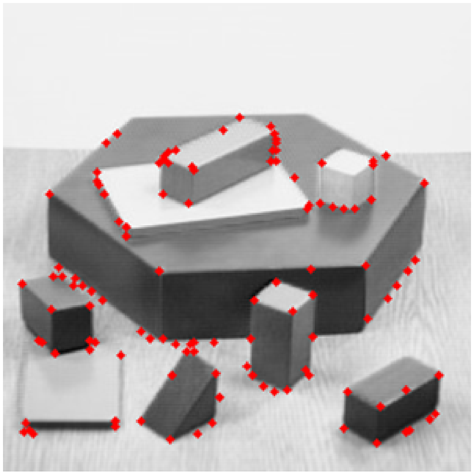
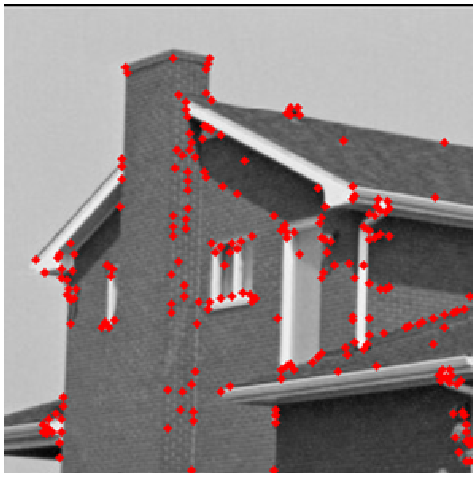
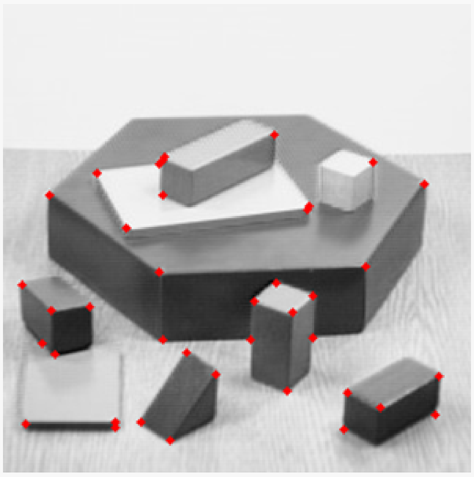
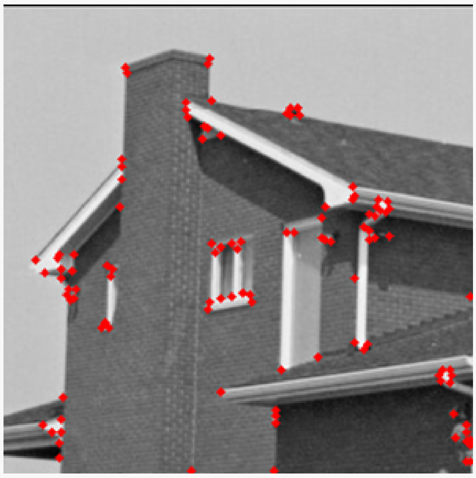
Report for Lab exercise 2 for Computer Vision

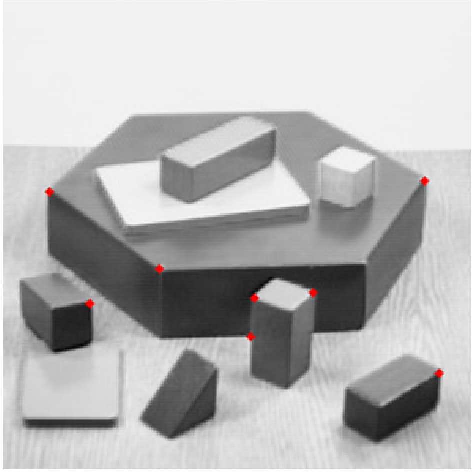
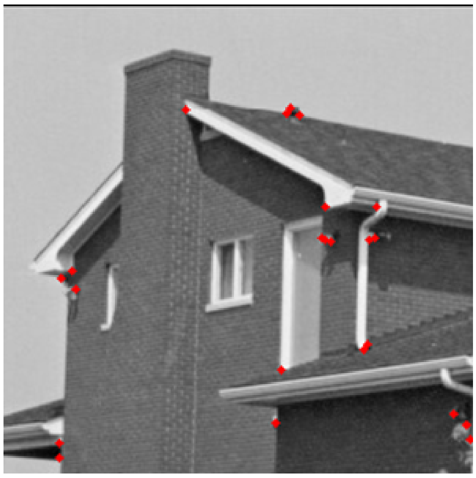
Detection

Parameter changes results

- Change of thresh from 1e-6 to 1e-4:

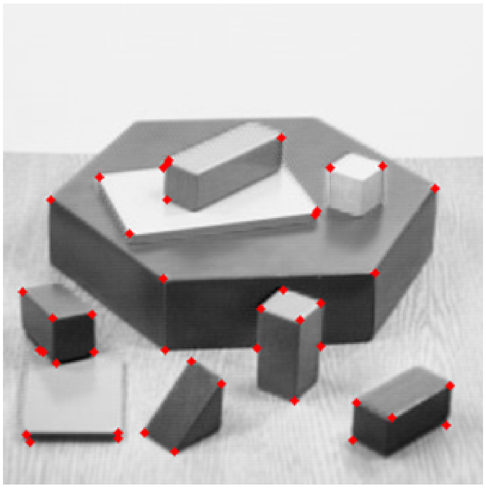
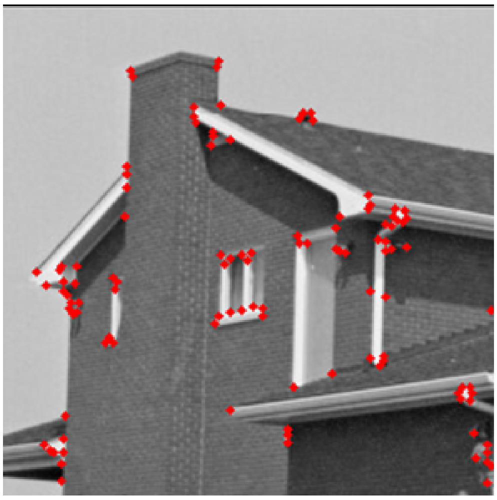
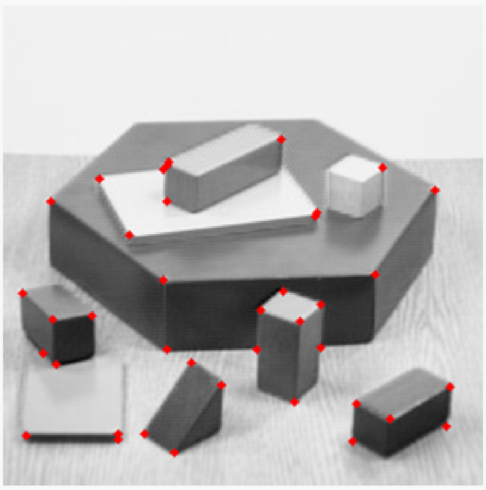
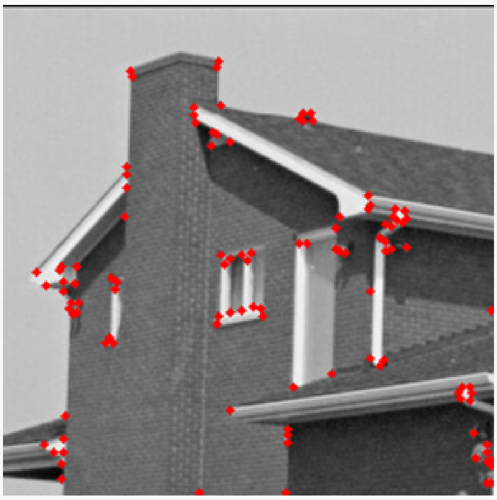
  Sigma = 1.0 K = 0.05

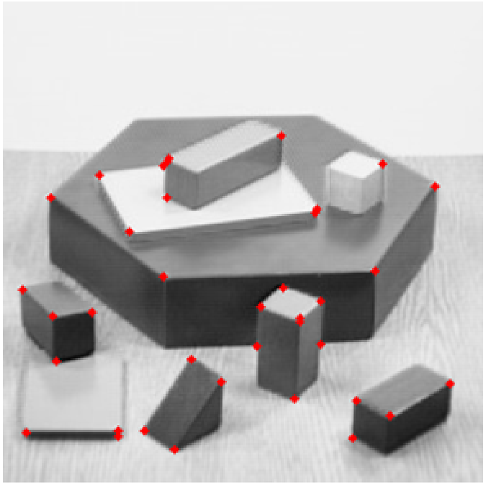
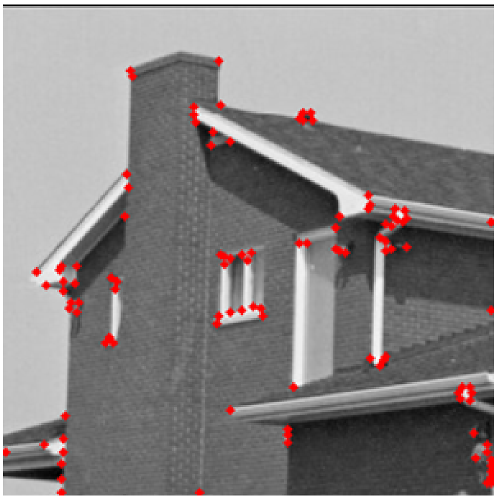
thresh	Blocks	House
1e-6		
1e-5		

thresh	Blocks	House
1e-4		

- Change of K from 0.04 to 0.06:

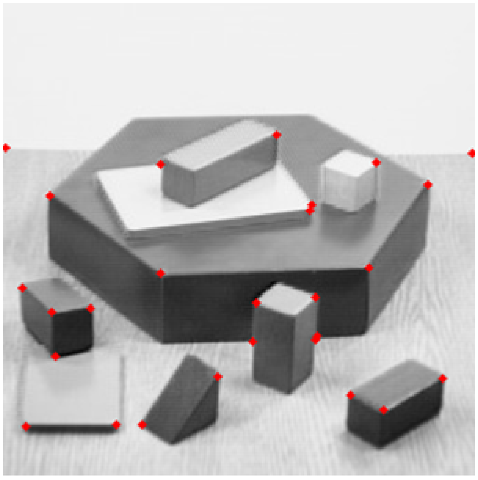
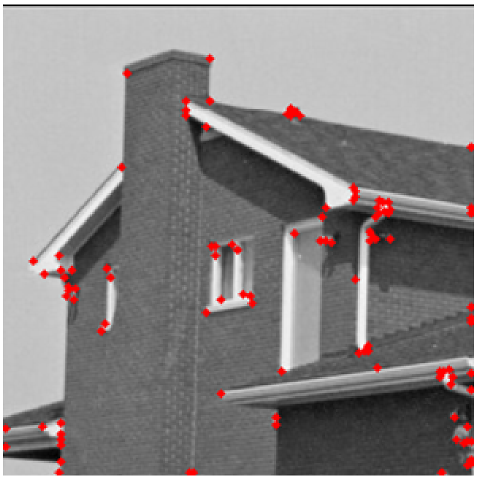
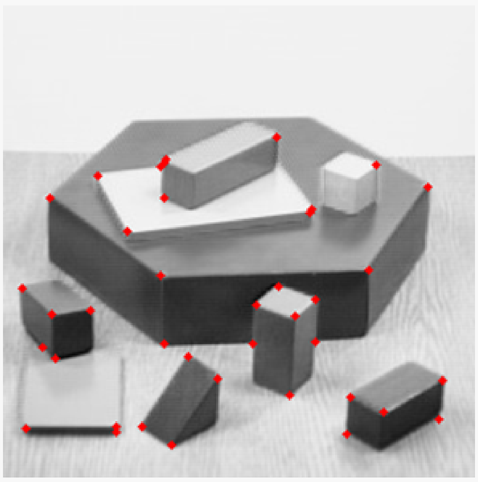
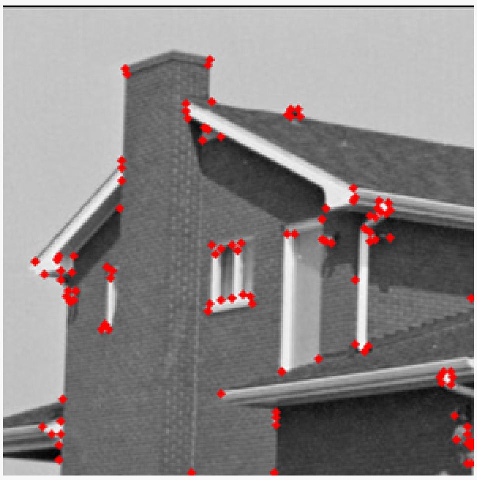
  Sigma = 1.0 thresh = 1e-5

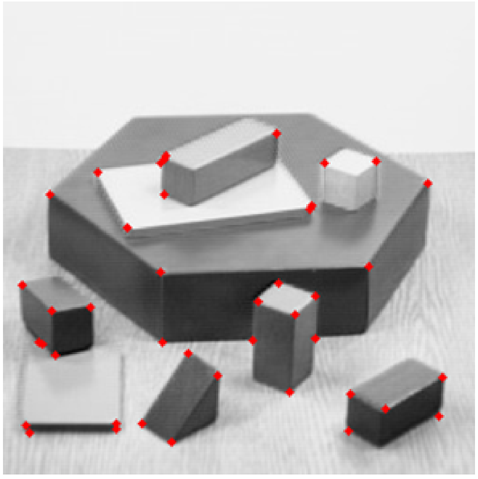
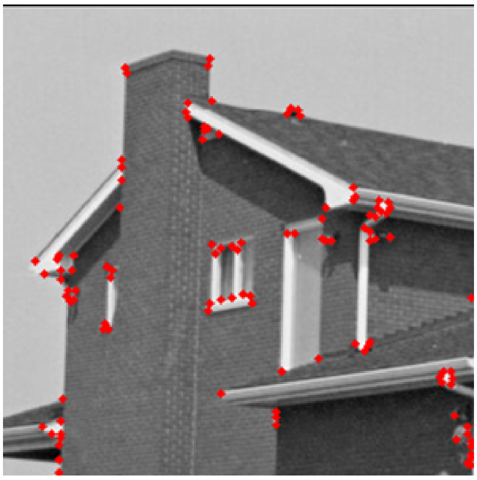
K	Blocks	House
0.04		
0.05		

K	Blocks	House
0.06		

- Change of Sigma from 0.5 to 2.0:

K = 0.05 thresh = 1e-5

Sigma	Blocks	House
0.5		
1.0		

Sigma	Blocks	House
2.0		

The appropriate values of the parameters above is $\begin{cases} \sigma = 1.0 \\ K = 0.05 \\ Thresh = 1e-5 \end{cases}$

Main issues of detected keypoints

The main issues of detected points are listed below:

- Shadows could have a strong inference with the detection. Many corners are not detected while some points at edges in the shadow could be detected.
- The points lying at the intersection of two lines (especially at the margin of images) could be detected as corners because they also have obvious change in two directions.
- The size of local maximum check patch is small, leading to multiple points gathering together.

Main steps of the implementation

```
# To compute the image gradients Ix and Iy by convolving operators in two
directions respectively
    operator_y = 0.5 * np.array([[-1],
                                 [0],
                                 [1]])
    operator_x = 0.5 * np.array([[-1, 0, 1]])
    # Ix and Iy matrix
    Ix_mat = signal.convolve2d(img, operator_x, mode='same')
    Iy_mat = signal.convolve2d(img, operator_y, mode='same')
```

```
# Compute the local auto-correlation matrix at each pixel. I divide the whole Mp matrix into four parts. For example, Mp_unsmooth11 means x image gradients square at each pixel. Then Mp_11 could be obtained by gaussianblur from Mp_unsmooth11
    Mp_unsmooth11 = Ix_mat * Ix_mat
    Mp_unsmooth12 = Ix_mat * Iy_mat
    Mp_unsmooth21 = Mp_unsmooth12
    Mp_unsmooth22 = Iy_mat * Iy_mat

    Mp_11 = cv2.GaussianBlur(Mp_unsmooth11, (3, 3), sigma, cv2.BORDER_REPLICATE)
    Mp_12 = cv2.GaussianBlur(Mp_unsmooth12, (3, 3), sigma, cv2.BORDER_REPLICATE)
    Mp_21 = Mp_12
    Mp_22 = cv2.GaussianBlur(Mp_unsmooth22, (3, 3), sigma, cv2.BORDER_REPLICATE)
```

```
# Then I can get C from four parts of auto-correlation matrix above
    C = (Mp_11 * Mp_22 - Mp_12 * Mp_21) - k * (Mp_11 + Mp_22) ** 2
```

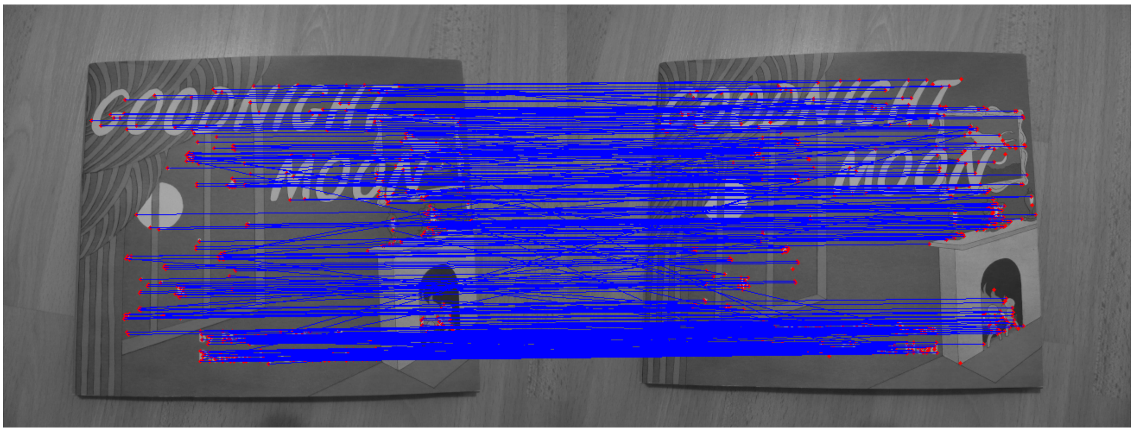
```
# First use scipy.ndimage.maximum_filter to blur all pixels to the maximum of their neighbors
# Then, the original maximum pixels in their neighborhood would have the same value in C and H, we find every index of pixel if C[i][j] == H[i][j]. Those follow the equations must have the local maximality of the response. Using argwhere could avoid for-loop, which decrease process time.
# Finally, we delete the index which lies at the periphery of imgs
    H = scipy.ndimage.maximum_filter(C, size=3)

    corners_idx = np.argwhere((C == H) & (C > thresh))
    corners = np.zeros((len(corners_idx), 2))
    corners[:, 0] = corners_idx[:, 1]
    corners[:, 1] = corners_idx[:, 0]
    corners = np.delete(corners, np.where((corners[:, 0] == 0) | (corners[:, 0] == img.shape[1]-1)| (corners[:, 1] == 0) | (corners[:, 1] == img.shape[0]-1))[0], axis=0)
    return corners.astype(int), C
```

Avoid for-loop by using np.argwhere to decrease process time!!!!!!!!!!!!!!!!!!!!!!!!

Description and Matching

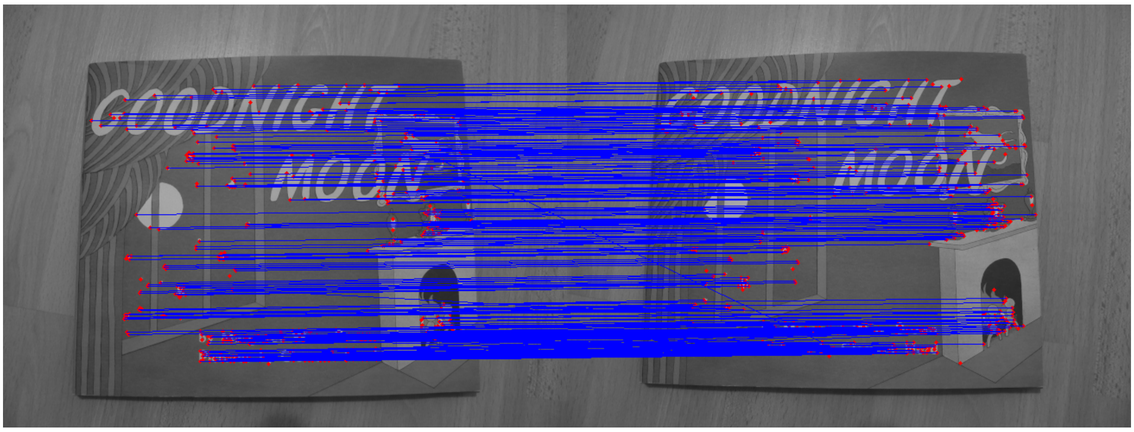
Results of one-way\mutual\ratio match

- One way



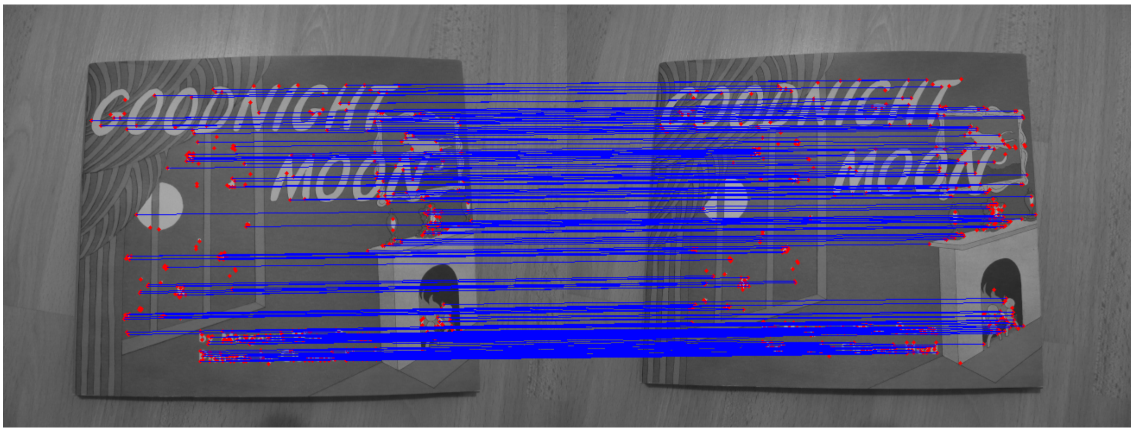
We can see there are a lot of wrong matches from img1 to img2. This is because the number of key features in img1 is larger than img2, so it's just a many to 1 match. The second reason could be there are some similar descriptors in img2.

- Mutual

  To decrease the wrong match, we implement the one way match from 2 to 1 in return. And we only choose those pairs have the corresponding results. So there are no many to 1 match problem. We can see the number of wrong match decreases effectively. However, there still exists some wrong match just because of some similar descriptors distances.

- Ratio

  Since it's avoidable to have some points from img 1 to have multiple corresponding points in img2. We just delete these points, and only choose the definite correct points by using ratio. This method only choose the pairs whose distances smaller than any other wrong pairs obviously.

From one-way to mutual to ratio, the wrong match decreases and thus the effects getting better and better. We can see there are almost no wrong match in Ratio-match method.

Main steps of the implementation

```
# I filter out keypoints that are too close to the edges.
# np.floor(patch_size/2.0) means the distance from the center points to the periphery of patches. Through the for-loop, the inappropriate positions could be deleted from the keypoints matrix
    h, w = img.shape[0], img.shape[1]
    for i in range(len(keypoints)):
        if keypoints[i][0] < np.floor(patch_size/2.0) or keypoints[i][0] >= w-(np.floor(patch_size/2.0)) or keypoints[i][1] < np.floor(patch_size/2.0) or keypoints[i][1] >= h-(np.floor(patch_size/2.0)):
            keypoints = np.delete(keypoints, i, axis=0)
    return keypoints
```

```
# For descriptors from two images, I compute distances of q1 each descriptor from image 1 to q1 each descriptor from image 2
# To avoid for-loop, distances[i][j] uses vectorized computation in Python (np.sum and vec ** 2)
    q1 = desc1.shape[0]
    q2 = desc2.shape[0]
    # print(q1, q2)
    distances = np.zeros((q1, q2))
    for i in range(q1):
        for j in range(q2):
            distances[i][j] = np.sum((desc1[i] - desc2[j]) ** 2)

    return distances
```

- One way match

```
# find the index of desc2 which has the minimum distance with No.i descriptor1, and stack [i,j] in rows
for i in range(q1):
    matches = np.row_stack((matches, np.array([i, np.argmin(distances[i])])))
```

- Mutual match

```
# Not only compute match matrix from 1 to 2 but also from 2 to 1
    matches1to2 = np.empty((0, 2), int)
    matches2to1 = np.empty((0, 2), int)
# Compute 1 to 2 match matrix
    for i in range(q1):
        matches1to2 = np.row_stack((matches1to2, np.array([i, np.argmin(distances[i])])))
# Compute 2 to 1 match matrix
    distancesT = distances.T
    for i in range(q2):
        matches2to1 = np.row_stack((matches2to1, np.array([i, np.argmin(distancesT[i])])))

# matches1to2[i][1] = j, if matches2to1[j][1] == 1, then these two key patches match
    for i in range(q1):
        if matches2to1[matches1to2[i][1]][1] == i:
```

```
            matches = np.row_stack((matches, np.array([i, matches1to2[i]
[1]])))
```

- Ratio match

```
# np.min(distances[i]) could get the min of distances from desc1[i] to the
index of desc2
# np.partition of sort_distances could get the second min of distances from
desc1[i] to index of desc2
# if min < ratio * sec_min, add this pair[i,j] to the match matrix
    sort_distances = np.sort(distances, axis=1)
    for i in range(q1):
        if np.min(distances[i]) <= ratio_thresh *
np.partition(sort_distances, kth=1, axis=1)[i][1]:
            matches = np.row_stack((matches, np.array([i,
np.argmin(distances[i])])))
```