

### Results

*Figure 1: SIFT correspondences for input images 1,2*

*Figure 2: Inliers (green) and outliers (white) in the input images 1,2*

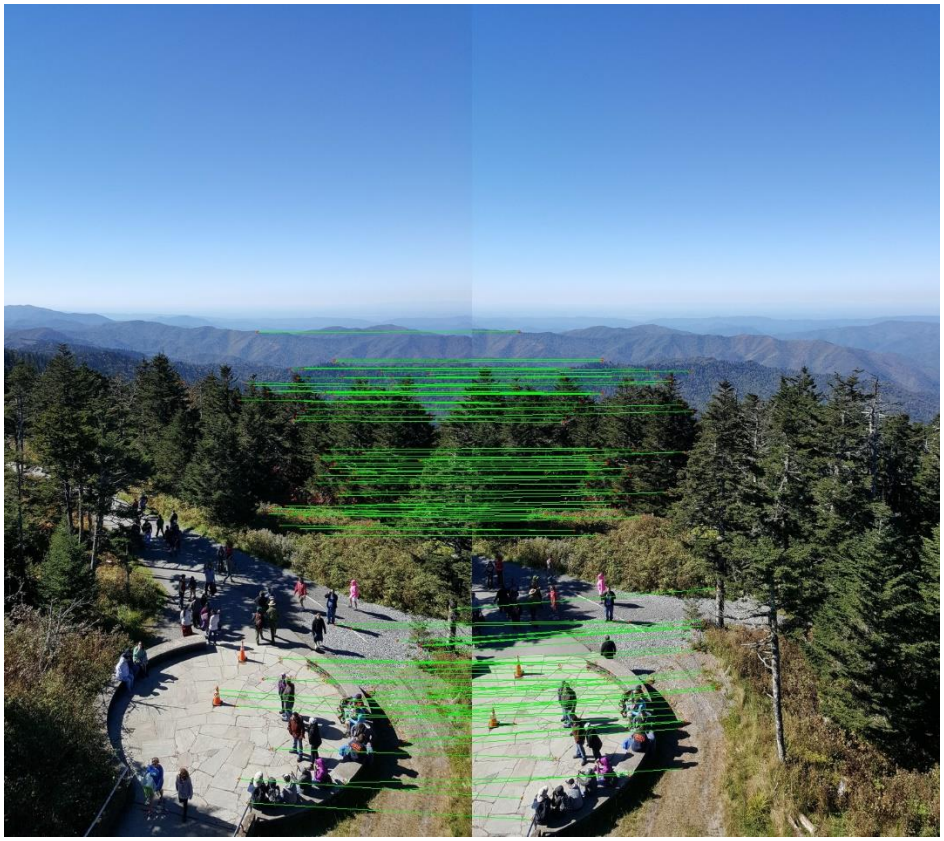


*Figure 3: SIFT correspondences for input images 2,3*

*Figure 4: Inliers (green) and outliers (white) in the input images 2,3*

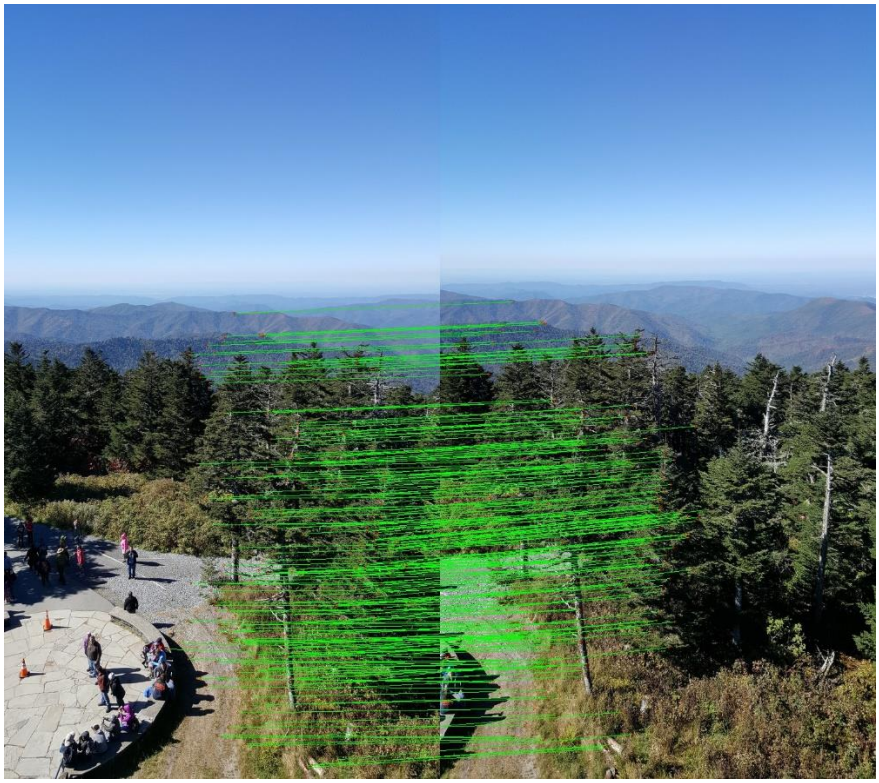


*Figure 5: SIFT correspondences for input images 3,4*

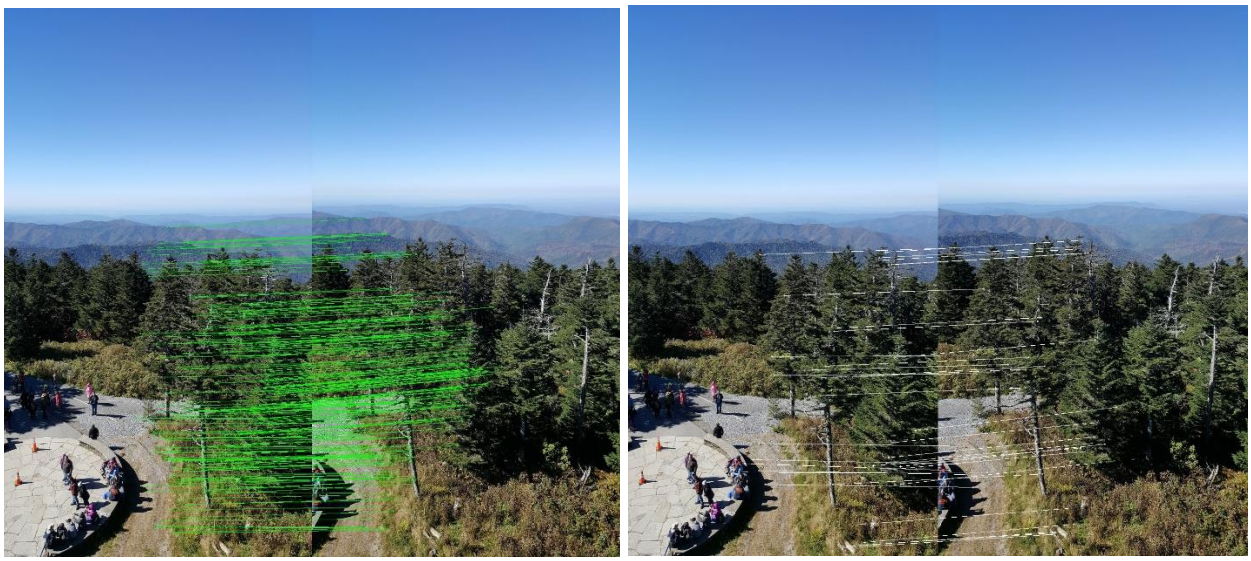


*Figure 6: Inliers (green) and outliers (white) in the input images 3,4*

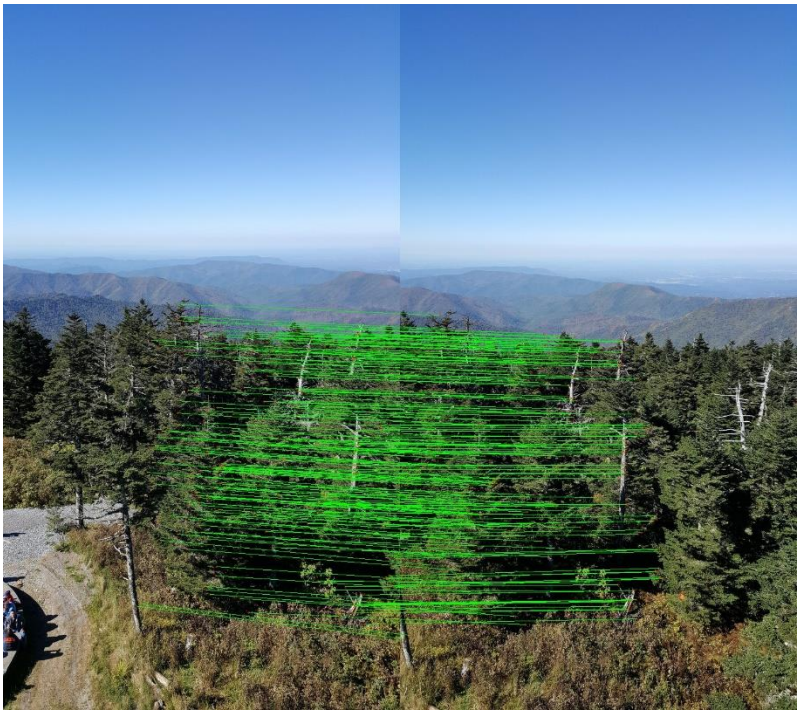


*Figure 7: SIFT correspondences for input images 4,5*

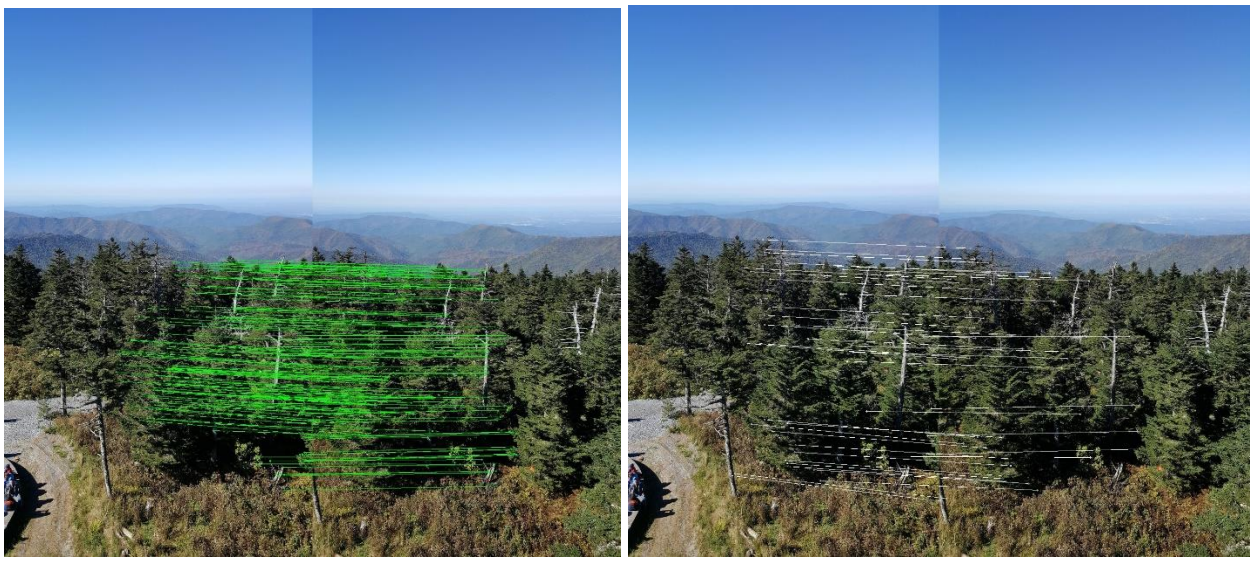


*Figure 8: Inliers (green) and outliers (white) in the input images 4,5*

*Figure 9: Panorama after stitching the images without using LM algorithm*

*Figure 10: Panorama after stitching the images with using LM algorithm*

We can observe that the images using and not LM method are pretty similar in this case. The only observation is a slight reduce in the seam lines, but otherwise the RANSAC algorithm can be said as a robust algorithm.

## TASK-2

All the methods similar to task-1 are performed on a set of five own images to obtain a panorama.

*Input image 6*

*Input image 7*

*Input image 8*

*Input image 9*

*Input image 10*

## Results

*Figure 11: SIFT correspondences for input images 6,7*

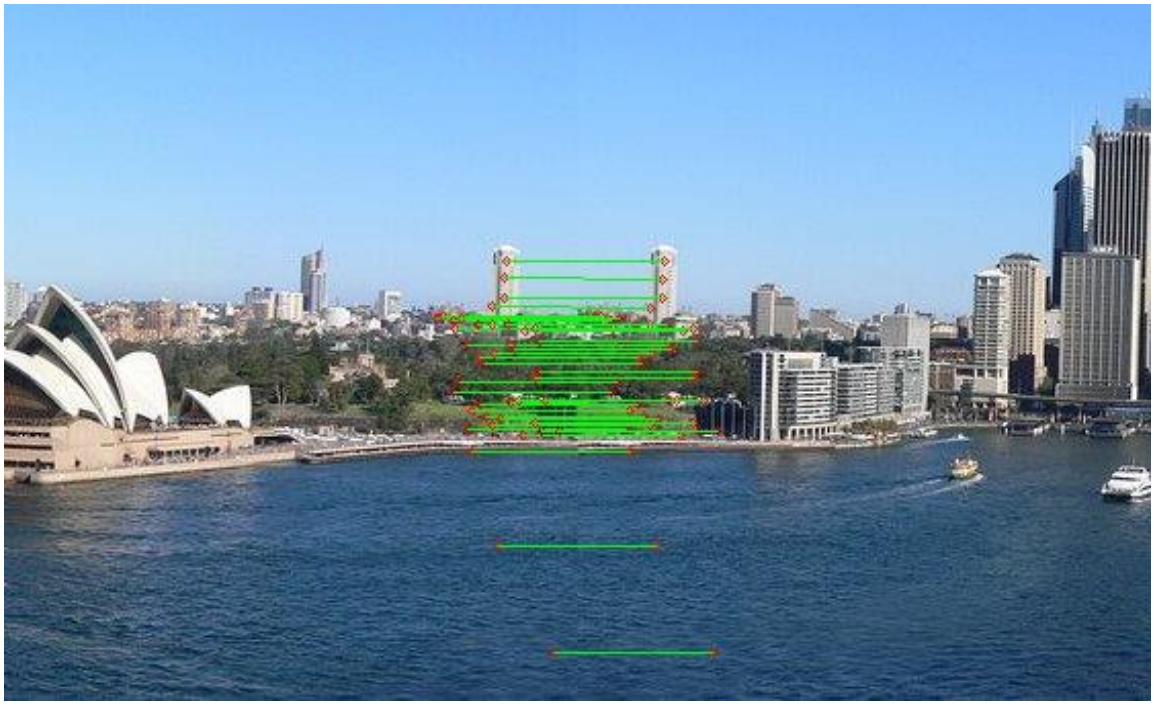


*Figure 12: SIFT correspondences for input images 7,8*

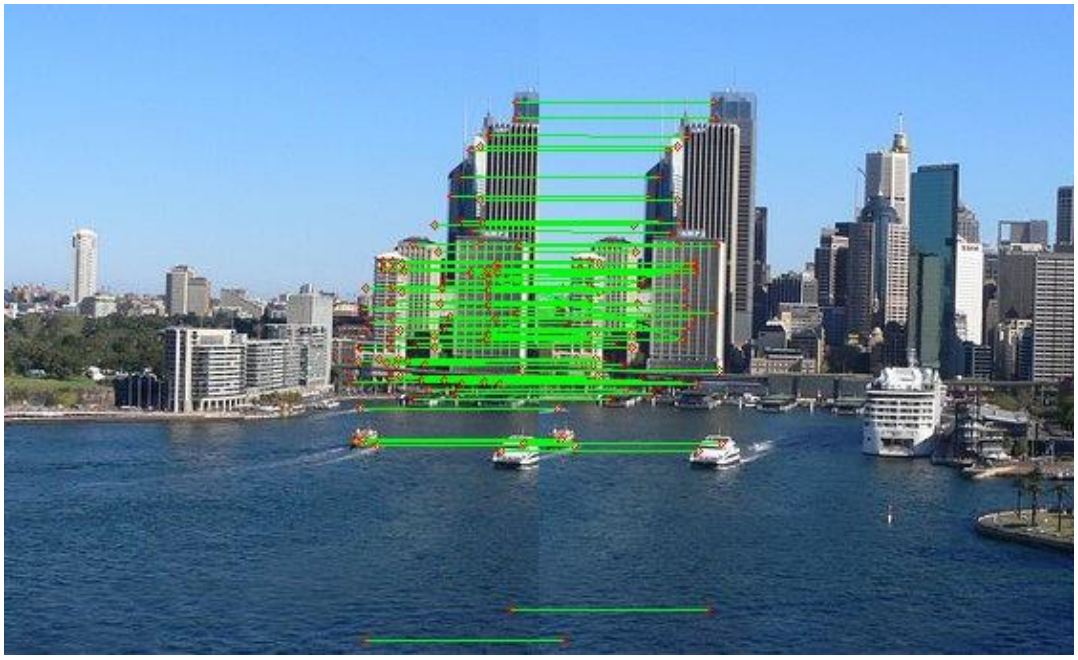


*Figure 13: SIFT correspondences for input images 8,9*

*Figure 14: Inliers (green) and outliers (white) in the input images 8,9*

*Figure 15: SIFT correspondences for input images 9,10*

*Figure 16: Inliers (green) and outliers (white) in the input images 9,10*

*Figure 17: Panorama after stitching the images without using LM algorithm*

*Figure 18: Panorama after stitching the images with using LM algorithm*

In the two images using LM, we can observe there is a very slight difference in the sharp lines but otherwise RANSAC performed as good as LM method for these images.

## Source Code:

```
#!/usr/bin/env python
# coding: utf-8

# <h2><center>ECE661 COMPUTER VISION</center></h2>
# <h3><center>Homework - 5 </center></h3>
# <h3><center>Sahithi Kodali - 34789866</center></h3>
# <h3><center>kodali1@purdue.edu</center></h3>

# In[1]:


# Import libraries needed

import numpy as np
import matplotlib.pyplot as plt
import skimage.io as sk
import cv2
import math


# In[65]:


#Image files list
images_all = ["0.jpg", "1.jpg", "2.jpg", "3.jpg", "4.jpg"]
# own_images_all = ["01.jpg", "02.jpg", "03.jpg", "04.jpg", "05.jpg"]


# In[27]:
```

```
#convert image into grey scale of channel 1
def normalize_img2(image):
    if len (image.shape) == 3:
        norm_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        norm_img = image
    return norm_img


# In[28]:


#Compute the distance metrics SSD and NCC to compute distance required to match
corresponsing interest points in the image pair

def distance_metric(w1,w2, metric = 'SSD'):

    #compute SSD and NCC accoording to the formula in the theory concepts
    if metric == 'SSD':
        distance = np.sum( (w1 - w2) ** 2)
    elif metric == 'NCC':
        mean1 = w1.mean()
        mean2 = w2.mean()

        ncc_num = np.sum ((w1 - mean1) * (w2 - mean2))
        ncc_den = np.sqrt (np.sum((w1 - mean1) ** 2) * np.sum((w2 - mean2) ** 2))

        distance = 1 - (ncc_num/ncc_denom)

    return distance


# In[29]:


#compute the SIFT matching using OpenCV techniques

def SIFT(image1, image2):

    img1_read = cv2.imread(image1)
    img2_read = cv2.imread(image2)

    img1 = normalize_img2(img1_read)
    img2 = normalize_img2(img2_read)
```

```
    #create SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    #compute the keypoints and descriptors for the respective images
    kps1, desc1 = sift.detectAndCompute(img1, None)
    kps2, desc2 = sift.detectAndCompute(img2, None)

    #finding correspondences
    images_concat = np.concatenate((img1_read, img2_read), axis = 1)

    domain_coord = np.array([[0,0]])
    range_coord = np.array([[0,0]])

    #get the best match fro each point in image1
    for i in range(desc1.shape[0]):
        dist_tmp = []
        kp1 = kps1[i].pt

        for j in range(desc2.shape[0]):
            kp2 = kps2[j].pt
            dist = distance_metric(desc1[i], desc2[j])
            dist_tmp.append(dist)

        best_kp2_coord = list(kps2[np.argsort(dist_tmp)[0]].pt)
        best_kp2_coord = list(map(int, best_kp2_coord))

        if np.min(dist_tmp) < 10000:
            domain_coord = np.append(domain_coord, [list(map(int, kp1))], axis =
0)
            range_coord = np.append(range_coord, [best_kp2_coord], axis = 0)

            final_kp1 = tuple(list(map(int, kp1)))
            final_kp2 = (best_kp2_coord[0] + img1_read.shape[1],
best_kp2_coord[1])

            #draw the matching correspondences
            cv2.circle (images_concat, final_kp1 , 2, (0,0,255) , 1)
            cv2.circle (images_concat, final_kp2 , 2, (0,0,255) , 1)
            cv2.line (images_concat, final_kp1 , final_kp2, (0,255,0) , 1)

    #remove the first coordinates that are initialized as (0,0)
    domain_coord = np.delete(domain_coord, 0, axis = 0)
    range_coord = np.delete(range_coord, 0, axis = 0)
```

```
    return images_concat, domain_coord, range_coord


# In[30]:


# Homography matrix for overdetermined system using Linear Least-Squares
Minimization (Inhomogeneous equations)

def homo_matrix_LS(X,X_prime):

    #give the matrix A shape
    n = X.shape[0]
    A = np.zeros((2*n,8))

    #fill the matrix A
    for i in range(n):
        A[2*i,0] = X[i,0]
        A[2*i,1] = X[i,1]
        A[2*i,2] = 1
        A[2*i,6] = -1*X[i,0]*X_prime[i,0]
        A[2*i,7] = -1*X[i,1]*X_prime[i,0]
        A[2*i+1,3] = X[i,0]
        A[2*i+1,4] = X[i,1]
        A[2*i+1,5] = 1
        A[2*i+1,6] = -1*X[i,0]*X_prime[i,1]
        A[2*i+1,7] = -1*X[i,1]*X_prime[i,1]


    B = X_prime.copy()
    B = B.reshape((-1,1))

    #compute the H = A_pseudoInverse * b, for each correspondences and obtain the
Homography matrix
    A_pseudoInverse = np.matmul(np.linalg.inv(np.dot(A.T,A)), A.T)
    K = np.dot(A_pseudoInverse, B)

    H = np.concatenate((K,np.array([[1]])),axis = 0)
    H = H.reshape((3,3))

    return H


# In[110]:
```

```
# Ransac algorithm to return the Homography using the best inlier set obtained

def ransac(domain_coord, range_coord):

    # Initializations (refer to theory)
    delta = 2
    n = 6
    eps = 0.4
    prob = 0.99

    # Total number of correspondences
    n_total = domain_coord.shape[0]

    # Calculate number of trails
    N = int((math.log(1 - prob)/math.log(1 - (1 - eps) ** n)))

    # Calculate Minimum size of the inlier set
    M = int(((1 - eps) * n_total))

    best_inlier_indxs = []
    inlier_percent = 0
    indices_range = list(range(n_total))

    #Check for the maximum inlier set > M among the trails
    for trail in range(N):
        rand_corr_indices = np.random.choice(indices_range,n)

        rand_domain_coord = domain_coord[rand_corr_indices]
        rand_range_coord = range_coord[rand_corr_indices]

        H_initial = homo_matrix_LS(rand_domain_coord, rand_range_coord)

        inliers_tmp_indices = find_inliers(H_initial, domain_coord, range_coord,
delta)
        num_inliers = inliers_tmp_indices.shape[0]

        if num_inliers >= M and num_inliers/n_total > inlier_percent:
            inlier_percent = num_inliers/n_total
            best_inlier_indxs = inliers_tmp_indices

#     print('Best inliers are')
#     print(best_inlier_indxs)

    # compute the final homography using the best inlier set obtained
```

```
    H_final = homo_matrix_LS(domain_coord[best_inlier_indxs],
range_coord[best_inlier_indxs])
#     print(H_final)

    return H_final, best_inlier_indxs



# In[111]:


# Function to find the inlier set by comparing the difference of original and
transformed coordinates to delta (distance threshold)

def find_inliers(H_initial, domain_coord, range_coord, delta):

    domain_coord_HC = np.insert(domain_coord, 2, 1, axis = 1)
    range_coord_HC = np.insert(range_coord, 2, 1, axis = 1)

    Y = np.matmul(H_initial, domain_coord_HC.T).T
    Y = Y.T/Y.T[2,:]

    #calculate error and use it to get the distance
    error = (np.abs(Y.T - range_coord_HC))**2
    dist_squared = np.sum(error, axis = 1)

    #obtian the best inlier coordinates indexes
    inlier_indices = np.where(dist_squared <= delta ** 2)[0]

    return inlier_indices


# In[112]:


# Plot the inliers and outliers in a pair of images

def plot_inliers_outliers(image1, image2, domain_coord, range_coord,
inlier_idxs):

    #read images
    img1_read = cv2.imread(image1)
    img2_read = cv2.imread(image2)

    #create image frame
    images_concat_inliers = np.concatenate((img1_read, img2_read), axis = 1)
```

```
    images_concat_outliers = np.concatenate((img1_read, img2_read), axis = 1)

    #plot for the inliers and outliers saparately
    for idx in range(domain_coord.shape[0]):
        if idx in inlier_idxs:
            pt1 = tuple(domain_coord[idx])
            pt2 = range_coord[idx]
            pt2[0] = pt2[0] + img1_read.shape[1]
            pt2 = tuple(pt2)

            cv2.circle (images_concat_inliers, pt1 , 2, (0,255,0) , 1)
            cv2.circle (images_concat_inliers, pt2 , 2, (0,255,0) , 1)
            cv2.line (images_concat_inliers, pt1 , pt2, (0,255,0) , 1)

        else:
            pt1 = tuple(domain_coord[idx])
            pt2 = range_coord[idx]
            pt2[0] = pt2[0] + img1_read.shape[1]
            pt2 = tuple(pt2)

            cv2.circle (images_concat_outliers, pt1 , 2, (255,255,255) , 1)
            cv2.circle (images_concat_outliers, pt2 , 2, (255,255,255) , 1)
            cv2.line (images_concat_outliers, pt1 , pt2, (255,255,255) , 1)

    return images_concat_inliers, images_concat_outliers


# In[113]:


# Levenberg-Marquardt (LM) algorithm initialization using the LM threshold

def LM_homo(H, best_indices, domain_coord, range_coord):
    H_initial = np.ravel(H)

    H_tmp,cost = LM_algo(H_initial, jacob_cost_func, args =
(domain_coord[best_indices], range_coord[best_indices]), delta_threshold =
0.0001)
    H_refined = H_tmp/H_tmp[-1]

    H_refined = H_refined.reshape(3,3)

    return H_refined
```

```
# In[156]:


# Levenberg-Marquardt (LM) algorithm to give the refined homography and the
cost(geometric error)
def LM_algo(H_p0, j_cost_func, args = None, delta_threshold=0.0001):

    #check for inliers to call the Jacobian cost function
    def check_inliers_exist(H, args):
        if args == None:
            return j_cost_func(H)
        else:
            return j_cost_func(H, args[0], args[1])

    #initialize T (usually between 0.5 and 2)
    T = 0.5

    #intialize intital H (p0) and find the cost and jacobian
    pk = H_p0
    [e, Je] = check_inliers_exist(H_p0, args)

    #initialize the damping coefficient
    JtJ = np.matmul(Je.T, Je)
    u0 = T * np.diag(JtJ).max()
    uk = u0

    I = np.eye(JtJ.shape[0])

    # until the LM threshold is not reached, repeat the steps of computing step,
Jacobian, cost function and damping coefficent
    # obtain the new reduced geometric error (cost) and the refined homography
    while(True):
        [e,Je] = check_inliers_exist(pk, args)

        JtJ = np.matmul(Je.T, Je)
        JTE = - np.matmul(Je.T, e)

        delta_p = np.matmul(np.linalg.inv(JtJ + uk * I), JTE)

        pk_1 = pk + delta_p

        [e_next, Je_next] = check_inliers_exist(pk_1, args)

        initial_cost = np.matmul(e.T, e)
#         print(initial_cost)
```

```
        new_cost = np.matmul(e_next.T, e_next)
#         print(new_cost)

        ratio_num = initial_cost - new_cost
        ratio_denom = np.matmul(delta_p.T, JTE) + np.matmul(delta_p.T, uk *
delta_p)
        LM_ratio = ratio_num / ratio_denom

        uk_1 = uk * max(1/3, 1 - ((2 * LM_ratio) - 1)**3)
        uk = uk_1

        if ratio_num >= 0:
            pk = pk_1

            if np.linalg.norm(delta_p) < delta_threshold:
                break


    return pk, new_cost


# In[157]:


# Function to compute the partial differences required for Jacobian matrix
def jacob_cost_func(h, domain_coord, range_coord):

    X = []
    F = []
    J = []

    for i in range(domain_coord.shape[0]):

        p1 = domain_coord[i]
        p2 = range_coord[i]

        X.extend([p2[0], p2[1]])

        f1num = h[0] * p1[0] + h[1] * p1[1] + h[2]
        f2num = h[3] * p1[0] + h[4] * p1[1] + h[5]
        fdenom = h[6] * p1[0] + h[7] * p1[1] + h[8]
        F.extend([f1num/fdenom, f2num/fdenom])

        ele1 = -f1num/fdenom**2
```

```
        ele2 = -f2num/fdenom**2
        J.extend([p1[0]/fdenom, p1[1]/fdenom, 1/fdenom, 0, 0, 0, p1[0] * ele1,
p1[1] * ele2, ele1])
        J.extend([0, 0, 0, p1[0]/fdenom, p1[1]/fdenom, 1/fdenom, p1[0] * ele1,
p1[1] * ele2, ele2])

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

    #error/cost is determined
    E = X - F
    cost = E
    E = E.reshape(-1,1)

    return cost, jacobian


# In[167]:


#Get the homography and the images plots for each pair of images given/not LM
def imgPairs_H(img1, img2, LM = False):
    SIFT_img, domain_coord, range_coord = SIFT(img1, img2)
    H, best_indices = ransac(domain_coord, range_coord)
    inliers_img, outliers_img = plot_inliers_outliers(img1, img2, domain_coord,
range_coord, best_indices)

    if LM:
        H = LM_homo(H, best_indices, domain_coord, range_coord)

    return H, SIFT_img, inliers_img, outliers_img


# In[168]:


# Dtermine the valid pixel coordinates using x, y min and max values
def get_validPixels(coord, H_coord, x_min, x_max, y_min, y_max):

    i = H_coord[0, :] >= x_min
    coord = coord[:, i]
    H_coord = H_coord[:, i]
```

```
    i = H_coord[0, :] <= x_max
    coord = coord[:, i]
    H_coord = H_coord[:, i]

    i = H_coord[1, :] >= y_min
    coord = coord[:, i]
    H_coord = H_coord[:, i]

    i = H_coord[1, :] <= y_max
    coord = coord[:, i]
    H_coord = H_coord[:, i]

    return [coord, H_coord]


# In[169]:


#Obtain the weighted pixel values of the valid coordinates
def get_weighted_PixelValues(img, valid_coord):

    coord = np.array([valid_coord[0], valid_coord[1]])
    x = int(np.floor(valid_coord[0]))
    y = int(np.floor(valid_coord[1]))

    #computer distance using L2 normalization
    d_00 = np.linalg.norm(coord - np.array([x,y]))
    d_01 = np.linalg.norm(coord - np.array([x,y+1]))
    d_10 = np.linalg.norm(coord - np.array([x+1,y]))
    d_11 = np.linalg.norm(coord - np.array([x+1,y+1]))

    w_num = img[y][x][:] * d_00 + img[y+1][x][:] * d_01 + img[y][x+1][:] * d_10 +
img[y+1][x+1][:] * d_11
    w_den = d_00 + d_01 + d_10 + d_11

    w_pixelVal = w_num/w_den

    return w_pixelVal



# In[170]:


#Function that maps the changed homographies to the whole canvas in the center
frame
```

```
def mapping_img2canvas(img, canvas, H_pair):

    h = img.shape[0]
    w = img.shape[1]

    canvas_h = canvas.shape[0]
    canvas_w = canvas.shape[1]

    H = np.linalg.inv(H_pair)

    #original canvas coordinates
    canvas_coord = np.array([[i,j,1] for i in range(canvas_w) for j in
range(canvas_h)])
    canvas_coord = canvas_coord.T

    #transformed canvas coordinates
    coord_translated = np.dot(H, canvas_coord)
    coord_translated = coord_translated/coord_translated[2, :]

    #check the validity of the coordinates
    [valid_canvas_coord, valid_canvas_translated] = get_validPixels(canvas_coord,
coord_translated, 0, w - 2, 0, h - 2)

    #obtian the pixel values for the valid points and project onto canvas
    for i in range(valid_canvas_coord.shape[1]):

        valid_point_coord = valid_canvas_translated[:, i]
        canvas[valid_canvas_coord[1][i]][valid_canvas_coord[0][i]][:] =
get_weighted_PixelValues(img, valid_point_coord)

    return canvas



# In[171]:


# function to get the homographies of image pairs and transformed to middle frame
# Further the transformed images are stitched together to get a panorama

def panorama(images_all):

    num_images = len(images_all)
    H_pairs = []
```

```
    #get all image pairs homographies
    for i in range(num_images - 1):
        H, SIFT_img, inliers_img, outliers_img = imgPairs_H(images_all[i],
images_all[i+1])

        #Save the SIFT, iniliers and outliers images for each image pair adn
append all homographies
        filename1 = 'D:\Purdue\ECE661_CV\HW5_outputs\SIFT' + str(i) + str(i+1)
+'.jpg'
        cv2.imwrite(filename1 ,  SIFT_img)

        filename2 = 'D:\Purdue\ECE661_CV\HW5_outputs\Inliers' + str(i) + str(i+1)
+'.jpg'
        cv2.imwrite(filename2,  inliers_img)

        filename3 = 'D:\Purdue\ECE661_CV\HW5_outputs\Outliers' + str(i) +
str(i+1) +'.jpg'
        cv2.imwrite(filename3,  outliers_img)

        H_pairs.append(H)

#     print('appended Hpairs')
#     print(len(H_pairs))
#     print(H_pairs)

    #get the center image location
    center_img_loc = int((num_images + 1)/2) - 1

    #shifting right half imgs to center frame
    H_ref = np.eye(3, dtype = np.float64)

    for i in range(center_img_loc, len(H_pairs)):
        H_ref = np.matmul(H_ref,np.linalg.inv(H_pairs[i]))
        H_pairs[i] = H_ref

    #shifting left half imgs to center frame
    H_ref = np.eye(3, dtype = np.float64)

    for i in range(center_img_loc - 1, -1, -1):
        H_ref = np.matmul(H_ref,H_pairs[i])
        H_pairs[i] = H_ref

    #insert center_img frame homography
    H_pairs.insert(center_img_loc, np.eye(3, dtype = np.float64))
#     print('centered Hpairs')
```

```
#     print(H_pairs)

    #Translating the center image location to middle frame of panorama and
translate the homographies of image pairs
    tx = 0

    for i in range(center_img_loc):
        each_img = cv2.imread(images_all[i])
        tx = tx + each_img.shape[1]

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

    for i in range(len(H_pairs)):
        H_pairs[i] = np.matmul(H_translated, H_pairs[i])

#     print('Translated Hpairs')
#     print(H_pairs)

    #stitch the images using their new translated homographies
    h = 0
    w = 0
    all_imgs_stitching = []

    for img in images_all:
        each_img = cv2.imread(img)
        h = max(h, each_img.shape[0])
        w = w + each_img.shape[1]
        all_imgs_stitching.append(each_img)

    panorama_canvas = np.zeros((h,w,3), np.uint8)

    for img_idx, each_img in enumerate(all_imgs_stitching):
        panorama_mapped = mapping_img2canvas(each_img, panorama_canvas,
H_pairs[img_idx])

    return panorama_mapped


# In[174]:


panorama_result = panorama(images_all)
```

```
# In[175]:


#Save the panorama images with LM
filename = 'D:\Purdue\ECE661_CV\HW5_outputs' + '\panorama_withLM' + 'given_imgs'
+'.jpg'
cv2.imwrite(filename,  panorama_result)


# In[176]:


# #Save the panorama images without LM
# filename = 'D:\Purdue\ECE661_CV\HW5_outputs' + '\panorama_withoutLM' +
'given_imgs' +'.jpg'
# cv2.imwrite(filename,  panorama_result)


# ### Task -2

# In[166]:


own_images_all = ["001.jpg", "002.jpg", "003.jpg", "004.jpg", "005.jpg"]
panorama_result_own = panorama(own_images_all)


# In[109]:


#Save the panorama image of own images with LM
filename = 'D:\Purdue\ECE661_CV\HW5_outputs' + '\panorama_withLM' + 'own_imgs'
+'.jpg'
cv2.imwrite(filename,  panorama_result_own)


# In[172]:


# #Save the panorama image of own images without LM
# filename = 'D:\Purdue\ECE661_CV\HW5_outputs' + '\panorama_withoutLM' +
'own_imgs' +'.jpg'
# cv2.imwrite(filename,  panorama_result_own)
```

******