ECE661: Homework 5

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

October 10, 2018

For this Homework we have to automatically compute Homographies between two images using RANSAC & linear least squares and subsequently refine it using nonlinear least-squares optimization using Levenberg-Marquardt algorithm. We have to then use our code to stitch together 5 images which have sufficient overlap.

1 Logic and Methodology

The task of this homework can be further divided into the following sub-tasks:

a) Linear Least squares Homography estimation

b) RANSAC for finding inliers

c) Non-Linear Least Squares optimization using Levenberg-Marquardt

d) Image mosiacing to generate panoramic image

1.1 Linear Least Squares Homography

For this sub-problem we have the equation $H * x = x'$;where H is the homography, x is the homogeneous coordinate of a point at the source image and x' is the homogeneous coordinate of a point at the destination image.

Now, we need to find the elements of the H,$3x3$ matrix. Also, as any other homography $k*H$ is equivalent to H and thus the information given by H is only in the ratios of the elements. Therefore, We can assume the $(3,3)$ index of H as 1, and will only need to find 8 other elements of H namely $h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}$ & h_{32}.

For any point $x = [x_1, y_1, 1]^T$ we have the equation:

$$H * x = x'$$

$$\Rightarrow \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix}$$

Thus we get:

$$h_{11}x_1 + h_{12}y_1 + h_{13} = x'_1$$
$$h_{21}x_1 + h_{22}y_1 + h_{23} = x'_2$$

$$h_{31}x_1 + h_{32}y_1 + 1 = x'_3$$

Now in the above equation we have the information of only the $\mathbb{R}^2$ coordinates. That is $x' = x'_1/x'_3$ and $y' = x'_2/x'_3$. Therefore, in order to make the right hand side as a vector of known quantities we need to divide the matrix equation by the scalar x'_3

Thus we get:

$$\frac{1}{(h_{31}x_1 + h_{32}y_1 + 1)} * \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ 1 \end{bmatrix}$$

Writing individual equation we get:

$$x'_1 = \frac{(h_{11}x_1 + h_{12}y_1 + h_{13})}{(h_{31}x_1 + h_{32}y_1 + 1)}$$

$$\Rightarrow h_{11}x_1 + h_{12}y_1 + h_{13} - h_{31}x_1x'_1 - h_{32}y_1x'_1 = x'_1$$

$$y'_1 = \frac{(h_{21}x_1 + h_{22}y_1 + h_{23})}{(h_{31}x_1 + h_{32}y_1 + 1)}$$

$$\Rightarrow h_{21}x_1 + h_{22}y_1 + h_{23} - h_{31}x_1y'_1 - h_{32}y_1y'_1 = y'_1$$

Note that, in the above equations we need to solve h_{ij}, therefore we will need atleast 4 more points in order to get atleast 8 equations to solve the 8 unknowns.

We can write out the system of equations in the form of $A\vec{x} = \vec{b}$. Also, if we order our vectors $\vec{x}$ and $\vec{b}$ in a particular order then we get a block Matrix for A, which will be suited in the future when we will have more than 4 points.

Let the four points in the source plane be denoted by $P = [P_1, P_2]^T$,$Q = [Q_1, Q_2]^T$,$R = [R_1, R_2]^T$ and $S = [S_1, S_2]^T$
and those in Destination Plane be denoted by $P' = [P'_1, P'_2]^T$,$Q' = [Q'_1, Q'_2]^T$,$R' = [R'_1, R'_2]^T$ and $S' = [S'_1, S'_2]^T$

Taking $\vec{x} = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}]^T$ and $\vec{b} = [P_1, Q_1, R_1, S_1, P_2, Q_2, R_2, S_2]^T$ we get the Matrix A as:

$$A = \begin{bmatrix} P_1 & P_2 & 1 & 0 & 0 & 0 & -P'_1P_1 & -P'_1P_2 \\ Q_1 & Q_2 & 1 & 0 & 0 & 0 & -Q'_1Q_1 & -Q'_1Q_2 \\ R_1 & R_2 & 1 & 0 & 0 & 0 & -R'_1R_1 & -R'_1R_2 \\ S_1 & S_2 & 1 & 0 & 0 & 0 & -S'_1S_1 & -S'_1S_2 \\ 0 & 0 & 0 & P_1 & P_2 & 1 & -P'_2P_1 & -P'_2P_2 \\ 0 & 0 & 0 & Q_1 & Q_2 & 1 & -Q'_2Q_1 & -Q'_2Q_2 \\ 0 & 0 & 0 & R_1 & R_2 & 1 & -R'_2R_1 & -R'_2R_2 \\ 0 & 0 & 0 & S_1 & S_2 & 1 & -S'_2S_1 & -S'_2S_2 \end{bmatrix}$$

Clearly the above matrix can be constructed using basic vectorization and if we have more points then those can be accommodated easily. We can then solve for using the pseudo inverse of A.

$$\vec{x} = (A^TA)^{-1}A^T\vec{b}$$

NOTE: We could have also constructed using all the 9 components of H, which will not have the constraint that $h_{33} = 1$ and then construct A matrix using a similar approach. In this case we would be needing to solve the equation $A\vec{x} = \vec{0}$, the solution to this equation which minimizes the error would be the eigen vector in V corresponding to the smallest eigen value, where by SVD $A = UDV^T$. The solution that we would get from this approach will be identical to what i have done using pseudo inverse of A.

The function written out in Python for this task is:

```
# Homograhpy using points
def HMat_Pts(world,img):
    """
    Returns a homography mat such that H*world=img and H[2,2]=1 ie push forward
    img is on destination Image
    world is on source Image
    img,world=np.array([[x1,y1],[x2,y2],[...],[...]])
    """
    n=np.shape(world)[0]
    m=np.shape(world)[1]
    A=np.zeros((2*n,8))
    b=np.reshape(img.T,n*m)

    block1=np.zeros((n,m+1))
    block1[:,-1]=np.ones((1,n))
    block1[:,:2]=world[:,:]
    A[:n,:3]=block1
    A[n:2*n,3:6]=block1

    block2=np.multiply(world.T,-1*img[:,0])
    block3=np.multiply(world.T,-1*img[:,1])

    A[:n,6:8]=block2.T
    A[n:2*n,6:8]=block3.T
    #solve for h using psuedo inverse
    h=(np.linalg.inv((A.T)@A)@A.T)@b
    h=np.concatenate([h,[1]])
    H=np.reshape(h,(3,3))
    return(H)
```

1.2 RANSAC for finding inliers

RANSAC stands for **Ran**dom **S**ampling **a**nd **C**onsensus. For this homework, I am using SIFT points as interesting points on two images. After Estimation correspondences of interesting points between a set of images using a similarity threshold on the euclidean distance between the key point descriptors the set of points that we get will still contain false correspondences. This is where RANSAC comes into picture to estimate correct set of inliers using linear least squares homography.

Fundamental to the RANSAC algorithm is using *randomly-selected* least amount of data to construct an estimate and to then ascertain the extent of support that the rest of the data provides to the estimate. We accept the estimate only if the support exceeds a threshold. When an estimate made in this manner is accepted, the supported data constitutes the inliers and the rest of the data the outliers.

To quantify the support we say that for any point X on image 1 the estimated point $\tilde{x} = HX$ and the actual point is x'. We then find out the euclidean distance of the estimate point $\tilde{x}$ from the actual point x' and count the number of points that satisfy the criteria that $||x' - \tilde{x}|| < \delta$ where δ is the threshold. Then by randomly selecting different points to construct H for N trials, the set containing the maximum number of inliers is the true inlier set. The criteria of choosing a value for N is discussed in the following paragraph.

Let ϵ be the probability that a randomly chosen point is an outlier, then $1 - \epsilon$ is the probability that all n correspondences chosen for estimating H in a given trial are all inliers is $(1 - \epsilon)^n$. This implies that, in a given trial, the probability that at least one of the n correspondences used for calculating H is a n outlier is $(1 - (1 - \epsilon)^n)$. therefore, the probability that every one of the N trials will involve at least one outlier in the calculation of H is $[1 - (1 - \epsilon)^n]^N$. Therefore, the probability that at least one of the N trials will be free of outliers in the calculation of H is $p = 1 - [1 - (1 - \epsilon)^n]^N$.

This gives us the expression for the minimum number of trials N as:

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

Also, as we will never have a correct estimate of ϵ value for any case, therefore we need to have another constraint. This comes from bound on size of inlier set M. If we have the correct ϵ then we can say that the number of inliers in the data would be $M = (1-\epsilon) * N_{total}$, where N_{total} is the total number of matched points. Therefore, in my code i am starting from an optimistic value of ϵ and check if i meet the size requirement, if not then i am increasing the value of ϵ and updating N, the number of trials.

For my implementation, I have chosen the following parameters:

i) probability that at least one of the N trials will be free of outliers $p = 0.99$

ii) minimal set of correspondences chosen randomly for constructing homography $n = 6$

iii) threshold to construct inlier set $\delta = 2$

iv) starting probability that a correspondence is an outlier $\epsilon = 0.1$

The function written out in Python for this task is:

```
# function for Random sampling and consensus
def RANSAC(pts1,pts2):
    """
    Input: pts1,pts2: matching points with inliers and outliers
                      format:[[x1,y1],[x2,y2],....] list of lists of points

    Output: in_indices_store: indices of inlier points of image  1 and image 2 respectively
                            format:[index1,index2,....] list of index numbers
    """
    #-----define RANSAC parameters:-----#
    p=0.99  # probability that at least one of the N trials will be free of outliers
    n=6     # minimal set of correspondences chosen randomly for constructing homography
    delta=2 # decision threshold to construct inlier set
    e=0.1   # probability that a correspondence is an outlier
    N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n))) #number of iterations needed
    #-----------------begin---------------------#
    n_total=len(pts1) # total number of correspondences
    # convert pts lists to array and into homogeneous form
    pts1_hc=np.ones((n_total,3))
    pts1_hc[:,:-1]=pts1
    pts2_hc=np.ones((n_total,3))
    pts2_hc[:,:-1]=pts2

    badinliers=True
    while badinliers:
        count=0
        size_old=0
        while count<N:
            #randomly sample n points from pts1 and pts2
            rand_indices=np.random.randint(n_total,size=n)
            ipts1=pts1_hc[rand_indices,:]
            ipts2=pts2_hc[rand_indices,:]
            # find homography between these points: linear least sqaures
            iH=HMat_Pts(ipts1[:,:-1],ipts2[:,:-1])#gives H12*ipts1=ipts2: push fwd
            # find estimated pts2 and error distance
            est_pts2=iH@(pts1_hc.T) # est pts as col vectors
            est_pts2=est_pts2.T # est pts as row vectors
            # scale est_pts2 to get third col as 1
            temp=est_pts2[:,-1]
```

```
            est_pts2=np.linalg.inv(np.diag(temp))@est_pts2
            error=pts2_hc[:,:-1]-est_pts2[:,:-1]
            distance=np.power(error,2)
            distance=distance@np.ones((2,1))
            distance=np.sqrt(distance)
            # consensus
            in_indices=np.where(distance<=delta)[0] # inlier indices
            size_new=len(in_indices) #size of ith inlier set
            if size_new>size_old:
                in_indices_store=in_indices #store the set of indices with largest size
                size_old=size_new
            count+=1
        if len(in_indices_store)>(1-e)*n_total:
            badinliers=False
        else:
            e=e*2 #bad dataset than what was assumed
            #update N ie more iterations of RANSAC
            N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n)))
    print('RANSAC max inlier set was'+str(len(in_indices_store)))
    return(in_indices_store)
```

1.3 Non-Linear Least Squares optimization using Levenberg-Marquardt

After identifying the inliers and outliers with the help of RANSAC and then estimating the homography using all the inliers, the homography received is still not accurate and needs to be corrected. We carry out non-linear least squares minimization to obtain a better homography. Levenberg-Marquardt(LM) algorithm is one of the non-linear least squares methods. This method is an interpolation of Gauss-Newton method and Gradient-Descent method. To construct the optimization problem we define a cost function which is the sum of squares of distance of a point x' with its estimate $\tilde{x} = Hx$. Therefore our cost function becomes

$$\min_p C(p) = \Sigma_i^N ||x_i' - \tilde{x_i}||^2 = \Sigma_i^N ||x_i' - H(p)x_i||^2 = ||X - f(p)||^2 = ||\epsilon(p)||^2 = \epsilon^T(p)\epsilon(p)$$

where p_i is the unknown for which we are solving i.e. components $H_{i,j}$ of the final homography. This is an unconstrained minimization problem, which can be solved using Gradient Descent(GD) or Gauss-Newton(GN) method, but both of these methods have its own advantages and disadvantages. This is where LM comes into picture, which combines the best of both these methods.
The solution given by GN is: $\vec{\delta_p} = (J_f^t J_f)^{-1} J_f^T \epsilon(\vec{p})$ this can be re written as $(J_f^t J_f)\vec{\delta_p} = J_f^T \epsilon(\vec{p})$. The LM method adds a damping coefficient μ to this equation and the new equation becomes:

$$(J_f^t J_f + \mu_k I)\vec{\delta_p} = J_f^T \epsilon(\vec{p_k})$$

and the update equation for p is:

$$\vec{p}_{k+1} = \vec{p}_k + \vec{\delta}_p$$

In the LM method we decide the value of the damping coefficient μ_k using a ratio test:

$$\rho_{k+1} = \frac{C(p_k) - C(p_k)}{\delta_p^T J_f^T \epsilon(p_k) + \delta_p^T \mu_k I \delta_p}$$

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

It can be noted that when ρ is negative then $\mu_{k+1} \geq 2\mu_k$ ie a strong steer towards GD, but when $\rho > 2$ then $\mu_{k+1} = \mu_k/3$ ie conservative steer towards GN. The only thing that remains is the initialization of the sequence μ_k, $\mu_0 = \tau * max(diag(J_f^T J_f))$ for some $0 < \tau \leq 1$

In our context the initial guess for LM, p_0 is given as the homography obtained from linear least squares of all inliers and after the optimization we get a better final homography. In my code, I am using *least_sqaures* implementation from *scipy.optimize* library.

1.4 Image mosiacing to generate panoramic image

The Image mosiacing process now combines all the tasks discussed above to estimate pair-wise homographies and then stitch the images to the center image. The details of steps involved are:

1. Step-1: Read N images and find pairwise homography
   In my implementation, I am giving a folder path to my function. this folder will have only images in the folder and named sequentially from 1 to N. In the first step the function will read all the N images and then calls the function for automatic calculation of homography. The routine finds pairwise homographies $H_{i,i+1}$.

2. Step-2: Find the middle image and find forward homographies and inverse

   After finding $H_{i,i+1}$ we then find out the homographies $H_{i,m}$, where m is the index of the middle image. To find $H_{i,m}$ we just need to multiply sequentially the homographies $H_{i,i+1}$ till we get to index m. Then if $i < m$ the forward homography becomes the the same as the product found above but if $i >= m$ then we find the product of $\Pi_{m+1}^{i} H_{m,j}$ and invert this product to get the forward homographies.

   To get the set of inverse homographies from middle plane to ith image plane we just invert the homographies stored earlier.

   After finding the above two quantities, I am just appending one more H matrix for H mid to mid as identity at the middle index. So now out H and H inverse lists become the same size as the number of images.

3. Step-3: Find size of panoramic canvas
   I make use of one assumption here that all the images given to us are of the same sizes(mxn) Using this assumption I construct a matrix of corner coordinates of all images. Then using the H matrix constructed from step-2, I find the transformed coordinates of the the corners. Taking minimum and maximum of both x and y coordinates for all corners, I get the size of the panoramic canvas.

4. Step-4: Assign pixel values
   This procedure is very similar to what was done in homework 3 with the exception that we now have N images from where the pixel values can be taken from. The process involves steps:

   i) Iterate over p_{ij} on the canvas

   ii) Using H inverse computed from step-2 find the point coordinate on the ith image's plane

   iii) loop over k images check if the transformed point lies inside the mxn window

   iv) if yes, then copy the first pixel value and break loop on k images.

   v) assign pixel value to p_{ij} and move to next p_{ij}

   NOTE: In the above procedure, I am assuming that the pixel value found at the first instance of matched image will be very close to the pixel values in other images as its the same feature/object. Therefore to reduce computation by taking average of all pixel values, I am just assigning the first found value.

The python function to perform the task is:

```
#function for Panoramic image generation
def Panorama(readpath,savepath,savename,plot):
    """
    Input:  readpath: string type path of a folder containing all images ...
            named sequentially as 1.jpg,2,3.... folder should have only images
```

```
        assuming all images are of the same sizes m,n
        savepath: saving location of the panoramic image, string
        savename: name of the saved image, string
        plot: 1 or 0: 1= plot pairwise images with inliers an outliers
Output: saved image at savepath location , empty return
"""
print('Panorama begins')
img_names=os.listdir(readpath)
print(img_names)
N=len(img_names)#total number of images
# read all images in the sequence 1,2,3,4....
img=[];color_img=[]
for i in range(N):
    temp=cv2.imread(readpath+img_names[i],0)
    temp2=cv2.imread(readpath+img_names[i],-1)
    img.append(temp)
    color_img.append(temp2)
m=np.shape(img[0])[0];n=np.shape(img[0])[1]
print('Images read')
# find homographies sequentially ie H12, H23, H34, ...
H=[]
for i in range(N-1):
    temp=AutoHomoCalc(img[i],img[i+1],(plot,savepath+savename+str(i)+str(i+1)),color_img
        [i],color_img[i+1])
    H.append(temp)
print('Individual Homographies calculated')
# find middle image:
if N%2==0:
    mid=int(N/2)
else:
    mid=int((N+1)/2)
#find homographies to middle image ie projecting to mid-1 image: numbering from 0-(N-1)
H_to_mid=[]
for i in range(N-1):
    temp=np.identity(3)
    if i<mid-1:
        for j in range((mid-1)-i):
            temp=np.array(H[i+j])@temp
        H_to_mid.append(temp)
    else:
        for j in range(i-(mid-1)+1):
            temp=np.array(H[(mid-1)+j])@temp
        H_to_mid.append(np.linalg.inv(temp))
# add H=I for center image
if N%2==0:
    H_to_mid.insert(mid,np.identity(3))
else:
    H_to_mid.insert(mid-1,np.identity(3))
# find transformed corners of all images in the plane of mid
corners=np.array([[0,0,1],
                  [n,0,1],
                  [n,m,1],
                  [0,m,1]])
all_corners=np.kron(np.ones((N,1)),corners)# col vector 4Nx3
all_corners=all_corners.T # 3x4N vector last row all 1
all_H=np.zeros((3,3*N)) #row vector 3x3N: need as row vector only
for i in range(N): all_H[:,3*i:3*i+3]=H_to_mid[i]
all_H_inv=np.zeros((3,3*N))#row vector 3x3N: need as row vector only
for i in range(N): all_H_inv[:,3*i:3*i+3]=np.linalg.inv(H_to_mid[i])

temp=np.zeros((3*N,4*N))
for i in range(N): temp[3*i:3*i+3,4*i:4*i+4]=all_corners[:,4*i:4*i+4]
tr_corners=all_H@temp# tarnsformed corners 3x4N last row all 1
# scale tr_corners to real coordinates
tr_corners=tr_corners.T # 4Nx3
```

```
    tr_corners=np.linalg.inv(np.diag(tr_corners[:,-1]))@tr_corners
    #find canvas size
    xmin=int(np.floor(np.min(tr_corners[:,0])))
    xmax=int(np.ceil(np.max(tr_corners[:,0])))
    deltax=xmax-xmin
    ymin=int(np.floor(np.min(tr_corners[:,1])))
    ymax=int(np.ceil(np.max(tr_corners[:,1])))
    deltay=ymax-ymin
    #create empty canvas
    canvas=np.zeros(((ymax-ymin),(xmax-xmin),3)) #filan image of this size, no scaling
    print('canvas_size:'+str(ymax-ymin)+'x'+str(xmax-xmin))
    # loop over canvas indices to assign pixel values
    distype='RoundDown'
    for i in range(deltay):
        for j in range(deltax):
            #transform [j,i] on canvas to the block of interest
            x_tr=xmin+j
            y_tr=ymin+i
            hc_mp=[x_tr,y_tr,1] # hc coordinate in mid plane
            #find transformed back hc coordinate
            hc_op=all_H_inv@(np.kron(np.identity(N),hc_mp)).T # hc own plane 3xN
            #scale to convert to real coordinates
            hc_op=hc_op.T # Nx3
            hc_op=np.linalg.inv(np.diag(hc_op[:,-1]))@hc_op
            x=hc_op[:,:-1] # only x and y coordinates in all planes
            #find if coordinates lie in the image region of the specific plane
            count=0
            p=np.zeros((1,3))#pixel values
            for k in range(N):
                temp_x=x[k,:]
                if temp_x[0]>0 and temp_x[0]<n and temp_x[1]>0 and temp_x[1]<m:
                    p=p+InterpolatePixel(temp_x,color_img[k],distype)
                    count+=1
                    break
#             take average of summed pixel values and make integer
#             if count>0:
#                 p=p/count
#                 p=np.floor(p)
#                 p=p.astype(int)
            #assign pixel value
            canvas[i,j,:]=p
    # write image
    cv2.imwrite(savepath+savename+'.jpg',canvas)
    return()
```

NOTE:

a) I had to down size my input images so that the Euclidean match can take place faster and it also helps in reducing the size of the canvas.

b) I am not doing any scaling when making the panoramic image, if done then the process will becomes faster, but we might loose some information.

2 Results

2.1 Inlier and Outlier matches

All the Inliers in the following images are indicated with a green circle and outliers with a red circle.

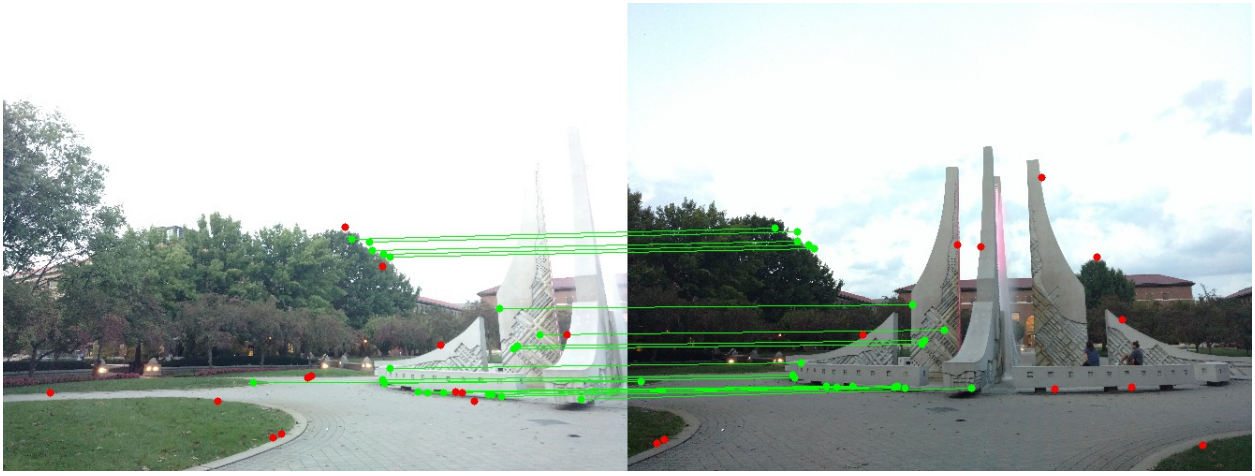

Figure 1: image 1 and 2

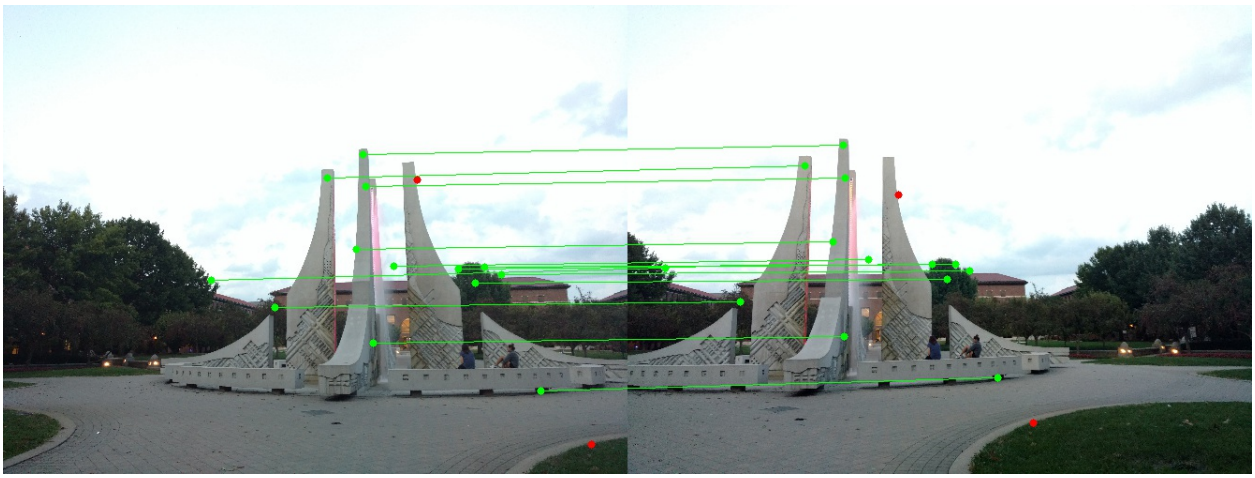

Figure 2: image 2 and 3

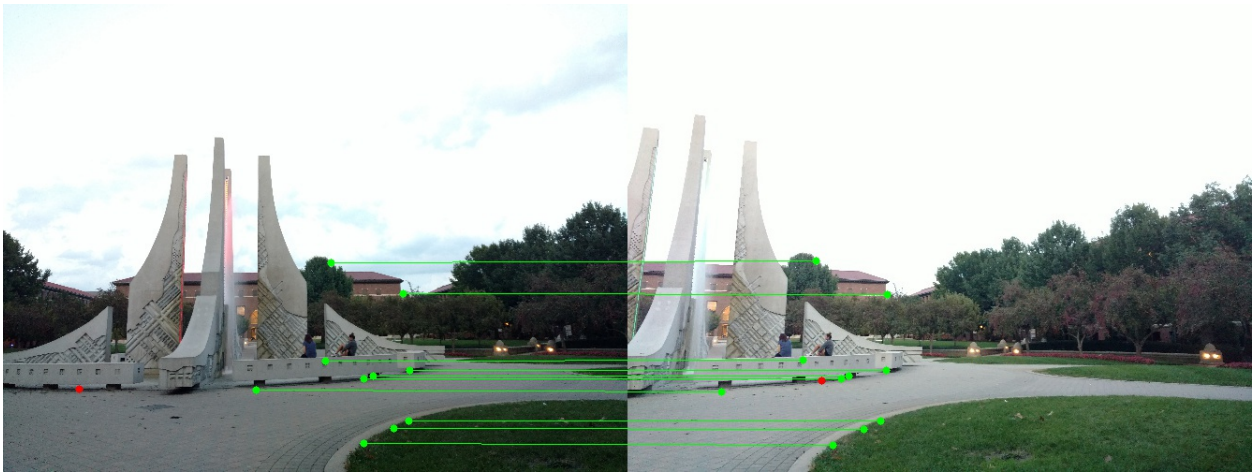

Figure 3: image 3 and 4

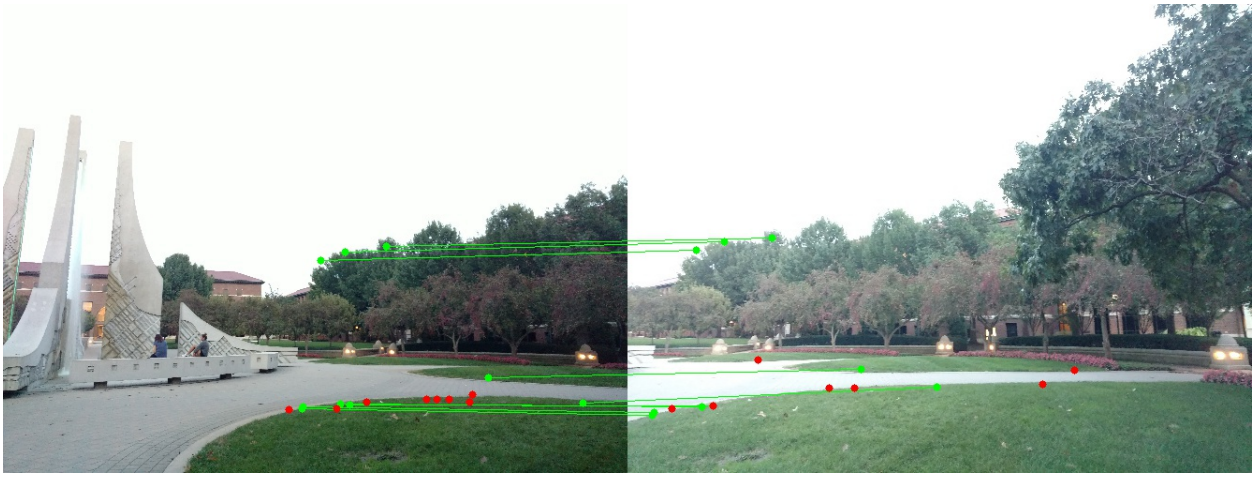

Figure 4: image 4 and 5

2.2 Panoramic Image

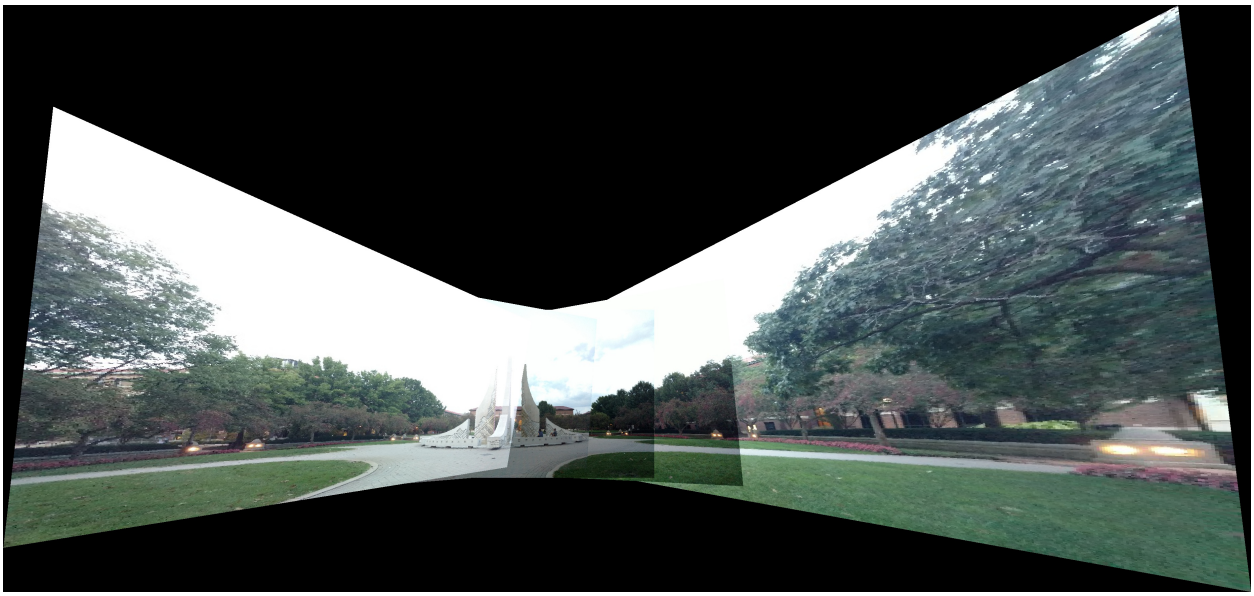

Figure 5: Engineering Fountain

3 Source Code:

3.1 main file

```
"""
ECE661: hw5 main file
@author: rahul deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""

#import libraries
import sys

sys.path.append('../../')
import MyCVModule as MyCV
#define path
readpath='../images/fountain/' # path of images to be read
savepath='../results/' #path for saving results of images
savename='fountain'

plot=1 # print pair wise images

MyCV.Panorama(readpath,savepath,savename,plot)
```

3.2 functions

```
import numpy as np
import cv2
from scipy.optimize import least_squares
import os

#%%
#-----------------------------------------HW5-----------------------------------------#
#%%
#function for Panoramic image generation
def Panorama(readpath,savepath,savename,plot):
    """
    Input: readpath: string type path of a folder containing all images ...
           named sequentially as 1.jpg,2,3.... folder should have only images
           assuming all images are of the same sizes m,n
           savepath: saving location of the panoramic image, string
           savename: name of the saved image, string
           plot: 1 or 0: 1= plot pairwise images with inliers an outliers
    Output: saved image at savepath location , empty return
    """
    print('Panorama begins')
    img_names=os.listdir(readpath)
    print(img_names)
    N=len(img_names)#total number of images
    # read all images in the sequence 1,2,3,4....
    img=[];color_img=[]
    for i in range(N):
        temp=cv2.imread(readpath+img_names[i],0)
        temp2=cv2.imread(readpath+img_names[i],-1)
        img.append(temp)
        color_img.append(temp2)
    m=np.shape(img[0])[0];n=np.shape(img[0])[1]
    print('Images read')
    # find homographies sequentially ie H12, H23, H34, ...
    H=[]
    for i in range(N-1):
        temp=AutoHomoCalc(img[i],img[i+1],(plot,savepath+savename+str(i)+str(i+1)),color_img
            [i],color_img[i+1])
```

```
        H.append(temp)
    print('Individual Homographies calculated')
    # find middle image:
    if N%2==0:
        mid=int(N/2)
    else:
        mid=int((N+1)/2)
    #find homographies to middle image ie projecting to mid-1 image: numbering from 0-(N-1)
    H_to_mid=[]
    for i in range(N-1):
        temp=np.identity(3)
        if i<mid-1:
            for j in range((mid-1)-i):
                temp=np.array(H[i+j])@temp
            H_to_mid.append(temp)
        else:
            for j in range(i-(mid-1)+1):
                temp=np.array(H[(mid-1)+j])@temp
            H_to_mid.append(np.linalg.inv(temp))
    # add H=I for center image
    if N%2==0:
        H_to_mid.insert(mid,np.identity(3))
    else:
        H_to_mid.insert(mid-1,np.identity(3))
    # find transformed corners of all images in the plane of mid
    corners=np.array([[0,0,1],
                      [n,0,1],
                      [n,m,1],
                      [0,m,1]])
    all_corners=np.kron(np.ones((N,1)),corners)# col vector 4Nx3
    all_corners=all_corners.T # 3x4N vector last row all 1
    all_H=np.zeros((3,3*N)) #row vector 3x3N: need as row vector only
    for i in range(N): all_H[:,3*i:3*i+3]=H_to_mid[i]
    all_H_inv=np.zeros((3,3*N))#row vector 3x3N: need as row vector only
    for i in range(N): all_H_inv[:,3*i:3*i+3]=np.linalg.inv(H_to_mid[i])

    temp=np.zeros((3*N,4*N))
    for i in range(N): temp[3*i:3*i+3,4*i:4*i+4]=all_corners[:,4*i:4*i+4]
    tr_corners=all_H@temp# tarnsformed corners 3x4N last row all 1
    # scale tr_corners to real coordinates
    tr_corners=tr_corners.T # 4Nx3
    tr_corners=np.linalg.inv(np.diag(tr_corners[:,-1]))@tr_corners
    #find canvas size
    xmin=int(np.floor(np.min(tr_corners[:,0])))
    xmax=int(np.ceil(np.max(tr_corners[:,0])))
    deltax=xmax-xmin
    ymin=int(np.floor(np.min(tr_corners[:,1])))
    ymax=int(np.ceil(np.max(tr_corners[:,1])))
    deltay=ymax-ymin
    #create empty canvas
    canvas=np.zeros(((ymax-ymin),(xmax-xmin),3)) #filan image of this size, no scaling
    print('canvas size:'+str(ymax-ymin)+'x'+str(xmax-xmin))
    # loop over canvas indices to assign pixel values
    distype='RoundDown'
    for i in range(deltay):
        for j in range(deltax):
            #transform [j,i] on canvas to the block of interest
            x_tr=xmin+j
            y_tr=ymin+i
            hc_mp=[x_tr,y_tr,1] # hc coordinate in mid plane
            #find transformed back hc coordinate
            hc_op=all_H_inv@(np.kron(np.identity(N),hc_mp)).T # hc own plane 3xN
            #scale to convert to real coordinates
            hc_op=hc_op.T # Nx3
            hc_op=np.linalg.inv(np.diag(hc_op[:,-1]))@hc_op
```

```
            x=hc_op[:,:-1] # only x and y coordinates in all planes
            #find if coordinates lie in the image region of the specific plane
            count=0
            p=np.zeros((1,3))#pixel values
            for k in range(N):
                temp_x=x[k,:]
                if temp_x[0]>0 and temp_x[0]<n and temp_x[1]>0 and temp_x[1]<m:
                    p=p+InterpolatePixel(temp_x,color_img[k],distype)
                    count+=1
                    break
#             take average of summed pixel values and make integer
#             if count>0:
#                 p=p/count
#                 p=np.floor(p)
#                 p=p.astype(int)
            #assign pixel value
            canvas[i,j,:]=p
    # write image
    cv2.imwrite(savepath+savename+'.jpg',canvas)
    return()
#%%
# function for automatic calculation of homography between two images
# using SIFT interest points, RANSAC, and Non-Linear Least squares Levenberg marqaudt
def AutoHomoCalc(img1,img2,options,img1c,img2c):
    """
    Input: img1,img2: grayscale image matrices
    img1c,img2c: colored images only for plotting purposes
    options=tuple(plot,name)
    plot: 0 or 1, 1= make a plot of inliers
    name: name of saved image, full name with path
    Output: H: homograhy
    """
    # find sift points of images
    pts1,des1=SIFT_detector(img1)
    pts2,des2=SIFT_detector(img2)
    # find euclidean matches between images
    match_pts1,match_pts2=Euclidean_sift_match(pts1,des1,pts2,des2) # returns list of lists
    # find inlier points using RANSAC
    in_indices=RANSAC(match_pts1,match_pts2)  # returns array of indices
    #store inliers and outlier separately
    in_pts1=np.array(match_pts1)
    in_pts1=in_pts1[in_indices,:]
    out_pts1=match_pts1
    for i in sorted(in_indices,reverse=True): del out_pts1[i]
    out_pts1=np.array(out_pts1)

    in_pts2=np.array(match_pts2)
    in_pts2=in_pts2[in_indices,:]
    out_pts2=match_pts2
    for i in sorted(in_indices,reverse=True): del out_pts2[i]
    out_pts2=np.array(out_pts2)
    ## plot inliers(green) and outliers(red) on one image and save image
    if options[0]==1:
        plot_inliers(img1c,img2c,in_pts1,out_pts1,in_pts2,out_pts2,options[1])

    # find Initial homography using Linear Least sqaures with all inliers
    H=HMat_Pts(in_pts1,in_pts2)
    # Non-Linear Least Squares fit to find better Homography
    # find intial solution for LM
    p0=np.reshape(H,np.shape(H)[0]*np.shape(H)[1]) # p0=[h11,h12,h13,h21,h22,h23,h31,h32,h33
        ]
    # call Least squares with cost function as argument
    optim_results=least_squares(cost_fun,p0,method='lm',args=(in_pts1,in_pts2))
#     print('----------------------LM results-----------------------------')
#     print('solution is :'+str(optim_results['x']))
```

```
#     print('min function value is:'+str(optim_results['cost']))
#     print('gradient is:'+str(optim_results['grad']))
#     print('number of function evalutaions were:'+str(optim_results['nfev']))
#     print('termination status:'+str(optim_results['status']))
#     print('termination message:'+str(optim_results['message']))

    p=optim_results['x']
    H=np.reshape(p,(3,3))
    H=H/H[-1,-1]
    return(H)
#%%
# cost function for Non-Linear Least Squares Fit
def cost_fun(p,in_pts1,in_pts2):
    """
    We are estimating homography such that H*in_pts1=in_pts2
    Input: p is an array of size 9 in the format np.array([h11,h12,h13,h21,h22,h23,h31,h32,
        h33])
           in_pts1,in_pts2: inlier points as np.array([[x1,y1],[x2,y2],...])
    Output: scalar cost function value: sum of sqaures of error
    """
    #reshape p to form H
    H=np.reshape(p,(3,3))
    #convert in_pts1 to hc representation
    in_pts1_hc=np.ones((np.shape(in_pts1)[0],3))
    in_pts1_hc[:,:-1]=in_pts1
    # find estimated point2 using homography
    est_hc= H@in_pts1_hc.T#estimated point as stacked col vectors
    est_hc=est_hc.T #points as row vectors
    #scale with 3rd ordinate to get real point
    est=np.linalg.inv(np.diag(est_hc[:,-1]))@est_hc
    est=est[:,:-1] # removing third column
    # define error and cost function
    X=np.reshape(in_pts2,np.shape(in_pts2)[0]*np.shape(in_pts2)[1]) #col vector
    f=np.reshape(est,np.shape(est)[0]*np.shape(est)[1]) #col vector
    error=X-f #col vector
    cost=error# Least sqaures function with lm option requires a M dimensional vector as
        objective function
    #cost= error.T@error
    return(cost)

#%%
def plot_inliers(img1,img2,in_pts1,out_pts1,in_pts2,out_pts2,name):
    """
    Input: img1,img2: colored images of the same sizes mxn
           in_pts1,in_pts2: np array [[x,y],..] inliers to be plotted in green circle
           out_pts1,out_pts2: np array [[x,y],..] outliers to be plotted in red circles
           name: filename of image with full path
    Output:  save the image, empty return
    """
    m=np.shape(img1)[0];n=np.shape(img1)[1]
    #convert in_pts and out_pts to integers
    in_pts1=in_pts1.astype(int)
    in_pts2=in_pts2.astype(int)
    out_pts1=out_pts1.astype(int)
    out_pts2=out_pts2.astype(int)
    # make empty canvas of twice width
    img=np.zeros((m,2*n,3))
    #copy original images onto the two halves of canvas
    img[:,:n,:]=img1
    img[:,n:,:]=img2
    #plot inliers in green
    r=4
    for i in range(len(in_pts1)):
        cv2.circle(img,tuple(in_pts1[i,:]),r,(0,255,0),-1)# left image
        cv2.circle(img,(n+in_pts2[i,0],in_pts2[i,1]),r,(0,255,0),-1)# right image
```

```
        cv2.line(img,tuple(in_pts1[i,:]),(n+in_pts2[i,0],in_pts2[i,1]),(0,255,0),1)#line
            joining inliers
    #plot outliers in red
    for i in range(len(out_pts1)):
        cv2.circle(img,tuple(out_pts1[i]),r,(0,0,255),-1)# left image
        cv2.circle(img,(n+out_pts2[i,0],out_pts2[i,1]),r,(0,0,255),-1)# right image
    #save image
    cv2.imwrite(name+'.jpg',img)
    return()
#%%
# function for Random sampling and consensus
def RANSAC(pts1,pts2):
    """
    Input: pts1,pts2: matching points with inliers and outliers
                      format:[[x1,y1],[x2,y2],....] list of lists of points

    Output: in_indices_store: indices of inlier points of image  1 and image 2 respectively
                          format:[index1,index2,....] list of index numbers
    """
    #----define RANSAC parameters:-----#
    p=0.99  # probability that at least one of the N trials will be free of outliers
    n=6     # minimal set of correspondences chosen randomly for constructing homography
    delta=4 # decision threshold to construct inlier set
    e=0.1   # probability that a correspondence is an outlier
    N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n))) #number of iterations needed
    #-----------------begin-----------------#
    n_total=len(pts1) # total number of correspondences
    # convert pts lists to array and into homogeneous form
    pts1_hc=np.ones((n_total,3))
    pts1_hc[:,:-1]=pts1
    pts2_hc=np.ones((n_total,3))
    pts2_hc[:,:-1]=pts2

    count=0
    size_old=0
    badinliers=True
    while badinliers:
        while count<N:
            #randomly sample n points from pts1 and pts2
            rand_indices=np.random.randint(n_total,size=n)
            ipts1=pts1_hc[rand_indices,:]
            ipts2=pts2_hc[rand_indices,:]
            # find homography between these points: linear least sqaures
            iH=HMat_Pts(ipts1[:,:-1],ipts2[:,:-1])#gives H12*ipts1=ipts2: push fwd
            # find estimated pts2 and error distance
            est_pts2=iH@(pts1_hc.T) # est pts as col vectors
            est_pts2=est_pts2.T # est pts as row vectors
            # scale est_pts2 to get third col as 1
            temp=est_pts2[:,-1]
            est_pts2=np.linalg.inv(np.diag(temp))@est_pts2
            error=pts2_hc[:,:-1]-est_pts2[:,:-1]
            distance=np.power(error,2)
            distance=distance@np.ones((2,1))
            distance=np.sqrt(distance)
            # consensus
            in_indices=np.where(distance<=delta)[0] # inlier indices
            size_new=len(in_indices) #size of ith inlier set
            if size_new>size_old:
                in_indices_store=in_indices #store the set of indices with largest size
                size_old=size_new
            count+=1
        if len(in_indices_store)>5:
            badinliers=False
        else:
            e=e*2 #bad dataset than what was assumed
```

```
            #update N ie more iterations of RANSAC
            N=int(np.ceil(np.log(1-p)/np.log(1-(1-e)**n)))
    print('RANSAC max inlier set was'+str(len(in_indices_store)))
    return(in_indices_store)
#%%
#function for Euclidean match for SIFT points
#V1: for HW5 script: make sure V0 is commented out!!
def Euclidean_sift_match(pts1,des1,pts2,des2):
    """
    Input: pts1,pts2= [[x,y],[],..] list of lists of interest points
           des1,des2 = vector of 128 size, sift descriptor stacked as row vectors
    Output: match_pts1,match_pts2= [[x1,y1],[x2,y2],....] list of lists of points
                                    both have same sizes
            Euclidean distance is less than a dynamic threshold
    """
    match_pts1=[]
    match_pts2=[]
    eu=np.zeros((len(pts1),len(pts2)))
    for i in range(len(pts1)):
        for j in range(len(pts2)):
            eu[i,j]=np.linalg.norm(des1[i,:]-des2[j,:])

    eu=eu/np.min(eu)
    dy_threshold=2
    for i in range(len(pts1)):
        eu_min=np.min(eu[i,:])
        if eu_min<dy_threshold:
            j_min=np.argmin(eu[i,:])
            match_pts1.append(pts1[i])
            match_pts2.append(pts2[j_min])
    print('size of Euclidean match was:'+str(len(match_pts1)))
    return(match_pts1,match_pts2)
#%%
#function for SIFT interest points detection
def SIFT_detector(img):
    """
    detects SIFT interest points using openCV function and returns pts and des
    in my particular format.
    Input: img: image matrix
    Output: pts: [[x,y],...] : list of lists of pt coordinates as row vectors
            des: 128 bit  SIFT descriptor matrix arranged as stacked row vector
            =[d1,d2,d3,...] di is row vector of size 128
    """
    #sift key points
    sift=cv2.xfeatures2d.SIFT_create() #defining structure
    sift_pts,des=sift.detectAndCompute(img,None)
    pts=convert_SIFT_myFormat(sift_pts)
    return(pts,des)
#%%
#function for converting sift key points into my format
#V1: to be used for HW5: make sure V0 is commented out!!
def convert_SIFT_myFormat(kp):
    """
    Input: kp: keypoint structure given by SIFT, point coordinates in kp.pt
    Output: [[x,y],...] : list of lists of pt coordinates as row vectors
    """
    mykp=[]
    for ikp in kp:
        mykp.append([ikp.pt[0],ikp.pt[1]])
    return(mykp)
#=============================================================================#

#=================================HW2=========================================#
# Homograhpy using points
def HMat_Pts(world,img):
```

```
    """
    Returns a homography mat such that H*world=img and H[2,2]=1 ie push forward
    img is on destination Image
    world is on source Image
    img,world=np.array([[x1,y1],[x2,y2],[...],[...]])
    """
    n=np.shape(world)[0]
    m=np.shape(world)[1]
    A=np.zeros((2*n,8))
    b=np.reshape(img.T,n*m)

    block1=np.zeros((n,m+1))
    block1[:,-1]=np.ones((1,n))
    block1[:,:2]=world[:,:]
    A[:n,:3]=block1
    A[n:2*n,3:6]=block1

    block2=np.multiply(world.T,-1*img[:,0])
    block3=np.multiply(world.T,-1*img[:,1])

    A[:n,6:8]=block2.T
    A[n:2*n,6:8]=block3.T
    #solve for h using psuedo inverse
    h=(np.linalg.inv((A.T)@A)@A.T)@b
    h=np.concatenate([h,[1]])
    H=np.reshape(h,(3,3))
    return(H)

# Interpolation of colors from the source image
def InterpolatePixel(x,srcImg,distype):
    """
    Input:
        x= n.array([X,Y]) coordinates of a point
        srcImg=source image object a n,m,3 matrix
        distype= a string L2, sqL2,BiLinear,RoundDown,RoundUp
    Output:
        cx= size(1,3) vector of pixel values with only integer values
    The function will interpolate the pixel values of the nearest neighbours and do a
        rounding
    to get integers
    """
    #finding the nearest interger neighbours
    p1=[int(np.floor(x[0])),int(np.floor(x[1]))]
    p2=[int(np.ceil(x[0])),int(np.floor(x[1]))]
    p3=[int(np.ceil(x[0])),int(np.ceil(x[1]))]
    p4=[int(np.floor(x[0])),int(np.ceil(x[1]))]
    #storing value of pixels at these points
    #x: col index & y:row index
    c1=srcImg[p1[1],p1[0],:]
    c2=srcImg[p1[1],p1[0],:]
    c3=srcImg[p1[1],p1[0],:]
    c4=srcImg[p1[1],p1[0],:]
    C=np.array([c1,c2,c3,c4])
    #storing weights for interpolation
    w=weights(np.array([p1,p2,p3,p4]),x,distype);#should return a np array
    #interpolating
    fcx=C.T@w
    fcx=np.floor(fcx)
    cx=fcx.astype(int)
    return(cx)

def weights(pts,x,distype):
    """
    function will give a list of weights
    Input:  pts = np.array[[X Y],[X Y]...]
```

```
    x=np.array[X,Y]
    distype is a string with options L2, sqL2,BiLinear,RoundDown,RoundUp
    output:
        w=[w1,w2,w3,w4]
    """
    w=[]
    if distype=='L2':
        #use L2 norm for distance
        for i in range(np.shape(pts)[0]):
            w.append(np.linalg.norm(pts[i,:]-x))
        wsum=sum(w)
        w=w/wsum
        return(w)
    elif distype=='sqL2':
        #use squared L2 norm for distance
        for i in range(np.shape(pts)[0]):
            w.append((np.linalg.norm(pts[i,:]-x))**2)
        wsum=sum(w)
        w=w/wsum
        return(w)
    elif distype=='BiLinear':
        #using bilinear shape functions as weights
        xmin=pts[0,0]
        xmax=pts[1,0]
        ymin=pts[0,1]
        ymax=pts[3,1]
        xi=2*((x[0]-xmin)/(xmax-xmin))-1
        eta=2*((x[1]-ymin)/(ymax-ymin))-1
        w.append((1-xi)*(1-eta)/4.0)
        w.append((1+xi)*(1-eta)/4.0)
        w.append((1+xi)*(1+eta)/4.0)
        w.append((1-xi)*(1+eta)/4.0)
        return(w)
    elif distype=='RoundDown':
        w=[1,0,0,0]
        return(w)
    elif distype=='RoundUp':
        w=[0,0,1,0]
        return(w)
#=============================================================#
```

some more panoramic images

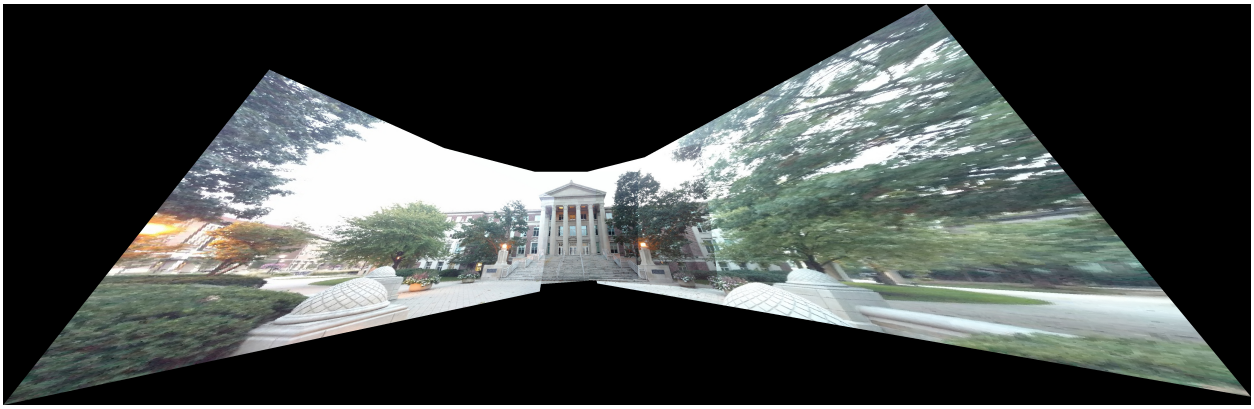

Figure 6: Hovde Hall with 5 images

Figure 7: Maheshwar temple in Indore, India with 3 images

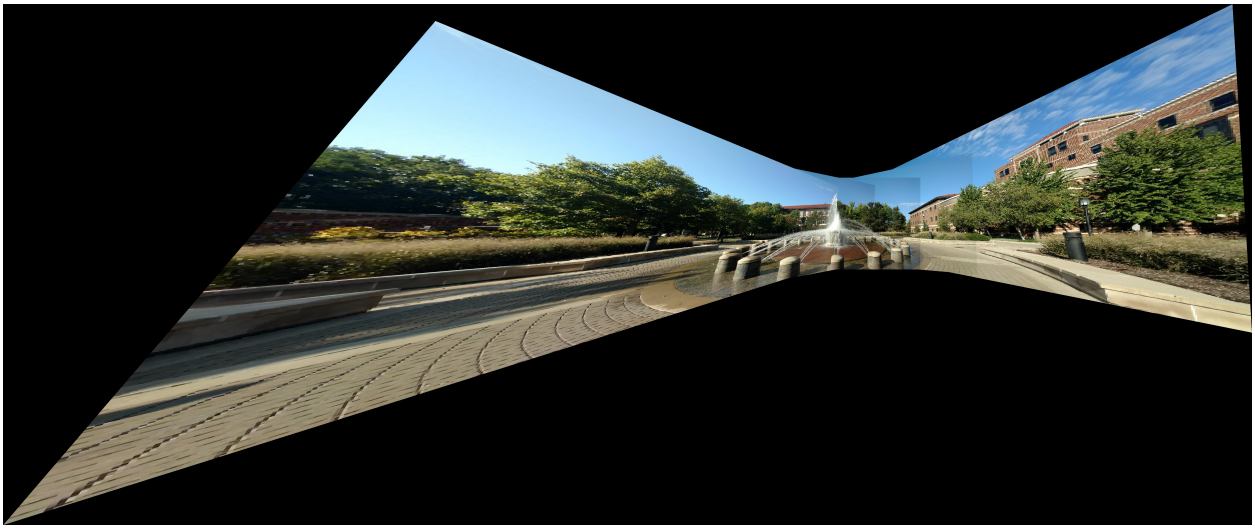

Figure 8: Sciences fountain with 5 images