# HW1

January 10, 2022

## 1 CSE 252B: Computer Vision II, Winter 2022 – Assignment 1

#### 1.0.1 Instructor: Ben Ochoa

#### 1.0.2 Due: Wednesday, January 12, 2022, 11:59 PM

### 1.1 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.
- 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. If you are uncertain about using a specific package, then please ask the instructional staff whether or not it is allowable.
- 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.

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

Download input data from the course website. The file price_center20.jpeg contains image 1 and the file price_center21.jpeg 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 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 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 Förstner corner point operator.

**Report your final values for:**

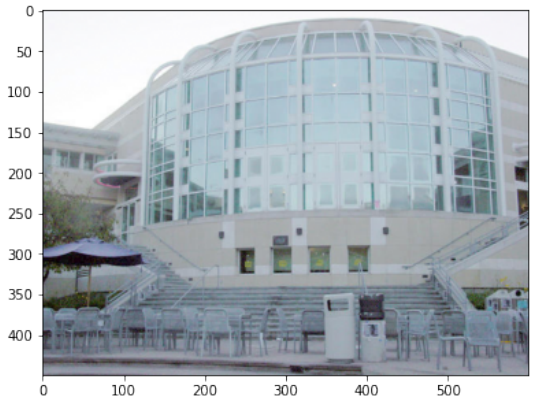
- the size of the feature detection window (i.e., the size of the window used to calculate the elements in the gradient matrix N)
- the minor eigenvalue threshold value
- the size of the local nonmaximum suppression window
- the resulting number of features detected (i.e., corners) in each image.

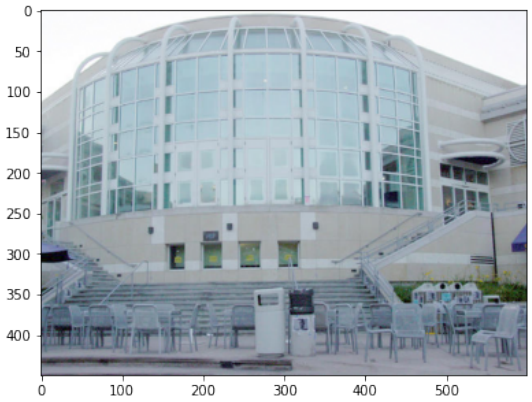
**Display figures for:**

- 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.

[1]:
```
%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 convolve
from scipy import ndimage

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




[2]:
```python
def ImageGradient(I):
    # inputs:
    # I is the input image (may be mxn for Grayscale or mxnx3 for RGB)
    #
    # 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]

    # Define filter kernel
    f = np.array([[-1,8,0,-8,1]])/12
    if(I.ndim == 3):
        # RGB Image
        filter_x = f[..., np.newaxis]
        filter_y = np.transpose(filter_x, (1,0,2))
    else:
        filter_x = f
        filter_y = f.T

    # Convolve to generate gradient images
    I_x = convolve(I, filter_x, mode = 'valid')
    I_y = convolve(I, filter_y, mode = 'valid')

    # Pad gradient image to restore original shape
    p = 2
    Iy = np.pad(I_y,((p,p),(0,0),(0,0)), 'constant', constant_values = 0)
    Ix = np.pad(I_x,((0,0),(p,p),(0,0)), 'constant', constant_values = 0)

    return Ix, Iy
```

```python
def MinorEigenvalueImage(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))

    # Pad the Image
    p = int(w/2)
    Ix = np.pad(Ix,((p,p),(p,p),(0,0)), 'constant', constant_values = 0)
    Iy = np.pad(Iy,((p,p),(p,p),(0,0)), 'constant', constant_values = 0)

    #Calculate your minor eigenvalue image J0.
    for i in range(p,p+m):
        for j in range(p,p+n):
            # Extract window
            Ix_window = Ix[i-p:i+p+1,j-p:j+p+1,:]
            Iy_window = Iy[i-p:i+p+1,j-p:j+p+1,:]
            assert Ix_window.shape[0] == w, "Window mismatch in minor eigen␣
 ↪value"
            # Compute gradient matrix entries
            Ix_2 = np.sum(np.multiply(Ix_window,Ix_window))
            Iy_2 = np.sum(np.multiply(Iy_window,Iy_window))
            Ixy = np.sum(np.multiply(Iy_window,Ix_window))
            # Compute trace and determininat
            T = Ix_2 + Iy_2
            D = Ix_2*Iy_2 - Ixy*Ixy
            k = np.maximum(T*T - 4*D,0)
            J0[i-p,j-p] =  (T - np.sqrt(k))/2

    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()
    J_max = ndimage.maximum_filter(J, size = (w_nms, w_nms), mode = ␣
 ↪'constant', cval = 0)
    J2[J < J_max] = 0


    return J2

def ForstnerCornerDetector(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.
    J0 = MinorEigenvalueImage(Ix, Iy, w)
    #Thresholding
    J1 = J0
    J1[J0<t] = 0

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

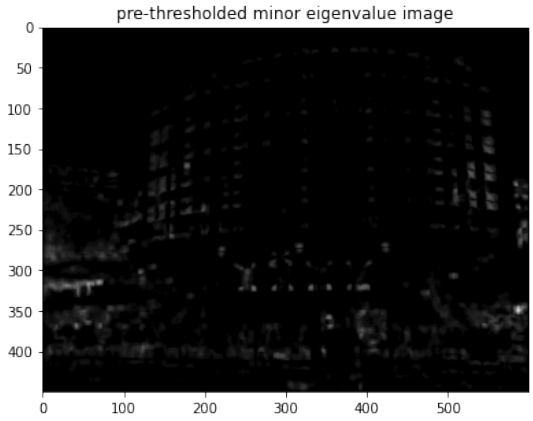
```
    #Detect corners.
    idx = np.array(np.where(J2 != 0))

    # Pad the Image
    p = int(w/2)
    Ix = np.pad(Ix,((p,p),(p,p),(0,0)), 'constant', constant_values = 0)
    Iy = np.pad(Iy,((p,p),(p,p),(0,0)), 'constant', constant_values = 0)

    pts = []
    for i in range(p,p+m):
        for j in range(p,p+n):
            if(J2[i-p,j-p] != 0):
                # Extract window
                Ix_window = Ix[i-p:i+p+1,j-p:j+p+1,:]
                Iy_window = Iy[i-p:i+p+1,j-p:j+p+1,:]
                assert Ix_window.shape[0] == w, "Window size mismatch in␣
↪ForstnerCornerDetector"

                # Compute entries of gradient matrix
                Ix_2 = np.multiply(Ix_window,Ix_window)
                Iy_2 = np.multiply(Iy_window,Iy_window)
                Ixy =  np.multiply(Iy_window,Ix_window)

                W_x = np.arange(j-p,j+p+1).reshape(-1,1)
                W_x = np.repeat(W_x, w, axis = 1)
                W_y = np.arange(i-p,i+p+1).reshape(1,-1)
                W_y = np.repeat(W_y, w, axis = 0)

                if(Ix_2.ndim == 3):
                    W_x = np.dstack([W_x]*3)
                    W_y = np.dstack([W_y]*3)

                b0 = np.sum(np.multiply(W_y, Ixy)) + np.sum(np.multiply(W_x,␣
↪Ix_2))
                b1 = np.sum(np.multiply(W_x, Ixy)) + np.sum(np.multiply(W_y,␣
↪Iy_2))
                b = np.array([b0, b1])

                N = np.array([[np.sum(Ix_2), np.sum(Ixy)],[np.sum(Ixy), np.
↪sum(Iy_2)]])
                pts.append(np.matmul(np.linalg.pinv(N),b)-p)


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

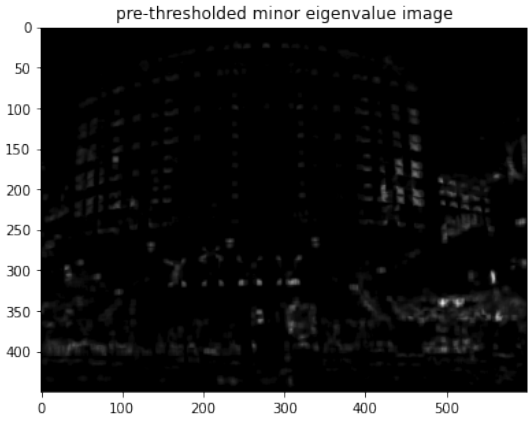
```
    return C, pts, J0, J1, J2


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

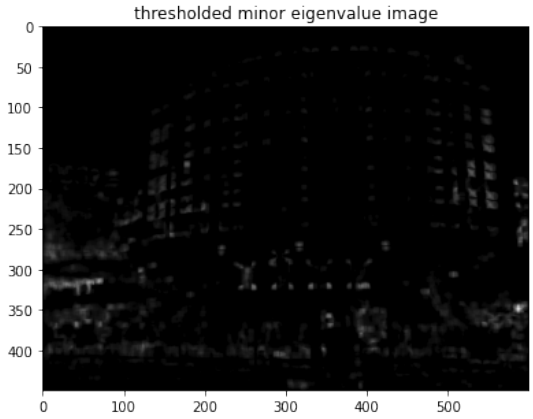
[3]:
```
# input images
I1 = np.array(Image.open('price_center20.jpeg'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.jpeg'), dtype='float')/255.

# parameters to tune
w = 7
t = 0.125
w_nms = 7

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

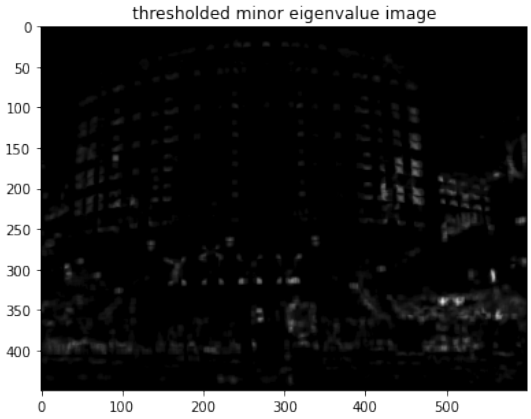
```
plt.imshow(I1)
for i in range(C1): # draw rectangles of size w around corners
    x,y = pts1[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
# plt.plot(pts1[0,:], pts1[1,:], '.b') # display subpixel corners
plt.title('found %d corners'%C1)

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()
```

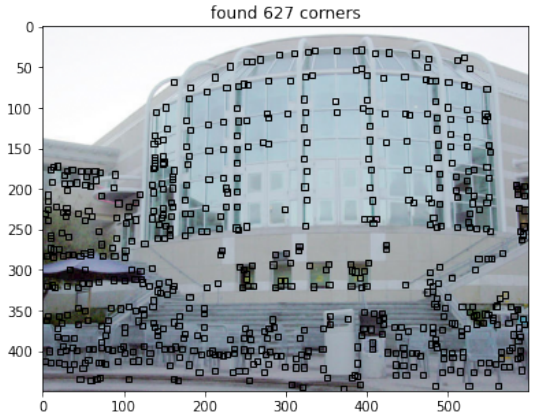
```
took 12.725629 secs
```

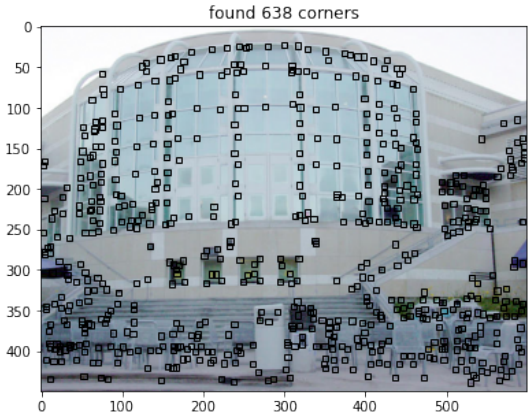

pre-thresholded minor eigenvalue image


pre-thresholded minor eigenvalue image


thresholded minor eigenvalue image


thresholded minor eigenvalue image


found 627 corners


found 638 corners

**Final values for parameters**

- w = 7
- t = 0.125
- w_nms = 7
- C1 = 627
- C2 = 638

## 2.1 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.

**Note: You must center each window at the sub-pixel corner coordinates while computing normalized cross correlation; 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

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

[4]:
```
# HELPER FUNCTIONS
def getPatch(I, xc, yc, w):
    '''
    Function performs bilinear interpolation and
    extracts patch centered at (xc,yc)
    '''
    x = int(xc)-int(w/2)
    y = int(yc)-int(w/2)
```

```
    p1 = I[x:x+w, y:y+w, :]
    p2 = I[x+1:x+w+1, y:y+w, :]
    p3 = I[x:x+w, y+1:y+w+1, :]
    p4 = I[x+1:x+w+1, y+1:y+w+1, :]

    I0 = p1*(x+1-xc) + p2*(xc-x)
    I1 = p3*(x+1-xc) + p4*(xc-x)
    if(I1.shape != I0.shape):
        print(x,y)
        print(p1.shape)
        print(p2.shape)
        print(p3.shape)
        print(p4.shape)
    D  = I0*(y+1-yc) + I1*(yc-y)

    return D
```

[5]:
```
def compute_NCC(p1, p2):
    '''
    given 2 patches, function computes
    the normalized cross correlation
    score between the 2 patches
    '''
    assert p1.shape == p2.shape, "Patches of different sizes"
    p1 = p1 - np.mean(p1)
    p2 = p2 - np.mean(p2)
    # Cross correlation term
    N = np.apply_over_axes(np.sum, np.multiply(p1,p2), [0,1])
    # Auto - correlation term
    D = np.sqrt(np.apply_over_axes(np.sum, np.multiply(p1,p1), [0,1])*np.sum(np.
 ↪apply_over_axes(np.sum, np.multiply(p2,p2),[0,1])))
    return np.min(N/D)
```

[6]:
```
def 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
    #
```

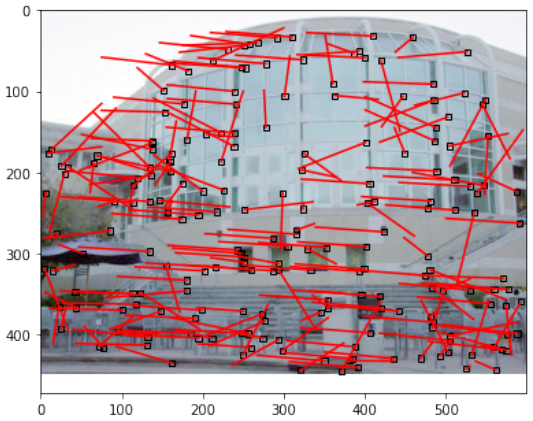
```
    # output:
    # normalized cross correlation matrix of scores between all windows in
    #    image 1 and all windows in image 2
    #

    C1 = pts1.shape[1]
    C2 = pts2.shape[1]

    P = int(w/2)
    scores = np.zeros((C1,C2))

    I1_pad = np.pad(I1.copy(), ((P,P),(P,P),(0,0)), 'constant', constant_values␣
↪= 0)
    I2_pad = np.pad(I2.copy(), ((P,P),(P,P),(0,0)), 'constant', constant_values␣
↪= 0)

    for i in range(C1):
        for j in range(C2):
            s = -1

            # proximity filter
            if np.linalg.norm(pts1[:,i]-pts2[:,j]) < p:
                # Extract patches
                xc1 = pts1[1,i] + P
                yc1 = pts1[0,i] + P
                xc2 = pts2[1,j] + P
                yc2 = pts2[0,j] + P

                p1 = getPatch(I1_pad, xc1, yc1, w)
                p2 = getPatch(I2_pad, xc2, yc2, w)

                assert p1.shape[0] == w, "Window size mismatch in NCC"
                assert p2.shape[0] == w, "Window size mismatch in NCC"

                s = compute_NCC(p1, p2)
                assert s>=-1 or s<=1, 'Invalid NCC value'

            scores[i,j] = s

    return scores


def Match(scores, t, d):
    # perform the one-to-one correspondence matching on the correlation␣
↪coefficient matrix
    #
    # inputs:
```

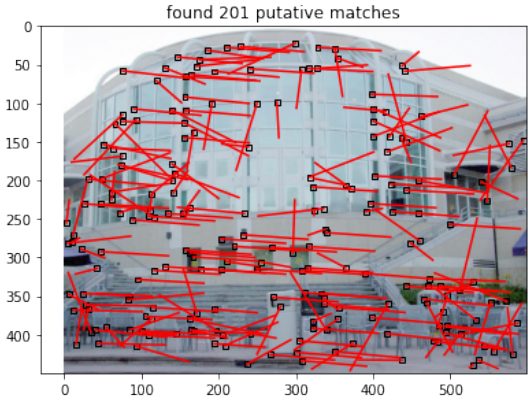
```
    # scores is the NCC matrix
    # t is the correlation coefficient threshold
    # d distance ration threshold
    #
    # output:
    # 2xM array of the feature coordinates in image 1 and image 2,
    # where M is the number of matches.

    inds = []
    mask = np.full(scores.shape, True, dtype=bool)

    while(np.max(scores) > t):
        idx = np.unravel_index(scores.argmax(), scores.shape)

        # Find max
        best_match = scores[idx]
        scores[idx] = -1

        # Find next best match
        next_best = np.maximum(np.max(scores[idx[0],:]),np.max(scores[:
↪,idx[1]]))
        scores[idx] = best_match

        # Append if match good enough
        if((1-best_match) < (1-next_best)*d):
            inds.append(idx)

        mask[idx[0],:] = False
        mask[:,idx[1]] = False

        #mask scores matrix
        scores[np.logical_not(mask)] = -1
    inds = np.array(inds).T
    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
```

[7]:
```
# parameters to tune
w = 7
t = 0.01
d = .95
p = 100

tic = time.time()
# run the feature matching algorithm on the input images and detected features
inds = RunFeatureMatching(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])
```

```
took 10.514711 secs
```




```
unique points in image 1: 201
unique points in image 2: 201
```

### Final values for parameters

- w = 7
- t = 0.01
- d = 0.95
- p = 100
- num_matches = 201