CSE 252B: Computer Vision II, Winter 2019 – Assignment 1

Instructor: Ben Ochoa

Due: Wednesday, January 16, 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.

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:

- gradient image heat map before thresholding
- gradient image heat map after thresholding
- original image with detected features

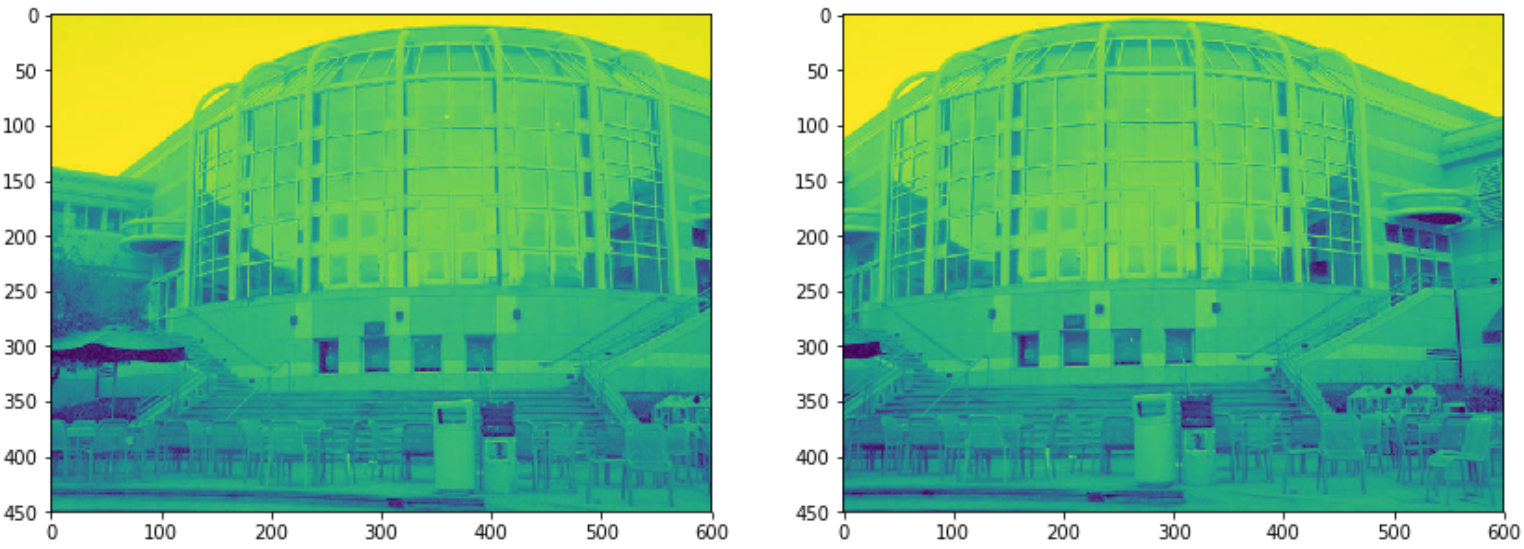
My implementation takes around 25 seconds to run. If yours is more than 250 seconds you may lose points.

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()
```



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
```

```
    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
    #
    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,pos_np


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

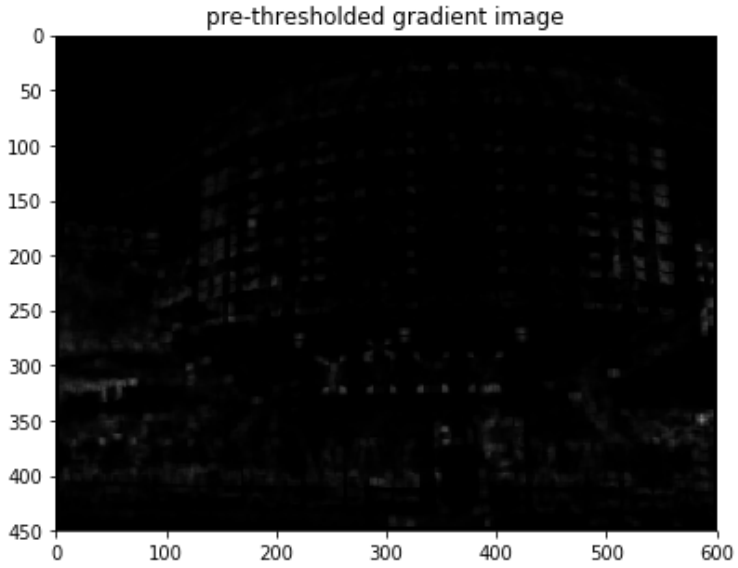
In [4]:

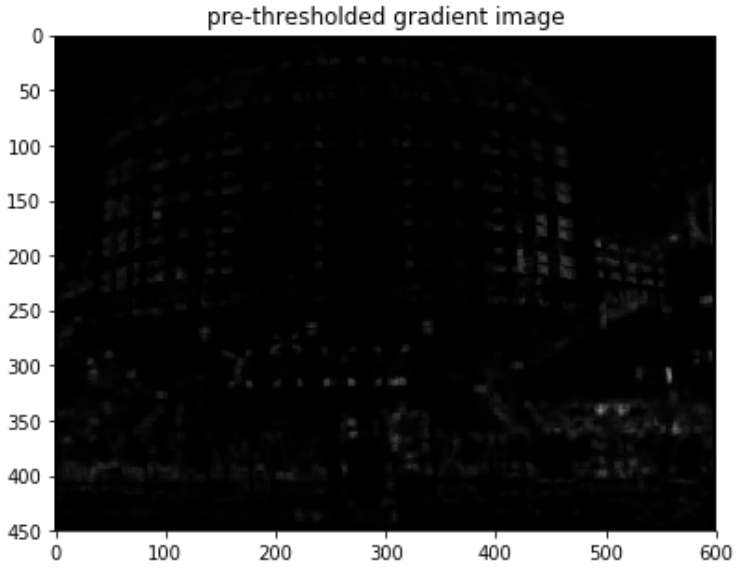
```
# input images
'''
w = 7
t1 = 8*10**-4
t2 = 8*10**-4
w_nms = 7
8.3 --- 606/632
8.7 -- /621
'''
I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.
I1_gray = grayscale(I1)
I2_gray = grayscale(I2)
# parameters to tune
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,N1,b1,pos1 = RunFeatureDetection(I1_gray, w, t1, w_nm
s)
C2, pts2, J2_0, J2_1, J2_2,N2,b2,pos2 = RunFeatureDetection(I2_gray, w, t2, w_nm
s)
toc = time.time() - tic

print('took %f secs'%toc)
```

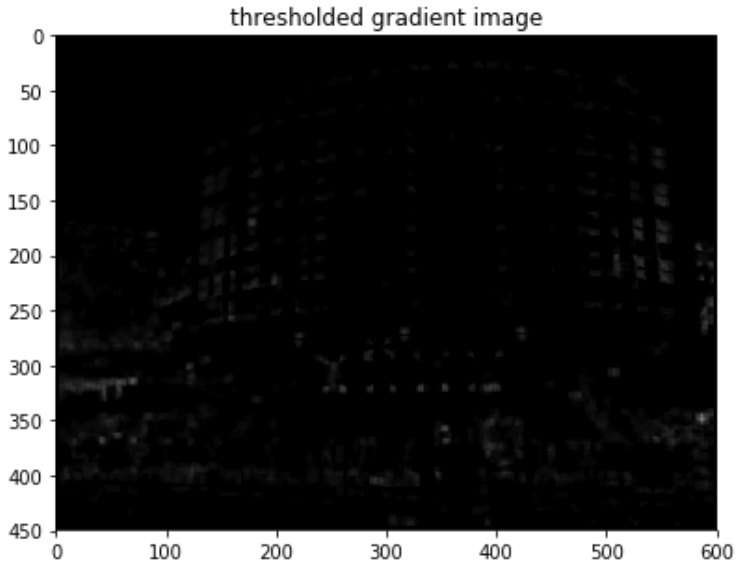
```
took 87.442113 secs
```

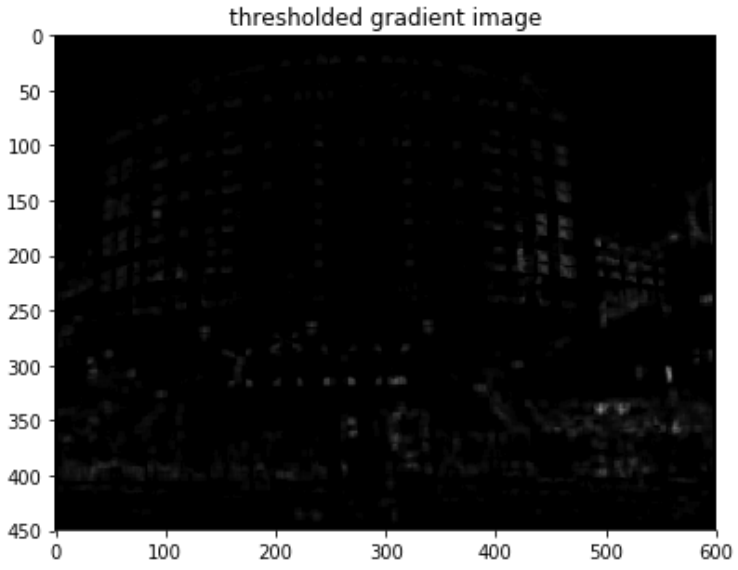
In [5]:

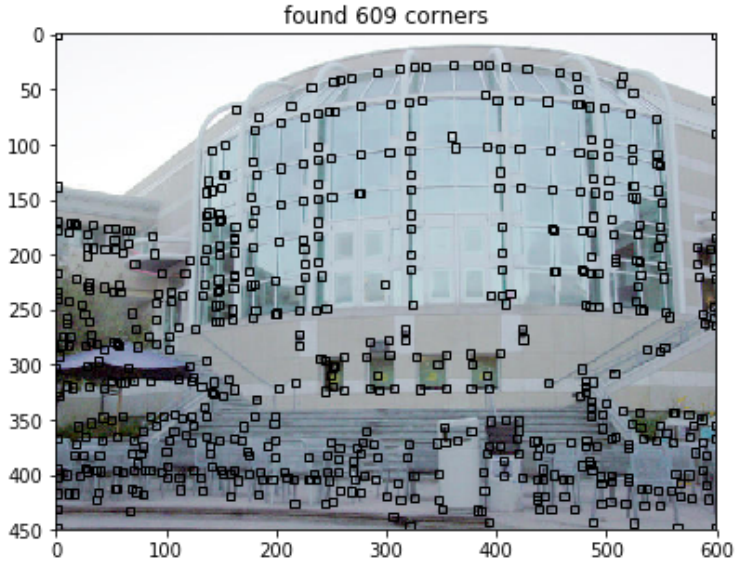
```
# display results
plt.figure(figsize=(14,24))

# show pre-thresholded corner heat map from gradient image function
plt.subplot(3,2,1)
plt.imshow(J1_0, cmap='gray')
plt.title('pre-thresholded gradient image')
plt.subplot(3,2,2)
plt.imshow(J2_0, cmap='gray')
plt.title('pre-thresholded gradient image')

# show thresholded corner heat map from gradient image function
plt.subplot(3,2,3)
plt.imshow(J1_1, cmap='gray')
plt.title('thresholded gradient image')
plt.subplot(3,2,4)
plt.imshow(J2_1, cmap='gray')
plt.title('thresholded gradient image')

# show corners on original images
ax = plt.subplot(3,2,5)
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(3,2,6)
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()
```

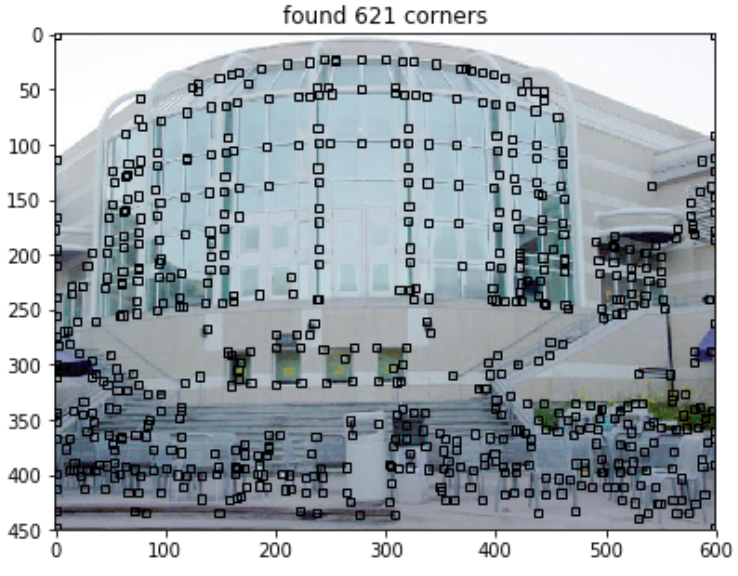

pre-thresholded gradient image


pre-thresholded gradient image


thresholded gradient image


thresholded gradient image


found 609 corners


found 621 corners

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

My implementation takes around 40 seconds to run. If yours is more than 400 seconds you may lose points.

In [102]:

```
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
```

```
    # # is the size of the matching window to compute correlation coefficients
    #
    # 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
    # p is the size of the proximity window
    #
    # output:
    # list of the feature coordinates in image 1 and image 2

    """your code here"""
    #inds = np.vstack((np.random.choice(pts1.shape[1],200,replace=False),
```

```
    #                   np.random.choice(pts1.shape[1],200,replace=False)))

    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,scores
```

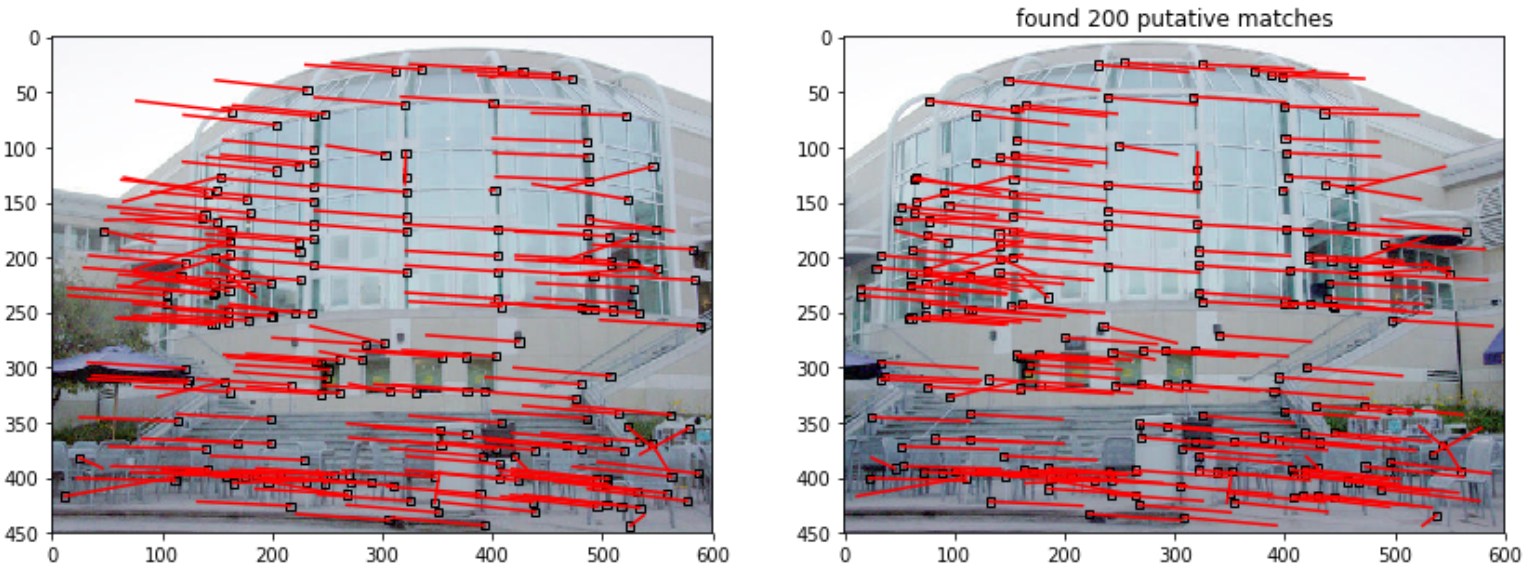
In [110]:

```
# parameters to tune
#7/0.85/1/280 190pts

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,scores = 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,:]]
```

```
took 5.132250 secs
```

In [111]:

```
# 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])
w1 = 7
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-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()

# test 1-1
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])
```




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

Final values for parameters

```
w = 7
t = 0.8
d = 9
p = np.array([100,200])
num_matches = 200
```