CS6550 Computer Vision Homework1 – Image Features

112062673 吳文雯

Q1.Histogram equalization

1.Implementation

step 1

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

img = cv2.imread('hw1-1.jpg',cv2.IMREAD_GRAYSCALE)

arr_i = img.flatten()

arr = np.zeros(256,dtype=int)

arr_c = np.zeros(256,dtype=int)
#存放 equalize 後的灰階值
arr_t = np.zeros(256,dtype=int)
...
```

◆ Read the image in grayscale, and store in the variable img. Then turn img into 1-dimensional NumPy array "arr_i" for plotting histogram.
◆ Construct 1-dimensional NumPy arrays: arr for storing how many times each gray-scale value appears; "arr_c" for storing the cumulatuve value of each gray-scale value; "arr_t" for storing the gray scale values after histogram equalized.

step 2

```
...
```

```
#h=rows, w=cols
h,w = img.shape[0:2]

#把灰階值出現次數存入 arr
for i in range(h):
    for j in range(w):
        v = img[i,j]
        arr[v] += 1

#計算 cumulative 灰階值
for i in range(256):
    sum_i = 0
    for j in range(i+1):
        sum_i += arr[j]
    arr_c[i] = sum_i

#進行 equalization
for i in range(256):
    arr_t[i] = round((255/(h*w))*arr_c[i])

#建立陣列儲存新圖片的灰階值
arr_e = np.zeros((h,w), dtype = int)
#更新灰階值  k 是新的灰階值
for i in range(h):
    for j in range(w):
        k = img[i,j]
        arr_e[i][j] = arr_t[k]
...
```

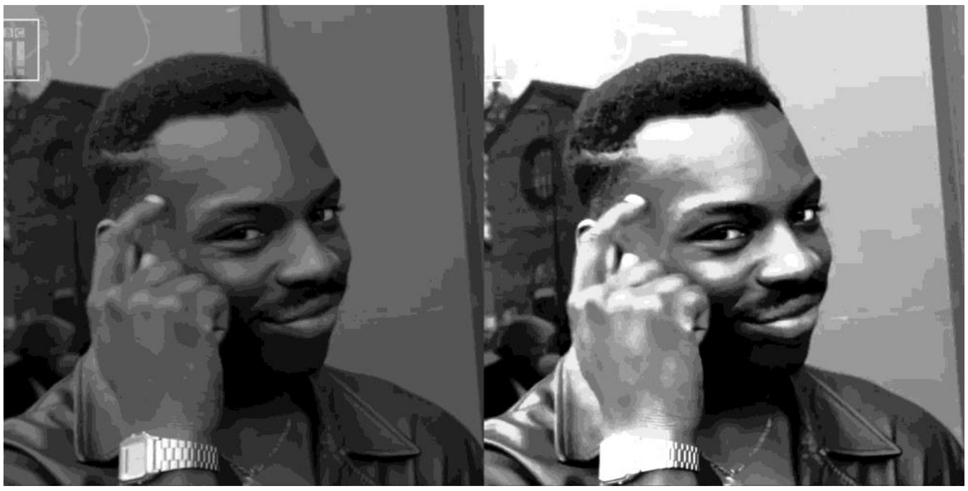
◆ Get the height and width of the image and store the two value in "h" and "w."
◆ Scan each pixel of the image and compute how many times the gray-scale value appears. The results are stored in the array "arr."
◆ Caculate the cumulative values of each gray-scale value and store the results in the array "arr_c."
◆ Performing equlization by using the formula: T[p]=round((G-1/NM)*Hc[p]) provided on page 3 in the ppt of unit 2.The results are stored in "arr_t."
◆ Construct 2-dimensional NumPy array "arr_e" for storing the gray-scale value after histogram equalization.

◆ Scan each pixel of the image, and update their gray-scale values according to "arr_t."

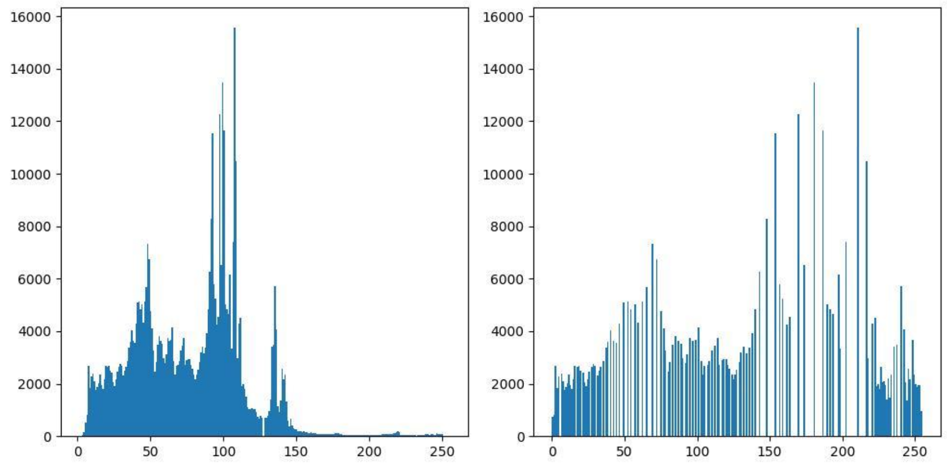
step 3

```
#把 arr_e 轉成 uint8 表示法
img_e = arr_e.astype(np.uint8)

#把兩張圖依水平方向放在一起
imgs = np.hstack((img,img_e))
cv2.imwrite('hw1-101', imgs)
cv2.imshow("image", imgs)
cv2.waitKey(0)

arr_e = arr_e.flatten()

#繪製原本灰階圖的 histogram
fig, ax = plt.subplots(1,2, figsize=(10,5))
ax[0].hist(arr_i, bins=256, range=None, density=None, cumulative=False,
histtype='bar', align='mid', orientation='vertical', rwidth=None, color=None,
label=None, stacked=False)

#繪製更新後灰階圖的 histogram
ax[1].hist(arr_e, bins=256, range=None, density=None, cumulative=False,
histtype='bar', align='mid', orientation='vertical', rwidth=None, color=None,
label=None, stacked=False)

plt.tight_layout()
plt.show()
plt.savefig('hw1-102.jpg')
```

◆ Convert the 2-d NumPy array "arr_e" into the type of uint8 so that cv2.imshow() can accept.
◆ Concatenate the original gray image and the one after histogram equalization horizontally, and show them together.



◆ Flatten the np.array after histogram equalization and construct its histogram and the original one by plt.subplots().
◆ Use plt.tight_layout() and plt.show() to show the 2 histogram one near the other one. Then save the image to the file by plt.savefig().



Q2.Harris corner detector

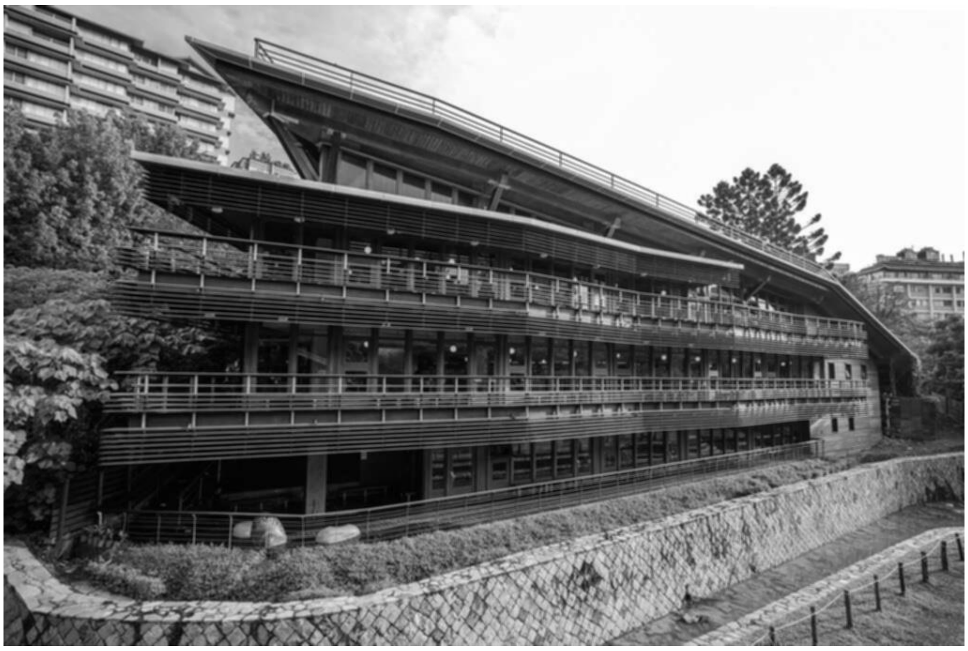
1.Implementation

step 1

```
-----------------------------------------------------------------------------------------------------------------------
#讀圖片且轉成灰階  要跟 code 放在同一個資料夾內
img = cv2.imread('hw1-2.jpg',cv2.IMREAD_GRAYSCALE)
rows,cols = img.shape[0:2]


#取得 kernel 內部參數
```

```
kernel_size = 3
sigma = 3
#利用一維參數取得二維參數 T 表示轉置
kernel_1d = cv2.getGaussianKernel(kernel_size, sigma)
kernel_2d = kernel_1d * kernel_1d.T
#print(kernel_2d)

#進行 gaussian filter(blur)
#-1 表示產出的圖跟原圖會有相同 depth
img_gb = cv2.filter2D(img, -1, kernel_2d)
#print(img_gb)
cv2.imwrite('hw1-2i.jpg', img_gb)
cv2.imshow("image", img_gb)
cv2.waitKey(0)
```

◆ Read the image in grayscale, and store in the variable img.
◆ Get the values of 1d kernel with the size and sigma being both 3 by cv2.getGaussianKernel(). And 2d Gaussian kernel can be obtained by 1d kernel and its transpose.
◆ Use cv2.filter2D() to convolve the gray image "img" and the 2d kernel to get an image after applying Gaussian blur.



step 2

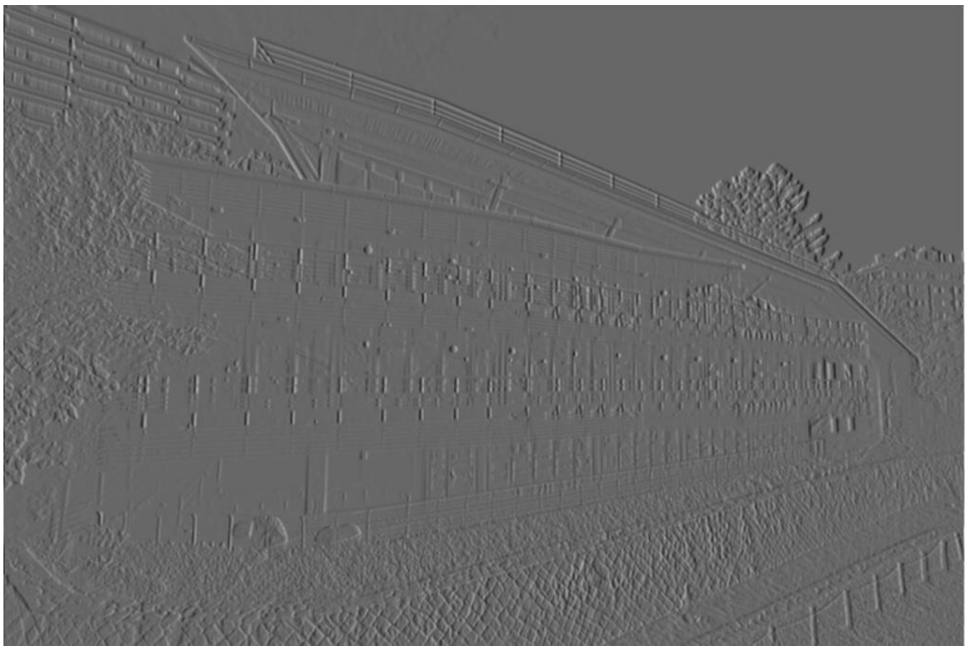
```
#apply Sobel operator Hx
```

```
Hx = np.array(
[[-1, 0, 1],
[-2, 0, 2],
[-1, 0, 1]]
)
gx = convolve2d(img_gb, Hx, mode='same')

#把原本的結果 轉成 0-255 印出
gmx = np.zeros((rows,cols), dtype = int)
for i in range(rows):
    for j in range(cols):
        gmx[i,j] = gx[i,j]

xmin = gx.min()
xmax = gx.max()
for i in range(rows):
    for j in range(cols):
        gmx[i,j] = ((gmx[i,j]-xmin)/(xmax-xmin))*255
gmx = gmx.astype(np.uint8)

cv2.imwrite('hw1-2ii01.jpg', gmx)
cv2.imshow("image", gmx)
cv2.waitKey(0)
```

◆ Consruct Sobel Operator Hx according to p.25 of the Unit_2 ppt, and convolve the image applied Gaussian blur with Hx.

◆ Convert the values of the gradient into the range of 0 to 255, and set the type to np.uint8.

◆ Show the image of the gradient in x direction.

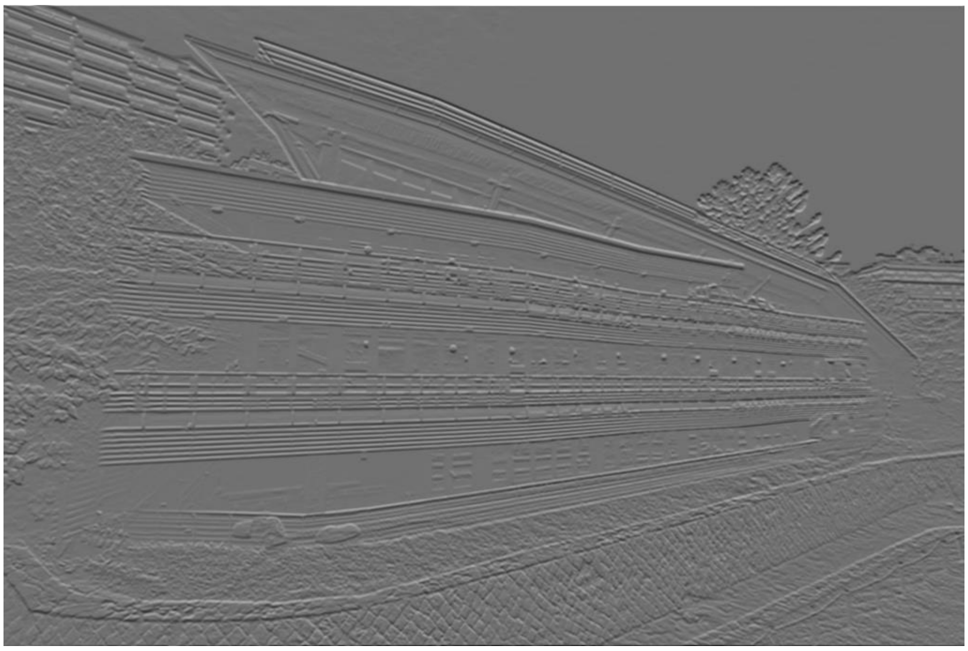


step 3

```
#apply Sobel operator Hy
Hy = np.array(
[[1, 2, 1],
[0, 0, 0],
[-1, -2, -1]]
)
gy = convolve2d(img_gb, Hy, mode='same')

#把原本的結果 轉成 0-255 印出
gmy = np.zeros((rows,cols), dtype = int)
for i in range(rows):
    for j in range(cols):
        gmy[i,j] = gy[i,j]

ymin = gy.min()
ymax = gy.max()
for i in range(rows):
    for j in range(cols):
        gmy[i,j] = ((gmy[i,j]-ymin)/(ymax-ymin))*255
gmy = gmy.astype(np.uint8)

cv2.imwrite('hw1-2ii02.jpg', gmy)
```

```
cv2.imshow("image", gmy)
cv2.waitKey(0)
```

◆ Consruct Sobel Operator Hx according to p.25 of the Unit_2 ppt, and convolve the image applied Gaussian blur with Hy.
◆ Convert the values of the gradient into the range of 0 to 255, and set the type to np.uint8.
◆ Show the image of the gradient in y direction.

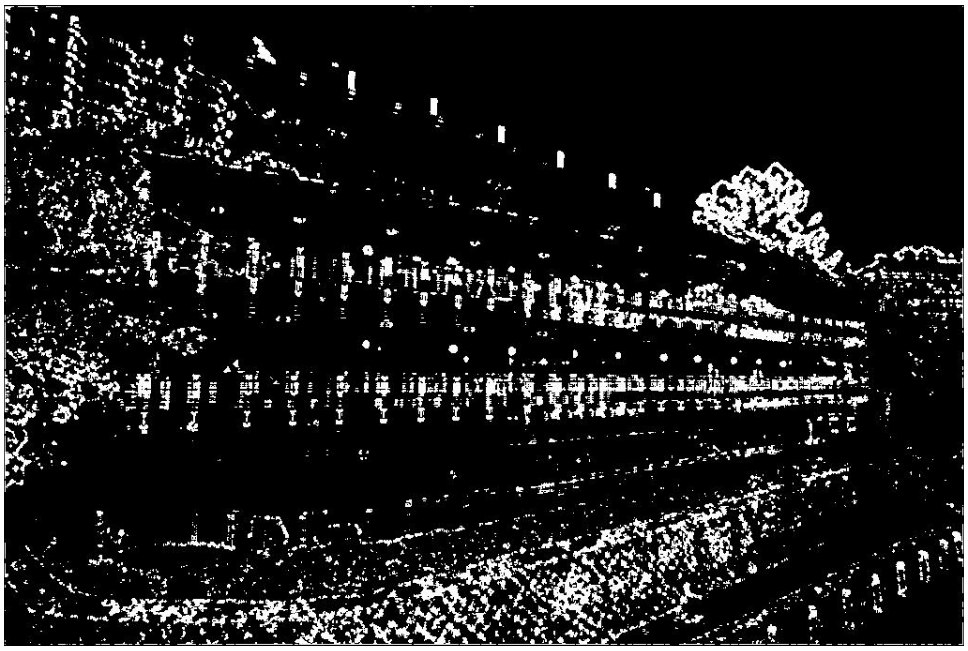


step 4

```
gmx = gmx.astype(np.int32)
gmy = gmy.astype(np.int32)
#compute Ix^2 Iy^2 Ix*Iy of each pixel
Ix2 = np.zeros((rows,cols), dtype = np.int32)
Iy2 = np.zeros((rows,cols), dtype = np.int32)
Ixy = np.zeros((rows,cols), dtype = np.int32)
for i in range(rows):
    for j in range(cols):
        Ix2[i,j] = gx[i,j]**2
        Iy2[i,j] = gy[i,j]**2
        Ixy[i,j] = gx[i,j] * gy[i,j]

#compute sum value according to window size
Sx = np.zeros((rows,cols), dtype = np.int32)
```

```
Sy = np.zeros((rows,cols), dtype = np.int32)
Sxy = np.zeros((rows,cols), dtype = np.int32)
#zero padding
for i in range(1,rows-1):
    for j in range(1,cols-1):
        #window size = 3
        for x in range(-1,2):
            for y in range(-1,2):
                Sx[i,j] += Ix2[i+x,j+y]
                Sy[i,j] += Iy2[i+x,j+y]
                Sxy[i,j] += Ixy[i+x,j+y]
#compute matrix H according to window size
img_H = np.zeros((rows,cols,2,2), dtype = int)
#zero padding
for i in range(1,rows-1):
    for j in range(1,cols-1):
        img_H[i,j,0,0] = Sx[i,j]
        img_H[i,j,0,1] = Sxy[i,j]
        img_H[i,j,1,0] = Sxy[i,j]
        img_H[i,j,1,1] = Sy[i,j]

#compute R of each pixel and stored in img_R
arr_R = np.zeros((rows,cols), dtype = np.float64)
k = 0.04
for i in range(rows):
    for j in range(cols):
        det = linalg.det(img_H[i,j])
        trace = img_H[i,j].trace()
        arr_R[i,j] = round(det-k*(trace**2))

rmax = arr_R.max()
#output only R>threshold
#list for collecting possible corners
#(value,列編號,行編號,0 表示沒檢查過)
list = []
threshold = 0.0005*rmax
img_R = np.zeros((rows,cols), dtype = np.uint8)
for i in range(rows):
    for j in range(cols):
        if arr_R[i,j] > threshold:
            list.append([arr_R[i,j], i, j, 0])
            img_R[i,j] = 255
img_R = img_R.astype(np.uint8)
cv2.imwrite('hw1-2iii.jpg', img_R)
cv2.imshow("image",img_R)
cv2.waitKey(0)
```

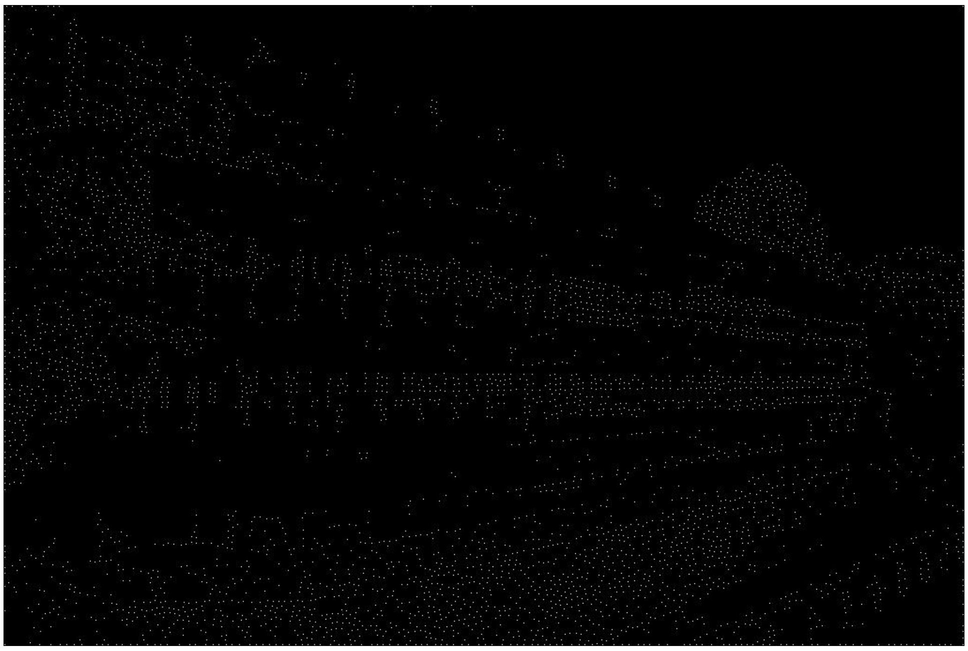
◆ Convert the maginitude of gradient into the type of np.int32 for later calculation.
◆ To compute the structure tensor H of each pixel according to the window size, I first compute Ix^2, Iy^2, and Ix*Iy of each pixel. Then sum the values of the 9 pixels in the window for the middle pixel, and construct the structure tensor of each pixel.
◆ Since the value of those pixels around the border of the image cannot be compute correctly, I use zero padding and let their values remain zero.
◆ Constuct an array arr_R to store the response of each pixel, and compute the response by the formula: R=detA-k(traceA)^2. K is usually 0.04-0.06 and I define k as 0.04 here.
◆ Use arr_R.max() to get the larger response value of the image, and store it in the variable "rmax."
◆ Construct a list to gather those points with reponse value higher than the threshold, defined as 0.0005*rmax here. The list will be used when performing non-maximal suppression.
◆ Construct a 2d np.array "img_R" with the same size of the original gray image, and scan over "arr_R." If the response of the pixel is higher than the threshold, I set its value to be 255, otherwise 0.
◆ Those points set to 255 will also be appended into the list in the form of [reponse value, row number, column number, 0] where 0 represents that the point has not been evaluted when performing non-max suppression.
◆ Convert the type of "img_R" into np.uint8 and show the image by cv2.imshow().



step 5

```
#non-maximum suppression
#sort the list in decreasing order
list.sort(reverse=True)
winsize = 5
#從 r 最大的開始檢查還沒看過的點
for c in list:
    if c[3] == 0:
        for k in list:
            if k[3] == 0:
                #check the value of other possible corner in the window
                dist = math.sqrt((k[1] - c[1])**2 + (k[2] - c[2])**2)
                #這些 k 都是 c 的鄰居且比 c 小  最後要濾掉 1 用來表示看過且要 suppress 的點
                if (dist <= winsize and dist > 0):
                    k[3] = 1
        #確認 local maximum 大於 threshold
        if c[0] < threshold:
            c[3] = 1
#最後留下的點放入 lmlist
lmlist = filter(lambda x: x[3] == 0, list)
img_NMS = np.zeros((rows,cols), dtype = np.uint8)
for c in lmlist:
    i = c[1]
    j = c[2]
    img_NMS[i,j] = 255;
img_NMS = img_NMS.astype(np.uint8)
cv2.imwrite('hw1-2iv.jpg', img_NMS)
cv2.imshow("image",img_NMS)
cv2.waitKey(0)
```

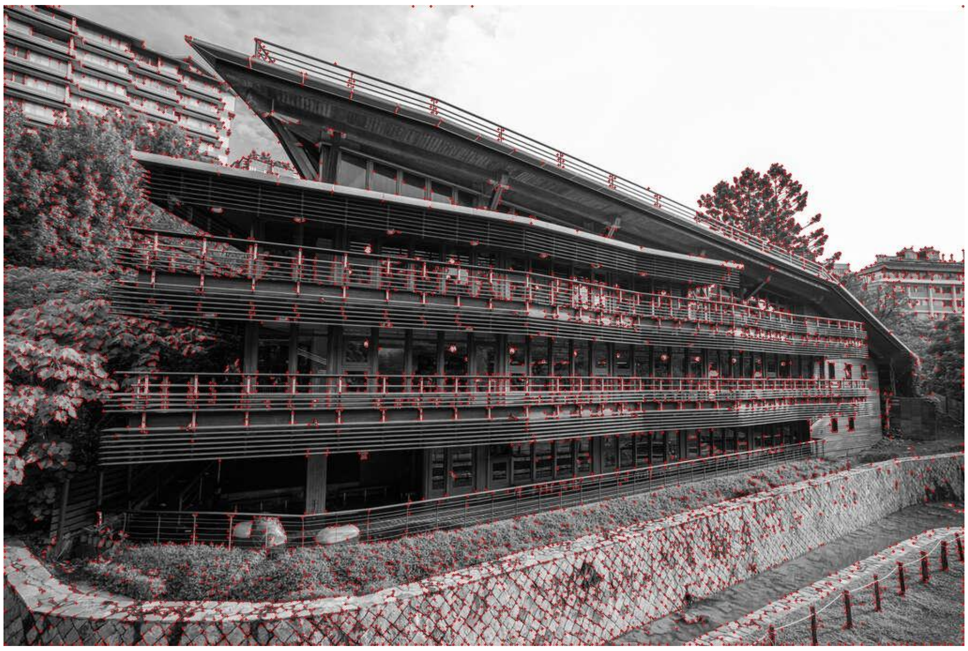
◆ First sort the list in decreasing order according to their response value and defined window size as 5.
◆ Take the first point "c", the one with the largest response, from the list and check the other points "k" in the list. If the distance of "c" and "k" is smaller than the window size, it means they are in the same window. Then I keep "c" as the local maximum and set k[3] as 1, which means this k has been evaluated and will be eliminated later.
◆Check whether the response of the local maximum c is higher than the threshold, if not, c[3] will be set to 1 so that it will not be shown on the image later.
◆ Then take the second point from the list and perform the before mentioned step, until all the points in the list have been evaluated. The check for c[3] and k[3] is to make sure that a point that is unwanted would not be evaluated again.
◆ Use filter to get only the local maximum points and put them into lmlist, and take the points in lmlist one by one.
◆ Construct a 2d np.array "img_NMS" with the same size of the original gray image, and set the value of those pixels in the lmlist to be 255.
◆ Convert the type of "img_NMS" into np.uint8 and show the image by cv2.imshow().



step 6

```
#把 corner 疊在灰階圖上
img_gray = cv2.imread('hw1-2.jpg', cv2.IMREAD_GRAYSCALE)
img_final = cv2.cvtColor(img_gray, cv2.COLOR_GRAY2BGR)
for i in range(rows):
    for j in range(cols):
        if img_NMS[i,j] == 255:
            x = j
            y = i
            cv2.circle(img_final, (x,y), radius=1, color=(0, 0, 255), thickness=-
1)

cv2.imwrite('hw1-2b.jpg', img_final)
cv2.imshow("image",img_final)
cv2.waitKey(0)
```

◆ Read the original image in gray scale and store it as "img_gray", and convert it into a 3-channel image by cv2.cvtColor(img_gray, cv2.COLOR_GRAY2BGR). By doing so, the red dots of the corners can be shown on the gray image.
◆ Use cv2.circle to draw only the points considered corners after performing non-maximal suppression, and then show the image.

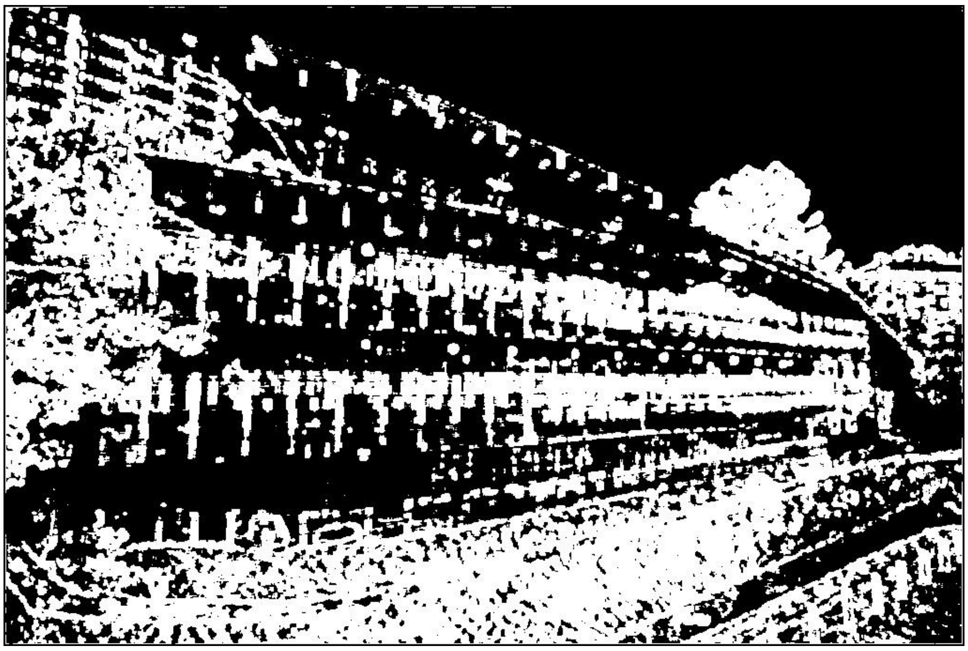


step 7 (different window size)
The following steps are similar to the prior steps, so I only show the different parts of

the code.

```
#compute sum value according to window size 5
Sx2 = np.zeros((rows,cols), dtype = np.int32)
Sy2 = np.zeros((rows,cols), dtype = np.int32)
Sxy2 = np.zeros((rows,cols), dtype = np.int32)
#zero padding
for i in range(2,rows-2):
    for j in range(2,cols-2):
        #window size = 5
        for x in range(-2,3):
            for y in range(-2,3):
                Sx2[i,j] += Ix2[i+x,j+y]
                Sy2[i,j] += Iy2[i+x,j+y]
                Sxy2[i,j] += Ixy[i+x,j+y]
#compute matrix H according to window size
img_H2 = np.zeros((rows,cols,2,2), dtype = int)
#zero padding
for i in range(2,rows-2):
    for j in range(2,cols-2):
        img_H2[i,j,0,0] = Sx2[i,j]
        img_H2[i,j,0,1] = Sxy2[i,j]
        img_H2[i,j,1,0] = Sxy2[i,j]
        img_H2[i,j,1,1] = Sy2[i,j]
```

◆ Since the window size is now set to 5, the values of more pixels around the image border cannot be correctly computed, I change the range for iteration as (2, rows-2) and (2, cols-2).
◆ Change the range for computing the neighbor points in the window as (-2,3) so that it would include 25 pixels around the middle pixel, including itself, of the window.
◆ The result image is the following one.



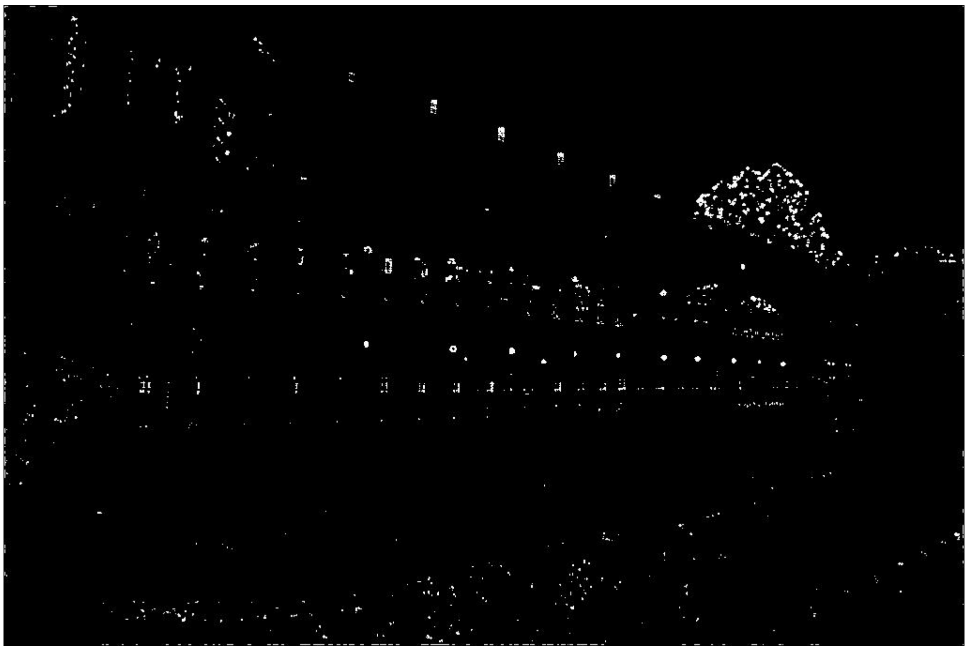
step 8 (different threshold)

The following steps are similar to steps 1 to 5, so I only show the different parts of the code.

```
list3 = []
threshold3 = 0.005*rmax
img_R3 = np.zeros((rows,cols), dtype = np.uint8)
for i in range(rows):
    for j in range(cols):
        if arr_R3[i,j] > threshold3:
            list.append([arr_R3[i,j], i, j, 0])
            img_R3[i,j] = 255
img_R3 = img_R3.astype(np.uint8)
cv2.imwrite('hw1-2cii.jpg', img_R3)
cv2.imshow("image",img_R3)
cv2.waitKey(0)
```

◆ The threshold here is set to 0.005*rmax, which is 10 times larger than the first threshold.

◆ The result image is the following one.



2.Discussion Section

■ **How does window size affect the result?**
When the window size becomes larger, it will include more pixels when computing the response of each pixel.Therefore, the reponse value of one pixel can be influenced by some further points comparing to the smaller window size. This can increase the response value of some pixels, and more points would be considered as corners.

■ **How does threshold affect the result?**
When the threshold becomes larger, fewer points would be reconized as corners since only the points with a higher corner response value will be considered as corners. Although the number of the false corners will decrease, some real corners will also be eliminated. Conversely, when the threshold becomes smaller, it is more likely to detect some weak corners. However, some points may also be mistakenly detected as corners.

Q3.SIFT object recognition

1.Implementation

step 1

```
#讀圖片且轉成灰階  要跟 code 放在同一個資料夾內
img01 = cv2.imread('hw1-3-1.jpg',cv2.IMREAD_GRAYSCALE)
img02 = cv2.imread('hw1-3-2.jpg',cv2.IMREAD_GRAYSCALE)
#print(img01.shape)
#print(img02.shape)

#建立 sift
sift = cv2.SIFT_create(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04,
edgeThreshold=10, sigma=1.6)

#取得特徵點及 descriptor
keypoints01, descriptors01 = sift.detectAndCompute(img01, None)
keypoints02, descriptors02 = sift.detectAndCompute(img02, None)
n_keypoints01 = len(keypoints01)
n_keypoints02 = len(keypoints02)
# print(n_keypoints01)
# print(n_keypoints02)
```

◆ Read the images in grayscale, and store them as "img01" and "img02."
◆ Use cv2.SIFT_create() to create SIFT object and get the keypoints and their descriptors by sift.detectAndCompute().
◆ The returned keypoints01 is a list with keypoint objects on the "image01" and the descriptors01 is a NumPy array with the shape of (number of keypoints01)* 128.
◆ Get the number of keypoints in "image01" and "image02" by len() and store them as n_keypoints01 and n_keypoints02.

step 2

```
#在圖 2 中找兩個跟 descriptor 最相近的 keypoint
min_1 = {}
min_2 = {}
match = {}
```

```
for k1, A in zip(range(n_keypoints01), descriptors01):
    min_dist1 = float('inf')
    min_b1 = None

    for k2, B in zip(range(n_keypoints02), descriptors02):
        dist = np.linalg.norm(A - B)
        if dist < min_dist1:
            min_dist1 = dist
            min_b1 = k2
    min_1[k1] = min_b1

    min_dist2 = float('inf')
    min_b2 = None
    for k2, B in zip(range(n_keypoints02), descriptors02):
        if k2 != min_b1:
            dist = np.linalg.norm(A - B)
            if dist < min_dist2:
                min_dist2 = dist
                min_b2 = k2
    min_2[k1] = min_b2
    #只接受 d(v,b1)/d(v,d2)<1/2 的點
    if (min_dist1/min_dist2)<0.5:
        match[k1] = [min_b1, min_dist1]
```

◆ Construct dictionary "min_1" to store the closest keypoints in image02 for each keypoint in image01, and dictionary "min_2" to store the second closest keypoints in image02.

◆ The distance between the keypoint in image01 and the keypoint in image02 is defined by the distance of their descriptors. The smaller their distance is, the more similar their features are. Use np.linalg.norm() to compute their distance.

◆ After getting the two closest keypoints in image02, I compute the value of min_dist1/min_dist2. Only when the value is larger than 1/2, I'd consider it as a real keypoint that matches the keypoint in image01 and put it into "match" dictionary.

◆ Since if the value is smaller than 1/2, it means the best and the second best keypoints in image02 can be similar. In this case, there may exist ambiguity for recognizing match-point, so the keypoint b1 cannot be considered as the real match for the keypoint in image01.

◆ In the dictionary "match", the keys are the keypoints of image01 and the value of each key is a list consists of the corresponding keypoint in image02 and the distance of the descriptors of the two keypoints.

step 3

```
#sort the match in ascending order
match_s = {k: v for k, v in sorted(match.items(), key=lambda item: [item[1][1]])}
print(match_s)

#new dictionary for at most 20 matches of each objects
match_20 = {}
front = 0.33*img02.shape[0]
medium = 0.66*img02.shape[0]
bottom = img02.shape[0]
print(img02.shape[0])
A = B = C = 0
for k1, (k2, d) in match_s.items():
    y = keypoints02[k2].pt[1]
    if y <= front and A<20:
        match_20[k1] = (k2,y)
        A+=1
    elif front<y<=medium  and B<20:
        match_20[k1] = (k2,y)
        B+=1
    elif medium<y<=bottom and C<20:
        match_20[k1] = (k2,y)
        C+=1

#print(f"A={A}, B={B}, C={C}")
#print(match_20)
#print(len(match_20))
```

◆ Sort the "match" dictionary in ascending order according to the distance of the descriptors of the two keypoints. Since the smaller the distance is, the more likely the matching of the two keypoints is correct.
◆ Since the first object in image02 is placed at the top 1/3 of the image, and the second object is in the middle while the third object is at the bottom, I compute the corresponding values of the height of image02.

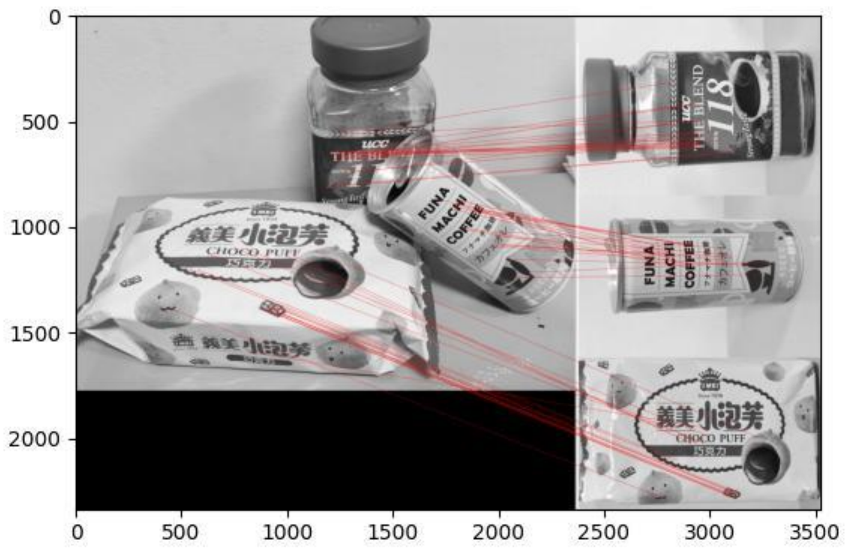
◆ Use A to store how many matches are selected for the object on the top; B for the object in the middle and C for the object at the bottom of the image02.
◆ Then take the matches from "match_s" one by one. If the k2 of the match appears in top 1/3 of image02 and the number of matches accepted in the same region is still lower than 20, it will be put into the dictionary "match_20" and A will be incremented. The other groups follow the same steps.
◆ With this method, at most the top 20 matches of each object will be shown on the final image.

step 4

```
#turn into 3 channels
img01_bgr= cv2.cvtColor(img01, cv2.COLOR_GRAY2BGR)
print(img01_bgr.shape)
img02_bgr = cv2.cvtColor(img02, cv2.COLOR_GRAY2BGR)
print(img02_bgr.shape)

#Draw the matches
matches = [[cv2.DMatch(k, v[0], 0) for k, v in match_20.items()]]
img_match = cv2.drawMatchesKnn(img01_bgr, keypoints01, img02_bgr, keypoints02,
matches,None, matchColor=(0, 0,
255),flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
# Convert BGR image to RGB
img_match_rgb = cv2.cvtColor(img_match, cv2.COLOR_BGR2RGB)
print(img_match_rgb.shape)

plt.imshow(img_match_rgb)
plt.savefig('hw1-3c.jpg')
plt.show()
```

◆ Convert image01 and image02 into 3-channel images by cv2.cvtColor(img0x, cv2.COLOR_GRAY2BGR) so that we can draw red matches on them.
◆ Create a nested-list "matches" in which store DMatch objects of the indices of each pair of matched keypoints. This would be the fifth parameter of cv2.drawMatchesKnn().
◆ Use cv2.drawMatchesKnn() to draw the matching on the two images and the new image is called img_match.
◆ Convert img_match into an image with channels in RGB order by

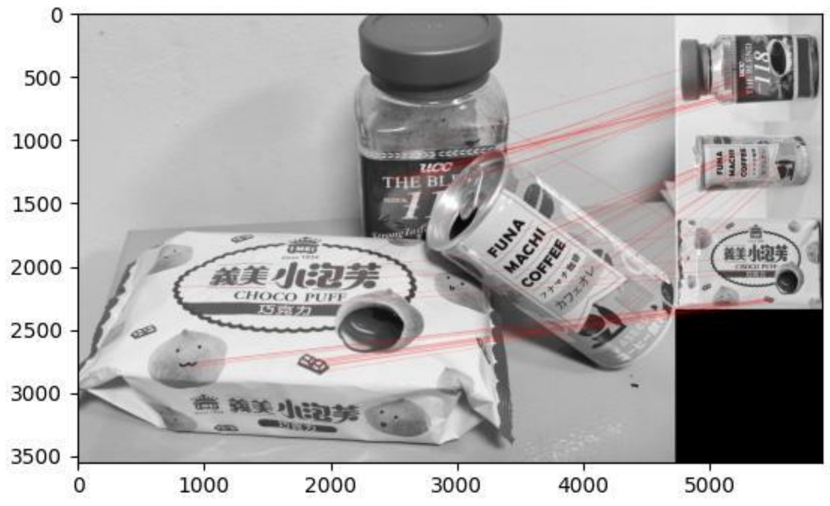
cv2.cvtColor(img_match, cv2.COLOR_BGR2RGB) so that it can be shown by plt.imshow.



step 5 (Scale the scene image to 2.0x)

```
# Resize image01
img01_2x = cv2.resize(img01, ((img01.shape[1])*2, (img01.shape[0])*2))
# print(img01_2x.shape)
# print(img02.shape)
```

◆ Resize “img01” with its height and width becoming twice as large and redo the previous steps. The resulting image is shown below.



2.Discussion Section

■ **Discuss the cases of mis-matching in the point correspondences in (c.) or (d.).**
In (c.), mismatches occur on the top object of the right image due to repeated small patterns on the label. Furthermore, for letters appearing more than once in the images, there are also some mismatched lines. This may happen because those similar patterns of the letters would have similar descriptors, making them prone to be mismatched.

■ **Discuss the difference between the results before and after the scale.**
◆ After scaling the left image, the number of mismatches increases. Some keypoints are even matched to the different objects whereas before scaling, keypoints were only matched on the same object.
◆ This result may happen because after scaling, the neighborhood of each pixel may not remain same features. To be precicely, it may lose some details of the features around the keypoints after scaling. Meanwhile, the descriptor of each pixel on the left image would also change, making them more challenging to be matched with the original keypoints. Consequently, an increasing number of mismatches occurs.