Q1 Histogram Equalization

1. Implementation

Step 1 of (a.)

- Code

```python
def equalizeHistogram(img):

    # caculate histogram
    histogram, bins = np.histogram(img.flatten(), bins=256, range=[0,256])

    # caculate cumulative histogram
    cdf = np.cumsum(histogram)

    # caculate normalized cumulative histogram
    cdf_norm = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())

    #apply cumulative distribution function transformation
    img_eq = cdf_norm[img].astype(np.uint8)

    return histogram, img_eq
```

- Explaination of code

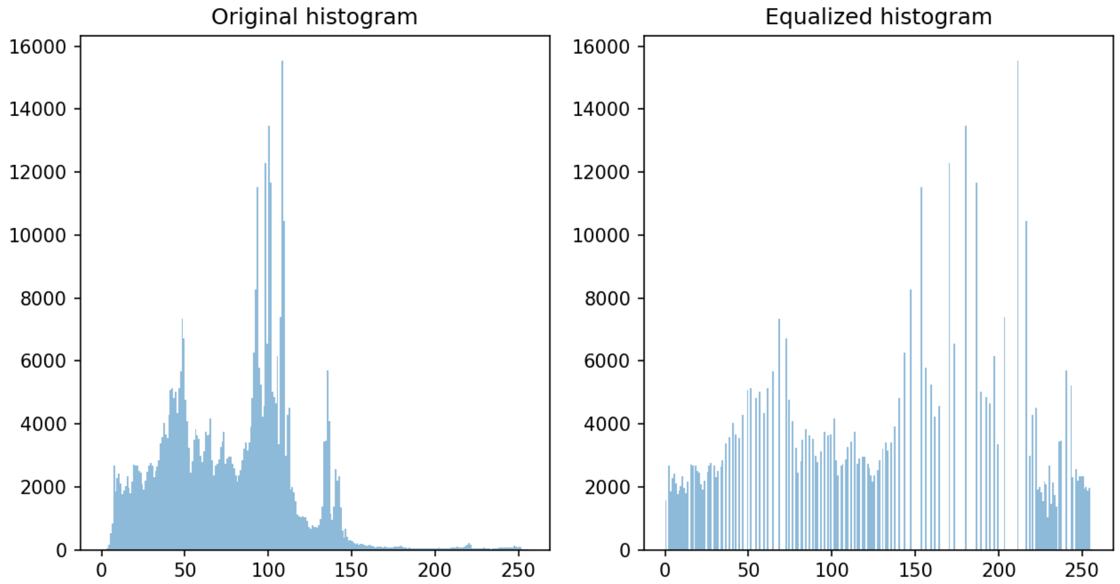
利用 np.histogram 去計算 input image 的 histogram，再將 histogram 利用 np.cunsum 去計算 cumulative histogram。最後則是將 input image mapping 到 normailzed 的 cumulative histogram 完成 histogram equalization。

Step 2 of (b.)

- Result image

Step 3 of (c.)

- Result image



Q2 Harris Corner Detector

1. Implementation of (a.)

Step 1 of (a.)(i)

- Code

```
def gaussian_blur(img, ksize, sigma):

    # generate 1D Gaussian kernel
    kernel_1d = cv2.getGaussianKernel(ksize, sigma)
    # get 2D Gaussian kernel by multiplying two 1D Gaussian kernel
    kernel_2d = np.outer(kernel_1d, kernel_1d)
    # normalized Gaussian kernel
    norm_kernel = kernel_2d / kernel_2d.sum()
    # apply normalized 2D Gaussian kenrel to image
    filtered_img = cv2.filter2D(img, -1, norm_kernel)

    return filtered_img
```

- Explanation of code

先利用 cv2.getGaussianKernel 生成 1D Gaussian kernel，2D Gaussian 則是取兩個 1D Gaussian kernel 的乘積。最後則是利用 cv2.filter2D 將 normalized 2D Gaussian kernel 應用到 image 上做 Guassian blur。

- Result image

Result after Gaussian blur

Step 2 of (a.)(ii)

- Code

```
def sobel_x(img_blur):
    # create Sobel filter to detect horizontal edges
    filter = np.array([[-1,0,1],
                       [-2,0,2],
                       [-1,0,1]])
    # apply filter to the ouput image of Gaussain_blur function
    gradient_x = cv2.filter2D(img_blur, cv2.CV_64F, filter)
    return gradient_x
```

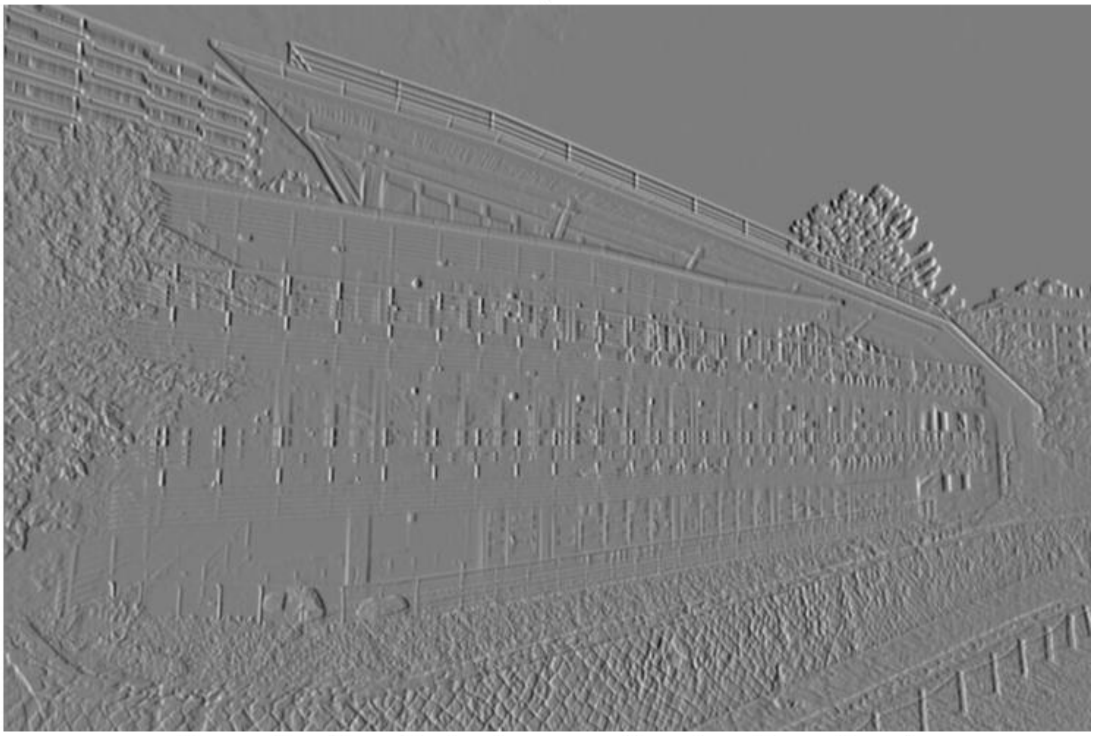
```
def sobel_y(img_blur):
    # create Sobel filter to detect vertical edges
    filter = np.array([[-1,-2,-1],
                       [0,0,0],
                       [1,2,1]])
    # apply filter to the ouput image of Gaussain_blur function
    gradient_y = cv2.filter2D(img_blur, cv2.CV_64F, filter)
    return gradient_y
```

- Explaination of code

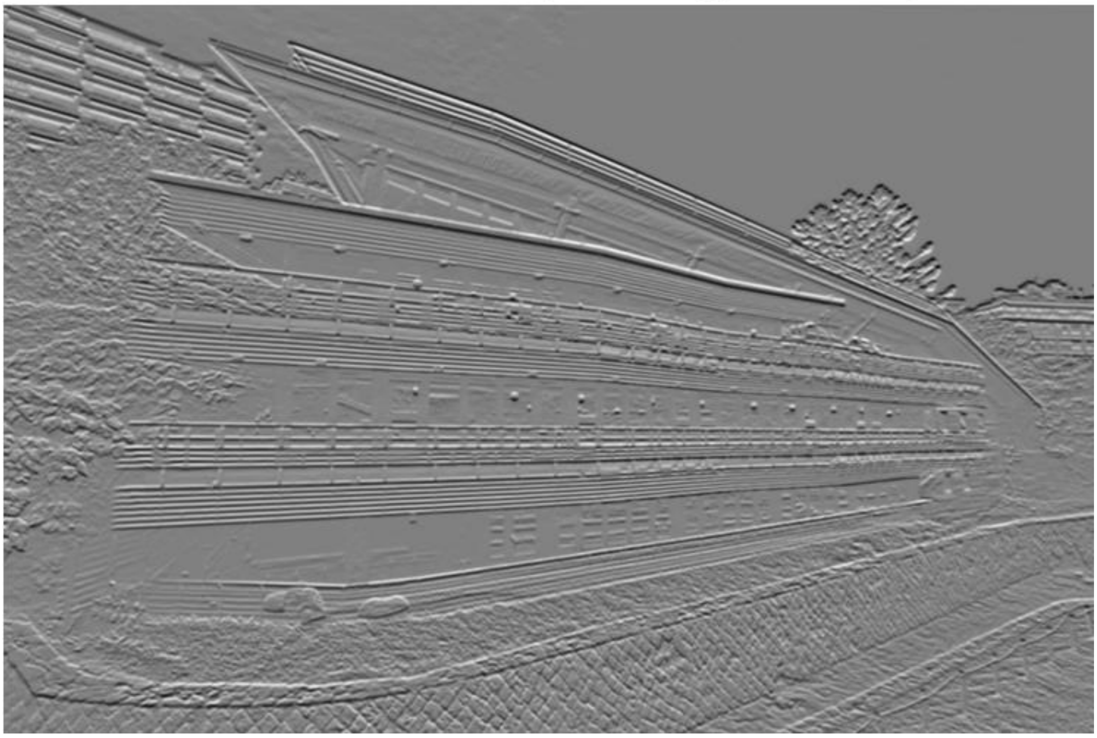
利用不同 Sobel filter 的設定，對 Gaussian blur function 輸出 image 做水平和垂直方向的邊緣偵測。

- Result images

Result after Sobel operation(x direction)



Result after Sobel operation(y direction)



Step 3 of (a.)(iii)

- Code

```
def corner_response(dx, dy, ksize, threshold):

    print('corner response...')
    k= 0.04
    height, width = dx.shape
    offset = ksize // 2

    # Caculate the structure matrix element
    dxx = gaussian_blur(dx * dx, ksize, 3)
    dyy = gaussian_blur(dy * dy, ksize, 3)
    dxy = gaussian_blur(dx * dy, ksize, 3)

    # calculate the harris response matrix R
    det = dxx * dyy - dxy ** 2  # determinant
    tr = dxx + dyy  # trace
    R = det - k * tr ** 2   # harris response matrix R

    img_corner = np.zeros_like(dx)
    #recaculate threshild because the harris response matrix is not normalized
    threshold = threshold * R.max()
    # thresholding : loop through the harris response matrix to check which position is larger than threshold
    for y in range (offset, height-offset):
        for x in range(offset, width-offset):
            # if it is large than threshold, then set the value to 255(white) of the same position in img_corner array
            if R[y, x] > threshold:
                img_corner[y, x] = 255

    return R, img_corner
```

- Explanation of code

這部分主要是將 Sobel_x function 輸出 array(dx)的平方(dx*dx)、Sobel_y function 輸出 array(dy)的平方(dy*dy)以及兩個 array 乘積(dx*dy)未入到 gaussian blur function 中分別取得 dxx, dyy, dxy 三個 structure matrix elements。再去計算 harris response matrix 的 determinant 和 trace，進而取得 harris response matrix。最後則是去檢查 harris response matrix 每個 position 的值，若值大於 threshold 則設成 255(白色)，其餘的設成 0(黑色)，得到最後結果影像。

- Result image

Harris response

Step 4 of (a.)(iv)

- Code

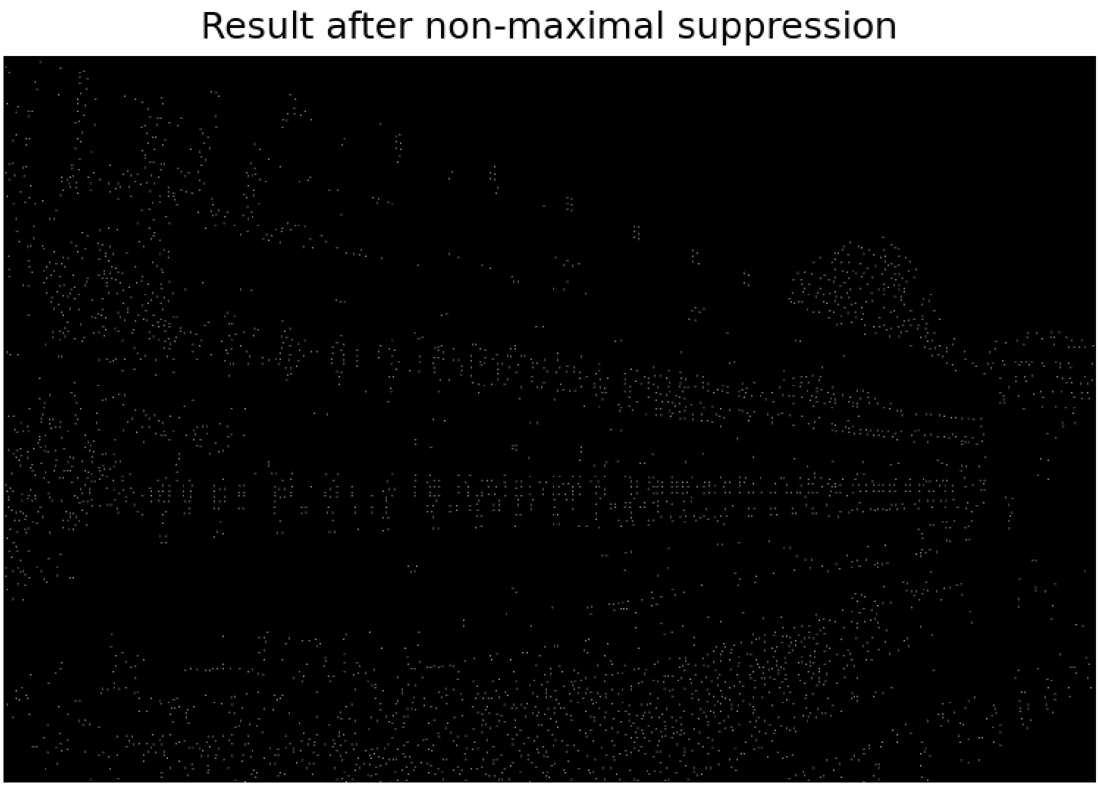
```
def nms(R, nms_window, threshold):

    print('nms...')
    height, width = R.shape
    offset = nms_window // 2
    img_nms = np.zeros_like(R)
    threshold = threshold * R.max()
    nms_pos = []
    # loop through the harris response matrix with a 5x5 window
    for y in range(offset, height - offset):
        for x in range(offset, width - offset):
            # local_max is the maximum number in the 5x5 window
            local_max = np.max(R[y - offset:y + offset + 1, x-offset : x+offset+1])
            # check whather the (y,x) positon is the local_max and larger than the threshld or not
            if R[y, x] == local_max and R[y, x] > threshold:
                # if the condition holds, then set the corresponding poisiton in img_mns array to 255(white)
                img_nms[y,x] = 255
                nms_pos.append((y,x))    # record the nms coordinate
    return nms_pos, img_nms
```

- Explanation of code

根據上一題所取得的 harris response matrix，利用 5x5 的 window 先找出，matrix 中每個位置所屬的 window 中最大值(local max)。再去比較說當前這個位置是否為 local max 並且有沒有大於 threshold。若都符合，則將值設為 255(白色)。

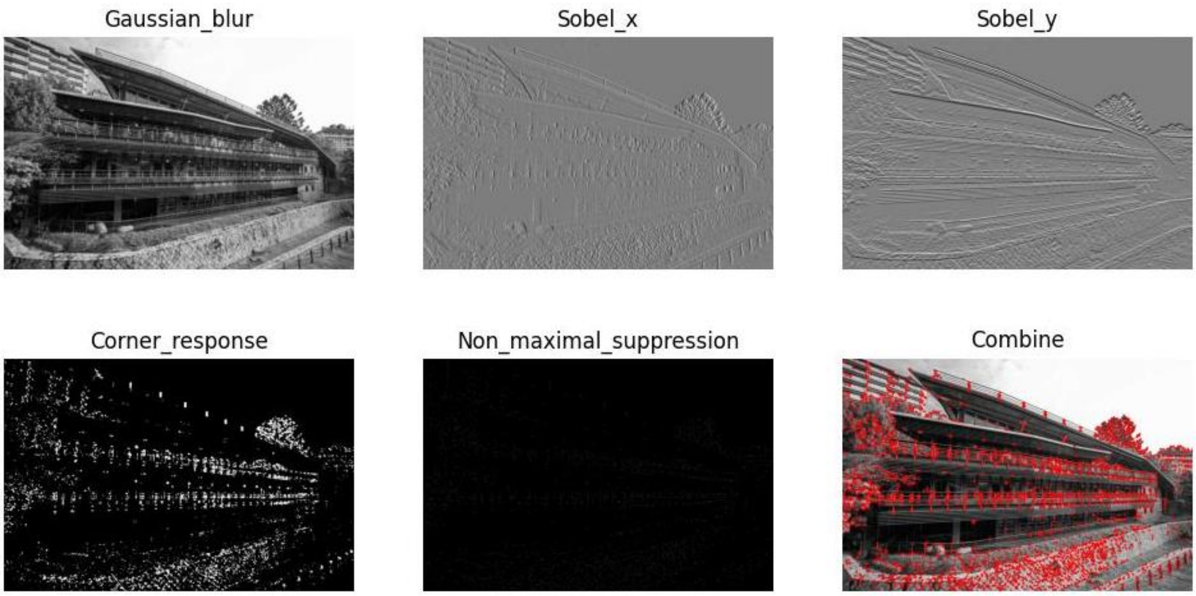
- Result image



1. Implementation of (b.)

Step 1 of (b.)(i)

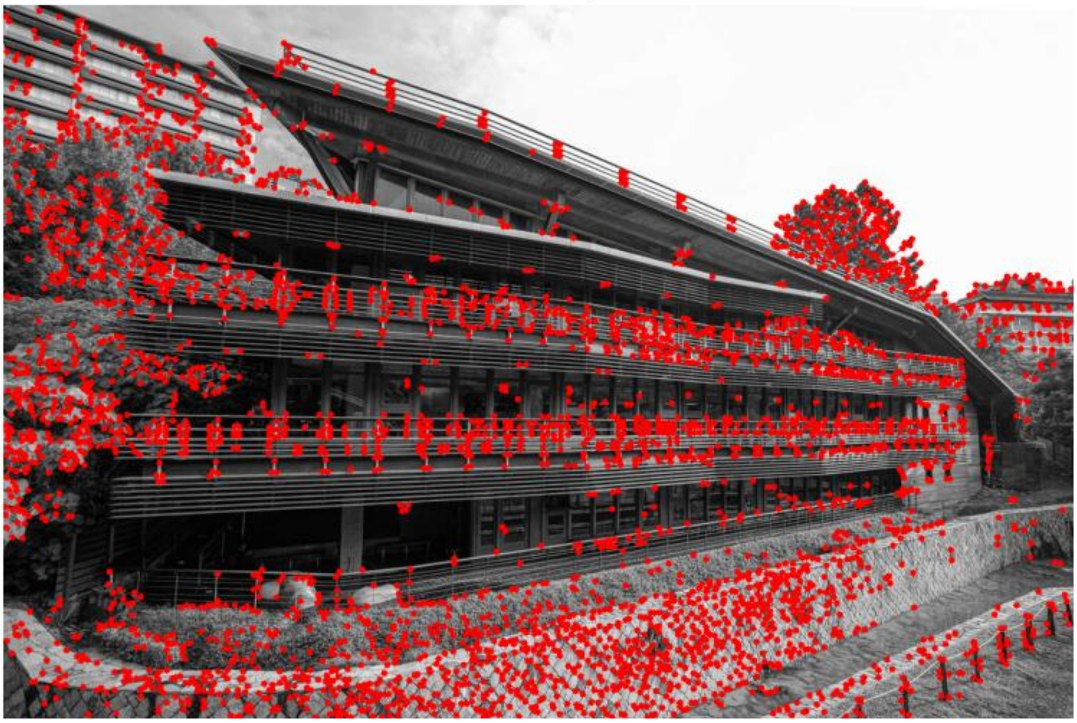
- Result image



Step 2 of (b.)(ii)

- Result image

Final output

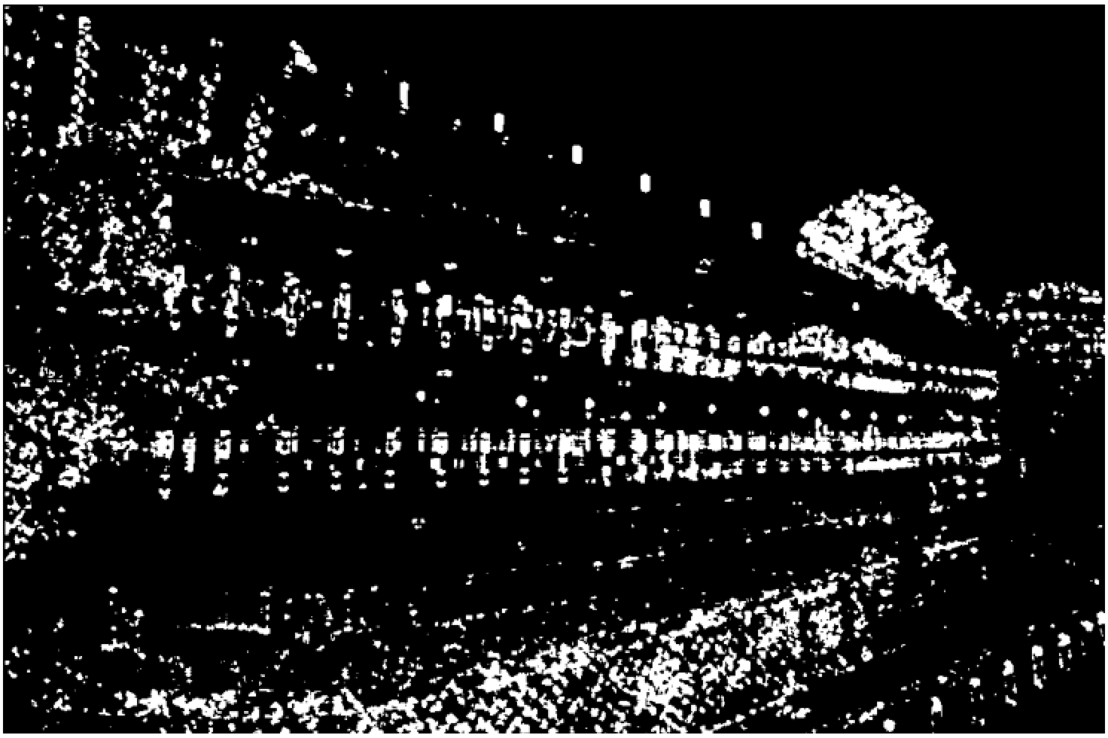


1. Implementation of (c.)

Step 1 of (c.)(i)

- Result images

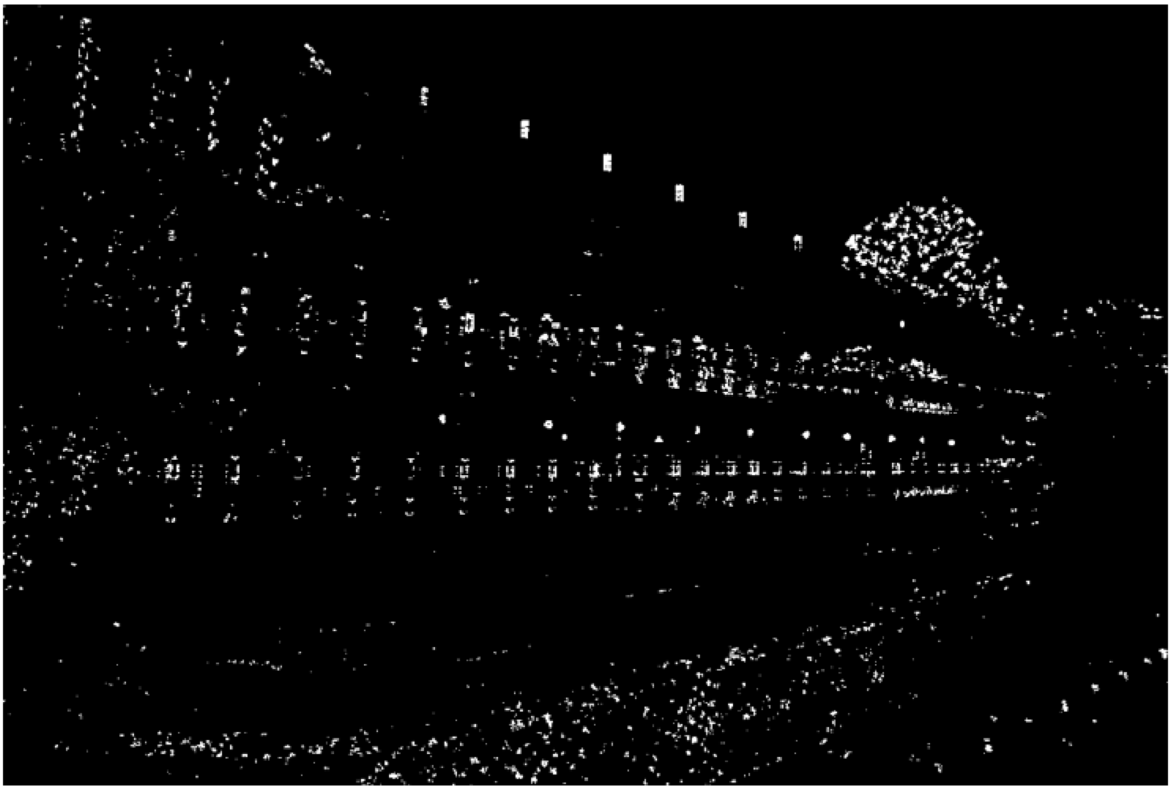
ksize=5



Step 2 of (c.)(ii)

- Result images

threshold=0.03



2. Discussion section of (d.)

- Discuss about how the window size affect the result

Window size 變大，根據結果影像顯 harris response 會變多。當 window size 增加，在做 gaussian blur 的過程可以讓 input image 亮度更平均，降低 noisy 的 noisy 的影響。相對來說 harris response 的點就會比較好偵測，所以數量變多。

- Discuss about how the threshold affect the result

原本 threshold=0.01，這裡將 threshold 調大至 0.03，導致 harris response 數量減少。因為在其他條件不變的情況下，harris response matrix 也不會變。所以滿足較大的 threshold 點相對就會比較少。

Q3 SIFT object recognition

1. Implementation

Step 1 of (a.)

- Code

```
def sift_extractor(img):
    # initializes the SIFT
    sift = cv2.SIFT_create()
    # compute keypoints and descriptors for input image
    keypoints, descriptors = sift.detectAndCompute(img, None)
    return keypoints, descriptors
```

- Explanation of code

先利用 cv2.SIFT_create 初始創建 sift，在利用 detectAndCompute 對 input image 計算 keypoint 和 descriptor。

Step 2 of (b.)

- Code

```
def feature_matching(keypoint_target, desc_source, desc_target):

    print('matching...')
    matches_ob1 = []
    matches_ob2 = []
    matches_ob3 = []
    ratio_threshold = 0.7

    # loop through all the descriptors of source image to find the best match in target image
    for i, desc1 in enumerate(desc_source):
        best_match = None

        # find distances between desc1 and all the descriptors of target image
        distances = np.linalg.norm(desc1 - desc_target, axis=1)

        # ratio testing
        indices = list(range(len(distances)))
        # sort the distances in order to find the best(sorted_indices[0])and the second best(sorted_indices[1])
        sorted_indices = sorted(indices, key=lambda i: distances[i])
        best_distance = distances[sorted_indices[0]]
        second_best_distance = distances[sorted_indices[1]]
        # if the best distance is smaller than 0.7*second best distance, then we consider it as a good match and store it
        if best_distance < ratio_threshold * second_best_distance:
            # i: index if source keypoint, sorted_indices[0]: index of target keypoint, best_distance: distance between two keypoint
            best_match = cv2.DMatch(i, sorted_indices[0], best_distance)
```

- Explanation of code

利用 for loop 針對每個 source descriptor 和所有 target descriptor 計算距離。再利用 sort 找出最短的兩個距離，若最短的距離又小於 threshold *第二短距離，視為 good match。根據 target descriptor y 座標大小判定屬於哪一個物體，分別將 good match 加入到不同的 list，將三個 list 根據距離大小做 sorting，每個分別只取前 20 好的。

Step 3 of (c.)

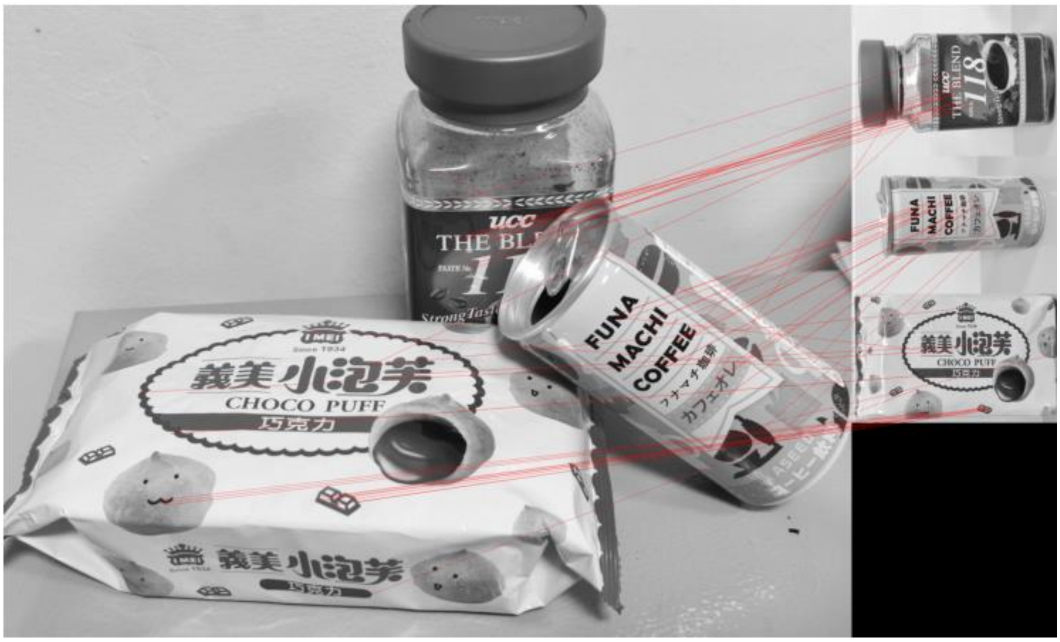
- Result image

Original

Step 4 of (d.)

- Result image

Scaled



2. Discussion section of (e.)

- Discuss about the mismatching point in (c.)

  在第(c.)小題中，地案個物體和第三個物體會也 mismatch 的情形出現。透過觀察，我認為因為罐子和泡芙本身顏色較相似，所以容易會出現罐子對應到泡芙以及泡芙對應到罐子的情況出現。

- Discuss about the difference between results before and after the scale.

  1.) 第一個物體經過 scaling 之後，會有 mismatch 的情形出現。

  我認為是因為物體二上的字母被放大又和物體二包裝顏色相似，所以容易找到字母上。

  2.) 第二個物體經過 scaling 之後，matching 的表現變好。同時相同的物體相同的點可以正確對應。我認為是因為罐子上的字母被放大再加上顏色較深，所以可以很好的對應。

3.) 第三個物體經過 scaling 之後，matching 的表現變好。同時相同的物體相同的點大致上可以正確對應。我認為可能因為放大之後泡芙整體區域變大，可以更好的對應。