Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision
and Geometry Lab

ETH Zurich
Computer Science Department

Computer Vision

Lab Assignment - Local Features

Lara Nonino - lnonino@student.ethz.ch

October 2022

Student ID:
22-937-833

Section 1

Harris Corner Detection

The *Harris detector* is a corner detector used to extract corners and infer features of an image. In this section, we are asked to implement one in order to find points of major interest in an image. To do so I proceeded as follows.

1.1 Image gradients

In the first place, I computed the image gradients I_x and I_y in the x and y directions respectively.

$$I_x(i,j) = \frac{I(i,j+1) - I(i,j-1)}{2} \qquad I_y(i,j) = \frac{I(i+1,j) - I(i-1,j)}{2}$$

To implement above as convolution, I used the following filters

$$I_x(i,j) = I(i,j) * \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} \qquad I_y(i,j) = I(i,j) * \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}^T$$

With the function `scipy.signal.convolve2d`, the implementation resulted in

```
Ix = signal.convolve2d(img, np.array([-0.5, 0, 0.5]).reshape(1, 3), 'same')
Iy = signal.convolve2d(img, np.array([-0.5, 0, 0.5]).reshape(3, 1), 'same')
```

1.2 Local auto-correlation matrix

Afterward, we were asked to compute the elements of the auto correlation matrix defined as

$$M_p = \sum_{p' \in N(p)} w_{p'} \begin{bmatrix} I_x(p')^2 & I_x(p')I_y(p') \\ I_y(p')I_x(p') & I_y(p')^2 \end{bmatrix} = \begin{bmatrix} g(I_x^2) & g(I_xI_y) \\ g(I_yI_x) & g(I_y^2) \end{bmatrix}$$

where N(p) is a neighbourhood of point p.
As suggested, I chose to blur the gradients using a gaussian with standard deviation σ (centered in $(0,0)$). With the function `cv2.GaussianBlur`, the implementation resulted in

```
Ixx = Ix * Ix
Iyy = Iy * Iy
Ixy = Ix * Iy

Mxx = cv2.GaussianBlur(Ixx, ksize=(3, 3), sigmaX= sigma, borderType=cv2.BORDER_REPLICATE)
Myy = cv2.GaussianBlur(Iyy, ksize=(3, 3), sigmaX= sigma, borderType=cv2.BORDER_REPLICATE)
Mxy = cv2.GaussianBlur(Ixy, ksize=(3, 3), sigmaX= sigma, borderType=cv2.BORDER_REPLICATE)
```

1.3 Harris response function

The next step consisted in computing the Harris response function. This one informs us whether a pixel is part of a corner, an edge or a flat region. It can be defined as

$$C(i,j) = det(M_{i,j}) - k * Tr^2(M_{i,j})s$$

$$C(i,j) = g(I_x^2)g(I_y^2) - [g(I_xI_y)]^2 - \alpha[g(I_x^2) + g(I_y)^2]^2$$

where k is an empirically determined constant. Implementing the second version of the formula, the code resulted in

```
C = Mxx * Myy - (Mxy ** 2) - k * ((Mxx + Myy) ** 2)
```

1.4 Detection criteria

The last part of this section consisted in selecting the keypoints of the image if the following two conditions are satisfied:

- The Harris response function $C(i,j)$ is above a certain detection threshold T;
- The Harris response function $C(i,j)$ is a local maxima in its 3 x 3 neighborhood.

To compute the local maximum, I took advantage of the `scipy.ndimage.maximum_filter` function which gives as output the maximum value of the Harris response function computed among the 3 x 3 pixels' window.
Once the two conditions were defined, I found the keypoints using the `np.argwhere` function, which returns the indices (coordinates) of the points grouped by element. Since the coordinates are returned in a swapped way (the indices of a matrix are in reverse order of the standard way the (x, y) coordinates are defined), it was necessary to invert the coordinates at last. The code follows.

```
case1 = C > thresh
case2 = C == scipy.ndimage.maximum_filter(C, size=(3, 3))
corners = np.argwhere(case1 & case2)
corners[:,[0,1]] = corners[:,[1,0]]
```

By performing different simulations with varying values σ, k and T, it is possible to get different keypoints sets.

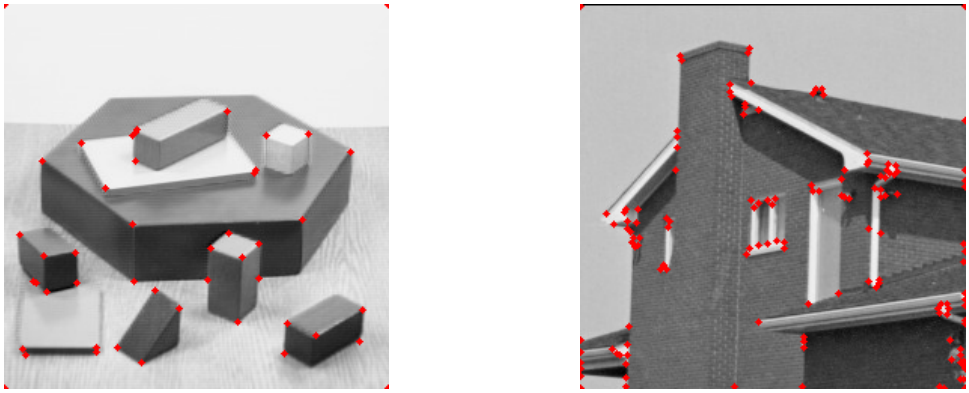

Figure 1.1: $\sigma = 1.0,\ k = 0.04,\ T = 1 \times 10^{-5}$ (keypoints: 43 - 139)

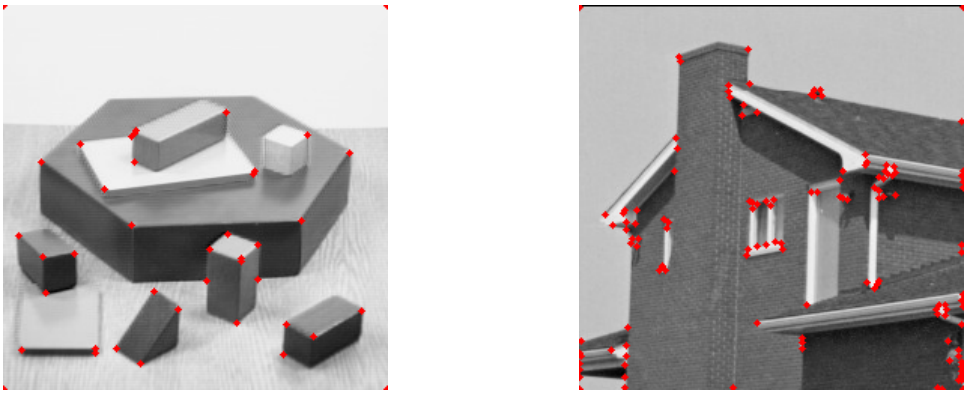

Figure 1.2: $\sigma = 1.0,\ k = 0.06,\ T = 1 \times 10^{-5}$ (keypoints: 41 - 129)

From the comparison between Figure 1.1 and Figure 1.2, we can understand the meaning of the k **parameter**. Indeed, according to the chosen value, we can score the *cornerness* of a point: with a bigger k, we will get less wrong corners and, for example, we will emphasize the real corners over edges. However, we will also miss more real corners.
With a smaller k we will get more corners, missing less true corners. On the other hand, we will get a lot of false ones.

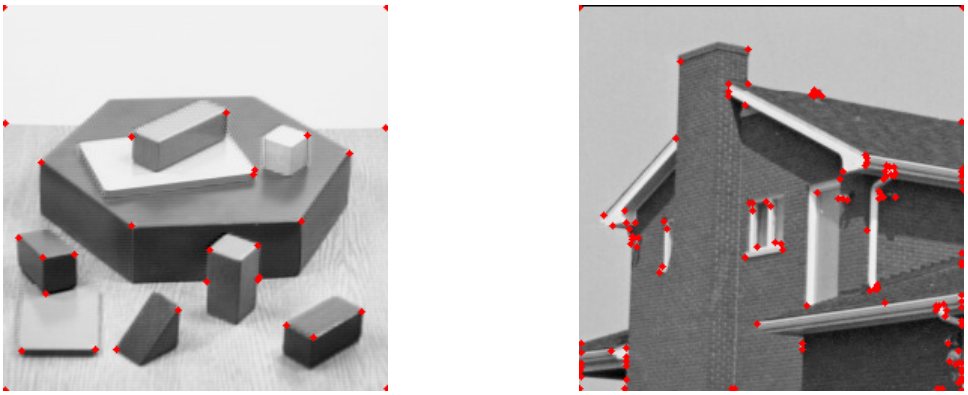

Figure 1.3: $\sigma = 0.5,\ k = 0.05,\ T = 1 \times 10^{-5}$ (keypoints: 31 - 124)

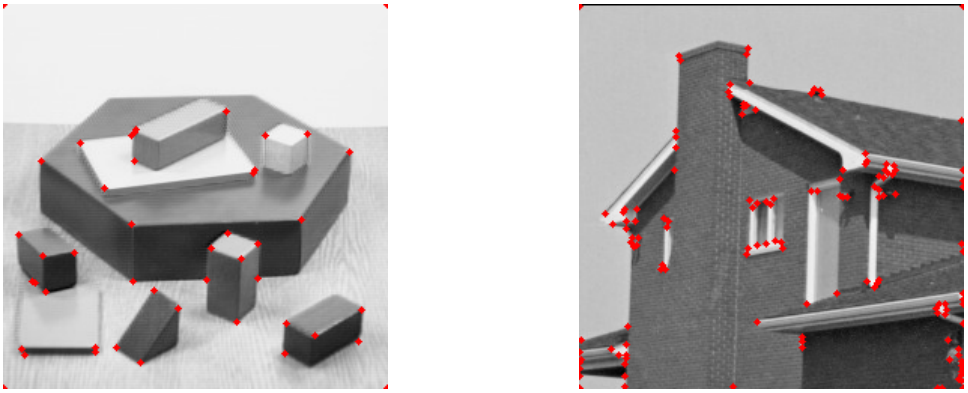

Figure 1.4: $\sigma = 2.0,\ k = 0.05,\ T = 1 \times 10^{-5}$ (keypoints: 46 - 135)

From the comparison between Figure 1.3 and Figure 1.4, we can understand the meaning of the σ parameter. This parameter is used to set the variance of the gaussian used to blur the gradients in order to reduce the level of noise of the image. In this way the number of corners detected might decrease (since a corner could be blurred out). However, it could also happen that the Harris detector is not able to detect a corner due to the high level of noise. In this case, blurring the gradients with a bigger σ, would help the detector finding the right corner, and therefore the number of keypoints would increase.

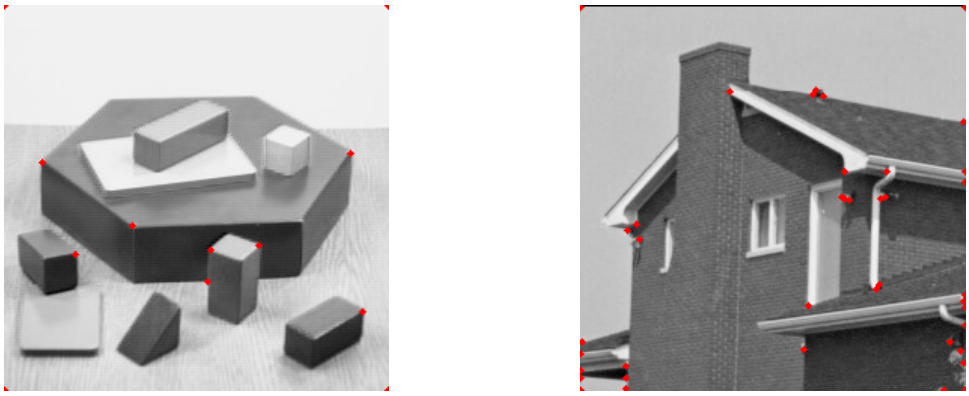

Figure 1.5: $\sigma = 1.0,\ k = 0.05,\ T = 1 \times 10^{-4}$ (keypoints: 12 - 40)

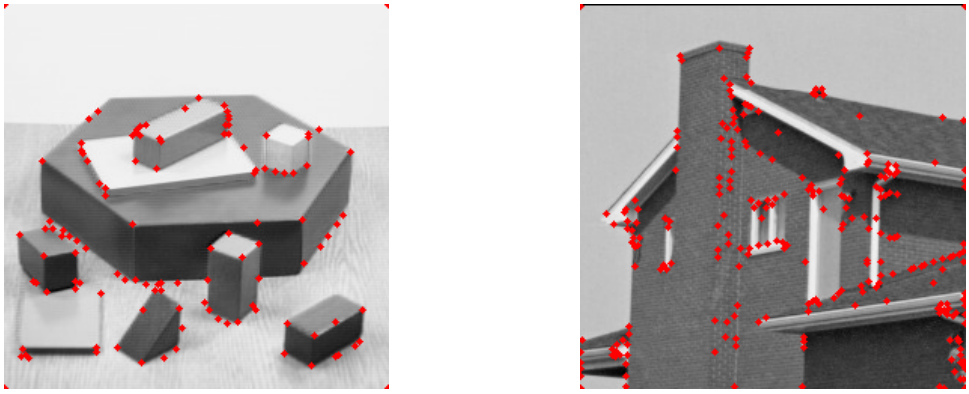

Figure 1.6: $\sigma = 1.0,\ k = 0.05,\ T = 1 \times 10^{-6}$ (keypoints: 118 - 246)

From the comparison between Figure 1.5 and Figure 1.6, we can understand the meaning of the T **parameter**. This is the threshold applied to the Harris response function and, as the pictures show, its changing has a significant influence on the set of points selected as corners: as T increases, the number of keypoints will decrease.

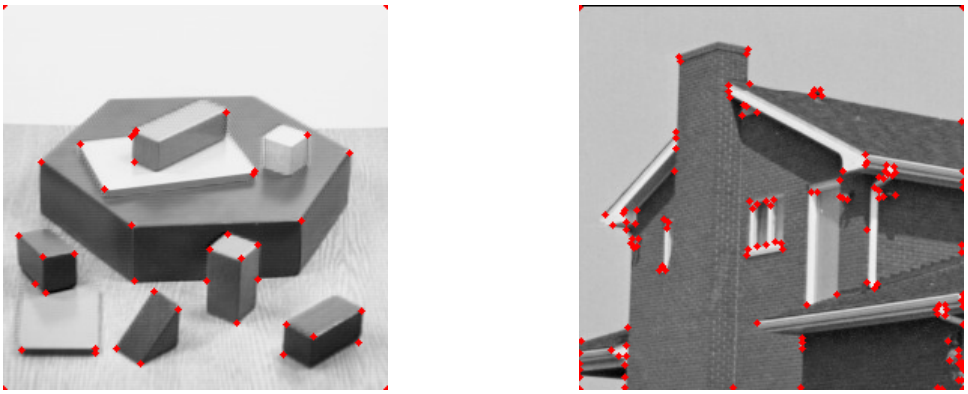

Figure 1.7: $\sigma = 1.0,\ k = 0.05,\ T = 1 \times 10^{-5}$ (keypoints: 43 - 133)

For the rest of the assignment I will use the hyperparameters of Figure 1.7, which are $\sigma = 1.0,\ k = 0.05,\ T = 1 \times 10^{-5}$.

On the whole, the feature detector implemented seems to work properly. However, some points are classified as corner even if they are not and some right corners are not detected. The cause of this issue might lie in the tuning of the hyperparameters and the trade-off between high precision and high recall.

Section 2

Description & Matching

In this section, we are asked to implement a matching protocol for image patches and test it out on a provided image pair. In order to achieve such a result, I proceeded as follows.

2.1 Local descriptors

As a first step, I needed to extract patches around each detected keypoint, which will be used as descriptors. This is done the provided `extract_descriptors` function which extracts 9 x 9 patches.
To avoid out-of-bounds issues, it was necessary to filter out the keypoints that are were close to the image boundaries. To do so, I decided to keep only those corners that had a margin of at least half of the size of the patch (9 x 9). Thus, in this case I used a 4 pixel margin.

```
def filter_keypoints(img, keypoints, patch_size = 9):

    Xs = np.array(keypoints[:, 1])
    Ys = np.array(keypoints[:, 0])
    offset = int(np.floor(patch_size / 2.0))
    mask = (Xs > offset) & (Xs < img.shape[0] - offset) & \
            (Ys > offset) & (Ys < img.shape[1] - offset)
    mask = np.column_stack((mask, mask))
    return keypoints[mask].reshape(-1,2)
```

2.2 SSD one-way nearest neighbors matching

Once the points that were too close to the image boundaries were filtered out, we needed to compute the sum-of-squared-differences (SSD) between the descriptors of all features from the first image and the second image. Since the SSD is defined as

$$SSD(p, q) = \sum_i (p_i - q_i)^2$$

I took advantage of the function `scipy.spatial.distance.cdist` which computes the distance between each pair of the two collections of inputs. To obtain the actual SSD, I also needed to squared it.

```
def ssd(desc1, desc2):
    assert desc1.shape[1] == desc2.shape[1]
    distances = cdist(desc1, desc2) ** 2
    return distances
```

In order to implement a one-way nearest neighbors matching protocol where each feature from the first image is associated to its closest feature from the second image, I followed these steps:

- I found the minimum gradient distance between each pair of keypoints (each one belonging to a different image). Using the `np.amin` function, I therefore obtained an array of minimum distances called min (the i^{th} element of the array will contain the minimum gradient distance between the i^{th} keypoint of the first image and the closest one in the second image).
- Using np.where I was able to find the indices of two matching point. In particular, `xind` contains the index of all the first image corners ([0, 1, 2, . . . , n]) while `yind` will be an array where the i^{th} element will be the index of the keypoint of the second image whose gradient is closer to the i^{th} keypoint of the first image.
- I created the (n x 2) metrix that contains the matches of the keypoints between the first and the second image.

The resulting implementation follows

```
min = np.amin(distances, axis=1)
xind, yind = np.where(distances == min[:, None])
matches = np.column_stack((xind, yind))
```

As the following image shows, most of the keypoints are correctly matched (horizontal lines), while others are not (diagonal lines). 'Mutual nearest neighbors matching' and 'Ratio test matching' will improve the performance of the protocol.

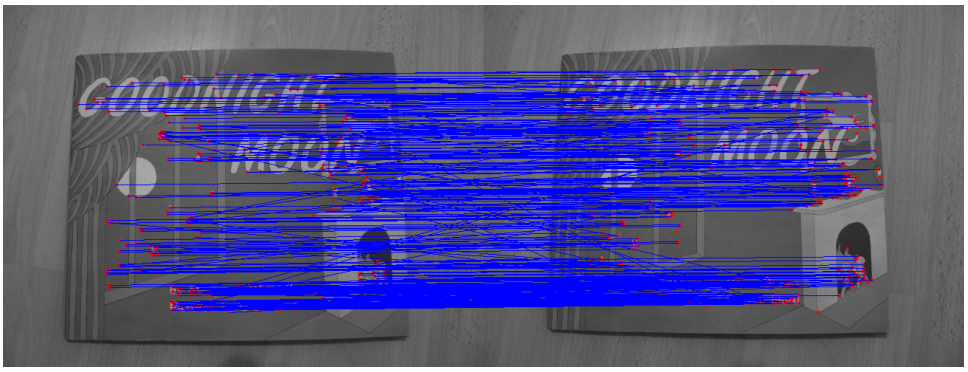

Figure 2.1: One-way nearest matching

2.3 Mutual nearest neighbors

'Mutual nearest neighbors protocol' consists in comparing each one-way match obtained from two images with each one-way match obtained from the same two images, but in reverse order. If the two matches are between the same pair of corners, than that match will be considered valid.

The implementation is similar to the one-way matching, but is repeated for the two ways of comparison.

```
min_ow = np.amin(distances, axis=1)
xind, yind = np.where(distances == min_ow[:, None])
matches_ow= np.column_stack((xind, yind))
```

```
min_mut = np.amin(distances, axis=0)
xind, yind = np.where(distances == min_mut[None, :])
matches_mut = np.column_stack((xind, yind))

matches = np.unique(matches_ow[np.where(cdist(matches_ow, matches_mut) == 0)[0]], axis=0)
```

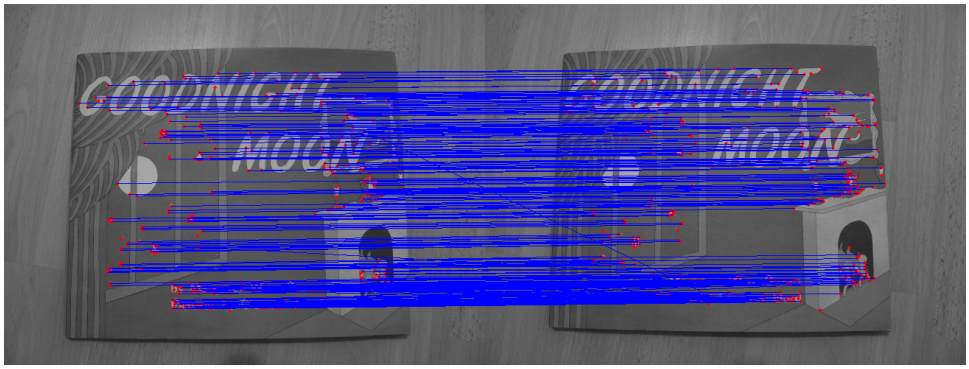

Figure 2.2: Mutual nearest neighbors protocol

As predicted, some wrong matches from the one-way nearest matching have been corrected. However, the presence of some diagonals lines means that some corners are still incorrectly matched.

2.4 Ratio test

'Ratio test protocol' consists in executing a one-way match and consider it valid only if the ratio between the first and the second nearest neighbor is lower than a given threshold.

To implement this protocol I took advantage of the `np.partition` function which helped me selecting the 2 minimum distances from each point feature (first two elements of each row of `ordered_distances`). After having computed the ratio between the first and the second minimum distance from each point feature, and selected the matches between the two nearest points features (`np.where`), I masked the results in such a way that only those points whose ratio was lower than the threshold were kept.

```
ordered_distances = np.partition(distances, 1, axis = 1)
min = ordered_distances[:,0]
ratios = ordered_distances[:, 0] / ordered_distances[:, 1]
xind, yind = np.where(distances == min[:, None])
matches = np.column_stack((xind, yind))
matches = matches[ratios < ratio_thresh]
```

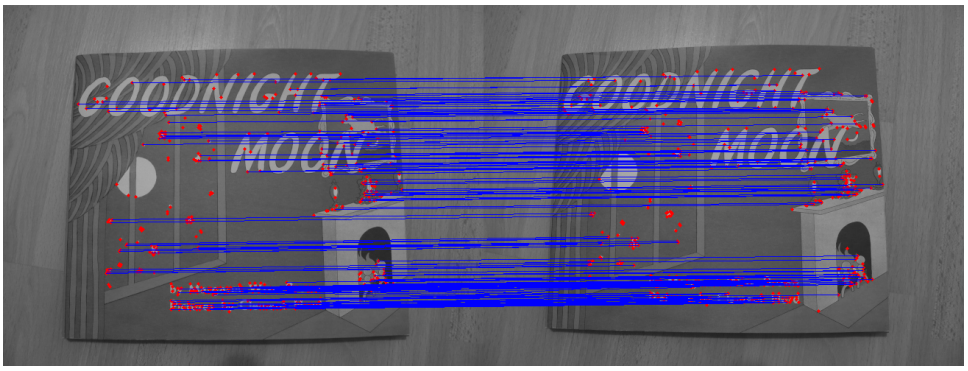

Figure 2.3: Ratio test protocol (threshold = 0.3)

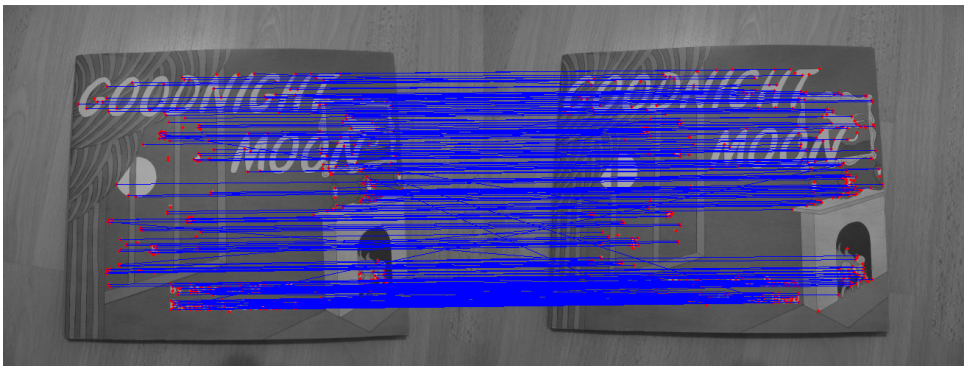

Figure 2.4: Ratio test protocol (threshold = 0.8)

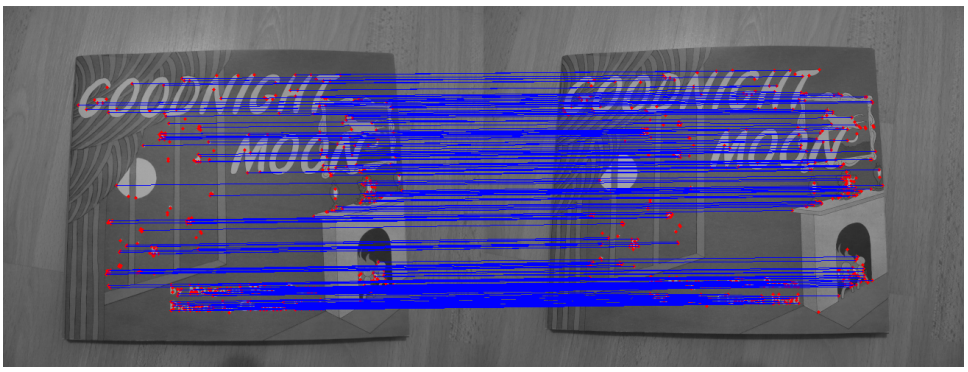

Figure 2.5: Ratio test protocol (threshold = 0.5)

Finally, using threshold = 0.5 with this protocol, all the matches seem to be correctly evaluated. Thus, is it possible to say that in this case this protocol is the most efficient.