Computer Vision
Lab Assigment Report
Local Features

Tuan Mate Nguyen (tunguyen@student.ethz.ch)

1 Detection

1.1 Image gradients

Computing the image gradients in the x and y direction, as described by

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

(where i and j are coordinates in the y and x direction, respectively) can be achieved by convolving the grayscale image with the following (1, 3) filter:

```
26  dx = np.array([[-0.5, 0, 0.5]])
```

In the y direction the kernel is just the transpose of the kernel for x direction. In case the image has too much noise, the above filters could be replaced with Sobel-filters (of the corresponding direction) which include some Gaussian blur.

1.2 Local auto-correlation matrix

Since $I_x(p')$ and $I_y(p')$ take the gradient values at p' and then these values are multiplied, we first need to do an element-wise multiplication of the image gradients:

```
37  Ix2 = Ix * Ix
38  Iy2 = Iy * Iy
39  Ixy = Ix * Iy
```

Computing the weighted sum of the image gradient products in the neighbourhood of a given pixel p equals to applying a Gaussian blur on the image-gradient products:

```
43  Gx2 = cv2.GaussianBlur(Ix2, ksize=ksize, sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
44  Gy2 = cv2.GaussianBlur(Iy2, ksize=ksize, sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
45  Gxy = cv2.GaussianBlur(Ixy, ksize=ksize, sigmaX=sigma, borderType=cv2.BORDER_REPLICATE)
```

with the auto-correlation matrix being

$$M = \begin{bmatrix} Gx2 & Gxy \\ Gxy & Gy2 \end{bmatrix}$$

The kernel was set to $(0, 0)$ so it's automatically set based on σ.

1.3 Harris response function

Using the closed form formula from Harris & Stephens, the response function is

$$C = det(M) - ktrace^2(M)$$

which, using the previous results, yields

```
48  C = Gx2*Gy2 - Gxy*Gxy - k * (Gx2 + Gy2)*(Gx2 + Gy2)
```

Here C is represented by a matrix where each element equals to the Harris-response at the same position of the image (again, element-wise addition and multiplication are needed).

1.4 Detection criteria

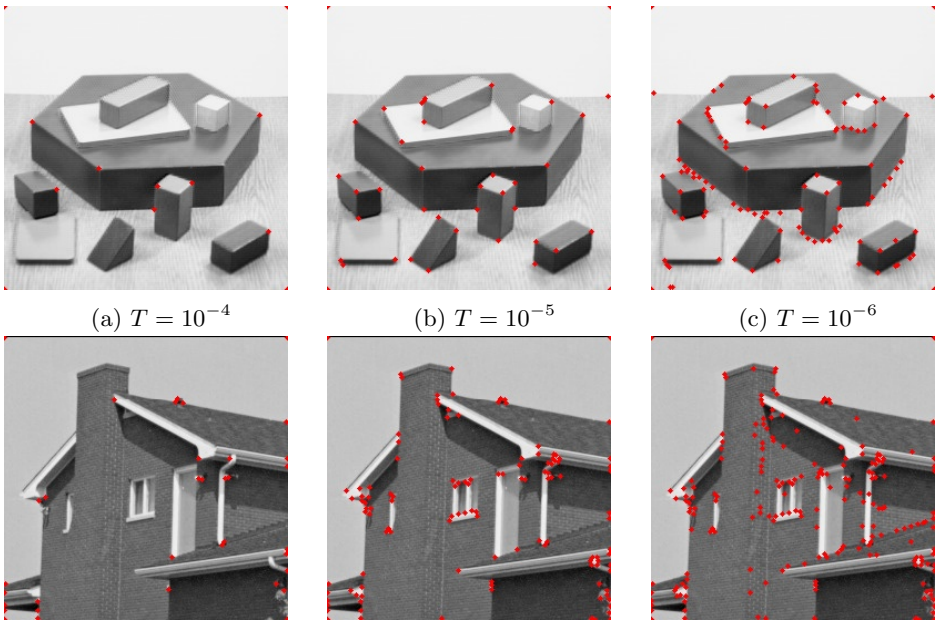
Common issues

In all images the four corner points of the image are detected as false-positive corners. The reason is that independently from border padding the pixel values will always form a corner except when neighboring pixels have exactly the same value, which is rare. However, later these will be filtered because they are too close to the edges, so they won't cause problems.

The two most frequent problems of keypoints are ambiguity, meaning detecting several keypoints for one actual corner, and detecting false positives in parts where there is only an edge or even just a (noisy) flat area.

Parameter sweep

Threshold T: For $T = 10^{-4}$ too few corners are detected, while for $T = 10^{-}6$ too many (multiple keypoints for a single corner, keypoints along edges). As a result $T = 10^{-5}$ was selected.



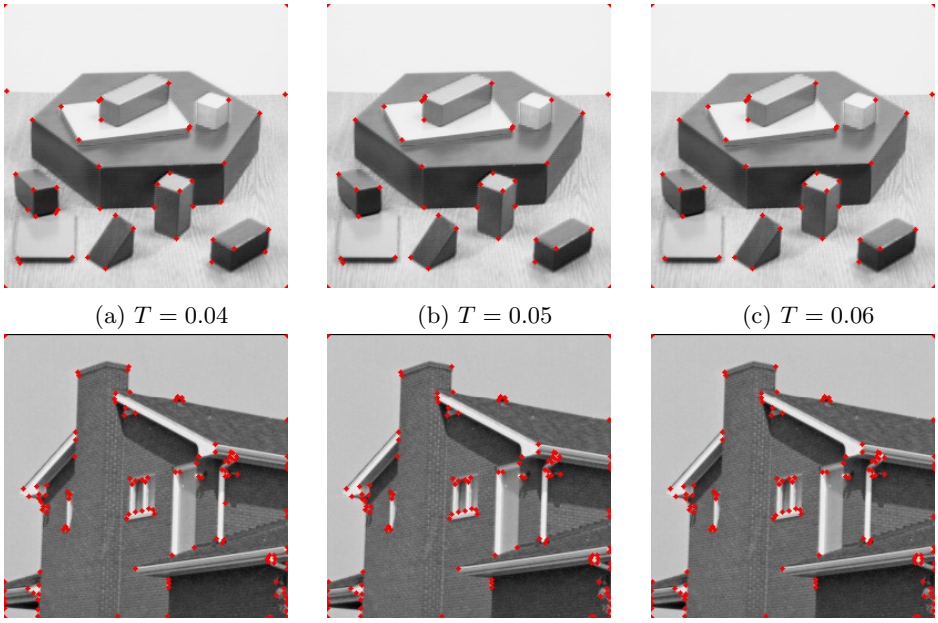
(a) $T = 10^{-4}$ (b) $T = 10^{-5}$ (c) $T = 10^{-6}$

(d) $T = 10^{-4}$ (e) $T = 10^{-5}$ (f) $T = 10^{-6}$

Figure 1: Parameter sweep for T with $\sigma = 1.0$ and $k = 0.05$

Constant k: For different k values there is not much difference between the obtained keypoints. In general a lower k value results in slightly more detected keypoints, however, sometimes among these are a few that don't correspond to corners or several keypoints are present for the same corner. For further experiments $k = 0.05$ was used.

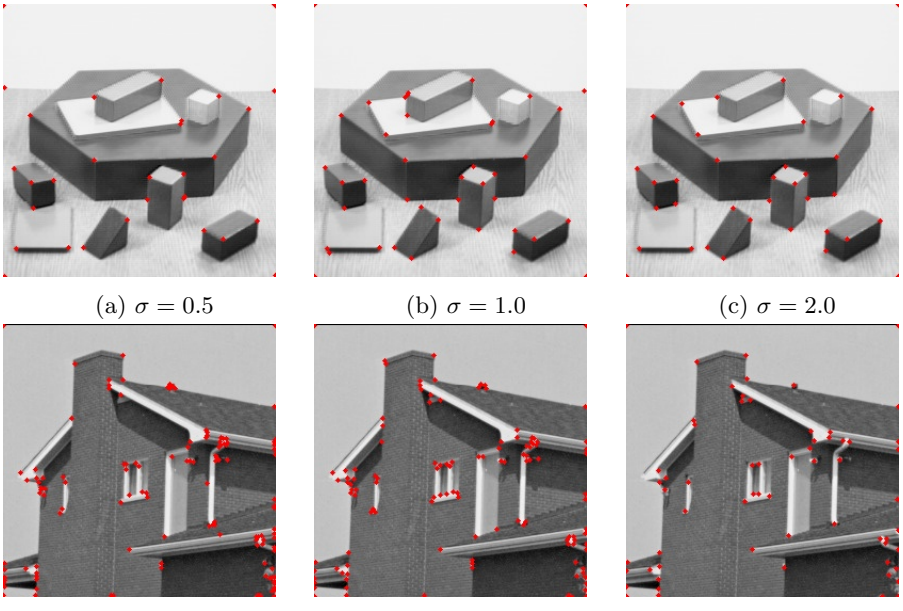
Standard deviation σ: With small ($\sigma = 0.5$) standard deviation obvious corners are missed (see blocks image). Applying bigger blur ($\sigma = 1.0$) gives more points but these are often redundant. The best result is achieved with $\sigma = 2.0$, which detects most of the sharp corners while avoiding duplicate keypoints.



(a) $T = 0.04$ (b) $T = 0.05$ (c) $T = 0.06$

(d) $T = 0.04$ (e) $T = 0.05$ (f) $k = 0.06$

Figure 2: Parameter sweep for k with $\sigma = 1.0$ and $T = 10^{-5}$



(a) $\sigma = 0.5$ (b) $\sigma = 1.0$ (c) $\sigma = 2.0$

(d) $\sigma = 0.5$ (e) $\sigma = 1.0$ (f) $\sigma = 2.0$

Figure 3: Parameter sweep for σ with $k = 0.05$ and $T = 10^{-5}$

2 Description and matching

2.1 Local descriptors

Patches are centered around the keypoint so they need to have an odd pixel size.

```
8   assert(patch_size % 2 == 1)
```

Only those keypoints are kept that are at least half patch-size pixel away from the edges in both direction. The accepted area is a subrectangle with the following top-left and bottom-right coordinates:

```
11  d = patch_size // 2
12
13  height, width = img.shape
14  # Points' coordinates must be "min. distance" far away from both edges,
15  # along both axises
16  min_w_idx = min_h_idx = d
17  max_w_idx = (width-1)-d
18  max_h_idx = (height-1)-d
19
20  # These intervals define a sub-rectangle in the image, with the following
21  # top-left/bottom-right coordinates:
22  tl = np.array([min_w_idx, min_h_idx])
23  br = np.array([max_w_idx, max_h_idx])
```

2.2 Sum of squared differences - one way

In order to be able to iterate through features of all pairs of keypoints, a meshgrid is first created for their indices. This just contains coordinates of a 2-D mesh where each coordinate will correspond to a keypoint index from the first and second image. The result are two 2-D arrays containing these "index coordinates" of the mesh.

```
20  grid_idx1, grid_idx2 = np.meshgrid(idx1, idx2, indexing='ij')
```

The SSD can then simply be computed in a vectorized way:

```
24  ssd = np.sum(np.square(desc1[grid_idx1,:] - desc2[grid_idx2,:]), -1)
```

For the one-way nearest neighbor matching first an array is created with the keypoint indices of from the first image. Then for each of them, the SSDs from the comparison with descriptors from the second image are sorted in ascending order. The smallest SSD will define mathcing keypoints.

```
45      x = np.arange(q1)
46      y = np.argmin(distances, 1)
47      matches = np.array([x, y]).transpose()
```

As it can be seen in Fig.4, all keypoints from the first image match with some points from the second. On the other hand in the second image some keypoints are left unpaired. It is worth mentioning that this would be the case even if the two images would be completely different. An inherent flaw of this approach is that the corresponances can have many-to-one relation ship, resulting in ambiguity. This can happen if several descriptors from the first image are closest to the same descriptor from the second image. These have to be filtered out, otherwise the homography cannot be determined.

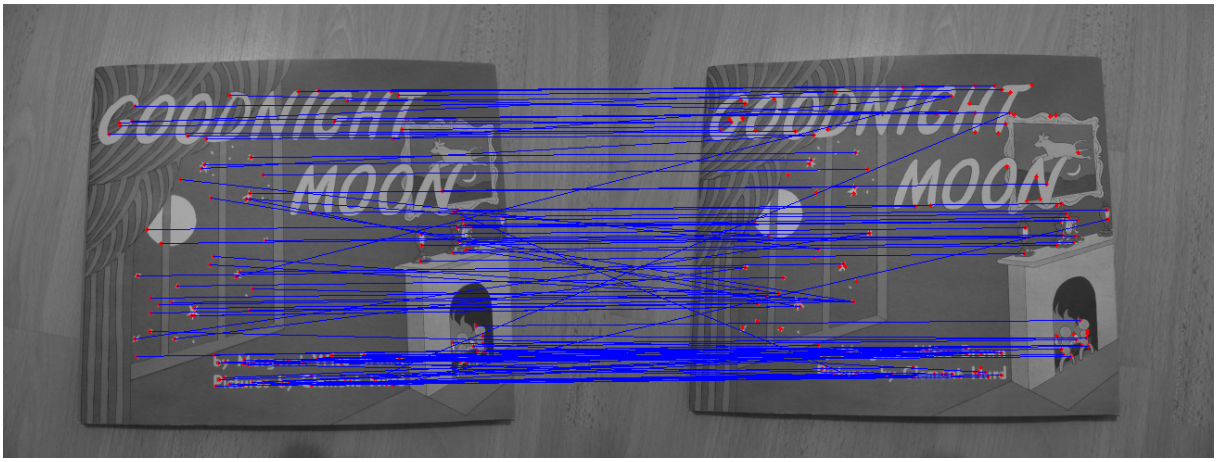


Figure 4: Matching keypoints for one-way nearest neighbor matching

Mutual nearest neighbors

One solution is to verify the closest descriptor from the other image's perspective as well. The mutual proximity can be checked by getting each keypoint (in the first image) $x1$'s nearest neighbor (in the second image) $y1$'s nearest neighbor (in the first image) $x2$. If this is the same as the original keypoint, they match:

```
57  x2 = np.argmin(distances, 0)
58  x = x1[np.where(x1 == x2[y1])]
```

Mutual matches guarantee one-to-one relationships. A consequence is that now a match is not guaranteed for all keypoints in either images (in theory one could end up without having any matches). This is nicely shown in Fig.5. The quality of the matching is higher compared to the one-way matching, since not every keypoints are forced to have a pair, only those that are good matches, meaning being mutually close.

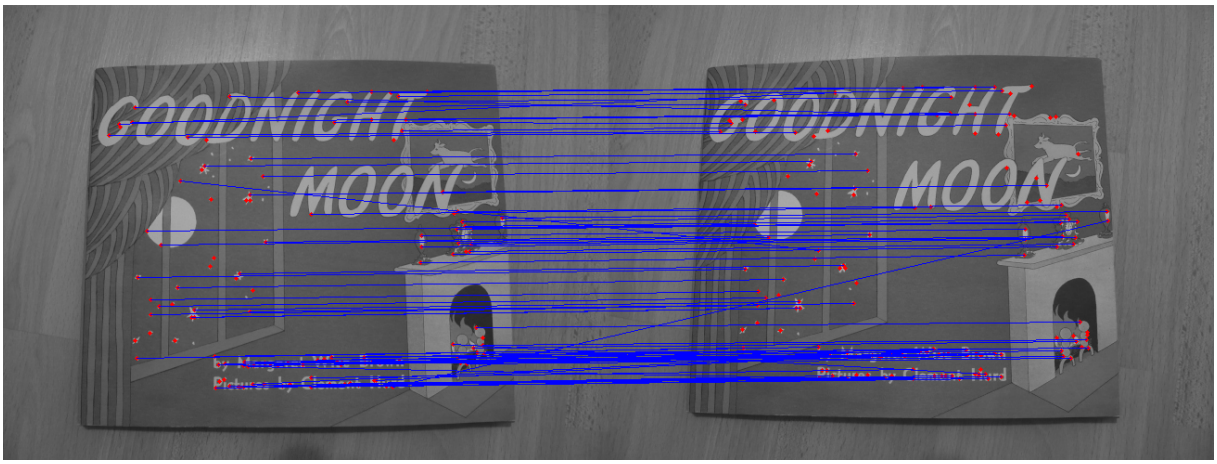


Figure 5: Matching keypoints for mutual nearest neighbor matching

For the ratio test matching, we partially sort (with *np.partition*) the SSDs from the comparison with descriptors from the second image, so that the first two elements - (0,1) - in every row are the nearest and second nearest neighbors.

```
68      closest_matches = np.partition(distances, (0,1), 1)
```

When dividing, we also must ensure that the second SSD is not zero. This can be the case when the second image contains two or more features that are bitwise identical to one in the first image (for example synthetic image with repetitive patterns). In this case the value can be set to np.nan and be ignored because the match would be ambiguous anyway.

Their ratio are then thresholded to find the final matches:

```
75      x = np.where((~np.isnan(closest_matches[:,1])) &
76                   (closest_matches[:, 0]/closest_matches[:, 1] < ratio_thresh))[0]
77      y = np.argmin(distances, 1)[x]
78      matches = np.array([x, y]).transpose()
```

Since the thresholding criteria doesn't necessarily hold, there are again unpaired keypoints. There are also fewer matches, as the requirement for sufficiently distinctive features further eliminates candidates.

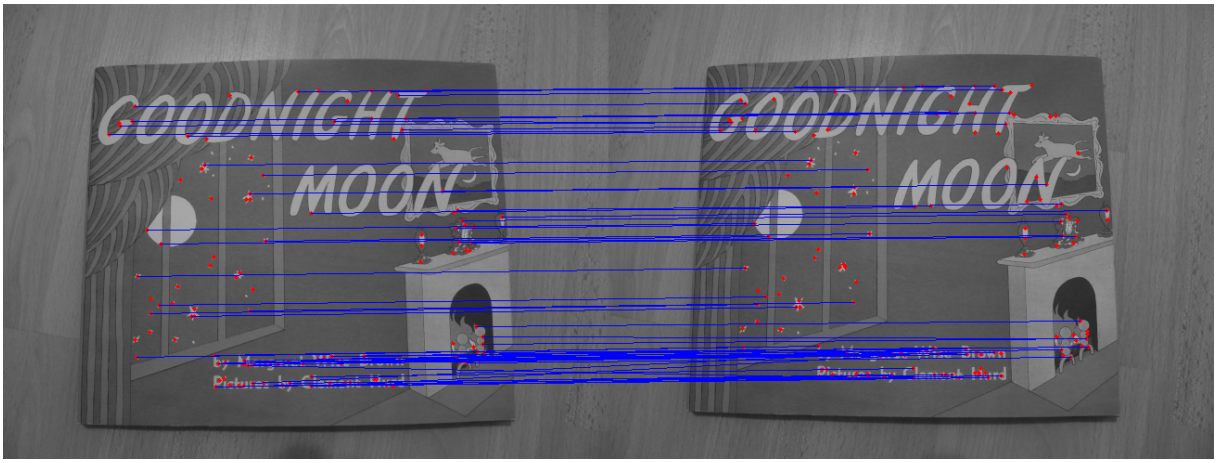


Figure 6: Matching keypoints for ratio test matching