HW4

March 7, 2018

1 CSE 252B: Computer Vision II, Winter 2018 – Assignment 4

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, March 7, 2018, 11:59 PM

1.1 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.

1.2 Problem 1 (Programing): Feature Detection (20 points)

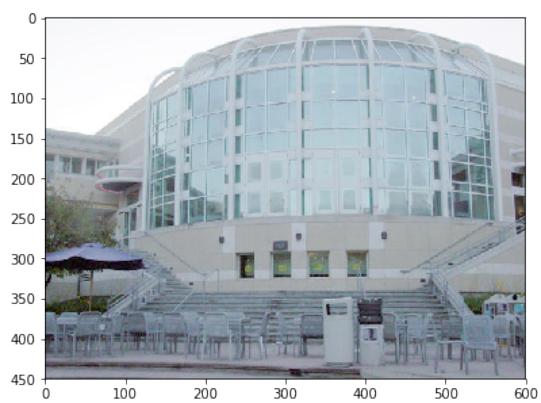
Download input data from the course website. The file price_center20.JPG contains image 1 and the file price_center21.JPG contains image 2.

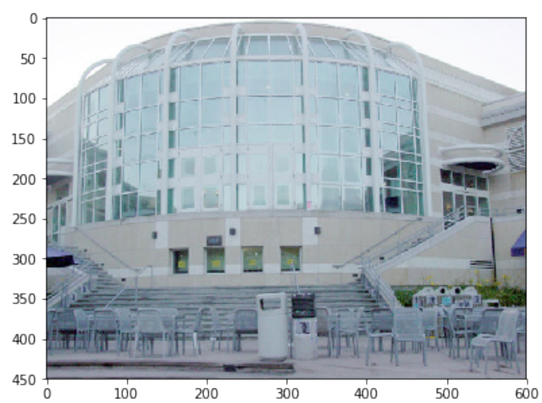
For each input image, calculate an image where each pixel value is the minor eigenvalue of the gradient matrix

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

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) non-maximum 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.

In [2]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy import signal
import random
# 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.

# Display the input images
plt.figure(figsize=(14,8))
plt.subplot(1,2,1)
plt.imshow(I1)
plt.subplot(1,2,2)
plt.imshow(I2)
plt.show()

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





In [3]:
```
def corner(I, w, t, w_nms):
    # inputs:
    # I is the input image (may be mxn for BW or mxnx3 for RGB)
```

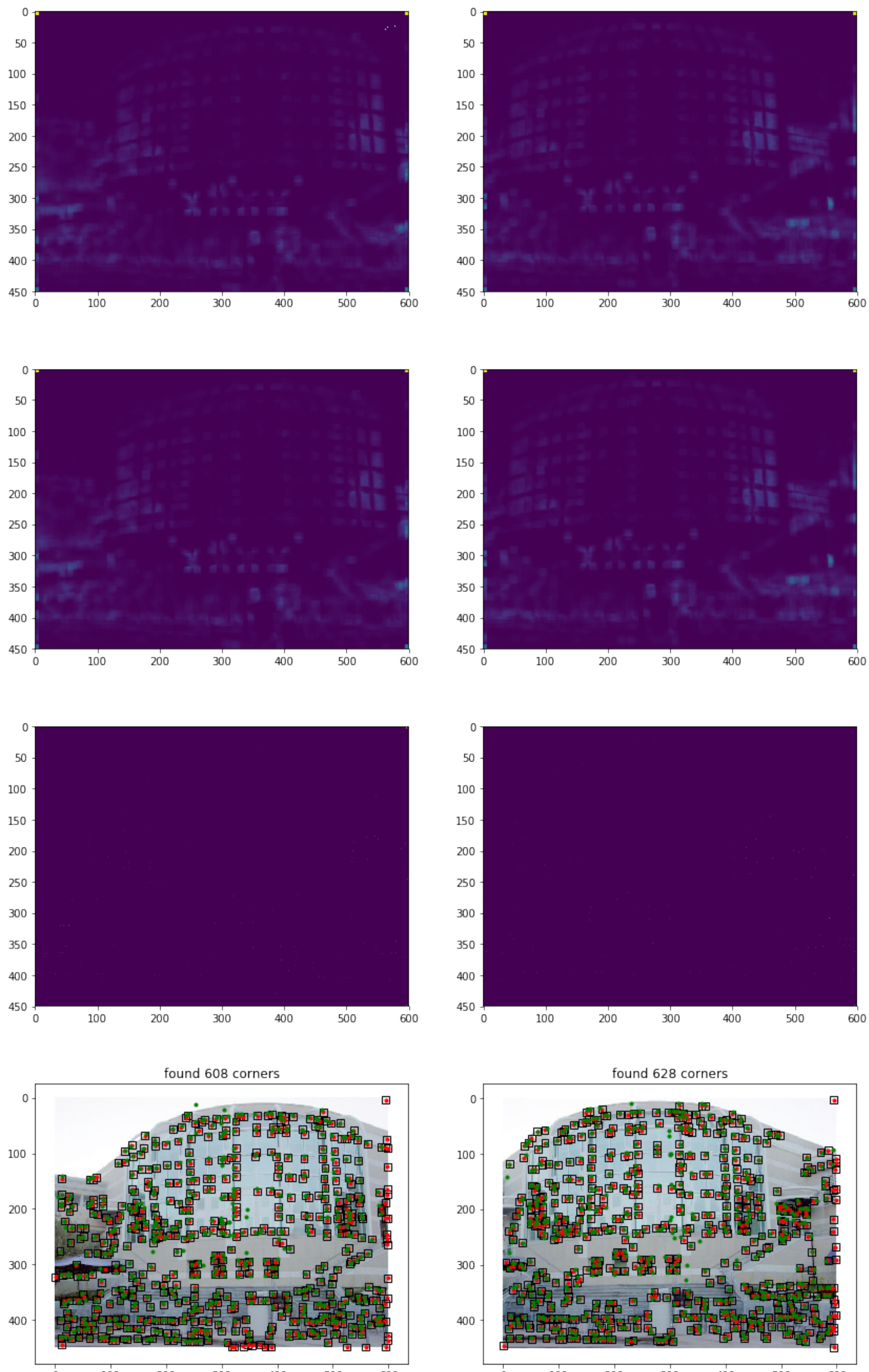
```
# w is the size of the window used to compute the gradient matrix N
# t is the minor eigenvalue threshold
# w_nms is the size of the window used for nonmaximal supression
# outputs:
# J0 is the mxn image of minor eigenvalues of N before thresholding
# J1 is the mxn image of minor eigenvalues of N after thresholding
# J2 is the mxn image of minor eigenvalues of N after nonmaximal supression
# pts0 is the 2xk list of coordinates of (pixel accurate) corners
#     (ie. coordinates of nonzero values of J2)
# pts1 is the 2xk list of coordinates of subpixel accurate corners
#     found using the Forstner detector

"""your code here"""
m, n = I.shape[:2]
J0 = np.zeros((m, n))
J1 = np.zeros((m, n))
J2 = np.zeros((m, n))

#xgradient Ix matrix
#ygradient Iy matrix
#translation and summation to calculate the gradient matrices
xgradient = np.zeros((m, n, 3))
xgradient[2:, :, :] = xgradient[2:, :, :] + np.dot(-1, I[:-2, :, :])
xgradient[1:, :, :] = xgradient[1:, :, :] + np.dot(8, I[:-1, :, :])
xgradient[:-1, :, :] = xgradient[:-1, :, :] + np.dot(-8, I[1:, :, :])
xgradient[:-2, :, :] = xgradient[:-2, :, :] + np.dot(1, I[2:, :, :])
ygradient = np.zeros((m, n, 3))
ygradient[:, 2:, :] = ygradient[:, 2:, :] + np.dot(-1, I[:, :-2, :])
ygradient[:, 1:, :] = ygradient[:, 1:, :] + np.dot(8, I[:, :-1, :])
ygradient[:, :-1, :] = ygradient[:, :-1, :] + np.dot(-8, I[:, 1:, :])
ygradient[:, :-2, :] = ygradient[:, :-2, :] + np.dot(1, I[:, 2:, :])

#N1 average IxIx matrix
#N2 average IxIy matrix
#N3 average IyIy matrix
#N01 IxIx matrix N02 IxIy N03 IyIy
N1 = np.zeros((m, n))
N2 = np.zeros((m, n))
N3 = np.zeros((m, n))
N01 = np.zeros((m, n))
N02 = np.zeros((m, n))
N03 = np.zeros((m, n))
xx = np.zeros((m, n,3))
yy = np.zeros((m, n,3))
xy = np.zeros((m, n,3))
xx = xgradient*xgradient/w/w
yy = ygradient*ygradient/w/w
xy = xgradient*ygradient/w/w
```

```python
convkernal=np.ones((w,w))
for i in range(3):
    N01 += xx[:,:,i]
    N02 += xy[:,:,i]
    N03 += yy[:,:,i]
N1 = signal.correlate2d(N01, convkernal, mode='same',boundary='symm')
N2 = signal.correlate2d(N02, convkernal, mode='same',boundary='symm')
N3 = signal.correlate2d(N03, convkernal, mode='same',boundary='symm')

#calculate minor eigenvalues
#same as the equation that use trace and determint
J0= (N1 + N3 - np.sqrt(np.square(N1 + N3) - 4 * (N1 * N3 - np.square(N2)))) / 2
for x in range(m):
    for y in range(n):
        J1[x][y] = J0[x][y] if J0[x][y] > t else 0
lenw = (w_nms - 1) // 2
flag = False
#nonmaximum suppression
for x in range(m):
    for y in range(n):
        cmax = J1[x][y]
        for x1 in range(x - lenw, x + lenw+1):
            for y1 in range(y - lenw, y + lenw+1):
                if x1 >= 0 and x1 < m and y1 >= 0 and y1 < n:
                    cmax = max(cmax, J1[x1][y1])
        J2[x][y] = J1[x][y] if J1[x][y] == cmax else 0
        if not J2[x][y] == 0:
            if not flag:
                pts0 = np.array([[y], [x]])
                flag = True
            else:
                pts0 = np.hstack((pts0, np.array([[y], [x]])))
ptsize=pts0.shape[1]
pts1=np.zeros(pts0.shape)
lenw = (w - 1) // 2
for i in range(ptsize):
    x=pts0[1][i]
    y=pts0[0][i]
    FCb1=0
    FCb2=0
    for x1 in range(x - lenw, x + lenw + 1):
        for y1 in range(y - lenw, y + lenw + 1):
            if x1 >= 0 and x1 < m and y1 >= 0 and y1 < n:
                FCb1+=x1*N01[x1][y1]+y1*N02[x1][y1]
                FCb2+=x1*N02[x1][y1]+y1*N03[x1][y1]
    det=N1[x][y]*N3[x][y]-N2[x][y]*N2[x][y]
    pts1[1][i]=int(1/det*(N3[x][y]*FCb1-N2[x][y]*FCb2))
    pts1[0][i]=int(1/det*(-N2[x][y]*FCb1+N1[x][y]*FCb2))
```

```
    pts0=pts0.astype(int)
    pts1=pts1.astype(int)
    #drop the pts near the boundary
    ilist=[i for i in range(pts1.shape[1]) if m-lenw > pts1[1][i] >=lenw and n-lenw > p
    pts1=pts1[:,ilist]
    pts0=pts0[:,ilist]
    return J0, J1, J2, pts0, pts1


# parameters to tune
w=13
t=.1
w_nms=7

# extract corners
J1_0, J1_1, J1_2, pts1_0, pts1_1 = corner(I1, w, t, w_nms)
J2_0, J2_1, J2_2, pts2_0, pts2_1 = corner(I2, w, t, w_nms)

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

# show pre-thresholded corner heat map
plt.subplot(4,2,1)
plt.imshow(J1_0)
plt.subplot(4,2,2)
plt.imshow(J2_0)

# show thresholded corner heat map
plt.subplot(4,2,3)
plt.imshow(J1_1)
plt.subplot(4,2,4)
plt.imshow(J2_1)

# show corner heat map after nonmaximal supression
plt.subplot(4,2,5)
plt.imshow(J1_2)
plt.subplot(4,2,6)
plt.imshow(J2_2)

# show corners on origional images
ax = plt.subplot(4,2,7)
plt.imshow(I1)
# draw rectangles of size w around corners
for i in range(pts1_0.shape[1]):
    x,y = pts1_0[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
plt.plot(pts1_0[0,:], pts1_0[1,:], '.r') # display pixel accurate corners
plt.plot(pts1_1[0,:], pts1_1[1,:], '.g') # display subpixel corners
```

```
    plt.title('found %d corners'%pts1_0.shape[1])
    ax = plt.subplot(4,2,8)
    plt.imshow(I2)
    for i in range(pts2_0.shape[1]):
        x,y = pts2_0[:,i]
        ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
    plt.plot(pts2_0[0,:], pts2_0[1,:], '.r')
    plt.plot(pts2_1[0,:], pts2_1[1,:], '.g')
    plt.title('found %d corners'%pts2_0.shape[1])

    plt.show()
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel_launche
```


found 608 corners
found 628 corners

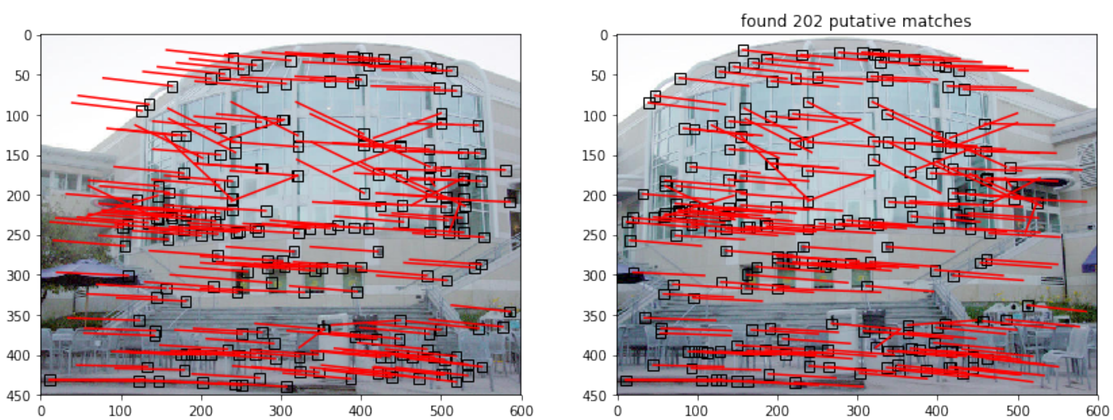
1.3 Problem 2 (Programing): Feature Matching (15 points)

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.

```
In [4]: def match(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 window to compute correlation coefficients
            # t is the correlation coefficient threshold
            # d distance ration threshold
            # p is the proximity threshold
            # 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 is a vector of length k that contains the correlation
            #     coefficients of the matches

            """your code here"""
            mask=np.ones((pts1.shape[1],pts2.shape[1]))
            lenw=(w-1)//2

            #calculate correlation coefficients
            corr0=np.zeros((pts1.shape[1],pts2.shape[1],3))
            corr=np.zeros((pts1.shape[1],pts2.shape[1]))
            for i in range(pts1.shape[1]):
                for j in range(pts2.shape[1]):
                    #calculate for 3 channels then take the minimum
                    for k in range(3):
                        x1=pts1[1][i]
                        y1=pts1[0][i]
                        window1=I1[x1-lenw:x1+lenw+1,y1-lenw:y1+lenw+1,k]
                        list1=np.reshape(window1,(1,w*w))
                        x2=pts2[1][j]
                        y2=pts2[0][j]
                        window2=I2[x2-lenw:x2+lenw+1,y2-lenw:y2+lenw+1,k]
                        list2=np.reshape(window2,(1,w*w))
```

```
                #avoid nan
                corr0[i][j][k]=np.corrcoef(list1,list2)[0][1] if np.corrcoef(list1,lis
            corr[i][j]=min(corr0[i][j][0],corr0[i][j][1],corr0[i][j][2])

    #select match
    flag=False
    inds,scores=0,0
    while True:
        curcorr=np.multiply(mask,corr)
        position=np.argmax(curcorr)
        m, n = divmod(position, corr.shape[1])
        #if the current max coef is smaller than threshhold, break the loop
        if corr[m][n]<t:
            break
        cmax=corr[m][n]
        corr[m][n]=-1

        #find next best
        nextbest=np.maximum(np.amax(corr[:,n]),np.amax(corr[m,:]))

        #distance ratio and proximity restrains
        if cmax>nextbest and (1-cmax)<(1-nextbest)*d \
        and np.sqrt(np.square(pts1[0][m]-pts2[0][n])+np.square(pts1[1][m]-pts2[1][n]))
            if not flag:
                inds = np.array([[m], [n]])
                scores=cmax
                flag = True
            else:
                inds = np.hstack((inds,np.array([[m], [n]])))
                scores=np.hstack((scores,cmax))
        mask[m,:]=np.zeros(corr.shape[1])
        mask[:,n]=np.zeros(corr.shape[0])
    return inds, scores

# parameters to tune
w1 = 13
t1 = 0.65
d1 = 5
p1 = 130

# do the matching
inds, scores = match(I1, I2, pts1_1, pts2_1, w1, t1, d1, p1)

# create new arrays of points which are corresponding
pts1=pts1_1[:,inds[0,:]]
pts2=pts2_1[:,inds[1,:]]

# display the results
```

```
plt.figure(figsize=(14,8))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
plt.title('found %d putative matches'%inds.shape[1])
ax2.imshow(I2)
for i in range(inds.shape[1]):
    ii = inds[0,i]
    jj = inds[1,i]
    x1 = pts1_1[0,ii]
    x2 = pts2_1[0,jj]
    y1 = pts1_1[1,ii]
    y2 = pts2_1[1,jj]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w1/2,y1-w1/2),w1,w1, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w1/2,y2-w1/2),w1,w1, fill=False))
plt.show()
```



1.4 Problem 3 (Programing): 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

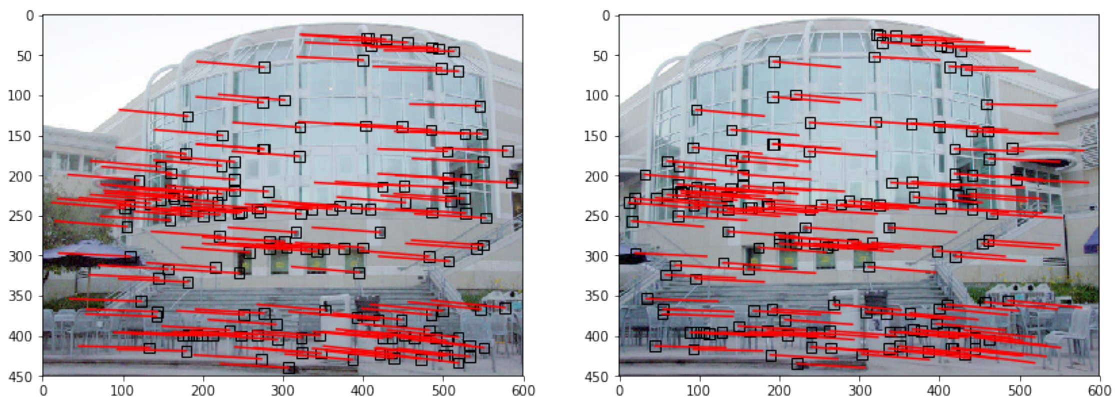
```
In [17]: def normalize2d(x):
             x0mean=np.mean(x[0,:])
             x1mean=np.mean(x[1,:])
```

```python
    x0var=np.var(x[0,:])
    x1var=np.var(x[1,:])
    sigma=np.sqrt(x0var+x1var)
    s2d=np.sqrt(2/(sigma**2))
    H2d=np.array([[s2d,0,-s2d*x0mean],[0,s2d,-s2d*x1mean],[0,0,1]])
    x_nl=H2d.dot(toHomo(x))
    return x_nl[0:2,:],H2d

def fourpointH(x):
    e1=np.array([[1],[0],[0]])
    e2=np.array([[0],[1],[0]])
    e3=np.array([[0],[0],[1]])
    e4=np.array([[1],[1],[1]])
    lam=np.linalg.inv(x[:,0:3]).dot(x[:,3])
    H=np.linalg.inv(np.hstack([lam[0]*x[:,0:1],lam[1]*x[:,1:2],lam[2]*x[:,2:3]]))
    return H

def matrixA(x1,x2):
    x1=toHomo(x1.reshape(2,1)).reshape(1,3)
    A=np.zeros((2,9))
    A[0,3:6]=-x1
    A[0,6:9]=x2[1]*x1
    A[1,0:3]=x1
    A[1,6:9]=-x2[0]*x1
    return A

def matrixJ(H,x1,x2):
    J=np.zeros((2,4))
    J[0,0]=-H[1,0]+x2[1]*H[2,0]
    J[1,0]=H[0,0]-x2[0]*H[2,0]
    J[0,1]=-H[1,1]+x2[1]*H[2,1]
    J[1,1]=H[0,1]-x2[0]*H[2,1]
    J[0,3]=x1[0]*H[2,0]+x1[1]*H[2,1]+H[2,2]
    J[1,2]=-J[0,3]
    return J

def MSAC(pts1, pts2, max_iters, p,alpha,sigma):
    """your code here"""

    maxtrails=np.inf
    mincost=np.inf
    iters = 0 # number of MSAC itterations executed
    threshhold=20 # lower bound for the size of acceptable consensus set
    #look at the chi square distribution table and find that F2^-1(0.95)=5.99
    tolerance=5.99

    num_inlier=0
    while(iters<maxtrails):
```

```python
        iters=iters+1
        #select a random sample
        indices=np.array(random.sample(range(pts1.shape[1]),4))

        #calculate model
        H1=fourpointH(toHomo(pts1[:,indices]))
        H2=fourpointH(toHomo(pts2[:,indices]))
        H=np.linalg.inv(H2).dot(H1)
        h=H.reshape(9,1)

        #print(H.dot)

        error=np.zeros((pts1.shape[1],1))
        cost=0

        #calculate sampson error
        for i in range(pts1.shape[1]):
            Ai=matrixA(pts1[:,i],pts2[:,i])
            epsilon=Ai.dot(h)
            J=matrixJ(H,pts1[:,i],pts2[:,i])
            lam=-np.linalg.inv(J.dot(J.T)).dot(epsilon)
            delta_x=J.T.dot(lam)
            error[i]=delta_x.T.dot(delta_x)
            cost+=error[i] if(error[i]<=tolerance) else tolerance

        #print(error)
        if(cost<mincost):
            minerror=error
            mincost=cost
            mincost_H=H

        ##calculate set of inliers, update max_trails
        indices_inlier=[i for i in range(pts1.shape[1]) if minerror[i]<=tolerance]
        num_inlier=len(indices_inlier)
        #if(num_inlier<=threshhold):
         #   break
        #print(num_inlier)
        maxtrails=np.log(1-p)/np.log(0.9-((num_inlier)/pts1.shape[1])**pts1.shape[1])

    H = mincost_H # estimated H matrix

    H=H/np.linalg.norm(H)
    inliers = indices_inlier # indices of inliers (must be sorted)

    return H, inliers, mincost, iters
```

```python
# MSAC hyperparameters (add any additional hyperparameters necessary here. For exampl
# You should pass these hyperparameters as additional paramters to MSAC(...)

p=0.99
alpha=0.95
sigma=1
max_iters=1

H_MSAC, inliers, cost_MSAC, iters_MSAC = MSAC(pts1, pts2, max_iters,p,alpha,sigma)
print('inliers: ', inliers)
print('inlier count: ', len(inliers))

print('cost_MSAC=%f'%cost_MSAC)
print('||H_MSAC||=%f'%np.sqrt(np.sum(H_MSAC**2)))
print('H_MSAC')
print(H_MSAC/np.sqrt(np.sum(H_MSAC**2)))

# display the results
plt.figure(figsize=(14,8))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
for i in range(inds[:,inliers].shape[1]):
    ii = inds[0,inliers[i]]
    jj = inds[1,inliers[i]]
    x1 = pts1_1[0,ii]
    x2 = pts2_1[0,jj]
    y1 = pts1_1[1,ii]
    y2 = pts2_1[1,jj]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w1/2,y1-w1/2),w1,w1, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w1/2,y2-w1/2),w1,w1, fill=False))
plt.show()
```

```
inliers:  [0, 2, 3, 5, 6, 10, 11, 12, 13, 14, 15, 16, 20, 21, 24, 27, 28, 29, 30, 31, 32, 34, 3
inlier count:  151
cost_MSAC=490.802891
||H_MSAC||=1.000000
H_MSAC
[[ 1.16153595e-02 -1.12215389e-04 -9.95329576e-01]
 [ 2.23583693e-04  1.12193510e-02 -9.45271737e-02]
 [ 1.07016581e-06 -2.23481887e-07  1.10812729e-02]]
```



1.5 Problem 4 (Programing): 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. Include the numerical values of the resulting $\boldsymbol{H}_{\text{DLT}}$, scaled such that $||\boldsymbol{H}_{\text{DLT}}||_{\text{Fro}} = 1$

In [18]:
```
#sampson correction as initial guess
def sampsoncorrection(pts1,pts2,H):
    h=H.reshape(9,1)
    delta_x1=np.zeros((2,pts1.shape[1]))
    for i in range(pts1.shape[1]):
        Ai=matrixA(pts1[:,i],pts2[:,i])
        epsilon=Ai.dot(h)
        J=matrixJ(H,pts1[:,i],pts2[:,i])
        lam=-np.linalg.inv(J.dot(J.T)).dot(epsilon)
        delta_x=J.T.dot(lam)
        delta_x1[:,i:i+1]=delta_x[0:2]
    return pts1+delta_x1


def DLT(pts1, pts2):
    """your code here"""
    pts1_nl,H2d1=normalize2d(pts1)
    pts2_nl,H2d2=normalize2d(pts2)
    pts1_nl=toHomo(pts1_nl)
    pts2_nl=toHomo(pts2_nl)
    v=np.zeros((3,pts1.shape[1]))
    A=np.zeros((2*pts1.shape[1],9))
    for i in range(pts1.shape[1]):
        #calculate v vector
        v[:,i]=(pts2_nl[:,i].reshape(3,1)+np.sign(pts2_nl[0,i])*np.sqrt(np.dot(pts2_nl
        ,pts2_nl[:,i]))*np.array([[1],[0],[0]])).ravel()
```

```
            #calculate H_v matrix
            H_v=np.eye(3,3)-2*v[:,i].reshape(3,1)*v[:,i].reshape(3,1).T/np.dot(v[:,i],v[:
            A[2*i:2*i+2,:]=np.kron(H_v[1:3,:],pts1_nl[:,i].reshape(1,3))

        u,sigma,vv=np.linalg.svd(A)

        index=np.where(sigma==np.min(sigma))
        Hcol=vv[index[0][0],:]
        H=Hcol.reshape(3,3)

        #back to unnormalized frame
        H = np.linalg.inv(H2d2).dot(H).dot(H2d1)
        H=H/np.linalg.norm(H)

        ptssi=sampsoncorrection(pts1,pts2,H)
        print(ptssi.shape)
        print(pts1.shape)
        #normalize and propagate covariance
        pts1_nl,H2d1=normalize2d(pts1)
        pts2_nl,H2d2=normalize2d(pts2)

        s1=H2d1[0,0]
        s2=H2d2[0,0]

        Sigmax1=s1**2*np.eye(2,2)
        Sigmax2=s2**2*np.eye(2,2)

        #calculate current cost

        x_proj1,_=normalize2d((ptssi))
        x_proj2,_=normalize2d(fromHomo(H.dot(toHomo(ptssi))))
        epsilon_1=pts1_nl-x_proj1
        epsilon_2=pts2_nl-x_proj2
        cost=0
        for i in range(pts1.shape[1]):
            cost=cost+epsilon_1[:,i].T.dot(np.linalg.inv(Sigmax1)).dot(epsilon_1[:,i])
            cost=cost+epsilon_2[:,i].T.dot(np.linalg.inv(Sigmax2)).dot(epsilon_2[:,i])
        return H,cost


    H_DLT, cost_DLT = DLT(pts1[:,inliers], pts2[:,inliers])
    print('cost_DLT=%f'%cost_DLT)
    print('||H_DLT||=%f'%np.sqrt(np.sum(H_DLT**2)))
    print('H_DLT')
    print(H_DLT/np.sqrt(np.sum(H_DLT**2)))
```

```
(2, 151)
```

```
(2, 151)
cost_DLT=50.407445
||H_DLT||=1.000000
H_DLT
[[ 1.09851058e-02 -1.67292267e-05 -9.84599281e-01]
 [ 3.23773338e-04  1.06915168e-02 -1.73851988e-01]
 [ 1.26205180e-06  1.03263564e-07  1.02301086e-02]]
```

1.6 Problem 5 (Programing): 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). 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 [20]:
```
def sinc(x):
    if x==0:
        return 1
    else:
        return np.sin(x)/x
def frompavec(v):
    norm=np.sqrt(np.dot(v.T,v))
    a=np.cos(norm/2)
    b=sinc(norm/2)/2*v
    return np.vstack((a,b))
def topavec(P):
    a=P[0]
    b=P[1:]
    v=2/sinc(np.arccos(a))*b
    norm=np.sqrt(np.dot(v.T,v))
    if norm>np.pi:
        v=(1-2*np.pi/norm*np.ceil((norm-np.pi)/(2*np.pi)))*v
    return v

#calculate  deparameterized vector partial derivatives
def partialpv(v):
    length=len(v)
    P0=frompavec(v)
    norm=np.sqrt(np.dot(v.T,v))
    a=P0[0]
    b=P0[1:]

    if norm==0:
```

```
            partialav=np.zeros((length,1))
            partialbv=1/2*np.eye(length,length)
        else:
            partialav=-1/2*b.T
            partialbv=sinc(norm/2)/2*np.eye(length,length)+1/4/norm*(np.cos(norm/2)/norm*
            -np.sin(norm/2)/norm/norm*4)*np.dot(v,v.T)
        partialpv=np.vstack((partialav,partialbv))
        return partialpv

    # input vector of H h, 2D scene points
    def calAi(x,h):
        partialxh=np.zeros((2,9))
        XT=x.reshape(1,3)
        w=XT.dot(h[6:9])
        x_proj=fromHomo(h.reshape(3,3).dot(x))
        Ai=np.zeros((2,9))
        Ai[0,:3]=XT
        Ai[1,3:6]=XT
        Ai[0,6:9]=-x_proj[0]*XT
        Ai[1,6:9]=-x_proj[1]*XT
        partialxh=1/w*Ai
        return partialxh.dot(partialpv(topavec(h)))

    # input 2d homogeneous points xsi
    def calBi(x,h):
        partialxxsi=np.zeros((2,3))
        x_nl=x/np.linalg.norm(x)
        h1=h[0:3]
        h2=h[3:6]
        h3=h[6:9]
        XT=x.reshape(1,3)
        w=XT.dot(h[6:9])
        x_proj=fromHomo(h.reshape(3,3).dot(x))
        #print(x_proj)
        partialxxsi[0,:]=h1.T-x_proj[0]*h3.T
        partialxxsi[1,:]=h2.T-x_proj[1]*h3.T
        partialxxsi=partialxxsi/w
        #print(partialxxsi)
        return partialxxsi.dot(partialpv(topavec(x_nl)))
```

```
In [23]: # feel free to modify the function signature as needed (pass parameterized H instead 
    def LMstep(H, pts1, pts2, ptssi, l, v):
        """your code here"""

        #normalize and propagate covariance
        pts1_nl,H2d1=normalize2d(pts1)
        pts2_nl,H2d2=normalize2d(pts2)
```

```
s1=H2d1[0,0]
s2=H2d2[0,0]

Sigmax1=s1**2*np.eye(2,2)
Sigmax2=s2**2*np.eye(2,2)

#calculate current cost
x_proj1,_=normalize2d((ptssi))
x_proj2,_=normalize2d(fromHomo(H.dot(toHomo(ptssi))))
epsilon_1=pts1_nl-x_proj1
epsilon_2=pts2_nl-x_proj2
cost=0
for i in range(pts1.shape[1]):
    cost=cost+epsilon_1[:,i].T.dot(np.linalg.inv(Sigmax1)).dot(epsilon_1[:,i])
    cost=cost+epsilon_2[:,i].T.dot(np.linalg.inv(Sigmax2)).dot(epsilon_2[:,i])

U=np.zeros((8,8))
V=np.zeros((pts1.shape[1],2,2))
W=np.zeros((pts1.shape[1],8,2))
epsilon_a=np.zeros((8,1))
epsilon_b=np.zeros((2,pts1.shape[1]))
for i in range(pts1.shape[1]):
    Ai=calAi(toHomo(ptssi[:,i:i+1]),H.reshape(9,1))
    Bi1=calBi(toHomo(ptssi[:,i:i+1]),np.eye(3).reshape(9,1))
    Bi2=calBi(toHomo(ptssi[:,i:i+1]),H.reshape(9,1))

    U=U+Ai.T.dot(np.linalg.inv(Sigmax1)).dot(Ai)
    V[i,:,:]=Bi1.T.dot(np.linalg.inv(Sigmax1)).dot(Bi1)+Bi2.T.dot(np.linalg.inv(Si
    W[i,:,:]=Ai.T.dot(np.linalg.inv(Sigmax2)).dot(Bi2)
    epsilon_a=epsilon_a+Ai.T.dot(np.linalg.inv(Sigmax2)).dot(epsilon_2[:,i]).resha
    epsilon_b[:,i]=Bi1.T.dot(np.linalg.inv(Sigmax1)).dot(epsilon_1[:,i])+Bi2.T.dot
flag=True
while flag:
    #print(l)
    sumwvw=np.zeros((8,8))
    sumwvb=np.zeros((8,1))
    for i in range(pts1.shape[1]):
        sumwvw=sumwvw+W[i,:,:].dot(np.linalg.inv(V[i,:,:]+l*np.eye(2))).dot(W[i,:
        sumwvb=sumwvb+W[i,:,:].dot(np.linalg.inv(V[i,:,:]+l*np.eye(2))).dot(epsil
    S=U+l*np.eye(8)+sumwvw
    e=epsilon_a-sumwvb
    delta_a=np.linalg.inv(S).dot(e)
    delta_b=np.zeros((2,pts1.shape[1]))
    for i in range(pts1.shape[1]):
        delta_b[:,i:i+1]=np.linalg.inv(V[i,:,:]+l*np.eye(2)).dot(epsilon_b[:,i].re

    # calculate candidate H matrix and scene points
    hvec=topavec(H.reshape(9,1))
```

```
        Hcandidate=frompavec(hvec+delta_a).reshape(3,3)
        ptscandidate=np.zeros((2,pts1.shape[1]))

        for i in range(pts1.shape[1]):
            ptsvec=toHomo(ptssi[:,i:i+1])
            ptsvec=ptsvec/np.linalg.norm(ptsvec)
            ptscandidate[:,i:i+1]=fromHomo(frompavec(topavec(ptsvec)+delta_b[:,i:i+1]

        # calculate projection
        x_proj1,_=normalize2d((ptscandidate))
        x_proj2,_=normalize2d(fromHomo(H.dot(toHomo(ptscandidate))))
        epsilon_11=pts1_nl-x_proj1
        epsilon_22=pts2_nl-x_proj2
        cost_1=0
        for i in range(pts1.shape[1]):
            cost_1=cost_1+epsilon_11[:,i].T.dot(np.linalg.inv(Sigmax1)).dot(epsilon_1
            cost_1=cost_1+epsilon_22[:,i].T.dot(np.linalg.inv(Sigmax2)).dot(epsilon_2


        if cost_1<cost:
            #print('error = %.20f'%(error_1))
            flag=False
            l=0.1*l
            cost=cost_1
        else:
            l=10*l

    #print(sampsoncorrection(pts1,pts2,Hcandidate)-ptscandidate)
    return Hcandidate, cost, ptscandidate, l

# LM hyperparameters
l=.001
v=10
max_iters=10

H_LM = H_DLT
prevcost=np.inf
# sampson correction as initial guess
ptssi=sampsoncorrection(pts1[:,inliers],pts2[:,inliers],H_DLT)
# LM optimization loop
i=0
print ('iter %d cost %.10f'%(i, cost_DLT))
while True:
    H_LM, cost_LM, ptssi, l = LMstep(H_LM, pts1[:,inliers], pts2[:,inliers],ptssi, l,
    print ('iter %d cost %.10f'%(i+1, cost_LM))
    if (1-cost_LM/prevcost)<1e-5:
        break
    prevcost=cost_LM
```

```python
            i+=1

        print('||H_LM||=%f'%np.sqrt(np.sum(H_LM**2)))
        print('H_LM')
        print(H_LM/np.sqrt(np.sum(H_LM**2)))
```

```
iter 0 cost 50.4074449629
iter 1 cost 50.4074406475
iter 2 cost 50.4062323097
iter 3 cost 50.4050374343
iter 4 cost 50.4038558551
iter 5 cost 50.4026874085
iter 6 cost 50.4015319328
iter 7 cost 50.4003892687
iter 8 cost 50.3992592592
iter 9 cost 50.3981417490
iter 10 cost 50.3970365854
iter 11 cost 50.3959436174
iter 12 cost 50.3948626962
iter 13 cost 50.3937936750
iter 14 cost 50.3927364089
iter 15 cost 50.3916907549
iter 16 cost 50.3906565722
iter 17 cost 50.3896337215
iter 18 cost 50.3886220656
iter 19 cost 50.3876214690
iter 20 cost 50.3866317980
iter 21 cost 50.3856529208
iter 22 cost 50.3846847072
iter 23 cost 50.3837270287
iter 24 cost 50.3827797586
iter 25 cost 50.3818427717
iter 26 cost 50.3809159447
iter 27 cost 50.3799991557
iter 28 cost 50.3790922844
iter 29 cost 50.3781952121
iter 30 cost 50.3773078216
iter 31 cost 50.3764299974
iter 32 cost 50.3755616253
iter 33 cost 50.3747025927
iter 34 cost 50.3738527882
iter 35 cost 50.3730121022
iter 36 cost 50.3721804262
iter 37 cost 50.3713576533
iter 38 cost 50.3705436780
iter 39 cost 50.3697383958
iter 40 cost 50.3689417040
iter 41 cost 50.3681535008
```

```
iter 42 cost 50.3673765616
iter 43 cost 50.3665257410
iter 44 cost 50.3663374100
||H_LM||=1.000000
H_LM
[[ 1.09859436e-02 -1.73211746e-05 -9.84599270e-01]
 [ 3.23882285e-04  1.06917116e-02 -1.73851984e-01]
 [ 1.26362250e-06  1.02596698e-07  1.02301323e-02]]
```