Lab2

Section1 Harris corner detection

1.Gradient Calculation

$$I_x = \frac{I(x+1,y) - I(x-1,y)}{2} \tag{1}$$

$$I_y = \frac{I(x,y+1) - I(x,y-1)}{2} \tag{2}$$

According to $(1)(2)$, I built the kernels below to calculate the gradients in x and y direction.

```
    # gradient kernel
    x_grad_kernel = np.array([[1, 0, -1]]) / 2
    y_grad_kernel = np.array([[1], [0], [-1]]) / 2

    # gradient calculation
    gradient_Ix = signal.convolve2d(img, x_grad_kernel, mode='same',
boundary='symm')
    gradient_Iy = signal.convolve2d(img, y_grad_kernel, mode='same',
boundary='symm')
    gradient_Ixy = gradient_Ix * gradient_Iy
    gradient_Ix_square = gradient_Ix**2
    gradient_Iy_square = gradient_Iy**2
```

One thing need to notice is that when padding the images, we shouldn't use default value **0**, because this operation would create key point in the edge of image. I choose symmetric padding as the code above.

2.Gradient Blurring

To avoid the influence of noise, I used gaussian kernel to smooth the gradients.

```
    blurred_gradient_Ix_square = cv2.GaussianBlur(gradient_Ix_square, ksize=[3,
3], sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
    blurred_gradient_Iy_square = cv2.GaussianBlur(gradient_Iy_square, ksize=[3,
3], sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
    blurred_gradient_Ixy = cv2.GaussianBlur(gradient_Ixy, ksize=[3, 3],
sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
```

Actually, we can smooth the original image first, and then to calculate the gradients. But because convolution operation is commutive, we can do the smoothing on the gradients after calculating.

3."M" Matrix Calculation

$$M = \sum_{(x,y)\in W} W(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix} \tag{3}$$

I choose the gaussian function($\sigma = 1$) as the window function, and built the gaussian kernel. And I choose the window size to be 3 which means the kernel size is also 3.

```
    # 2D gaussian kernel
    window_func_1D = cv2.getGaussianKernel(ksize=3, sigma=sigma)
    window_func_2D = window_func_1D * window_func_1D.transpose()
    # sum to build matrix "M"
    M_Ix_square = signal.convolve2d(blurred_gradient_Ix_square, window_func_2D,
mode='same', boundary='symm')
    M_Iy_square = signal.convolve2d(blurred_gradient_Iy_square, window_func_2D,
mode='same', boundary='symm')
    M_Ixy = signal.convolve2d(blurred_gradient_Ixy, window_func_2D, mode='same',
boundary='symm')
```

Then I get the "M" matrix for every windows with 3×3 size, has the following form:

$$M = \begin{bmatrix} \sum_{(x,y)\in W} W(x,y)I_x^2 & \sum_{(x,y)\in W} W(x,y)I_xI_y \\ \sum_{(x,y)\in W} W(x,y)I_yI_x & \sum_{(x,y)\in W} W(x,y)I_y^2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{4}$$

4.Response Calculation

$$C = \lambda 1\lambda 2 - k \times trace(M)^2 \tag{5}$$

$$\lambda 1\lambda 2 = |M| = ad - bc \tag{6}$$

$$\lambda 1 + \lambda 2 = trace(M) = a + d \tag{7}$$

Then $C = (ab - bc) - k \times (a + d)^2$:

```
    det_M = M_Ix_square * M_Iy_square - M_Ixy**2
    trace_M = M_Ix_square + M_Iy_square
    C = det_M - k * trace_M**2
```

5.Non-maximum suppression

Sometimes there are crowd of points satisfying the condition in a small field, but actually I don't need all of them, because any one of them with the field could include any of other points. So I need to remove other points with less response (although satisfying the condition).

Totally there are two conditions for any candidate points:

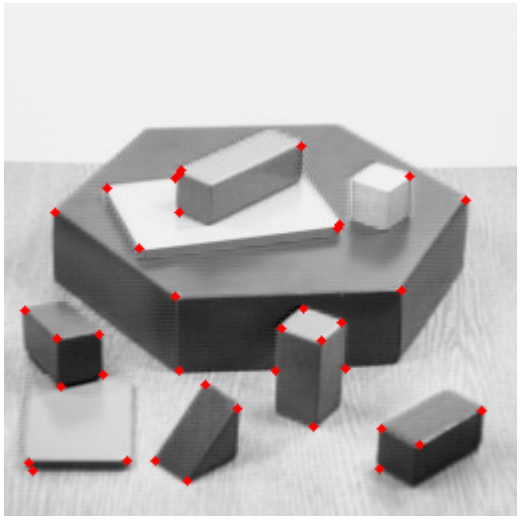
1. Response bigger than the thresh
2. Maximum in the field

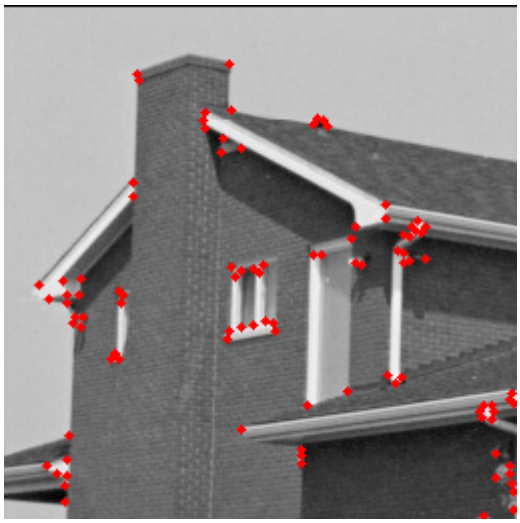
```
    max_filtered_C = ndimage.maximum_filter(C, size=3, mode='reflect')
    #(element > thresh) & element is max
    coordinate_y, coordinate_x = np.where((C > thresh) & (C == max_filtered_C))
```

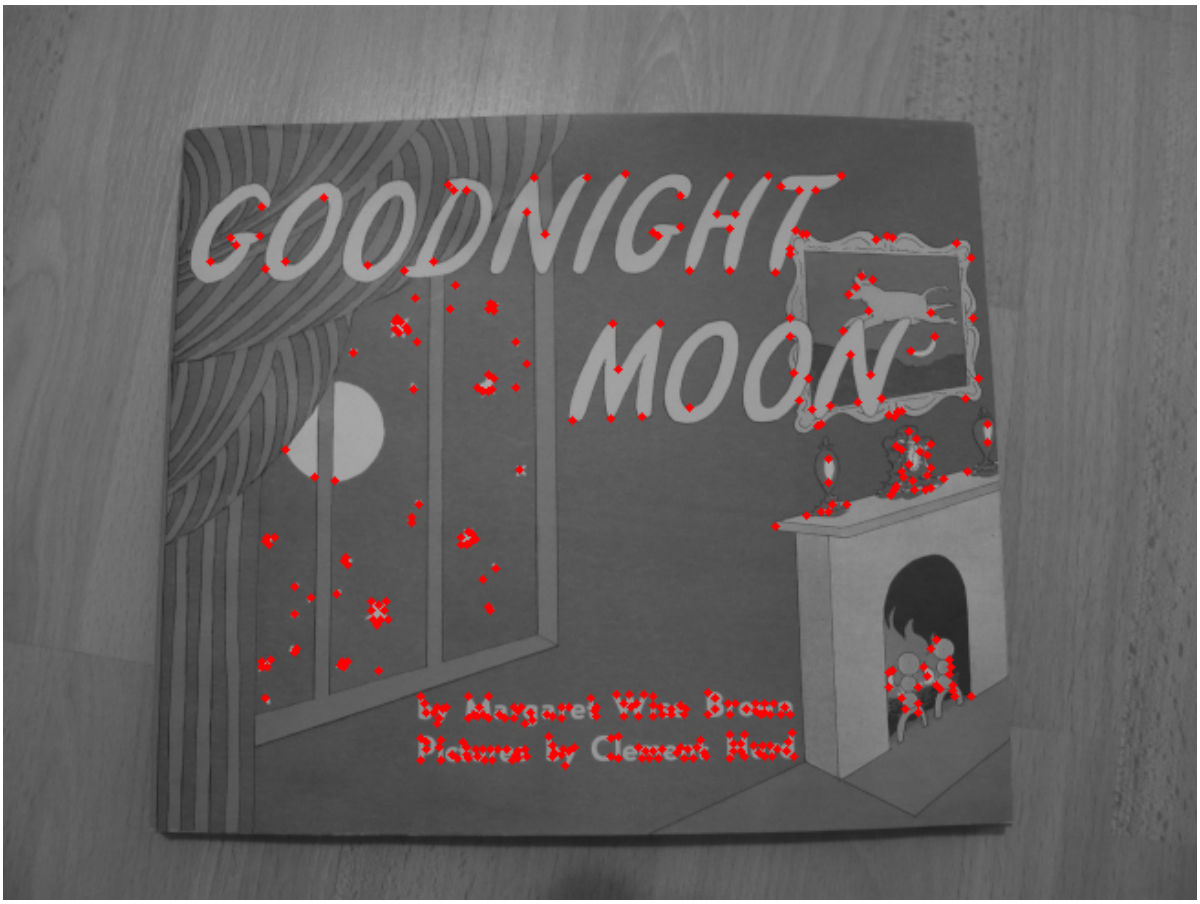
Then stack the coordinate to finish.

```
corners = np.stack((coordinate_x, coordinate_y), axis=-1)
```

At last, I'll show the results of the Harris corner detection:





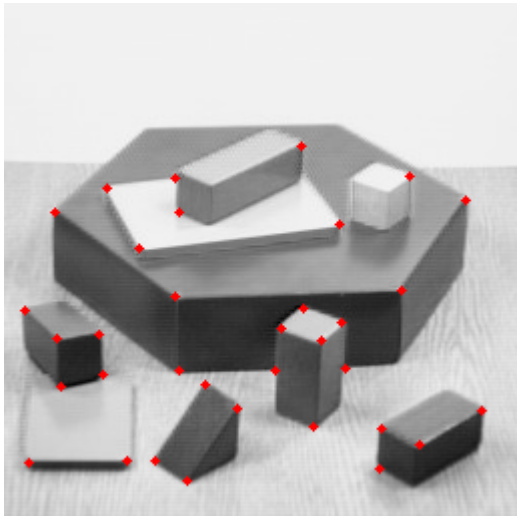

GOODNIGHT
MOON
by Margaret Wise Brown
Pictures by Clement Hurd

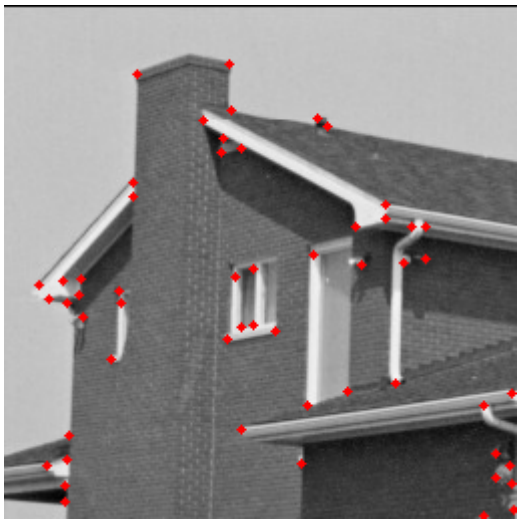
With the following outputs:

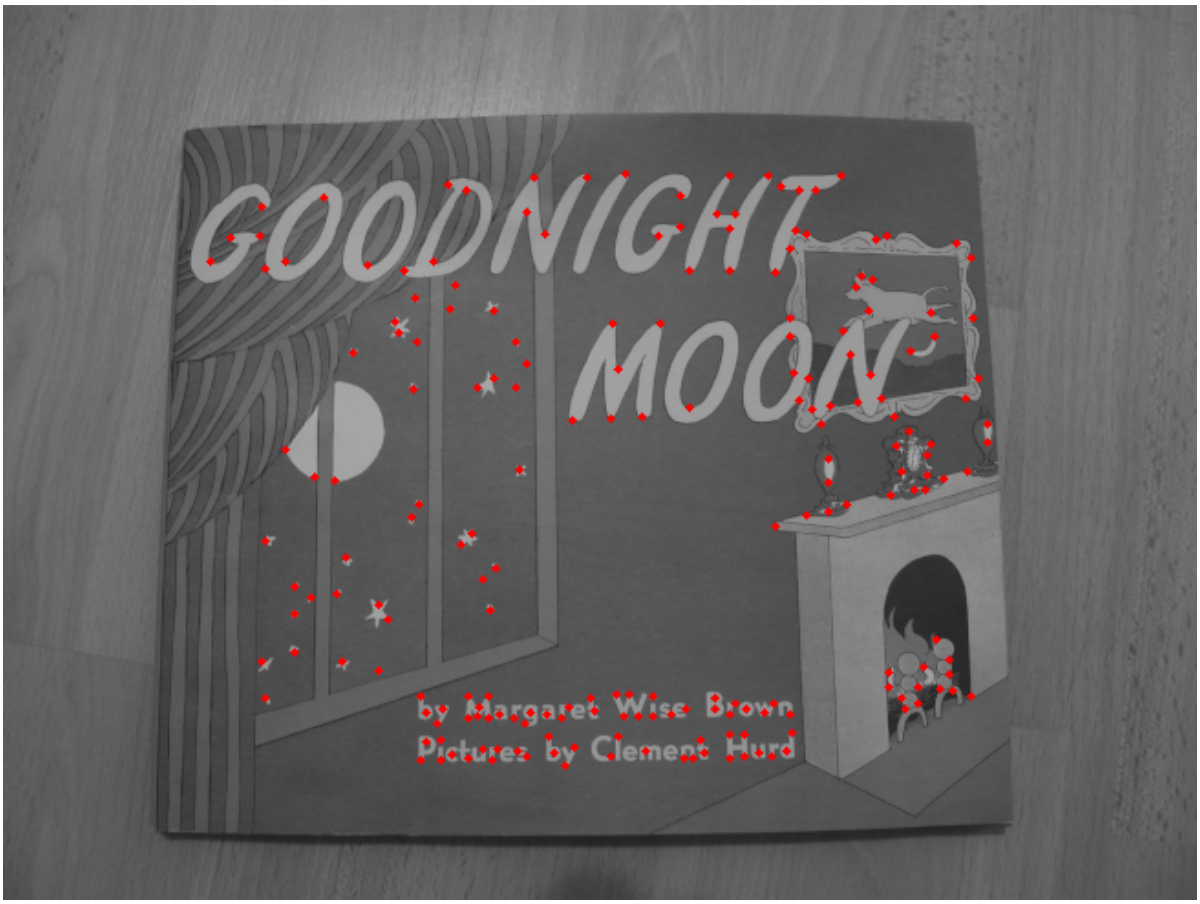
```
[LOG] Number of keypoints: 38. Writing keypoints visualization to
blocks_harris.png
[LOG] Number of keypoints: 99. Writing keypoints visualization to
house_harris.png
[LOG] Number of keypoints: 355. Writing keypoints visualization to I1_harris.png
[LOG] Number of keypoints: 372. Writing keypoints visualization to I2_harris.png
```

The results are fine, and the detector seems work well, and can detect lots of the corner. But I also noticed there are some corners it didn't detect out. I think it has relationship with the hyperparameters and the noise.

Besides, I found in last two images, the corner points are too much and too crowded, then I found that's because when I'm doing the NMS operation, the size is **3** (given in the template) which means too small and the suppression effect is not so good. After I changed it into **9**, the results are much better as below:






GOODNIGHT
MOON
by Margaret Wise Brown
Pictures by Clement Hurd

Section2 Description & Matching

1.Filter Key points & Extract Patches

Because it's possible that some key points are too close to the edge of the image, I need to remove them, and the filtered key points should satisfy:

```
    h, w = img.shape[0], img.shape[1]
    length = patch_size // 2
    # x + length < w && x - length >= 0 && y + length < h && y - length >= 0
    condition = (keypoints[:, 0] + length < w) & (keypoints[:, 0] - length >= 0)
\
                & (keypoints[:, 1] + length < h) & (keypoints[:, 1] - length >=
0)
```

Then extract the corresponding patch according to every key points.

2.Patch Description

$$SSD(p,q) = \sum_{i=1}^{N}(p_i - q_i)^2 \tag{8}$$

Thus, I can calculate the SSD (distance between two points), and the vector is just the descriptor:

```
    # distance calculation
    desc1 = np.expand_dims(desc1, axis=1)
    desc2 = np.expand_dims(desc2, axis=0)
    distances = ((desc1 - desc2)**2).sum(axis=2)
```

Then the SSD matrix is calculated, every element means the distance(similarity) between two key points.

3.Matching

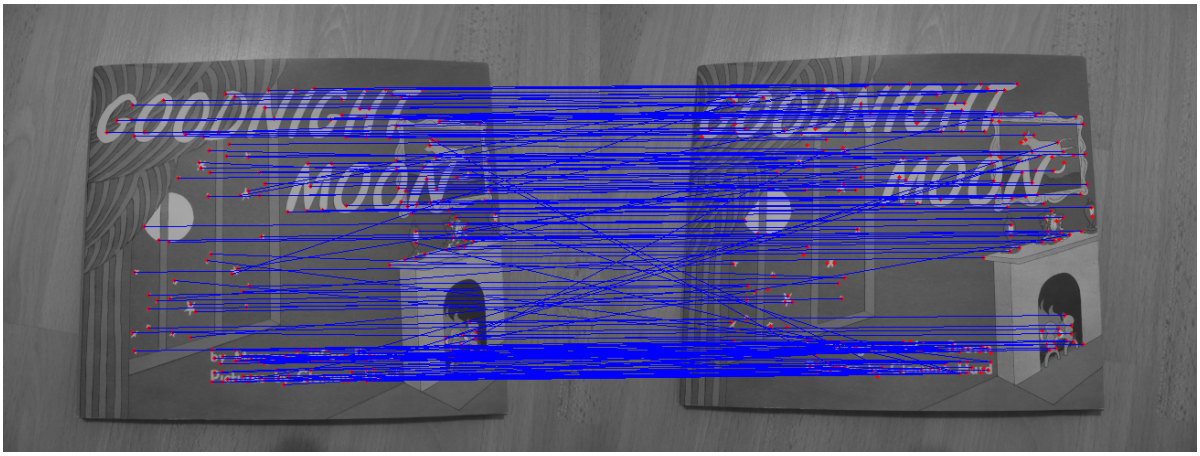
There are three kinds of matching method:

1.One Way

Only need to consider the distance from image1 to image2 which means for every key point in image1, I just need to find the points with the least distance for it and don't need to consider the opposite way:

```
indices_y = np.argmin(distances, axis=1)
indices_x = np.arange(q1)
matches = np.stack((indices_x, indices_y), axis=-1)
```

The result is like below:



With following output:

```
[LOG] Number of keypoints: 205. Writing keypoints visualization to I1_harris.png
[LOG] Number of keypoints: 221. Writing keypoints visualization to I2_harris.png
[LOG] Number of matches: 205. Writing matches visualization to match_ow.png
```

I found that almost every key points from image1 matched with a key points in image2, but some are not, which shows it's a kind of forcing match. And of course, the result is not good. As it must do the wrong matching. For example, there are some points in the top right of image1 matching with the bottom points in image2 (I call it cross matching), which is obviously wrong. And that's because one way matching only consider the direction from image1 to image2 and not the opposite ('forcing match' I've mentioned before).

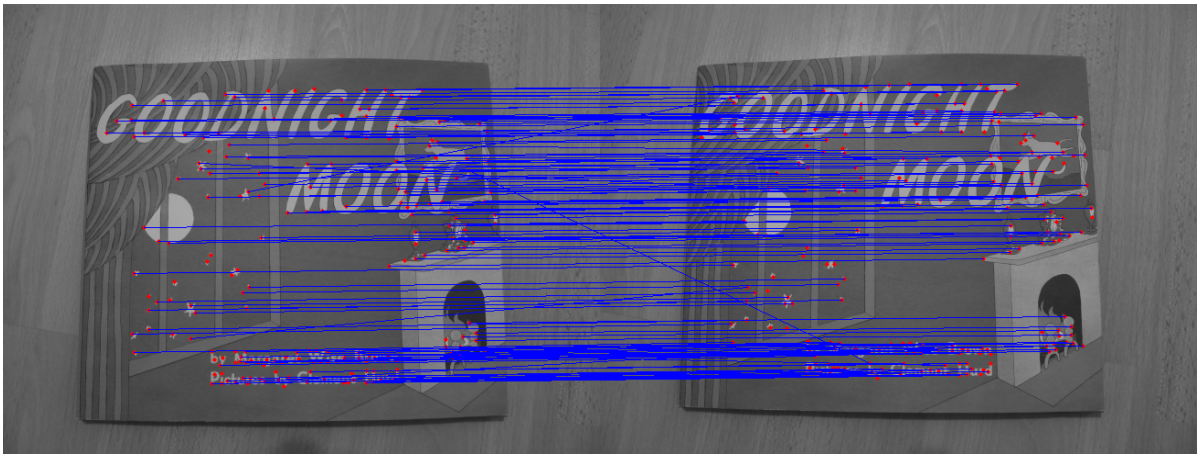
2.Mutual

Based on the One Way matching, image2 should check if the matching is also great from image2 to image1:

```
        indices_y = np.argmin(distances, axis=1)
        indices_x = np.arange(q1)
        #matches of q1 to q2
        matches = np.stack((indices_x, indices_y), axis=-1)
        #q1 to q2 values
        q1_to_q2_values = np.min(distances, axis=1)
        q2_to_q1_values = np.min(distances, axis=0)
        # to check if the value is the minimum
        mutual_indices = q1_to_q2_values == q2_to_q1_values[matches[:, 1]]
        matches = matches[mutual_indices]
```

First I do the same operation as the One Way matching, which means do the matching from image1 to image2, and then I go and check if the values are the minimum from the perspective of image2 to image1, and only keep those matching pairs satisfying two conditions.

Also the result and output:




```
[LOG] Number of keypoints: 205. Writing keypoints visualization to I1_harris.png
[LOG] Number of keypoints: 221. Writing keypoints visualization to I2_harris.png
[LOG] Number of matches: 145. Writing matches visualization to match_mutual.png
```

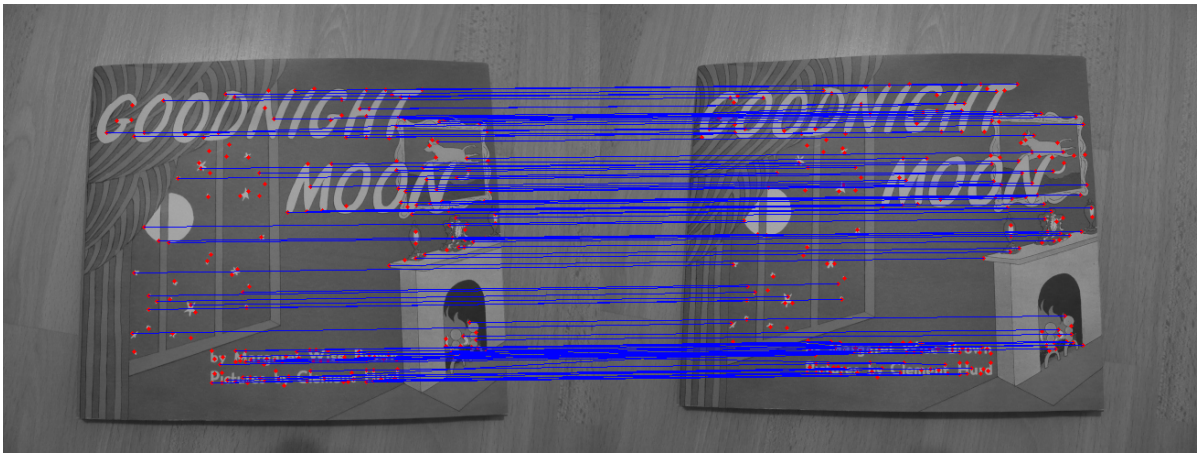
As is shown, the matching is than the One Way matching and obviously wrong matching (e.g., cross) is less as well, which means this method decreased the unreasonable matching.

3.Ratio

Again based on One Way matching, it asks the most matching pair and the second matching pair should have a difference which is control by the ratio. That means, only under the situation that a key point from image1 can distinguish the key point (similar candidate) in image2 (in matching perspective), a matching could happen. And the smaller the ratio is, the easier the distinguish should be (candidate key points are easier to judge), and the stricter the matching is,. Only satisfying the ratio can the matching happen:

```
        indices_y = np.argmin(distances, axis=1)
        indices_x = np.arange(q1)
        matches = np.stack((indices_x, indices_y), axis=-1)
        # find the smallest two
        smallest = np.partition(distances, 2, axis=1)
        ratio_indices = (smallest[:, 0] / smallest[:, 1]) < ratio_thresh
        matches = matches[ratio_indices]
```

Result and output:




```
[LOG] Number of keypoints: 205. Writing keypoints visualization to I1_harris.png
[LOG] Number of keypoints: 221. Writing keypoints visualization to I2_harris.png
[LOG] Number of matches: 108. Writing matches visualization to match_ratio.png
```

Obviously, the Mutual and Ratio Matching method should be better than the One Way Matching Method. But I found compared to the second method, some right matching is lost and some wrong matching is also corrected (because the Ratio Matching only care about the distinguishable degree between candidate key points instead of improving one way method from mutual perspective, so when the operation happens, it could lose the right matching and correct the wrong matching as well). I think it has relationship with the ratio, which's a hyperparameter we should adjust.