CSE 252B: Computer Vision II, Winter 2019 – Assignment 4

Instructor: Ben Ochoa

Due: Wednesday, March 6, 2019, 11:59 PM

Instructions

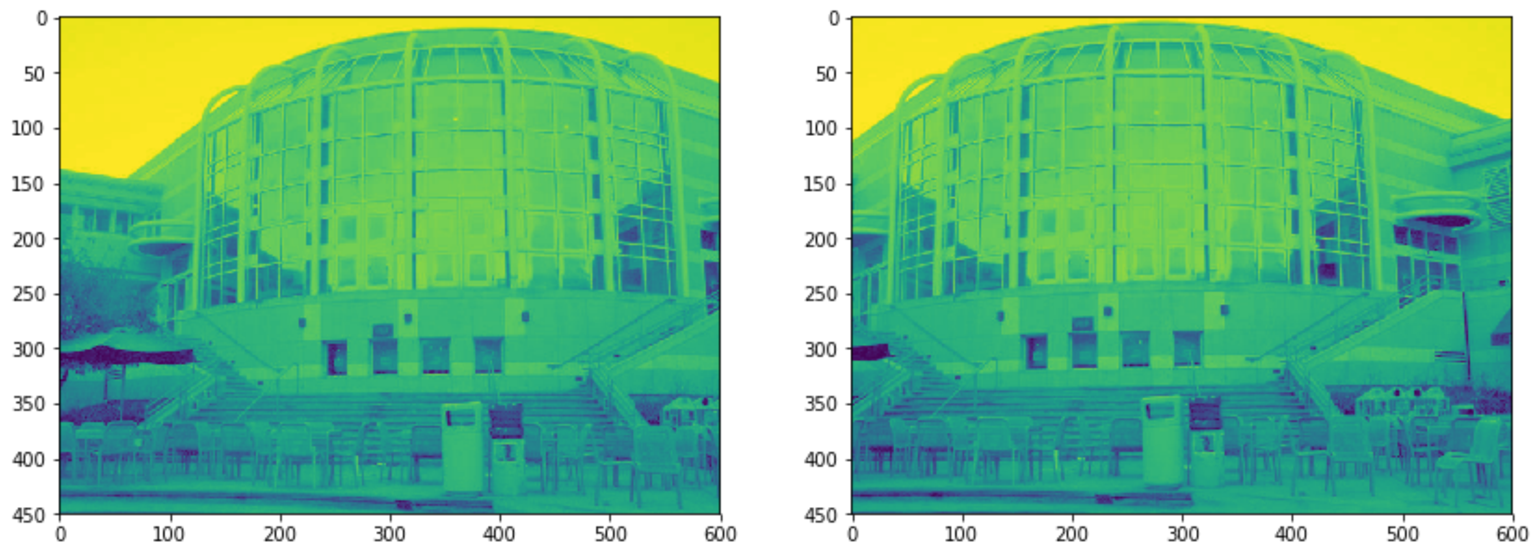
- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains both math and programming problems.
- All solutions must be written in this notebook
- Math problems must be done in Markdown/LATEX. Remember to show work and describe your solution.
- Programming aspects of this assignment must be completed using Python in this notebook.
- This notebook contains skeleton code, which should not be modified (This is important for standardization to facilate effeciant grading).
- You may use python packages for basic linear algebra, but you may not use packages that directly solve the problem. Ask the instructor if in doubt.
- You must submit this notebook exported as a pdf. You must also submit this notebook as an .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- It is highly recommended that you begin working on this assignment early.

In [1]:

```
def grayscale(img):
    gray=np.zeros((img.shape[0],img.shape[1]))
    gray=img[:,:,0]*0.2989+img[:,:,1]*0.5870+img[:,:,2]*0.1140
    return gray
```

In [2]:

```
%matplotlib inline
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import time
from scipy.signal import convolve2d as conv2

# open the input images
I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.
I1_gray = grayscale(I1)
I2_gray = grayscale(I2)



# Display the input images
plt.figure(figsize=(14,8))
plt.subplot(1,2,1)
plt.imshow(I1_gray)
plt.subplot(1,2,2)
plt.imshow(I2_gray)
plt.show()
```



Problem 1 (Programming): Feature detection (20 points)

Download input data from the course website. The file price_center20.JPG contains image 1 and the file price_center21.JPG contains image 2.

For each input image, calculate an image where each pixel value is the minor eigenvalue of the gradient matrix

$$N = \begin{bmatrix} \sum_w I_x^2 & \sum_w I_x I_y \\ \sum_w I_x I_y & \sum_w I_y^2 \end{bmatrix}$$

where w is the window about the pixel, and I_x and I_y are the gradient images in the x and y direction, respectively. Calculate the gradient images using the fivepoint central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead of the sum in N allows for adjusting the size of the window without changing the threshold value). Apply an operation that suppresses (sets to 0) local (i.e., about a window) nonmaximum pixel values in the minor eigenvalue image. Vary these parameters such that around 600–650 features are detected in each image. For resulting nonzero pixel values, determine the subpixel feature coordinate using the Forstner corner point operator.

Report your final values for:

- the size of the feature detection window (i.e. the size of the window used to calculate the elements in the gradient matrix N)
- the minor eigenvalue threshold value
- the size of the local nonmaximum suppression window
- the resulting number of features detected (i.e. corners) in each image.

Display figures for:

- original images with detected features

In [3]:

```
def ImageGradient(I, w, t):
    # inputs:
    # I is the input image (may be mxn for Grayscale or mxnx3 for RGB)
    # w is the size of the window used to compute the gradient matrix N
    # t is the minor eigenvalue threshold
    #
    # outputs:
    # N is the 2x2xmxn gradient matrix
    # b in the 2x1xmxn vector used in the Forstner corner detector
    # J0 is the mxn minor eigenvalue image of N before thresholding
    # J1 is the mxn minor eigenvalue image of N after thresholding

    m,n = I.shape[:2]
    N = np.zeros((2,2,m,n))
```

```
    b = np.zeros((2,1,m,n))
    J0 = np.zeros((m,n))
    J1 = np.zeros((m,n))

    """your code here"""
    #Compute gradient
    kernel_5pts = np.array([[-1,8,0,-8,1]]).T/12

    I_dx = conv2(I, kernel_5pts.T,mode = 'same')
    I_dy = conv2(I, kernel_5pts,mode = 'same')

    WTH = I.shape[1]
    LTH = I.shape[0]
    m = LTH
    n = WTH
    c_x = np.zeros(w)
    c_y = np.zeros(w)
    r = int(w/2)
    for i in range (r, WTH-r):
        for j in range (r, LTH-r):
            N[0,0,j,i] = (I_dx[j-r:j+r+1,i-r:i+r+1 ]**2).sum()#/(w**2)
            N[0,1,j,i] = (I_dx[j-r:j+r+1,
                                   i-r:i+r+1 ]* I_dy[j-r:j+r+1,
                                                   i-r:i+r+1]).sum()#/(w**2)
            N[1,0,j,i] = N[0,1,j,i]
            N[1,1,j,i] = (I_dy[j-r:j+r+1,
                                   i-r:i+r+1 ]**2).sum()#/(w**2)


            c_x = np.array([np.arange(i-r , i+r+1),]*w)
            c_y = np.array([np.arange(j-r , j+r+1),]*w).T

            b[0,0,j,i] = (c_x*I_dx[j-r:j+r+1,
                                      i-r:i+r+1 ]**2).sum() + (c_y * (I_dx[j-r:j+r+1
,
                                                                   i-r:i+r+1 ]* I_dy[j
-r:j+r+1,i-r:i+r+1 ])).sum()


            b[1,0,j,i] = (c_y*I_dy[j-r:j+r+1,
                                      i-r:i+r+1]**2).sum() + (c_x * (I_dx[j-r:j+r+1,
                                                                       i-r:i+r+1 ]* I_
dy[j-r:j+r+1,

i-r:i+r+1 ])).sum()


            #J0 before threshold, J1 after threshod
            J0[j,i] = (np.trace(N[:,:,j,i]/(w**2))- np.sqrt(np.around(np.trace(N
[:,:,j,i]/(w**2))**2,9)
                                                         -np.around(4*np.linalg.det(
N[:,:,j,i]/(w**2)),9)))/2
```

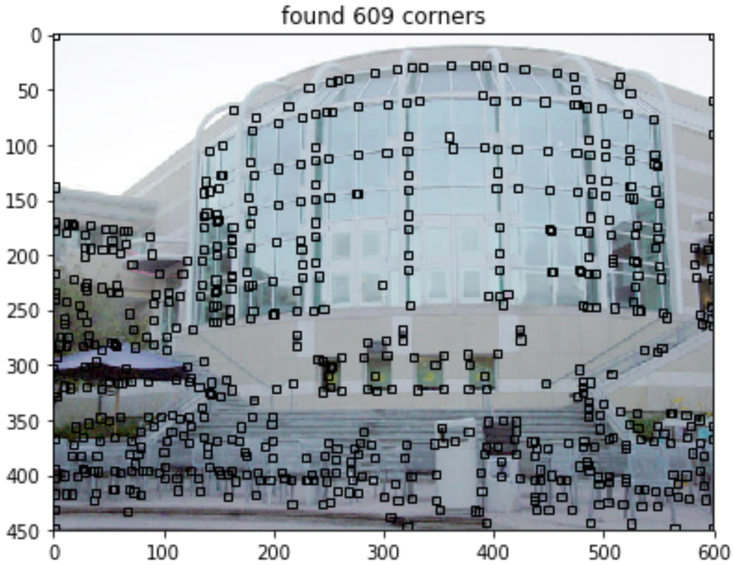
```python
    J1 = J0.copy()


    for i in range (WTH):
        for j in range (LTH):
            if J1[j,i] < t:
                J1[j,i] = 0

    return N, b, J0, J1


def NMS(J, w_nms):
    # Apply nonmaximum supression to J using window w
    # For any window in J, the result should only contain 1 nonzero value
    # In the case of multiple identical maxima in the same window,
    # the tie may be broken arbitrarily
    #
    # inputs:
    # J is the minor eigenvalue image input image after thresholding
    # w_nms is the size of the local nonmaximum suppression window
    #
    # outputs:
    # J2 is the mxn resulting image after applying nonmaximum suppression
    #

    J2 = J.copy()
    """your code here"""
    WTH = J.shape[1]
    LTH = J.shape[0]
    r = int(w_nms/2)

    J2 = J.copy()
    """your code here"""
    r = int(w/2)
    pos = []
    for i in range (r,WTH-r):
        for j in range (r, LTH-r):
            local_max = J[j-r:j+r+1,i-r:i+r+1].max()
            if local_max > J2[j,i]:
                J2[j,i] = 0

    return J2

#
def ForstnerCornerDetector(J, N, b):
    # Gather the coordinates of the nonzero pixels in J
    # Then compute the sub pixel location of each point using the Forstner opera
tor
    #
    # inputs:
    # J is the NMS image
    # N is the 2x2xmxn gradient matrix
    # b is the 2x1xmxn vector computed in the image_gradient function
    #
```

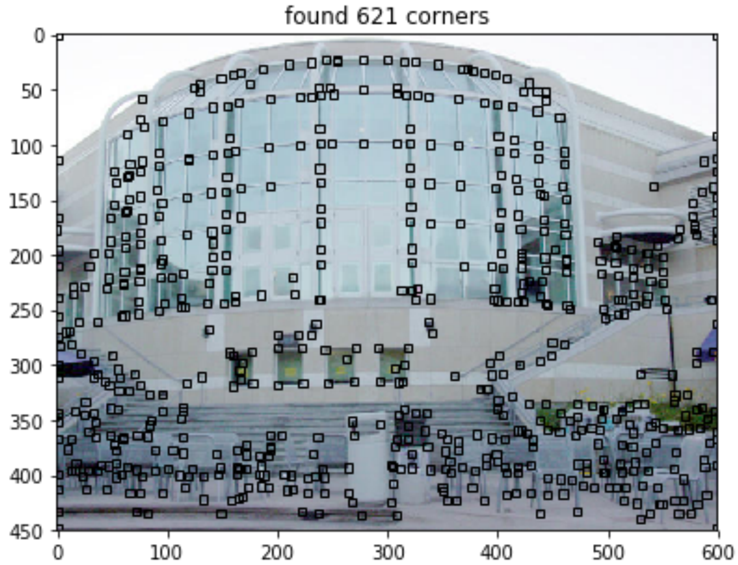
```
    # outputs:

    # C is the number of corners detected in each image
    # pts is the 2xC list of coordinates of subpixel accurate corners
    #     found using the Forstner corner detector

    """your code here"""
    WTH = J.shape[1]
    LTH = J.shape[0]
    pos = []
    for j in range (J.shape[0]):
        for i in range (J.shape[1]):
            if J[j,i] != 0:
                pos.append([j,i])

    pos_np = np.array(pos)

    C = pos_np.shape[0]
    pts = np.zeros((2,C))



    for k, (j, i) in enumerate(zip (pos_np[:,0],pos_np[:,1])):
        if np.linalg.det(N[:,:,j,i]) != 0 :
                pts[:,k] = np.dot(np.linalg.inv(N[:,:,j,i]),b[:,:,j,i]).reshape(
-1)

        else :
                #b[:,:,j,i] = np.matrix(b[:,:,j,i])
                pts[:,k] = np.dot(np.linalg.pinv(N[:,:,j,i]) ,b[:,:,j,i]).reshap
e(-1)





    return C, pts


# feature detection
def RunFeatureDetection(I, w, t, w_nms):
    N, b, J0, J1 = ImageGradient(I, w, t)
    J2 = NMS(J1, w_nms)
    C, pts = ForstnerCornerDetector(J2, N, b)
    return C, pts, J0, J1, J2
```

In [4]:

```
# input images
#I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
#I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.

# parameters to tune
w = 7
t1 = 8.3*10**-4
t2 = 8.7*10**-4
w_nms = 7




tic = time.time()

# run feature detection algorithm on input images
C1, pts1, J1_0, J1_1, J1_2 = RunFeatureDetection(I1_gray, w, t1, w_nms)
C2, pts2, J2_0, J2_1, J2_2 = RunFeatureDetection(I2_gray, w, t2, w_nms)
toc = time.time() - tic

print('took %f secs'%toc)

# display results
plt.figure(figsize=(14,24))

# show corners on original images
ax = plt.subplot(1,2,1)
plt.imshow(I1)
for i in range(C1): # draw rectangles of size w around corners
    x,y = pts1[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
# plt.plot(pts1[0,:], pts1[1,:], '.b') # display subpixel corners
plt.title('found %d corners'%C1)

ax = plt.subplot(1,2,2)
plt.imshow(I2)
for i in range(C2):
    x,y = pts2[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
# plt.plot(pts2[0,:], pts2[1,:], '.b')
plt.title('found %d corners'%C2)

plt.show()
```

```
took 86.668856 secs
```





Final values for parameters

- w = 7
- t1 = 8.3*10**-4
- t2 = 8.7*10**-4
- w_nms = 7
- C1 = 609
- C2 = 621

Problem 2 (Programming): Feature matching (15 points)

Determine the set of one-to-one putative feature correspondences by performing a brute-force search for the greatest correlation coefficient value (in the range [-1, 1]) between the detected features in image 1 and the detected features in image 2. Only allow matches that are above a specified correlation coefficient threshold value (note that calculating the correlation coefficient allows for adjusting the size of the matching window without changing the threshold value). Further, only allow matches that are above a specified distance ratio threshold value, where distance is measured to the next best match for a given feature. Vary these parameters such that around 200 putative feature correspondences are established. Optional: constrain the search to coordinates in image 2 that are within a proximity of the detected feature coordinates in image 1.

Report your final values for:

- the size of the matching window
- the correlation coefficient threshold
- the distance ratio threshold
- the size of the proximity window (if used)
- the resulting number of putative feature correspondences (i.e. matched features)

Display figures for:

- pair of images, where the matched features in each of the images are indicated by a square window about the feature and a line segment is drawn from the feature to the coordinates of the corresponding feature in the other image

In [5]:

```
def NCC(img1, img2, pts1, pts2, w, p):
    # compute the normalized cross correlation between image patches I1, I2
    # result should be in the range [-1,1]
    #
    # inputs:
    # I1, I2 are the input images
    # pts1, pts2 are the point to be matched
    # w is the size of the matching window to compute correlation coefficients
    # p is the size of the proximity window
    #
    # output:
    # normalized cross correlation matrix of scores between all windows in
    #    image 1 and all windows in image 2

    """your code here"""
    pts1_n = pts1.shape[1]
    pts2_n = pts2.shape[1]
    scores = np.zeros((pts1_n,pts2_n))
    R = int(w/2)
    pts1_int = pts1.astype(int)
```

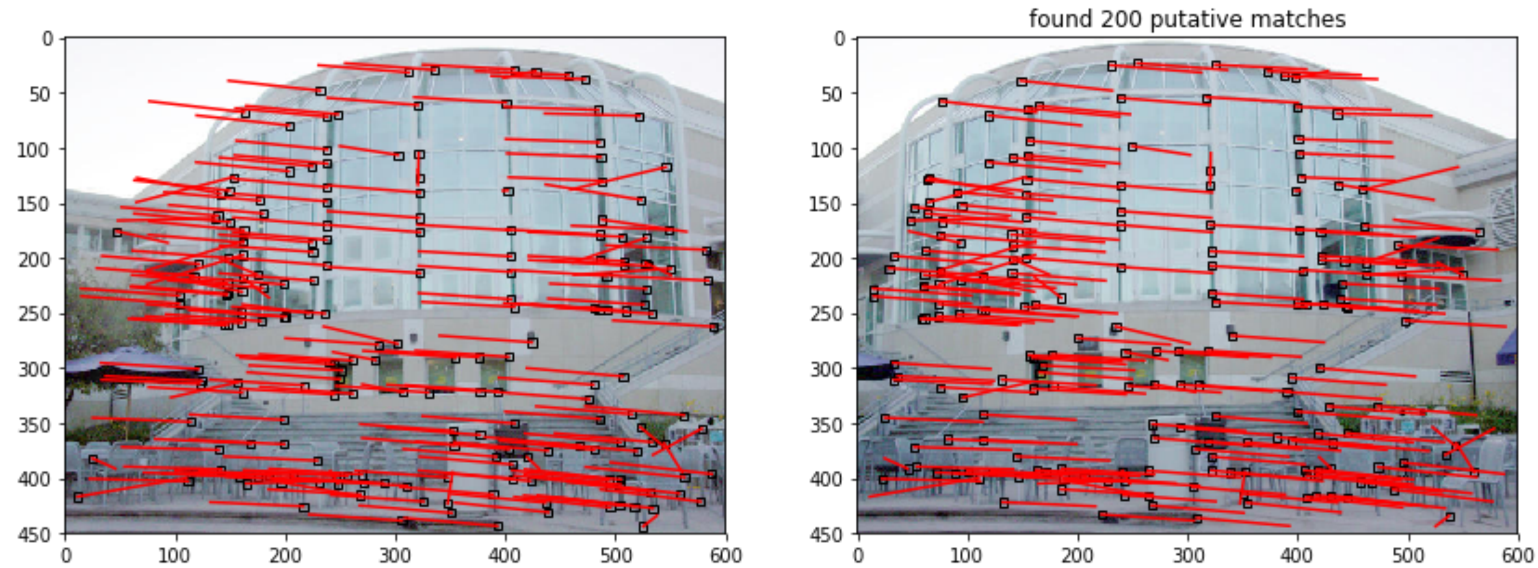
```
    pts2_int = pts2.astype(int)

    k = 0
    p_xr = int(p[1]/2)
    p_yr = int (p[0]/2)
    for j in range (pts1_n):
        x_1,y_1 = pts1_int[:,j]
        if R < x_1 < (int(img1.shape[1])-R) and R< y_1 < (int(img1.shape[0])-R):
            for i in range (pts2_n):
                x_2,y_2 = pts2_int[:,i]
                if R < x_2 < (int(img2.shape[1])-R) and R < y_2 < (int(img2.shap
e[0])-R) and x_1-p_xr < x_2 < x_1+p_xr and y_1-p_yr < y_2 < y_1+p_yr:
                    W1 = img1[y_1-R:y_1+R+1,x_1-R:x_1+R+1]
                    W2 = img2[y_2-R:y_2+R+1,x_2-R:x_2+R+1]
                    W1_mean = np.mean(W1)
                    W2_mean = np.mean(W2)
                    W1_num = W1 - W1_mean
                    W2_num = W2 - W2_mean
                    W1_tilde = W1_num/np.linalg.norm(W1_num)
                    W2_tilde = W2_num/np.linalg.norm(W2_num)
                    scores[j,i] = np.sum(W1_tilde*W2_tilde)
                    #print([j,i])
    return scores


def Match(scores, t, d):
    # perform the one-to-one correspondence matching on the correlation coeffici
ent matrix
    #
    # inputs:
    # scores is the NCC matrix
    # t is the correlation coefficient threshold
    # d distance ration threshold
    #
    # output:
    # list of the feature coordinates in image 1 and image 2

    """your code here"""
    for i in range (scores.shape[0]):
        for j in range (scores.shape[1]):
            if scores[i,j] < t:
                scores[i,j] = -1

    mask = (scores < 100)
    inds = np.zeros((2,200))
    #scores_copy = scores.copy()
    WTH = scores.shape[1]
    LTH = scores.shape[0]
    j = 0
    i = 0
    while True:
        if scores[mask].shape == (0,) :
            print('Not enough 200 matching points')
            break
```

```
        scores_copy = scores[mask].reshape((LTH-j,WTH-j))


        first_maximum = scores_copy.max()
        max_pos_copy = np.where(scores_copy == first_maximum)
        scores_copy[max_pos_copy] = -1
        next_maximum = max(scores_copy[max_pos_copy[0],:].max(),scores_copy[:,ma
x_pos_copy[1]].max())
        max_pos = np.where(scores == first_maximum)
        if (1-first_maximum) < (1- next_maximum)*d:
            inds[0,i] = max_pos[0]
            inds[1,i] = max_pos[1]
            i += 1
            if i == 200:
                break
        mask[max_pos[0],:] = False
        mask[:,max_pos[1]] = False
        j += 1


            #print(j)
    inds = inds.astype(int)

    return inds




def RunFeatureMatching(I1, I2, pts1, pts2, w, t, d, p):
    # inputs:
    # I1, I2 are the input images
    # pts1, pts2 are the point to be matched
    # w is the size of the matching window to compute correlation coefficients
    # t is the correlation coefficient threshold
    # d distance ration threshold
    # p is the size of the proximity window
    #
    # outputs:
    # inds is a 2xk matrix of matches where inds[0,i] indexs a point pts1
    #     and inds[1,i] indexs a point in pts2, where k is the number of matches

    scores = NCC(I1, I2, pts1, pts2, w, p)
    inds = Match(scores, t, d)
    return inds
```

In [6]:

```
# parameters to tune
w = 7
t = 0.8
d = 0.9
p = np.array([60,200])

tic = time.time()
# run the feature matching algorithm on the input images and detected features
inds = RunFeatureMatching(I1_gray, I2_gray, pts1, pts2, w, t, d, p)
toc = time.time() - tic

print('took %f secs'%toc)

# create new matrices of points which contain only the matched features
match1 = pts1[:,inds[0,:]]
match2 = pts2[:,inds[1,:]]

# # display the results
plt.figure(figsize=(14,24))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
plt.title('found %d putative matches'%match1.shape[1])
for i in range(match1.shape[1]):
    x1,y1 = match1[:,i]
    x2,y2 = match2[:,i]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w/2,y1-w/2),w,w, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w/2,y2-w/2),w,w, fill=False))

plt.show()

print('unique points in image 1: %d'%np.unique(inds[0,:]).shape[0])
print('unique points in image 2: %d'%np.unique(inds[1,:]).shape[0])
```

```
took 4.142421 secs
```




```
unique points in image 1: 200
unique points in image 2: 200
```

Final values for parameters

- w = 7
- t = 0.8
- d = 0.9
- p = np.array([60,200])
- num_matches = 200

Problem 3 (Programming): Outlier Rejection (15 points)

The resulting set of putative point correspondences should contain both inlier and outlier correspondences (i.e., false matches). Determine the set of inlier point correspondences using the M-estimator Sample Consensus (MSAC) algorithm, where the maximum number of attempts to find a consensus set is determined adaptively. For each trial, you must use the 4-point algorithm (as described in lecture) to estimate the planar projective transformation from the 2D points in image 1 to the 2D points in image 2. Calculate the (squared) Sampson error as a first order approximation to the geometric error.

hint: this problem has codimension 2

Report your values for:

- the probability p that as least one of the random samples does not contain any outliers
- the probability α that a given point is an inlier
- the resulting number of inliers
- the number of attempts to find the consensus set
- the tolerance for inliers
- the cost threshold

Display figures for:

- pair of images, where the inlier features in each of the images are indicated by a square window about the feature and a line segment is drawn from the feature to the coordinates of the corresponding feature in the other image

In [7]:

```
def Homogenize(x):
    # converts points from inhomogeneous to homogeneous coordinates
    return np.vstack((x,np.ones((1,x.shape[1]))))

def Dehomogenize(x):
    # converts points from homogeneous to inhomogeneous coordinates
    return x[:-1]/x[-1]
```

In [8]:

```
def four_pts_estH(x1,x2):
    #  estimate the planar projective transformation from the 2D pts in image 1
to the 2D pts in image 2 with 4 randomly chosen 2d inhomo pts
    # inputs:
    #     x1 - 4 2d inhomo pts vertical stacked from image 1
    #     x2 - 4 2d inhomo pts vertical stacked from image 2
    # outputs:
    #     H12 - planar projective matrix from img1 to img2 3*3 (x1 ---> x2)
    #   use H12 to est sampson error
    A1 = Homogenize(x1[:,:-1])
    B1 = Homogenize(x1[:,-1].reshape(-1,1))
    lamb1 = np.linalg.inv(A1) @ B1
    inv_H1 = np.zeros((3,3))
    for i in range (3):
        inv_H1[:,i] = lamb1[i] * A1[:,i]
    A2 = Homogenize(x2[:,:-1])
    B2 = Homogenize(x2[:,-1].reshape(-1,1))
    lamb2 = np.linalg.inv(A2) @ B2
    inv_H2 = np.zeros((3,3))
    for i in range (3):
        inv_H2[:,i] = lamb2[i] * A2[:,i]

    H12 = inv_H2 @ np.linalg.inv(inv_H1)


    return H12
```

In [23]:

```
def sampson_error(x1,x2, H):
    # Calculate the (squared) Sampson error as a first order approximation to th
e geometric error
    # inputs:
    #    x1,x2: SINGLE 2d inhomo (2*1) corresponding pts from WHOLE datapts (no
need to be one of the 4 pts used to est H before)
    #    H: Planar projection matrix map x1 to x2 (x1 --> x2)
    #    epsilon: residual error ah   2*1
    #    J: 2*4
    # output:
    #    sqaured sampson error
    #    cor_x: two 2d inhomo correspoding pts vertical stacked after sampson co
rrection
    h = H.reshape(-1,1)
    x1 = Homogenize(x1)
    #x2 = Homogenize(x2)
    a_left = np.vstack((np.zeros((1,3)), x1.T))
    a_mid = np.vstack((-x1.T, np.zeros((1,3))))
    a_right = np.vstack((x2[1,0]*x1.T, -x2[0,0]*x1.T))

    a = np.hstack((a_left,a_mid,a_right))
    epsilon = a @ h

    J = np.array([[-H[1,0]+x2[1,0]*H[2,0], -H[1,1]+x2[1,0]*H[2,1],0, x1[0,0]*H[2
,0]+x1[1,0]*H[2,1]+H[2,2]],
                  [H[0,0]-x2[0,0]*H[2,0],H[0,1]-x2[0,0]*H[2,1], -(x1[0,0]*H[2,0]
+x1[1,0]*H[2,1]+H[2,2]),0]])
    lamb = - np.linalg.inv(J @ J.T) @ epsilon
    error = J.T @ lamb
    cor_x = np.vstack((Dehomogenize(x1),x2)) + error
    #sqr_error = error.T @ error
    #sqr_error = np.linalg.norm(error)**2
    sqr_error = epsilon.T @ np.linalg.inv((J @ J.T))@ epsilon

    return sqr_error, cor_x
```

In [11]:

```
def Rej_outlier(match1,match2,H,tol):
    # reject outlier with sampson error
    # if the sampson error is greater than the tol, then that pair of pt will be regard as outlier. vice versa
    # inputs:
    #   match1,match2 : 2d inhomo matched features coordinates
    #   tolerance
    # outputs:
    #   N : number of inliers
    #   inliers: inlier index of matched pts
    #   cost
    cost = 0
    inliers = []
    for i in range (match1.shape[1]):
        error,_ = sampson_error(match1[:,i].reshape(-1,1),match2[:,i].reshape(-1,1), H)
        if error < tol :
            cost += error
            inliers.append(i)
        else:
            cost += tol
    N = len(inliers)
    return inliers, cost ,N
```

In [17]:

```
from scipy.stats import chi2
from math import log

def MSAC(match1, match2, thresh, tol, p):
    # Inputs:
    #    match1 - matched feature correspondences in image 1
    #    match2 - matched feature correspondences in image 2
    #    thresh - cost threshold
    #    tol - reprojection error tolerance
    #    p - probability that as least one of the random samples does not contain any outliers
    #    s - sample size for estimating max trail
    # Output:
    #    consensus_min_cost - final cost from MSAC
    #    consensus_min_cost_model - planar projective transformation matrix H
    #    inliers - list of indices of the inliers corresponding to input data
    #    trials - number of attempts taken to find consensus set

    """your code here"""
    trials = 0
    s = 4
    max_trials = np.inf
    consensus_min_cost = np.inf
    consensus_min_cost_model = np.zeros((3,3))
    while (trials < max_trials) and (consensus_min_cost > thresh):
```

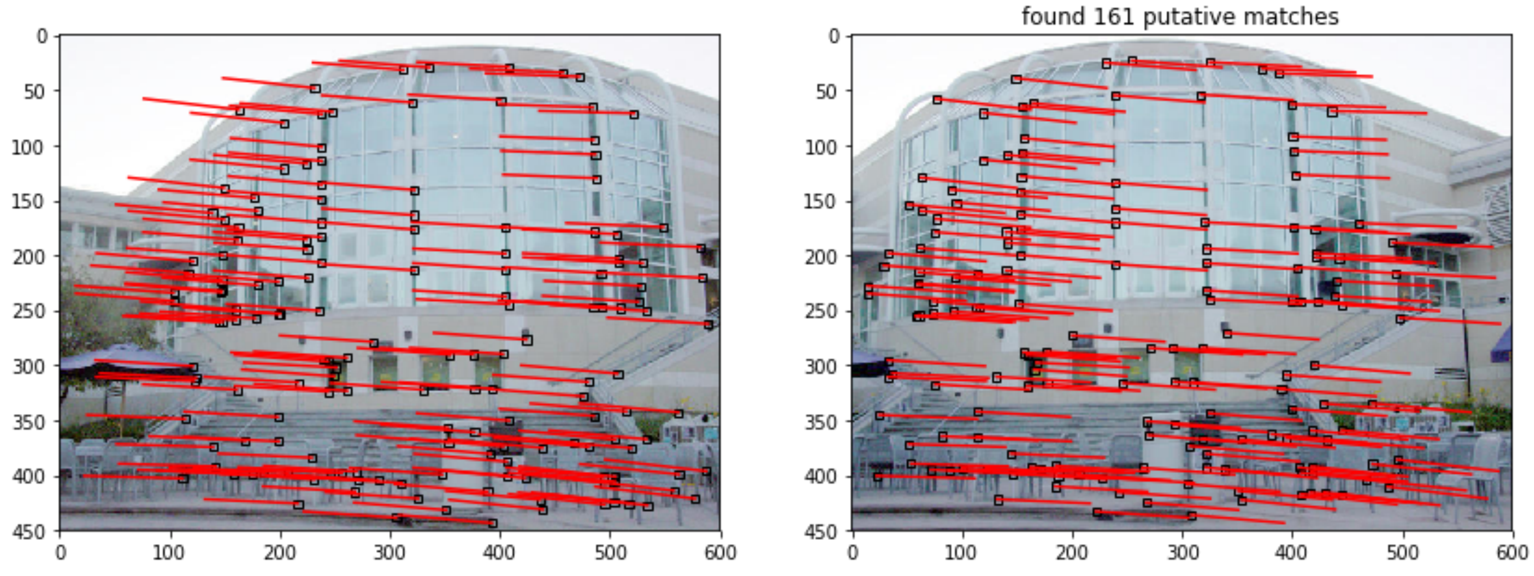
```
    while (trials < max_trials) and (consensus_min_cost > thresh):

    # consensus_min_cost_model --- camera projection matrix P

        idx = np.random.choice(match1.shape[1], size = 4, replace = False)   #[4
,53,51]
        #idx = np.array([3,4,5,6])
        x1_4pts = match1[:,idx]
        x2_4pts = match2[:,idx]
        H = four_pts_estH(x1_4pts,x2_4pts)
        inliers,cost, N = Rej_outlier(match1,match2,H,tol)
        if cost < consensus_min_cost:
            consensus_min_cost = cost
            consensus_min_cost_model = H
            global_inliers = inliers
            w = N / match1.shape[1]

            max_trials = log(1-p)/log(1-w**s)
        trials += 1




    return consensus_min_cost, consensus_min_cost_model, global_inliers, trials


# MSAC parameters
alpha = 0.95   #
df = 2
tol =  chi2.ppf(alpha,df)
thresh = 0
p = 0.99  # probability that as least one of the random samples does not contain
any outliers


tic=time.time()

cost_MSAC, H_MSAC, inliers, trials = MSAC(match1, match2, thresh, tol, p)

# choose just the inliers
new_match1 = match1[:,inliers]
new_match2 = match2[:,inliers]
outliers = np.setdiff1d(np.arange(pts1.shape[1]),inliers)

toc=time.time()
time_total=toc-tic

# display the results
print('took %f secs'%time_total)
print('%d iterations'%trials)
print('inlier count: ',len(inliers))
print('inliers: ',inliers)
print('MSAC Cost=%.9f'%cost_MSAC)
print('H_MSAC = ')
```

```
print(H_MSAC)
```

```
took 0.486661 secs
15 iterations
inlier count:  161
inliers:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
36, 38, 39, 40, 41, 42, 43, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55,
56, 57, 59, 60, 61, 62, 63, 64, 65, 66, 67, 69, 71, 72, 73, 74, 76,
77, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 91, 92, 93, 94, 95,
96, 98, 99, 102, 103, 104, 105, 106, 107, 108, 109, 112, 113, 115, 1
16, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
130, 132, 133, 134, 135, 136, 137, 139, 140, 141, 142, 143, 144, 145
, 147, 149, 150, 151, 152, 154, 157, 158, 160, 161, 163, 164, 166, 1
67, 168, 169, 171, 173, 177, 179, 180, 182, 183, 184, 185, 186, 187,
190, 191, 193, 195, 197, 198, 199]
MSAC Cost=288.634610095
H_MSAC =
[[ 1.00720870e+00 -4.01198714e-04 -9.00350326e+01]
 [ 2.59627657e-02  9.84944481e-01 -1.48398202e+01]
 [ 1.05158893e-04  1.01634461e-05  9.46085578e-01]]
```

In [18]:

```
# display the figures
"""your code here"""
plt.figure(figsize=(14,24))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
plt.title('found %d putative matches'%new_match1.shape[1])
for i in range(new_match1.shape[1]):
    x1,y1 = new_match1[:,i]
    x2,y2 = new_match2[:,i]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w/2,y1-w/2),w,w, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w/2,y2-w/2),w,w, fill=False))

plt.show()

print('unique points in image 1: %d'%len(inliers))
print('unique points in image 2: %d'%len(inliers))
```




```
unique points in image 1: 161
unique points in image 2: 161
```

Final values for parameters

- p = 0.99
- α = 0.95
- tolerance = 5.99146454710798
- threshold = 0
- num_inliers = 161
- num_attempts = 15

Problem 4 (Programming): Linear Estimate (15 points)

Estimate the planar projective transformation $\boldsymbol{H}_{\text{DLT}}$ from the resulting set of inlier correspondences using the direct linear transformation (DLT) algorithm (with data normalization). You must express $\boldsymbol{x}_i' = \boldsymbol{H}\boldsymbol{x}_i$ as $[\boldsymbol{x}_i']^\perp \boldsymbol{H}\boldsymbol{x}_i = \boldsymbol{0}$ (not $\boldsymbol{x}_i' \times \boldsymbol{H}\boldsymbol{x}_i = \boldsymbol{0}$), where $[\boldsymbol{x}_i']^\perp \boldsymbol{x}_i' = \boldsymbol{0}$, when forming the solution. Return $\boldsymbol{H}_{\text{DLT}}$, scaled such that $||\boldsymbol{H}_{\text{DLT}}||_{\text{Fro}} = 1$

In [201]:

```python
def Normalize(pts):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates
    # Outputs:
    #    pts - data normalized points homo
    #    T - corresponding transformation matrix
    """your code here"""
    dimension = pts.shape[0]
    variance = np.var(pts,axis = 1)
    mean = np.mean(pts,axis = 1)
    var_tol = variance.sum()
    S = np.sqrt(dimension/var_tol)
    T = np.eye(pts.shape[0]+1)
    T[:dimension, :dimension] = S * np.eye(dimension)
    for i in range (dimension) :
        T[i,-1] = -S*mean[i]
    pts_homo = Homogenize(pts)
    pts = T @ pts_homo  #home - W scale
    #pts = Dehomogenize(pts_normalized)   #inhomi - W/O scale

    return pts, T



def left_null_calculator(x):
    # calculate x's left null space (2 *  x.shape[0])
    #Inputs:
    #   x - homo W scale
    #   Hv - Household matrix
    #   v - Household matrix
    #Outputs:
    #   x_left - left null space of x
    x_left = np.zeros((x.shape[0]-1,x.shape[0]*x.shape[1]))
    e = np.zeros((x.shape[0],1))
    e[0] = 1
    for i in range (x.shape[1]):
        sign = np.sign(x[0,i])
        v = x[:,i].reshape(-1,1) + sign * np.linalg.norm(x[:,i]) *e
        Hv = np.eye((x.shape[0])) -2 * np.dot(v,v.T)/np.dot(v.T,v)
        x_left[:,i*x.shape[0]:(i+1)*x.shape[0]] = Hv[1:,:]
    return x_left
```

```
    return x_left

def ComputeCost(H, x1, x2):
    # Inputs:
    #    x1,x2 - 2D inhomogeneous image points NOT Normalized
    #
    #    H - planar projective transformation
    # Output:
    #    cost - Total reprojection error
    """your code here"""

    x1_sampson = np.zeros((2,x1.shape[1]))

    for i in range (x1.shape[1]):
        _, cor_x = sampson_error(x1[:,i].reshape(-1,1),x2[:,i].reshape(-1,1), H)
        x1_sampson[:,i] = cor_x[:2,0]


    x2_predict = Dehomogenize(H @ Homogenize(x1_sampson))


    cost = ((x2 - x2_predict)**2).sum() + ((x1- x1_sampson)**2).sum()
    return cost
```

In [202]:

```
def DLT(x1, x2, normalize=True):
    # Inputs:
    #    x1 - inhomogeneous inlier correspondences in image 1
    #    x2 - inhomogeneous inlier correspondences in image 1
    #    normalize - if True, apply data normalization to x1 and x2 which return
s homo normalized pts x1 x2
    #
    # Outputs:
    #    H - the DLT estimate of the planar projective transformation
    #    cost - linear estimate cost

    """your code here"""

    H = np.eye(3,3)

    # data normalization
    if normalize:
        x1, U = Normalize(x1)
        x2, T = Normalize(x2)
    else:
        x1 = Homogenize(x1)
        x2 = Homogenize(x2)

    """your code here"""
    x_left = left_null_calculator(x2)

    A = np.zeros((2*x2.shape[1],9))
    for i in range (x2.shape[1]):
```

```
        A[2*i:2*i+2,:] = np.kron(x_left[:,i*x2.shape[0]: (i+1)*x2.shape[0]],x1[:
,i].T)

    u,s,vh = np.linalg.svd(A)
    H_bar = vh[-1,:]
    # per rows
    H = H_bar.reshape(3,3)

    # data denormalize
    if normalize:
        H = np.linalg.inv(T) @ H @ U
        cost = ComputeCost(H,Dehomogenize(np.linalg.inv(U) @ x1),Dehomogenize(np
.linalg.inv(T) @ x2))
    else:
        cost = ComputeCost(H,Dehomogenize(x1),Dehomogenize(x2))

    return H, cost


# compute the linear estimate without data normalization
print ('Running DLT without data normalization')
time_start=time.time()
H_DLT, cost = DLT(new_match1, new_match2, normalize=False)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost = %.9f\n'%cost)


# compute the linear estimate with data normalization
print ('Running DLT with data normalization')
time_start=time.time()
H_DLT, cost = DLT(new_match1, new_match2, normalize=True)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost = %.9f'%cost)
```

```
Running DLT without data normalization
took 0.101834 secs
Cost = 28.252573996

Running DLT with data normalization
took 0.036860 secs
Cost = 27.514428744
```

In [203]:

```
# display your H_DLT, scaled with its frobenius norm
print('H_DLT = \n',H_DLT/np.linalg.norm(H_DLT))
```

```
H_DLT =
 [[ 1.09953289e-02 -1.40813364e-05 -9.84779819e-01]
 [ 3.15049403e-04  1.07157242e-02 -1.72822204e-01]
 [ 1.22603591e-06  8.98128653e-08  1.02649688e-02]]
```

Problem 5 (Programming): Nonlinear Estimate (45 points)

Use $\boldsymbol{H}_{\text{DLT}}$ and the Sampson corrected points (in image 1) as an initial estimate to an iterative estimation method, specifically the sparse Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the planar projective transformation that minimizes the reprojection error. You must parameterize the planar projective transformation matrix and the homogeneous 2D scene points that are being adjusted using the parameterization of homogeneous vectors (see section A6.9.2 (page 624) of the textbook, and the corrections and errata).

Report the initial cost (i.e. cost at iteration 0) and the cost at the end of each successive iteration. Show the numerical values for the final estimate of the planar projective transformation matrix $\boldsymbol{H}_{\text{LM}}$, scaled such that $||\boldsymbol{H}_{\text{LM}}||_{\text{Fro}} = 1$.

In [223]:

```
from numpy import sin, cos , pi
from math import ceil
def Sinc(x):
    # Returns a scalar valued sinc value
    """your code here"""
    if x == 0:
        y = 1
    else:
        y = sin(x)/x

    return y



def Parameterize(P):
    # wrapper function to interface with LM
    # takes all optimization variables and parameterizes all of them
    # in this case it is just P, but in future assignments it will
    # be more useful
    return ParameterizeHomog(P.reshape(-1,1))


def Deparameterize(p):
    # Deparameterize all optimization variables
    return DeParameterizeHomog(p).reshape(3,3)
```

```python
def ParameterizeHomog(V):
    ########## np.linalg.norm(V) must equal to 1#################


    # Given a homogeneous vector V return its minimal parameterization
    """your code here"""
    a = V[0]
    b = V[1:]
    v_hat = (2 * b)/Sinc(np.arccos(a))

    v_norm = np.linalg.norm(v_hat)
    if v_norm >= pi:
        v_hat = (1 - (2*pi/v_norm)*ceil((v_norm-pi)/(2*pi)))* v_hat
        #print(np.linalg.norm(v_hat)-pi )

    return v_hat


def DeParameterizeHomog(v):
    # Given a parameterized homogeneous vector return its deparameterization 11*
1 --> 12*1
    """your code here"""
    v_bar = np.zeros((v.shape[0]+1,1))
    v_bar[0] = cos(np.linalg.norm(v)/2)
    v_bar[1:] = Sinc(np.linalg.norm(v)/2)/2 * v
    return v_bar
```

In [224]:

```python
def corrected_pts(x1,x2,H):
    # Calculate the (squared) Sampson error as a first order approximation to th
e geometric error
    # the difference between this func and preview one (sampson_error) is that x
1,x2 in this func is 2*n instead of 2*1
    # inputs:
    #    x1,x2: 2d inhomo (2*n) corresponding pts
    #    H: Planar projection matrix map x1 to x2 (x1 --> x2)

    # output:
    #    x1_sampson : corrected pts by sampson correction  im img 1
    x1_sampson = np.zeros((2,x1.shape[1]))
    for i in range (x1.shape[1]):
        _, cor_x = sampson_error(x1[:,i].reshape(-1,1),x2[:,i].reshape(-1,1), H)
        x1_sampson[:,i] = cor_x[:2,0]



    return x1_sampson
```

In [231]:

```python
def normal_eq_parameters(A,B,B_prm,x1,x2,x1_sampson,inv_cov1,inv_cov2,lam,error1
```

```python
,error2):
    # Calculate the sigma_a ,sigma_b to correct Planar projection matrix H and s
ampson cprrected 2d inhomo pts respectively
    #
    # inputs:
    #    A,B,B_prm: Block matrix of Jacobian
    #    A - 2n * 8
    #    B,B_prm - 2n*2
    #    x1,x2: normalized 2d corresponding inhomo pts in two images  # only x1.
shape[0] used in this func
    #    x1_sampson: sampson corrected pts of image 1   #not using in this func
    #    error1 , error2 : 2n*1
    # output:
    #    sigma_a - correction for Planar transformation matrix  8*1
    #    sigma_b - correction for x1 sampson pts  2*2n
    U = np.zeros((8,8))
    V = np.zeros((2 * x1.shape[1],2 *x1.shape[1])) # 2n*2n
    W = np.zeros((8 ,2* x1.shape[1]))   # 8*2n
    epsilon_a = np.zeros((8,1))
    epsilon_b = np.zeros((2 * x1.shape[1],1))  #2n*1

    U = A.T @ inv_cov1 @ A

    epsilon_a = A.T @ inv_cov2 @ error2 # 8*1
    sigma_b = np.zeros((2, x1.shape[1]))

    for i in range (x1.shape[1]):

        V[2*i:2*i+2,2*i:2*i+2] = (B[2*i:2*i+2,:].T @ inv_cov1[0:2,0:2] @ B[2*i:2
*i+2,:]
                             + B_prm[2*i:2*i+2,:].T @ inv_cov2[0:2,0:2] @ B_prm[2*i
:2*i+2,:])

        W[:,2*i:2*i+2] = A[2*i:2*i+2,:].T @ inv_cov2[0:2,0:2] @ B_prm[2*i:2*i+2,
:]

        epsilon_b[2*i:2*i+2,:] = (B[2*i:2*i+2,:].T @ inv_cov1[0:2,0:2] @ error1[
2*i:2*i+2,:]
                                     + B_prm[2*i:2*i+2,:].T @ inv_cov2[0:2,0:2] @ e
rror2[2*i:2*i+2,:])


    inv_V = np.linalg.inv(V + lam * np.eye(2 *x1.shape[1]))   #2n*2n
    S = U + lam * np.eye(8) - W @ (inv_V) @ W.T
    e = epsilon_a - W @ inv_V @ epsilon_b
    sigma_a = np.linalg.inv(S) @ e   # 8*1



    for j in range (x1.shape[1]):

        #V_augm_inv = np.linalg.inv(V[2*j:2*j+2,2*j:2*j+2] + lam *np.eye(2))
        sigma_b[:,j] = (inv_V[2*j:2*j+2,2*j:2*j+2] @ (epsilon_b[2*j:2*j+2,:] - W
```

```
[:,2*j:2*j+2].T @ sigma_a)).reshape(-1)



    return sigma_a, sigma_b
```

In [229]:

```
def Jacobian(H,h,x1homo, x1,H2X = True):
    # compute the jacobian matrix
    #
    # Input:
    #    H - 3x3 planar projection matrix ##should be mormalized
    #    h - 8x1 homogeneous parameterization of H
    #    x1homo - homo sampson 2d pts       3*N
    #    x1 - Normalized 2d inhomo After the following process Sampson Corrected
--> Homo   --> para  2*n
    #    H --- x1homo
    #    h --- x1


    #    H2X(True) : x2 = J @ H    or    x = J @ P
    #    H2X(False) : x2 = J @ x1   or x = J @ X ,where x2 = H @ x1   x = P @ X
    # Output:
    #    J - 2nx8 jacobian matrix

    #J = np.zeros((2*x1.shape[1],h.shape[0]))

    """your code here"""
    x_homo = H @ x1homo  #x_estimate



    if H2X:
        H_bar = H.reshape(-1,1)  #9*1
        p = h

        b = H_bar[1:].reshape(-1,1)  #8*1
        a_diff = np.zeros((1,p.shape[0])) #1*8
        b_diff = np.eye(p.shape[0])/2
        n_p = np.linalg.norm(p)  #norm_p
        h_p = n_p/2              #half_norm_p
        diff_sinc = cos(h_p)/h_p - sin(h_p)/(h_p**2)
        if n_p != 0:
            a_diff = -b.T/2
            b_diff = Sinc(h_p)*np.eye(h.shape[0])/2 + (p @ p.T)*diff_sinc/(4*n_p
)

        P_bar_over_p = np.vstack((a_diff,b_diff)) # H_bar.shape[0] * (H_bar.shap
e[0]-1)


    else:
```

```
        #    H --- x1homo
        #    h --- x1

        P_bar_over_p = np.zeros((3*x1.shape[1],2)) #3n *2
        for i in range (x1.shape[1]):

            H_bar = x1homo[:,i].reshape(-1,1)

            p = x1[:,i].reshape(-1,1)

            b = H_bar[1:,0].reshape(-1,1)  #8*1
            a_diff = np.zeros((1,p.shape[0])) #1*8
            b_diff = np.eye(p.shape[0])/2
            n_p = np.linalg.norm(p)  #norm_p
            h_p = n_p/2              #half_norm_p
            diff_sinc = cos(h_p)/h_p - sin(h_p)/(h_p**2)
            if n_p != 0:
                a_diff = -b.T/2
                b_diff = Sinc(h_p)*np.eye(p.shape[0])/2 + (p @ p.T)*diff_sinc/(4
*n_p)

            P_bar_over_p[3*i: 3*i+3,:] = np.vstack((a_diff,b_diff)) # 3n*2



    x_in = Dehomogenize(x_homo) #x inhomo 2*N
    w = x_homo[-1]

    if H2X :
        x_hat_over_P_bar = np.zeros((2*x1.shape[1],h.shape[0]+1 )) #2n*9

        for i in range (x1.shape[1]):
            left = np.vstack((x1homo[:,i].reshape(-1,1).T,np.zeros([1,3])))
            mid = np.vstack((np.zeros([1,3]),x1homo[:,i].reshape(-1,1).T))
            right = np.vstack((-x_in[0,i]*x1homo[:,i].reshape(-1,1).T,-x_in[1,i]
*x1homo[:,i].reshape(-1,1).T))

            x_hat_over_P_bar[2*i:2*i+2,:] = (1/w[i])* np.hstack((left,mid,right)
)
        J = x_hat_over_P_bar @ P_bar_over_p

    else:
        x_hat_over_P_bar = np.zeros((2*x1.shape[1],x1.shape[0])) # 2n*3
        J = np.zeros((2*x1.shape[1],2))
        for i in range (x1.shape[1]):
            temp = (1/w[i]) * np.vstack((H[0,:] - x_in[0,i] * H[-1,:], H[1,:] -
x_in[1,i] * H[-1,:]))

            #x_hat_over_P_bar[2*i:2*i+2,:] = (1/w[i])* temp

            J[2*i:2*i+2,:] =   temp @ P_bar_over_p[3*i: 3*i+3,:]    #2n*2
```

```
    return J
```

In [232]:

```
from scipy.linalg import block_diag

def LM(H, x1, x2, max_iters, lam):
    #all pts are normarlized
    # keep updating x_sampson, h
    # Input:
    #    H - DLT estimate of planar projective transformation matrix
    #    x1 - inhomogeneous inlier points in image 1
    #    x2 - inhomogeneous inlier points in image 2
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    # Output:
    #    H - Final H (3x3) obtained after convergence

    # data normalization

    # USE x1_sampsonhomo to get inhomo pts. Do not use x1_sampson to calculate e
rror

    x2, T = Normalize(x2) #output-homo   input - inhomo     x-normalized
    x1, U = Normalize(x1) #homo   X-normalized
    """your code here"""
    H = T @ H @ np.linalg.inv(U)   #normarlized

    h = Parameterize(H)   # map from x1_sampson ---> x2
    H = Deparameterize(h)  # B_prm

    EYE_1 = np.eye(3)/np.linalg.norm(np.eye(3))  # initial H for B

    eye_1 = Parameterize(EYE_1)     # FOR B 2N*2
    EYE_1 = Deparameterize(eye_1)   #



    x1_sampson = corrected_pts(Dehomogenize(x1),Dehomogenize(x2),H)  #  2*N   h

    x1_sampsonhomo = Homogenize(x1_sampson)

    x1_para = np.zeros((2,x1.shape[1]))   #2*n
    x1_depara = np.zeros((3,x1.shape[1])) #3*N


    for j in range (x1.shape[1]):
        temp1 =  x1_sampsonhomo[:,j]
        x1_sampsonhomo[:,j] = temp1 /np.linalg.norm(temp1)     # 3*N
        x1_sampson[:,j] = Parameterize(x1_sampsonhomo[:,j]).reshape(-1)   # 2*N
#sampson corrected pts after homo and para
```

```
        x1_sampsonhomo[:,j] = (DeParameterizeHomog(x1_sampson[:,j].reshape(-1,1)
)).reshape(-1)

        #temp = Parameterize(x1_sampsonhomo[:,j]).reshape(-1)   # 2*N #sampson c
orrected pts after homo and para

        #x1_sampsonhomo[:,j] = (DeParameterizeHomog(temp.reshape(-1,1))).reshape
(-1)


    cov_x1 = U[0,0]**2 * np.eye(2*x1.shape[1]) # 2N*2N
    cov_x2 = T[0,0]**2 * np.eye(2*x1.shape[1])
    inv_cov1 = np.linalg.inv(cov_x1)
    inv_cov2 = np.linalg.inv(cov_x2)




    A = np.zeros((2*x1.shape[1] ,8))
    A = Jacobian(H,h,x1_sampsonhomo,x1_sampson) # 2N*8

    B = np.zeros((2*x1.shape[1] ,2))
    B = Jacobian(EYE_1,eye_1,x1_sampsonhomo,x1_sampson ,H2X = False) # 2N*2

    B_prm = np.zeros((2*x1.shape[1] ,2))  # B_prime
    B_prm = Jacobian(H,h,x1_sampsonhomo,x1_sampson, H2X = False) # 2N*2

    cost = ComputeCost(np.linalg.inv(T) @ H @ U, Dehomogenize(np.linalg.inv(U) @
x1),Dehomogenize(np.linalg.inv(T) @ x2))

    error1 = np.zeros((2*x1.shape[1],1))   # 2n *1
    error2 = np.zeros((2*x1.shape[1],1))   # 2n *1


    error1 = Dehomogenize(x1).reshape(-1,1,order = 'F') - Dehomogenize(x1_sampso
nhomo).reshape(-1,1,order = 'F') # 2N *1


    #error1 = Dehomogenize(x1).reshape(-1,1,order = 'F') - x1_sampson.reshape(-1
,1,order = 'F') # 2N *1
    error2 = Dehomogenize(x2).reshape(-1,1,order = 'F') - Dehomogenize(H @ x1_sa
mpsonhomo).reshape(-1,1,order = 'F') # 2N *1

    cost_old = (error1.T @ inv_cov1 @ error1 + error2.T @ inv_cov2 @ error2)
    tolerance = 1e-7

    for i in range(max_iters):

        sigma_a,sigma_b = normal_eq_parameters(A,B,B_prm,x1,x2,x1_sampson,inv_co
v1,inv_cov2,lam,error1,error2)

        h_new = h + sigma_a.reshape(-1,1)
        H = Deparameterize(h_new)  #AFTER depara, P is normalized 3*4
```

```
        x1_sampson_new = x1_sampson + sigma_b  # 2*n

        #print('sigma_b:      ',sigma_b)



        x1_sampsonhomo_new = np.zeros((3,x1.shape[1]))
        for k in range (x1.shape[1]):

            x1_sampsonhomo_new[:,k] = (DeParameterizeHomog(x1_sampson_new[:,k].r
eshape(-1,1))).reshape(-1) # 3 * N



        x2_new = Dehomogenize(H @ x1_sampsonhomo_new)

        error1_new = Dehomogenize(x1).reshape(-1,1,order = 'F') - Dehomogenize(x
1_sampsonhomo_new).reshape(-1,1,order = 'F') # 2N*1
        error2_new = Dehomogenize(x2).reshape(-1,1,order = 'F') - x2_new.reshape
(-1,1,order = 'F')


        cost_new = (error1_new.T @ inv_cov1 @ error1_new + error2_new.T @ inv_co
v2 @ error2_new)


        print ('iter %03d Cost %.9f'%(i+1, cost_new))
        if cost_new < cost:
            if tolerance > cost - cost_new:
                break
            h = h_new
            cost = cost_new
            error1 = error1_new
            error2 = error2_new
            lam = 0.1 * lam
            x1_sampson = x1_sampson_new
            x1_sampsonhomo = x1_sampsonhomo_new
            A = Jacobian(H,h,x1_sampsonhomo,x1_sampson)
            B = Jacobian(EYE_1,eye_1,x1_sampsonhomo,x1_sampson, H2X = False)
            B_prm = Jacobian(H,h,x1_sampsonhomo,x1_sampson, H2X = False)
            #print('lam_decre:      ',lam)
        else:
            lam = 10 *lam
            #print('lam_incre:      ',lam)

    # data denormalization
    H = np.linalg.inv(T) @ H @ U

    return H


# LM hyperparameters
lam = .001
max_iters = 10
```

```
#H_DLT/np.linalg.norm(H_DLT)



# Run LM initialized by DLT estimate with data normalization
print ('Running sparse LM with data normalization')
print ('iter %03d Cost %.9f'%(0, cost))
time_start=time.time()
H_LM = LM(H_DLT, new_match1, new_match2, max_iters, lam)
time_total=time.time()-time_start
print('took %f secs'%time_total)
```

```
Running sparse LM with data normalization
iter 000 Cost 27.514428744
iter 001 Cost 27.460563335
iter 002 Cost 27.460559311
iter 003 Cost 27.460559310
took 0.320162 secs
```

In [233]:

```
# display your converged H_LM, scaled with its frobenius norm
print('H_LM = \n', H_LM/np.linalg.norm(H_LM))
```

```
H_LM =
 [[ 1.09981868e-02 -1.56561862e-05 -9.84802189e-01]
 [ 3.14919608e-04  1.07171827e-02 -1.72694396e-01]
 [ 1.22939986e-06  9.07821151e-08  1.02653120e-02]]
```