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

Instructor: Ben Ochoa

Due: Wednesday, January 17, 2018, 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.

Problem 1 (Math): Line-plane intersection (5 points)

The line in 3D defined by the join of the points $\boldsymbol{X}_1 = (X_1, Y_1, Z_1, T_1)^\top$ and $\boldsymbol{X}_2 = (X_2, Y_2, Z_2, T_2)^\top$ can be represented as a Plucker matrix $\boldsymbol{L} = \boldsymbol{X}_1\boldsymbol{X}_2^\top - \boldsymbol{X}_2\boldsymbol{X}_1^\top$ or pencil of points $\boldsymbol{X}(\lambda) = \lambda\boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ (i.e., $\boldsymbol{X}$ is a function of λ). The line intersects the plane $\boldsymbol{\pi} = (a, b, c, d)^\top$ at the point $\boldsymbol{X}_{\boldsymbol{L}} = \boldsymbol{L}\boldsymbol{\pi}$ or $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$, where $\lambda_{\boldsymbol{\pi}}$ is determined such that $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top\boldsymbol{\pi} = 0$ (i.e., $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ is the point on $\boldsymbol{\pi}$). Show that $\boldsymbol{X}_{\boldsymbol{L}}$ is equal to $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ up to scale.

My solution:

It is obvious that the Plucker matrix $plücker$ matrix is a skew symmetric matrix. As the problem statement, there are two ways to see an intersection point:

- By using pencil point $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top \boldsymbol{\pi} = 0$,
$$(\lambda_\pi \boldsymbol{X_1^T} + (1-\lambda_\pi)\boldsymbol{X_2^T})\boldsymbol{\pi} = .0$$
The equation is then solved by $\lambda_\pi = \frac{\boldsymbol{X_2^T\pi}}{(\boldsymbol{X_2^T}-\boldsymbol{X_1^T})\boldsymbol{\pi}}$. Setting this parameter to the form pencil of point, the intersection point can be expressed as:
$\boldsymbol{X}(\lambda_\pi) = \frac{\boldsymbol{X_2^T\pi X_1 - X_1^T\pi X_2}}{(\boldsymbol{X_2}-\boldsymbol{X_1})^T\boldsymbol{\pi}} = \frac{1}{(\boldsymbol{X_2}-\boldsymbol{X_1})^T\boldsymbol{\pi}}(\boldsymbol{X_2^T\pi X_1} - \boldsymbol{X_1^T\pi X_2}) = c_0\boldsymbol{X_1} + c_1\boldsymbol{X_2}$, where c_0 and c_1 are sclars.
- By plugging the $plücker$ matrix in $\boldsymbol{X_L} = (\boldsymbol{X_1X_2}^{\boldsymbol{T}} - \boldsymbol{X_2X_1}^{\boldsymbol{T}})\boldsymbol{\pi}$.

$$\boldsymbol{L} = \begin{pmatrix} 0 & c_1 & c_2 & c_3 \\ -c_1 & 0 & c_3 & c_4 \\ -c_2 & -c_3 & 0 & c_5 \\ -c_3 & -c_4 & -c_5 & 0 \end{pmatrix} = (\boldsymbol{L}_1 + \boldsymbol{L}_2).$$

where
$c_1 = X_1Y_2 - X_2Y_1, c_2 = X_1Z_2 - X_2Z_1, c_3 = X_1T_2 - X_2T_1, c_4 = Y_1T_2 - Y_2T_1, c_5 = Z_1T_2 - Z_2T_1$ and $\boldsymbol{L}_1$ is lower triangle matrix, $\boldsymbol{L}_2$ an upper triangle matrix. Thus: $\boldsymbol{X_L} = \boldsymbol{L\pi} = (\boldsymbol{L}_1 + \boldsymbol{L}_2)\boldsymbol{\pi}$

When comparing the term $\boldsymbol{X_1X_2}^{\boldsymbol{T}}\boldsymbol{\pi} - \boldsymbol{X_2X_1}^{\boldsymbol{T}}\boldsymbol{\pi}$ and $\boldsymbol{X_2^T\pi X_1} - \boldsymbol{X_1^T\pi X_2}$. Their components are same.

Thus, it is summerised that only the scalar $\frac{1}{(\boldsymbol{X_2}-\boldsymbol{X_1})^T\boldsymbol{\pi}}$ differs $\boldsymbol{X_L}$ from $\boldsymbol{X}(\boldsymbol{\lambda_\pi})$.

Problem 2 (Math): Line-quadric intersection (5 points)

In general, a line in 3D intersects a quadric $\boldsymbol{Q}$ at zero, one (if the line is tangent to the quadric), or two points. If the pencil of points $\boldsymbol{X}(\lambda) = \lambda\boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ represents a line in 3D, the (up to two) real roots of the quadratic polynomial $c_2\lambda_{\boldsymbol{Q}}^2 + c_1\lambda_{\boldsymbol{Q}} + c_0 = 0$ are used to solve for the intersection point(s) $\boldsymbol{X}(\lambda_{\boldsymbol{Q}})$. Show that $c_2 = \boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_1 - 2\boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_2 + \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2$, $c_1 = 2(\boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_2 - \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2)$, and $c_0 = \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2$.

My solution:

A point X on a quadric Q can be interpreted as: $X^TQX = 0$. Applying the pencil of points, it is seen that:

$$X(\lambda_Q)^T QX(\lambda_Q) = (\lambda_Q X_1^T + (1-\lambda_Q)X_2^T)Q(\lambda_Q X_1 + (1-\lambda_Q)X_2).$$

Further, expand the equation and it is resulted in:

$$(X_1^TQX_1 - 2X_1^TQX_2 - X_2^TQX_2)\lambda_Q{}^2 + 2(X_1^TQX_1 - X_2^TQX_2)\lambda_Q + X_2^TQX_2 = 0.$$

Comparing this form with the given quadratic polynomial, it is true that:
$c_2 = X_1^\top QX_1 - 2X_1^\top QX_2 + X_2^\top QX_2$, $c_1 = 2(X_1^\top QX_2 - X_2^\top QX_2)$, and $c_0 = X_2^\top QX_2$

Problem 3 (Programing): Feature detection (20 points)

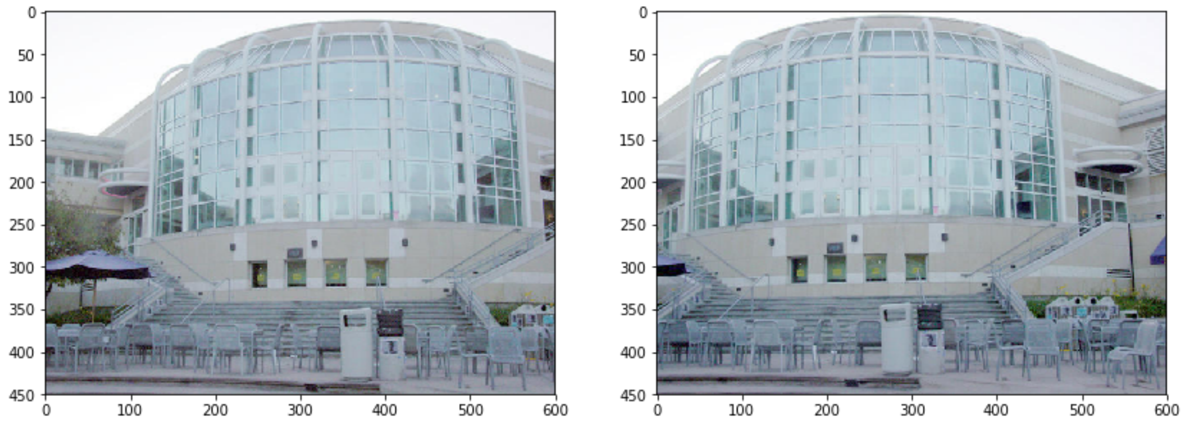
Download input data from the course website. The file price_center20.JPG contains image 1 and the file price_center21.JPG contains image 2.

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

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

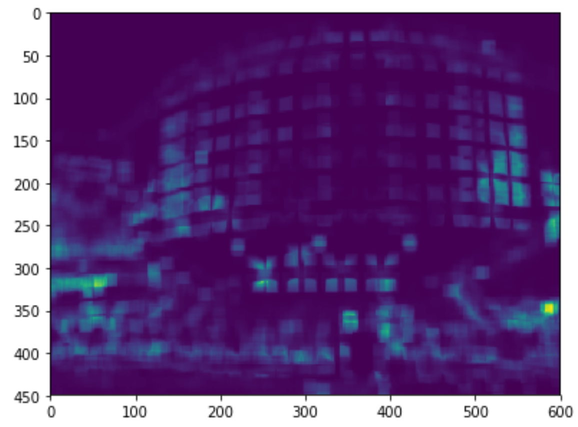
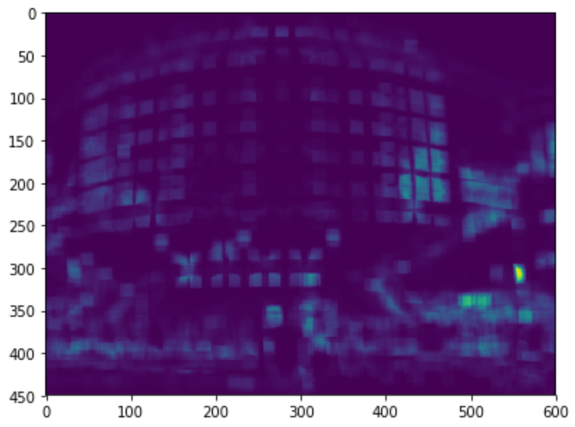
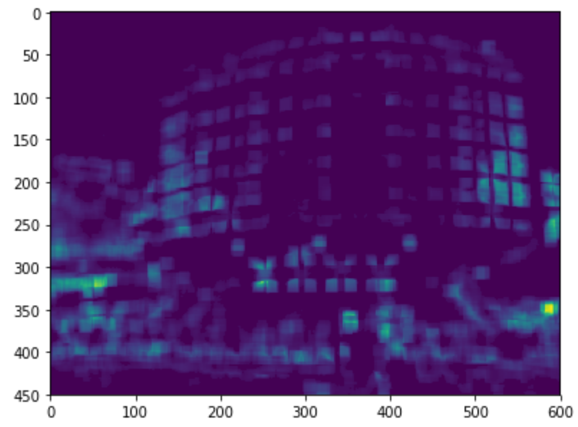
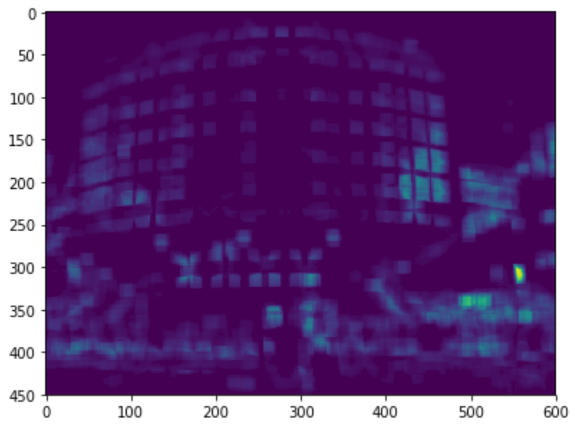
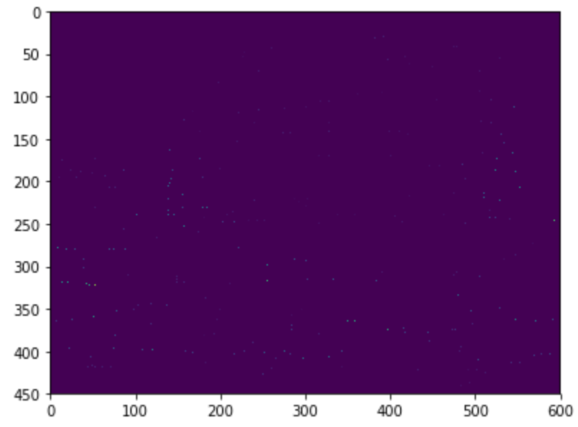
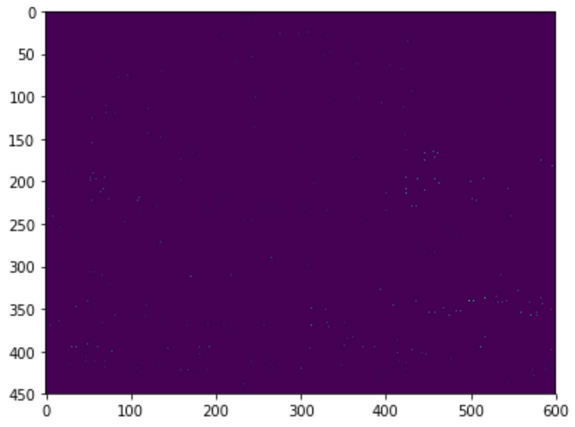
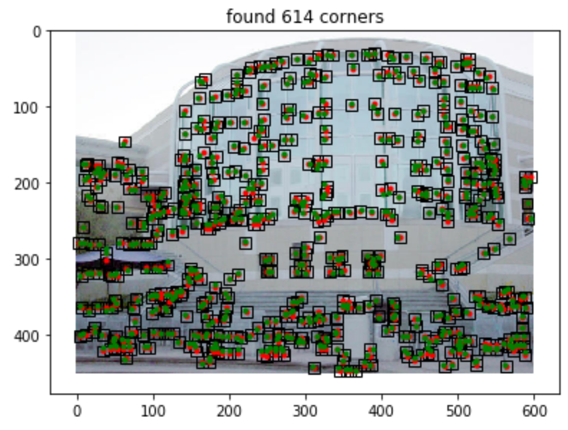
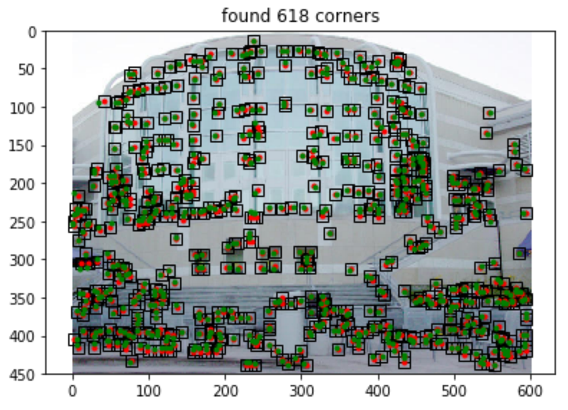
where w is the window about the pixel, and I_x and I_y are the gradient images in the x and y direction, respectively. Calculate the gradient images using the fivepoint central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead 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.

In [48]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy import ndimage, signal
import math
# open the input images, size of 600*450
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()
```



In [32]:
```
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 matr
ix N
    # t is the minor eigenvalue threshold
    # w_nms is the size of the window used for nonmaximal supressio
n
    # outputs:
    # J0 is the mxn image of minor eigenvalues of N before threshol
ding
    # J1 is the mxn image of minor eigenvalues of N after threshold
ing
    # J2 is the mxn image of minor eigenvalues of N after nonmaxima
l supression
    # pts0 is the 2xk list of coordinates of (pixel accurate) corne
rs
    #     (ie. coordinates of nonzero values of J2)
    # pts1 is the 2xk list of coordinates of subpixel accurate corn
ers
    #     found using the Forstner detector
    """my code starts"""
    if len(I.shape) == 3:
        I_gray = 0.2989*I[:,:,0] + 0.5870*I[:,:,1] + 0.1140*I[:,:,2
]
    else:
        I_gray = I

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

    gradient_kernel = 1/12*np.array([-1,8,0,-8,1])
    Ix = np.zeros((m,n))
    Iy = np.zeros((m,n))

    # image gradient using five-point-operator
    for i in range(m):
        Ix[i,:] = ndimage.convolve(I_gray[i,:],gradient_kernel,mode
= 'reflect')
    for j in range(n):
        Iy[:,j] = (ndimage.convolve(I_gray[:,j].T,gradient_kernel,m
ode = 'reflect')).T
    #magnitude = np.sqrt(Ix**2,Iy**2)
    Ixx = np.zeros((m,n))
    Ixy = np.zeros((m,n))
    Iyy = np.zeros((m,n))

    Ixx = np.multiply(Ix,Ix) # elementwise multiplication
    Ixy = np.multiply(Ix,Iy)
    Iyy = np.multiply(Iy,Iy)
    # window for computing N
    window = np.ones((w,w)) # no weight on pixels
    Nxx = signal.convolve2d(Ixx,window,'same','symm')
    Nxy = signal.convolve2d(Ixy,window,'same','symm')
    Nyy = signal.convolve2d(Iyy,window,'same','symm')
    #determinant and trace
    det_N = Nxx*Nyy - Nxy**2
    tr_N = Nxx + Nyy
    # assign minor ew of N before threshold to J0
    #J0 = [1/2*(tr_N-np.sqrt(tr_N**2-4*det_N)) for i in np.where((t
r_N**2-4*det_N)>0)]
    for i in range(m):
        for j in range(n):
            if (tr_N[i,j]**2-4*det_N[i,j])>=0:
                J0[i,j] = 1/2*(tr_N[i,j]-np.sqrt(tr_N[i,j]**2-4*det
_N[i,j]))
    # assign minor .... after threshold to J1
    for i in range(m):
        for j in range(n):
            if J0[i,j] >= t:
                J1[i,j] = J0[i,j]
    # nonmaximal supression
    w_nms_half = int((w_nms-1)/2)
    #for J2
    for i in range(w_nms_half, m-w_nms_half):
        for j in range(w_nms_half, n-w_nms_half):
            max_w_nms = np.amax(J1[i-w_nms_half:i+w_nms_half,j-w_nm
s_half:j+w_nms_half])
            # find where is the pick in J1 and fits in J2
            if J1[i,j] == max_w_nms:
                J2[i,j] = J1[i,j]
    # finding the corner points
    xIxx = np.zeros((m,n))
    yIyy = np.zeros((m,n))
    xIxy = np.zeros((m,n))
    yIxy = np.zeros((m,n))

    #center weighted center of gravity
    for i in range(m):
        xIxx[i,:] = i*Ixx[i,:]
        xIxy[i,:] = i*Ixy[i,:]
    for j in range(n):
        yIyy[:,j] = j*Iyy[:,j]
        yIxy[:,j] = j*Ixy[:,j]
    xIxx_w = signal.convolve2d(xIxx,window,'same','symm')
    xIxy_w = signal.convolve2d(xIxy,window,'same','symm')
    yIxy_w = signal.convolve2d(yIxy,window,'same','symm')
    yIyy_w = signal.convolve2d(yIyy,window,'same','symm')
    #corner points
    pts0_x = []
    pts0_y = []
    pts1_x = []
    pts1_y = []
    # solve equation Nx = b
    for i in range(m):
        for j in range(n):
            if J2[i,j]!=0:
                pts0_x.append(j)
                pts0_y.append(i) # cornidates of nonzero values of
J2
                N = np.array(([Nxx[i,j],Nxy[i,j]],
                              [Nxy[i,j],Nyy[i,j]]))
                b = np.array(([xIxx_w[i,j]+yIxy_w[i,j]],[xIxy_w[i,j
]+yIyy_w[i,j]]))
                x = np.dot(np.linalg.inv(N),b)
                pts1_x.append(int(x[1])) # found using the forstner
detector
                pts1_y.append(int(x[0]))

    pts0 = np.array([pts0_x,pts0_y])
    pts1 = np.array([pts1_x,pts1_y])


    """my code ends"""
    #pts0 = np.vstack((np.random.randint(0,n,(1,625)),np.random.ran
dint(0,m,(1,625)) ))
```

```
    #pts1 = pts0 + np.random.randn(2,625)

    return J0, J1, J2, pts0, pts1


# parameters to tune
w = 15  # original was 15
t = .08 # ori was .6
w_nms= 7 #ori was 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 c
orners
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()
```








found 614 corners

found 618 corners

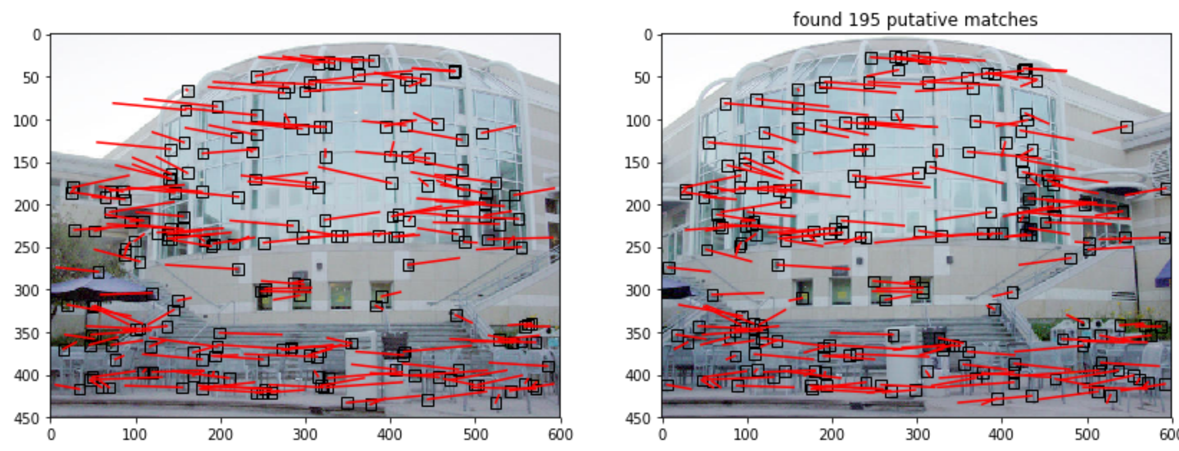
Problem 4 (Programing): Feature matching (15 points)

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 [65]:
```
size(I1.shape)
```

Out[65]: 3

In [74]:
```
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 coefficien
ts
    # 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 poin
t pts1
    #     and inds[1,i] indexs a point in pts2, where k is the numb
er of matches
    # scores is a vector of length k that contains the correlation
    #     coefficients of the matches

    """my code starts"""
    def gray_I(I):
        if len(I.shape) == 3:
            I_gray = 0.2989*I[:,:,0] + 0.5870*I[:,:,1] + 0.1140*I[:
,:,2]
        else:
            I_gray = I
        return I_gray

    I1_gray = gray_I(I1)
    I2_gray = gray_I(I2)

    m,n = I1.shape[:2]
    c1 = pts1.shape[1]
    c2 = pts2.shape[1]
    corr_m = np.zeros((c1,c2)) # correlation matrix 618*614
    pts1_x = pts1[1,:] # important!!! x is for column
    pts1_y = pts1[0,:]

    pts2_x = pts2[1,:]
    pts2_y = pts2[0,:]
    for i in range(c1): # 618
        for j in range(c2): # 614
            xmin1 = int(pts1_x[i]-w)
            xmax1 = int(pts1_x[i]+w)
            ymin1 = int(pts1_y[i]-w)
            ymax1 = int(pts1_y[i]+w)

            xmin2 = int(pts2_x[j]-w)
            xmax2 = int(pts2_x[j]+w)
            ymin2 = int(pts2_y[j]-w)
            ymax2 = int(pts2_y[j]+w)

            if xmin1<0 or ymin1<0 or xmin2<0 or ymin2<0:
                corr_m[i,j] = -1
            elif xmax1>m or xmax2>m or ymax1>n or ymax2>n:
                corr_m[i,j] = -1
            else:
                patch1 = I1_gray[xmin1:xmax1,ymin1:ymax1]
                patch2 = I2_gray[xmin2:xmax2,ymin2:ymax2]
                d1 = (patch1-np.mean(patch1))/np.std(patch1)
                d2 = (patch2-np.mean(patch2))/np.std(patch2)
                corr_m[i,j] = sum(d1*d2)
    #one to one matching algorithm
    mask = np.ones((c1,c2))

    pair1 = []
    pair2 = []
    scores = []
    temp_corr_m = np.multiply(mask,corr_m)
    while t<np.max(temp_corr_m):
        max_index = np.argmax(temp_corr_m)
        max_corr = np.max(temp_corr_m)
        row = int(max_index/c2)
        col = int(max_index/c2)
        corr_m[row,col] = -1
        max_next = max([corr_m[row,:].max(),corr_m[:,col].max()])
        corr_m[row,col] = max_corr
        if (1-corr_m[row,col])<((1-max_next)*d):
            if np.sqrt((pts1[0,row]-pts2[0,col])**2 + (pts1[1,row]-
pts2[1,col])**2) < p:
                pair1.append(row)
                pair2.append(col)
                scores.append(corr_m[row,col])
        mask[row,:] = np.zeros((1,pts2.shape[1]))
        mask[:,col] = np.zeros((1,pts1.shape[1]))
        temp_corr_m = np.multiply(mask,temp_corr_m)

    inds = np.array([pair1,pair2])
    scores = np.array(scores)
    #inds = np.vstack((np.random.choice(pts1.shape[1],200,replace=F
alse), np.random.choice(pts1.shape[1],200,replace=False)))
    #scores = np.random.rand(200)

    return inds, scores

# parameters to tune
w1 = 13 # ori: 13
t1 = 0.3 #0
d1 = 0.12 #12
p1 = 90 #np.inf

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

# display the results
plt.figure(figsize=(14,8))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
plt.title('found %d putative matches'%inds.shape[1])
ax1.imshow(I1)
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=F
alse))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w1/2,y2-w1/2),w1,w1, fill=F
alse))
plt.show()
```



```
In [ ]:
```