CSE 252B: Computer Vision II, Winter 2026 – Assignment 1

Instructor: Ben Ochoa

Assignment due: Wed, Jan 14, 11:59 PM

Name: Evan Cheng

PID: A69042831

Prior knowledge + certification of commencement of academic activity

Every course at UC San Diego, per the US Department of Education, is required to certify whether students have commenced academic activity for a class to be counted towards eligibility for Title IV federal financial aid. This certification must be completed during the first two weeks of instruction. For CSE 252B, this requirement will be fulfilled via an ungraded prior knowledge quiz, which will assist the instructional team by providing information about your background coming into the course. In Canvas (https://canvas.ucsd.edu), go to the CSE 252B course and navigate to Quizzes, then click on "First Day Survey: Prior Knowledge #FinAid"

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- All solutions must be written in this notebook.
- Math must be done in Markdown/LaTeX.
- You must show your work and describe your solution.
- Programming aspects of this assignment must be completed using Python in this notebook.
- Your code should be well written with sufficient comments to understand, but there is no need to write extra markdown to describe your solution if it is not explictly asked for.
- This notebook contains skeleton code, which should not be modified (this is important for standardization to facilate efficient grading).
- You may use python packages for basic linear algebra, but you may not use functions that directly solve the problem. If you are uncertain about using a specific package, function, or method, then please ask the instructional staff whether it is allowable.
- **You must submit this notebook as an .ipynb file, a .py file, and a .pdf file on Gradescope.**
  - You may directly export the notebook as a .py file. You may use nbconvert to convert the .ipynb file to a .py file using the following command `jupyter nbconvert --to script filename.ipynb`
  - There are two methods to convert the notebook to a .pdf file.
    - You may first export the notebook as a .html file, then print the web page as a .pdf file.
    - If you have XeTeX installed, then you may directly export the notebook as a .pdf file. You may use nbconvert to convert a .ipynb file to a .pdf file using the following command `jupyter nbconvert --allow-chromium-download --to webpdf filename.ipynb`
  - **You must ensure the contents in each cell (e.g., code, output images, printed results, etc.) are clearly visible, and are not cut off or partially cropped in the .pdf file.**
  - Your code and results must remain inline in the .pdf file (do not move your code to an appendix).
  - **While submitting on gradescope, you must assign the relevant pages in the .pdf file submission for each problem.**
- 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

$$\mathbf{M} = \begin{bmatrix} \sum_x \sum_y \left(R_x^2 + G_x^2 + B_x^2\right) & \sum_x \sum_y \left(R_x R_y + G_x G_y + B_x B_y\right) \\ \sum_x \sum_y \left(R_x R_y + G_x G_y + B_x B_y\right) & \sum_x \sum_y \left(R_y^2 + G_y^2 + B_y^2\right) \end{bmatrix}$$

$$\text{where} \;\; R_x = \frac{\partial R}{\partial x}, \quad R_y = \frac{\partial R}{\partial y}, \quad G_x = \frac{\partial G}{\partial x}, \quad G_y = \frac{\partial G}{\partial y}, \quad B_x = \frac{\partial B}{\partial x}, \quad B_y = \frac{\partial B}{\partial y}.$$

where R, G and B refer to the color channels of the input image, and x and y represent the horizontal and vertical gradient directions respectively. The gradient matrix M is computed over a local window of size w. Calculate the gradient images using the five-point central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead of the sum in M 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 600–650 features are detected in each image. For resulting nonzero pixel values, determine the subpixel feature coordinate using the Förstner corner point operator.

You may use scipy.signal.convolve to perform convolution operation and scipy.ndimage.maximum_filter for NMS operation.

Note: You must use the color images (not grayscale) for feature detection; otherwise you will lose points.

Report your final values for:

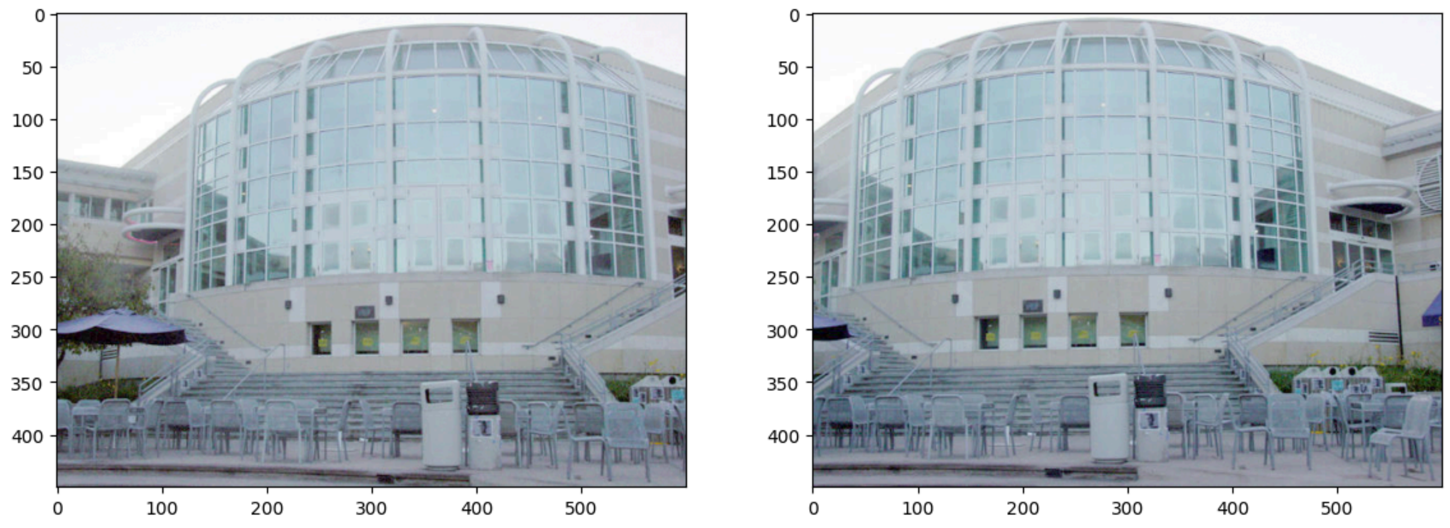
- the size of the feature detection window (i.e., the size of the window used to calculate the elements in the gradient matrix M)
- the minor eigenvalue threshold value
- the size of the local nonmaximum suppression window
- the resulting number of features detected (i.e., corners) in each image.

Display figures for:

- minor eigenvalue images before thresholding
- minor eigenvalue images after thresholding
- original images with detected features

A typical implementation takes around 30 seconds to run. If yours takes more than 120 seconds, you may lose points.

In [3]:
```
%matplotlib inline
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import time


# 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()
```



In [4]:
```
from scipy import signal
from scipy import ndimage

def image_gradient(I):
    # inputs:
    # I is the input RGB image of size mxnx3
    #
    # outputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y

    m, n = I.shape[:2]

    """your code here"""
    w = np.array([[-1/12, 8/12, 0, -8/12, 1/12]])
    w_t = w.T

    Ix = np.zeros((m, n, 3))
    Iy = np.zeros((m, n, 3))

    #red
    Ix[:,2:-2,0] = signal.convolve(I[:,:,0], w, mode='valid')
    Iy[2:-2,:,0] = signal.convolve(I[:,:,0], w_t, mode='valid')

    #green
    Ix[:,2:-2,1] = signal.convolve(I[:,:,1], w, mode='valid')
    Iy[2:-2,:,1] = signal.convolve(I[:,:,1], w_t, mode='valid')

    #blue
    Ix[:,2:-2,2] = signal.convolve(I[:,:,2], w, mode='valid')
    Iy[2:-2,:,2] = signal.convolve(I[:,:,2], w_t, mode='valid')

    return Ix, Iy

def minor_eigenvalue_image(Ix, Iy, w):
    # Calculate the minor eigenvalue image J
    #
    # inputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y
    # w is the size of the window used to compute the gradient matrix N
    #
    # outputs:
    # J0 is the mxn minor eigenvalue image of N before thresholding

    m, n = Ix.shape[:2]
    J0 = np.zeros((m,n))

    #Calculate your minor eigenvalue image J0.
    """your code here"""
    Rxx = Ix[:,:,0] ** 2
    Gxx = Ix[:,:,1] ** 2
    Bxx = Ix[:,:,2] ** 2

    Rxy = Ix[:,:,0] * Iy[:,:,0]
    Gxy = Ix[:,:,1] * Iy[:,:,1]
    Bxy = Ix[:,:,2] * Iy[:,:,2]

    Ryy = Iy[:,:,0] ** 2
    Gyy = Iy[:,:,1] ** 2
    Byy = Iy[:,:,2] ** 2

    Ixx = Rxx + Gxx + Bxx
    Iyy = Ryy + Gyy + Byy
    Ixy = Rxy + Gxy + Bxy

    sum = np.ones((w, w))
    Sxx = signal.convolve(Ixx, sum, mode='same')
    Syy = signal.convolve(Iyy, sum, mode='same')
    Sxy = signal.convolve(Ixy, sum, mode='same')

    detM = Sxx * Syy - Sxy * Sxy
    traceM = Sxx + Syy
    J0 = 0.5 * (traceM - np.sqrt(np.maximum(traceM**2 - 4 * detM, 0)))

    return J0

def nms(J, w_nms):
    # Apply nonmaximum supression to J using window w_nms
    #
    # 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"""
    R_max = ndimage.maximum_filter(J, w_nms)
    J2 = np.where(J == R_max, J, 0)

    return J2

def forstner_corner_detector(Ix, Iy, w, t, w_nms):
    # Calculate the minor eigenvalue image J
    # Threshold J
    # Run non-maxima suppression on the thresholded J
    # Gather the coordinates of the nonzero pixels in J
    # Then compute the sub pixel location of each point using the Forstner operator
    #
    # inputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y
    # 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 local nonmaximum suppression window
    #
    # outputs:
    # C is the number of corners detected in each image
    # pts is the 2xC array of coordinates of subpixel accurate corners
    #     found using 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
    # J2 is the mxn minor eigenvalue image of N after thresholding and NMS

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

    #Calculate your minor eigenvalue image J0 and its thresholded version J1.
    """your code here"""
    J0 = minor_eigenvalue_image(Ix, Iy, w)

    J1 = np.where(J0 > t, J0, 0)

    #Run non-maxima suppression on your thresholded minor eigenvalue image.
    J2 = nms(J1, w_nms)

    #Detect corners.
    """your code here"""
    y, x = np.where(J2 != 0)
    C = len(x)

    r = w // 2
    pts = []

    for c in range(C):
        i = x[c]
        j = y[c]

        Ix_w = Ix[j-r:j+r+1, i-r:i+r+1, :]
        Iy_w = Iy[j-r:j+r+1, i-r:i+r+1, :]

        X, Y = np.meshgrid(np.arange(i-r, i+r+1), np.arange(j-r, j+r+1))

        Rxx = Ix_w[:,:,0] ** 2
        Gxx = Ix_w[:,:,1] ** 2
        Bxx = Ix_w[:,:,2] ** 2

        Rxy = Ix_w[:,:,0] * Iy_w[:,:,0]
        Gxy = Ix_w[:,:,1] * Iy_w[:,:,1]
        Bxy = Ix_w[:,:,2] * Iy_w[:,:,2]

        Ryy = Iy_w[:,:,0] ** 2
        Gyy = Iy_w[:,:,1] ** 2
        Byy = Iy_w[:,:,2] ** 2

        Ixx = Rxx + Gxx + Bxx
        Iyy = Ryy + Gyy + Byy
        Ixy = Rxy + Gxy + Bxy

        Sxx = np.sum(Ixx)
        Syy = np.sum(Iyy)
        Sxy = np.sum(Ixy)

        A = np.array([[Sxx, Sxy], [Sxy, Syy]])

        bx = np.sum(X * Ixx + Y * Ixy)
        by = np.sum(X * Ixy + Y * Iyy)
        b = np.array([bx, by])

        corner_x, corner_y = np.linalg.solve(A, b)

        pts.append([corner_x, corner_y])

    pts = np.array(pts).T
    C = pts.shape[1]

    return C, pts, J0, J1, J2


# feature detection
def run_feature_detection(I, w, t, w_nms):
    Ix, Iy = image_gradient(I)
    C, pts, J0, J1, J2 = forstner_corner_detector(Ix, Iy, w, t, w_nms)
    return C, pts, J0, J1, J2
```

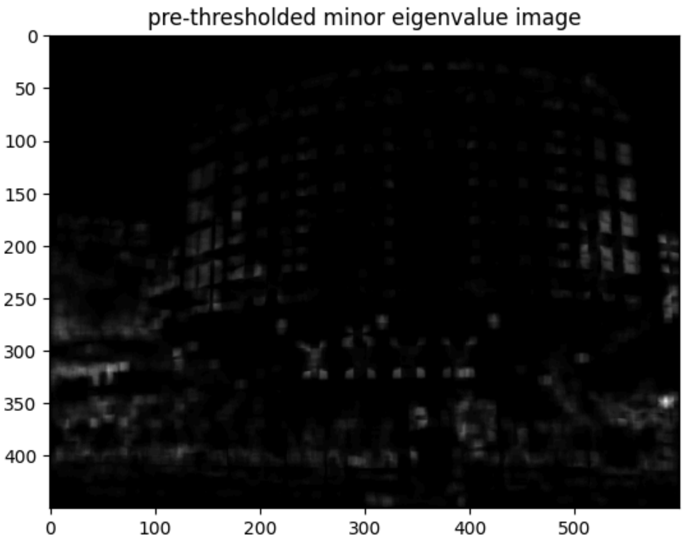
In [5]:
```
# ImageGradient() unit test
def check_values(I, target):
    eps = 1e-8  # Floating point error threshold
    I = I[2:-2, 2:-2]  # Ignore border values
    valid = np.all((I < target + eps) & (I > target - eps))
    print(f'Image is all equal to {target} +/- {eps}: {valid}')

def gray_to_RGB(I):
    h, w = I.shape
    I = np.expand_dims(I, axis=-1)
    return np.broadcast_to(I, (h, w, 3))

rampx = np.array(Image.open('rampx.png'), dtype='float')
rampy = np.array(Image.open('rampy.png'), dtype='float')

rampx = gray_to_RGB(rampx)
rampy = gray_to_RGB(rampy)

# rampx_Ix should be all ones, rampx_Iy should be all zeros (to floating point error)
rampx_Ix, rampx_Iy = image_gradient(rampx)
check_values(rampx_Ix, 1)
check_values(rampx_Iy, 0)

# rampy_Ix should be all zeros, rampx_Iy should be all ones (to floating point error)
rampy_Ix, rampy_Iy = image_gradient(rampy)
check_values(rampy_Ix, 0)
check_values(rampy_Iy, 1)
```

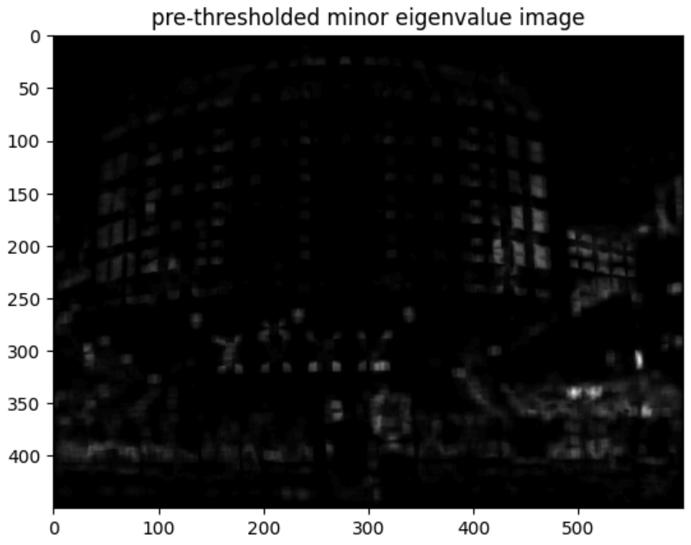
```
Image is all equal to 1 +/- 1e-08: True
Image is all equal to 0 +/- 1e-08: True
Image is all equal to 0 +/- 1e-08: True
Image is all equal to 1 +/- 1e-08: True
```

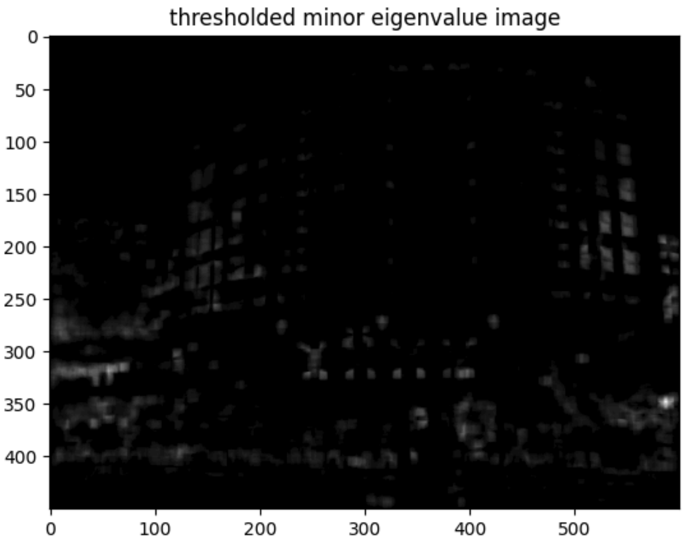
In [6]:
```
# input images
I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.

# parameters to tune
w = 9
t = 0.3
w_nms = 5

tic = time.time()
# run feature detection algorithm on input images
C1, pts1, J1_0, J1_1, J1_2 = run_feature_detection(I1, w, t, w_nms)
C2, pts2, J2_0, J2_1, J2_2 = run_feature_detection(I2, w, t, w_nms)
toc = time.time() - tic
```

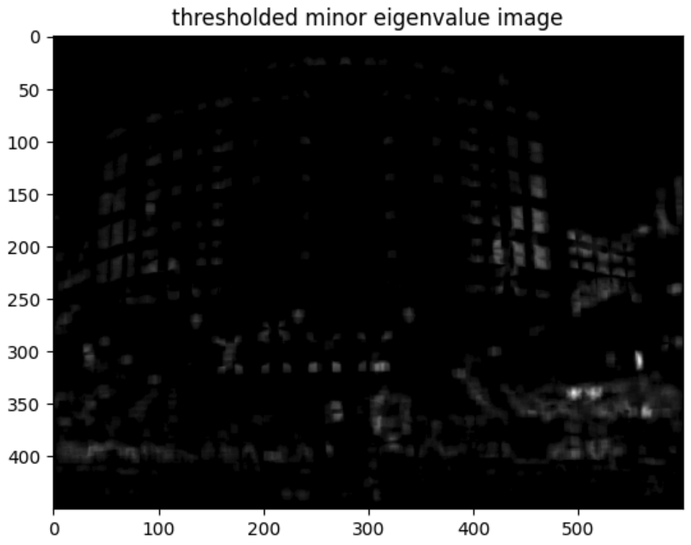
```
print('took %f secs'%toc)

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

# show pre-thresholded minor eigenvalue images
plt.subplot(3,2,1)
plt.imshow(J1_0, cmap='gray')
plt.title('pre-thresholded minor eigenvalue image')
plt.subplot(3,2,2)
plt.imshow(J2_0, cmap='gray')
plt.title('pre-thresholded minor eigenvalue image')

# show thresholded minor eigenvalue images
plt.subplot(3,2,3)
plt.imshow(J1_1, cmap='gray')
plt.title('thresholded minor eigenvalue image')
plt.subplot(3,2,4)
plt.imshow(J2_1, cmap='gray')
plt.title('thresholded minor eigenvalue 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()

# Final Parameter Values Print
print(f"\n\nFinal parameter values:\n w = {w}\n t = {t}\n w_nms = {w_nms}\n Time Taken = {toc} secs \n Corners in Image 1 - {
```

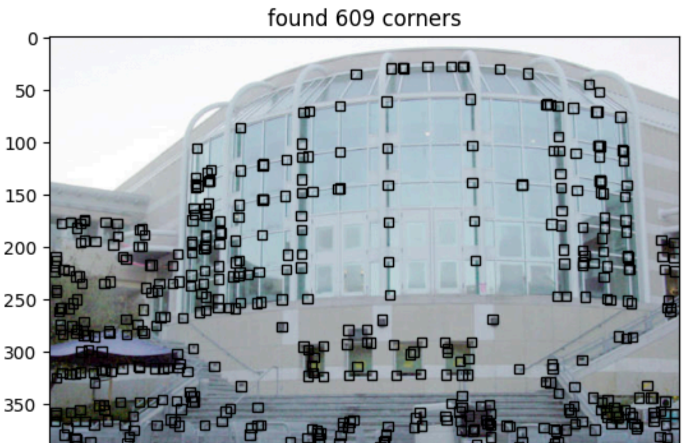
```
took 0.231956 secs
```

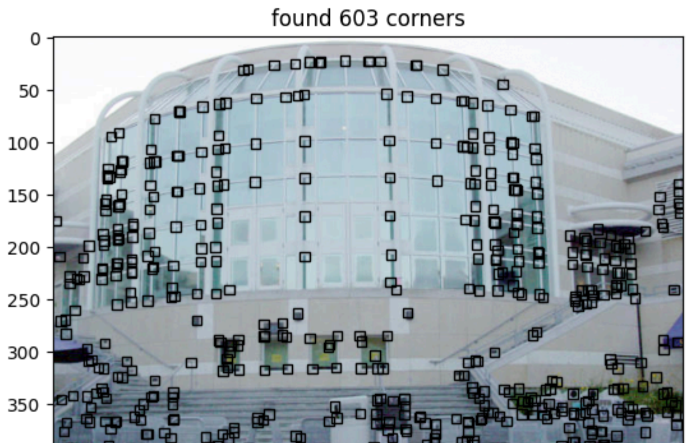

pre-thresholded minor eigenvalue image

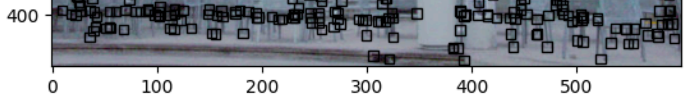
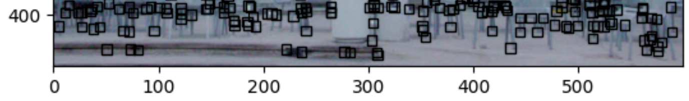

pre-thresholded minor eigenvalue image


thresholded minor eigenvalue image


thresholded minor eigenvalue image


found 609 corners


found 603 corners

```
Final parameter values:
 w = 9
 t = 0.3
 w_nms = 5
 Time Taken = 0.23195600509643555 secs
 Corners in Image 1 - 609
 Corners in Image 2 - 603
```

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 160-240 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. The proximity is calculated using the subpixel coordinates of the detected feature coordinates in image 1 and image 2. Given (x_1, y_1) in image 1 and (x_2, y_2) in image 2, you can think of a square with side length p, centered at (x_2, y_2). Then, (x_1, y_1) is within the proximity window if it lies inside that square.

Use the following formula to calculate the correlation coefficient (normalized cross correlation) between two image windows w in image 1 and w' in image 2:

$$\gamma = \frac{\sum_x \sum_y \left(w'(x,y) - \bar{w}'\right)\left(w(x,y) - \bar{w}\right)}{\sqrt{\left(\sum_x \sum_y \left(w'(x,y) - \bar{w}'\right)^2\right)\left(\sum_x \sum_y \left(w(x,y) - \bar{w}\right)^2\right)}}, \quad \gamma \in [-1, 1]$$

For color images:

$$\gamma = \min(\gamma_R, \gamma_G, \gamma_B)$$

where $w(x, y)$ is the pixel value of image 1 at (x, y) and $\bar{w}$ is the mean value of the window w. Similarly, $w'(x, y)$ and $\bar{w}'$ correspond to pixel values and mean of the window in image 2.

Note: You must center each window at the sub-pixel corner coordinates while computing normalized cross correlation, i.e., you must use bilinear interpolation to compute the pixel values at non-integer coordinates; Also, perform normalized cross correlation using the color image channels (not grayscale); otherwise, you will lose points.

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.

(You must use original (color) images to the draw boxes and correspondence lines)

A typical implementation takes around 10 seconds to run. If yours takes more than 120 seconds, you may lose points.

In [27]:
```python
from scipy.interpolate import RegularGridInterpolator

def bilinear_interpolation(pts,I_single_channel,w):
    # inputs:
    # pts: center points
    # I_single_channel: single channel input image (mxn)
    # w: window size
    #
    # output:
```

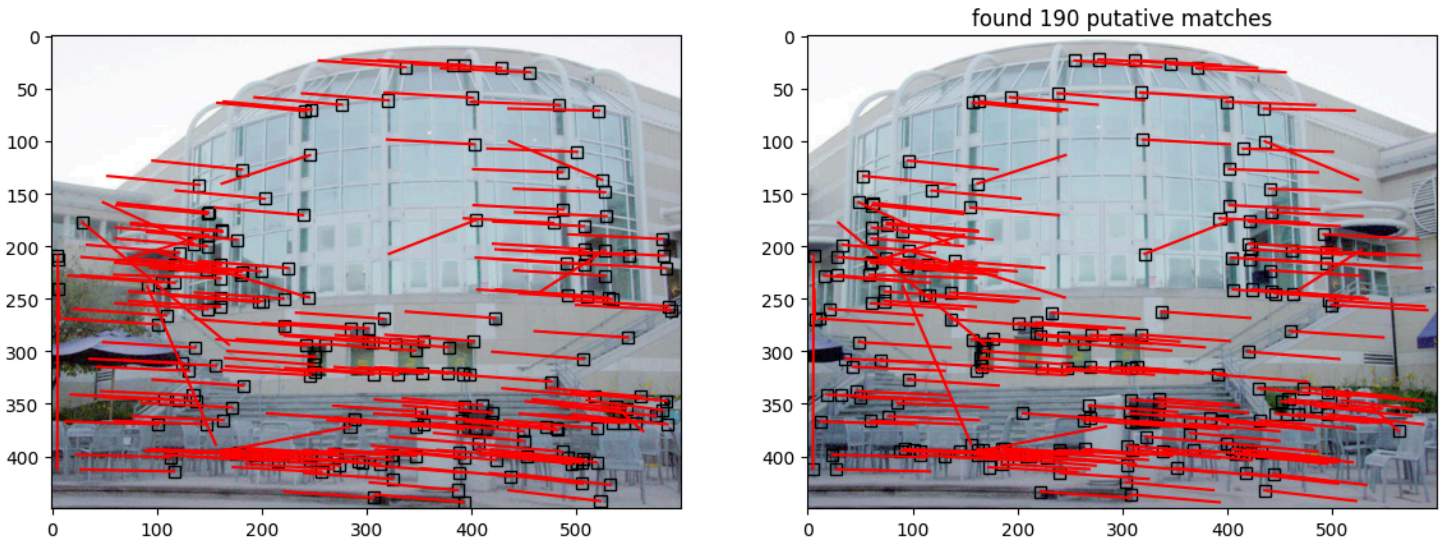
```
    # Interpolated pixel values for the corner windows

    half_win = w//2
    I_single_channel = np.pad(I_single_channel,pad_width=half_win)
    x = np.linspace(0,I_single_channel.shape[1]-1,I_single_channel.shape[1])
    y = np.linspace(0,I_single_channel.shape[0]-1,I_single_channel.shape[0])
    interp = RegularGridInterpolator((y, x),I_single_channel, bounds_error=False, fill_value=None)

    windows = []

    for c in range(pts.shape[1]):
      xx = np.linspace(pts[0][c]-half_win,pts[0][c]+half_win+1,2*half_win+1)
      yy = np.linspace(pts[1][c]-half_win,pts[1][c]+half_win+1,2*half_win+1)
      X, Y = np.meshgrid(xx, yy, indexing='ij')
      w1 = interp((Y,X))
      windows.append(w1)

    return windows


def compute_ncc(I1, I2, pts1, pts2, w, p):
    # compute the normalized cross correlation between image patches I1, I2
    # result should be in the range [-1,1]
    #
    # Do ensure that windows are centered at the sub-pixel co-ordinates
    #      while computing normalized cross correlation.
    #
    # 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"""
    scores = np.zeros((pts1.shape[1], pts2.shape[1]))
    w1 = [None] * 3
    w2 = [None] * 3

    #bilinear interpolation for RGB channels
    for c in range(3):
        w1[c] = bilinear_interpolation(pts1, I1[:,:,c], w)
        w2[c] = bilinear_interpolation(pts2, I2[:,:,c], w)

    for i in range(pts1.shape[1]):
        pt1 = pts1[:, i]
        for j in range(pts2.shape[1]):
            pt2 = pts2[:, j]

            #check proximity
            distance = np.sqrt((pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2)
            if distance > p:
                scores[i, j] = -np.inf
                continue

            ncc = np.zeros((3))

            #ncc for RGB channels
            for c in range(3):
                w_i = w1[c][i]
                w_j = w2[c][j]

                mean_i = np.mean(w_i)
                mean_j = np.mean(w_j)

                num = np.sum((w_j - mean_j) * (w_i - mean_i))
                denom = np.sqrt(np.sum((w_j - mean_j)**2) * np.sum((w_i - mean_i)**2))

                if denom != 0:
                    ncc[c] = num / denom
                else:
                    ncc[c] = 0
```

```python
                scores[i, j] = np.min(ncc)

    return scores


def perform_match(scores, t, d):
    # perform the one-to-one correspondence matching on the correlation coefficient matrix
    #
    # inputs:
    # scores is the NCC matrix
    # t is the correlation coefficient threshold
    # d distance ratio threshold
    #
    # output:
    # 2xM array of the feature coordinates in image 1 and image 2,
    # where M is the number of matches.

    """your code here"""
    inds = []

    m, n = scores.shape
    mask = np.ones((m, n), dtype=bool)

    masked_scores = np.where(mask, scores, -np.inf)

    while np.max(masked_scores) >= t:
        max_val = -np.inf
        i, j = 0, 0

        #1. find max and max index
        for row in range(m):
            for col in range(n):
                if masked_scores[row, col] > max_val:
                    max_val = masked_scores[row, col]
                    i, j = row, col

        #2. store best
        best = scores[i, j]

        #3. temporarily set to -1
        scores[i, j] = -1

        #4. find next best in row/col
        row_max = np.max(scores[i, :])
        col_max = np.max(scores[:, j])
        second_best = max(row_max, col_max)

        #5. set max value back to original value
        scores[i, j] = best

        #6. check unique
        if (1 - best) < (1 - second_best) * d:
            inds.append((i, j))

        #7. set best match row/col to false
        mask[i, :] = False
        mask[:, j] = False

        masked_scores = np.where(mask, scores, -np.inf)

    inds = np.array(inds).T

    return inds


def run_feature_matching(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 ratio threshold
    # p is the size of the proximity window
    #
    # outputs:
    # inds is a 2xK matrix of matches where inds[0,k] is an index into pts1
    #     and inds[1,k] is an index in pts2, where K is the number of matches
    scores = compute_ncc(I1, I2, pts1, pts2, w, p)
    inds = perform_match(scores, t, d)
    return inds
```

In [28]:

```
# parameters to tune
w = 11
t = 0.6
d = 0.8
p = 200

scores = compute_ncc(I1, I2, pts1, pts2, w, p)

tic = time.time()
# run the feature matching algorithm on the input images and detected features
inds = run_feature_matching(I1, I2, 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,:].astype('int')]
match2 = pts2[:,inds[1,:].astype('int')]

# 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()

# 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])

# Final Parameter Values Print
print(f"\n\nFinal parameter values:\n w = {w}\n t = {t}\n d = {d}\n p = {p}\n Time Taken = {toc} secs \n Matches Found - {mat
```

```
took 38.035335 secs
```



```
unique points in image 1: 190
unique points in image 2: 190


Final parameter values:
 w = 11
 t = 0.6
 d = 0.8
 p = 200
 Time Taken = 38.03533458709717 secs
 Matches Found - 190
```