Lab assignment 2

ETH Zurich, Computer Science Department

Computer vision

Report

Local Features

Author
Emilia Szymańska, 22-945-547

October 2022

The objective of this lab assignment was to implement a feature detector and a basic matching protocol to establish pixel-wise correspondences between images. The focus was on Harris detector and sum-of-squared-differences for feature matching.

1 Detection

1.1 Implementation

1.1.1 Image gradients

To compute the gradients I_x and I_y following the formulas (1), (2) I used the scipy.signal.convolve2d function. The crucial part was to define the kernels for both cases. For I_x, the kernel should contain a vector $[-0.5, 0, 0.5]$, while for I_y it has the form of $[-0.5, 0, 0.5]^T$. However, we should reverse the values in function's kernels to achieve the result we have in mind. Therefore, written in a form of a 3x3 matrix, the corresponding filter kernels look as depicted in (3), (4).

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

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

$$k_x = \begin{bmatrix} 0 & 0 & 0 \\ 0.5 & 0 & -0.5 \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

$$k_y = \begin{bmatrix} 0 & 0.5 & 0 \\ 0 & 0 & 0 \\ 0 & -0.5 & 0 \end{bmatrix} \tag{4}$$

The code implementation of this part is presented below:

```
kernel_x = [[ 0   , 0,  0   ],
            [ 0.5, 0, -0.5],
            [ 0   , 0,  0   ]]

kernel_y = [[0,   0.5, 0],
            [0,   0   , 0],
            [0, -0.5, 0]]

Ix = scipy.signal.convolve2d(img, kernel_x)
Iy = scipy.signal.convolve2d(img, kernel_y)
```

1.1.2 Local auto-correlation matrix

The next step was to compute the local auto-correlation matrix elements. I first caluclated I_x^2, I_y^2 and I_xI_y as element-wise multiplications. After that I applied the Gaussian blurring to the obtained matrices. We do not need to calculate the matrices corresponding to each pixel, because in the next step we can simply utilize I_x^2, I_y^2 and I_xI_y.

The code implementation of this part is presented below:

```
Ixx = np.multiply(Ix, Ix)
Iyy = np.multiply(Iy, Iy)
Ixy = np.multiply(Ix, Iy)

Ixxg = cv2.GaussianBlur(Ixx, ksize=(0,0), sigmaX=sigma, borderType=cv2.
    BORDER_REPLICATE)
Iyyg = cv2.GaussianBlur(Iyy, ksize=(0,0), sigmaX=sigma, borderType=cv2.
    BORDER_REPLICATE)
Ixyg = cv2.GaussianBlur(Ixy, ksize=(0,0), sigmaX=sigma, borderType=cv2.
    BORDER_REPLICATE)
```

1.1.3 Harris response function

Harris response for each pixel can be calculated using the formula (5).

$$C(i,j) = det(M_{ij}) - k \cdot Tr^2(M_{ij}) \tag{5}$$

The determinant of matrix M_{ij} is equal to $I_x^2(i,j) \cdot I_y^2(i,j) - (I_x(i,j)I_y(i,j))^2$, k is a parameter and a trace of the aforementioned matrix is $I_x^2(i,j) \cdot I_y^2(i,j)$. Therefore, to obtain C, I run the following computation (6) for each pixel.

$$C(i,j) = I_x^2(i,j) \cdot I_y^2(i,j) - (I_x(i,j)I_y(i,j))^2 - k \cdot I_x^2(i,j) \cdot I_y^2(i,j) \tag{6}$$

The code implementation of this part is presented below:

```
C = np.copy(img)
height = img.shape[0]
width  = img.shape[1]

for x in range(0, width):
    for y in range(0, height):
        C[y, x] = Ixxg[y, x] * Iyyg[y, x] - Ixyg[y, x] * Ixyg[y, x]  - k * Ixxg[y, x] * Iyyg[y, x]
```

1.1.4 Detection criteria

To detect the corners, I had to apply two conditions. First, I run the maximum_filter() function with a kernel size 3x3, to assign each pixel the maximum value from its neighborhood. Then I check if the pixel value is higher than the predefined threshold and if the center pixel response is higher than a pixel inside the window (by comparing $C(i,j)$ and the corresponding pixel in the maximum filtered matrix - if it was not changed by the filter, it means the maximum was in the center of the window). If both conditions are satisfied, I add the pixel to the detected corners.

The code implementation of this part is presented below:

```
max_filtered = scipy.ndimage.maximum_filter(C, size=(3,3))
corners = []
for x in range(0, width):
    for y in range(0, height):
        if max_filtered[y, x] > thresh and max_filtered[y, x] == C[y, x]:
            corners.append([x,y])

corners = np.array(corners)
```

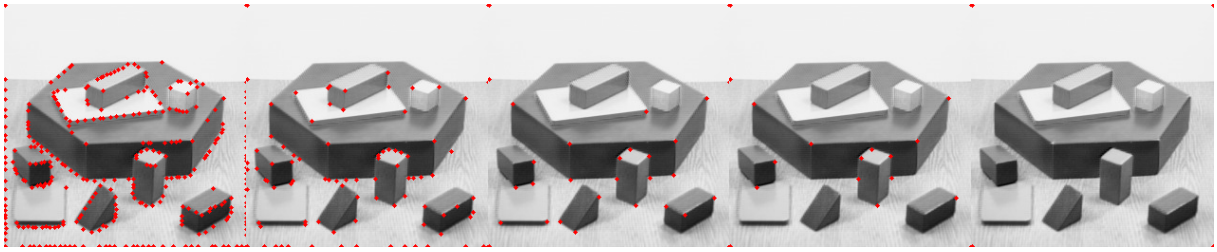
1.2 Results

The obtained detector was run for 5 different values of threshold T (Fig. 1,2), constant k (Fig. 3,4) and the standard deviation σ (Fig. 5,6).

Detected keypoints are not always true corners. Some of them are falsely detected on the edge of the image, some true ones are discarded. I think the hardest part of detecting corners is tuning all parameters so that only the correct corners are found.

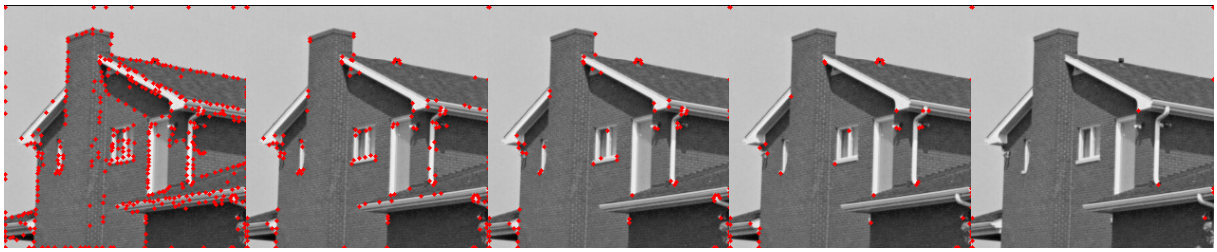
The higher the threshold, the fewer points are accepted as corners (including the falsely detected ones on the edges of the image). Therefore the selection of the threshold should be neither too high not to discard correctly detected corners, nor too low not to accept edge points as corners. With k parameter the difference between the images was not as obvious as in the previous case. However, it can be seen that with lower k some false corners are detected on the edges, although a higher k misaligned some of the real corners. As for σ, the number of false corners detected on the vertical edge of the image was much higher for a specific σ value than in other cases ($\sigma = 0.75$) and for $\sigma = 0.5$ the number of false detections on the horizontal edge was much lower than in other cases. Additionaly, it seems that a higher sigma allows for better detection of "bigger" corners (more eye-catching ones), while with a lower sigma more "smaller" corner are detected.

For feature matching, I selected $k = 0.055$, $\sigma = 1.25$ and $T = 1e-5$.



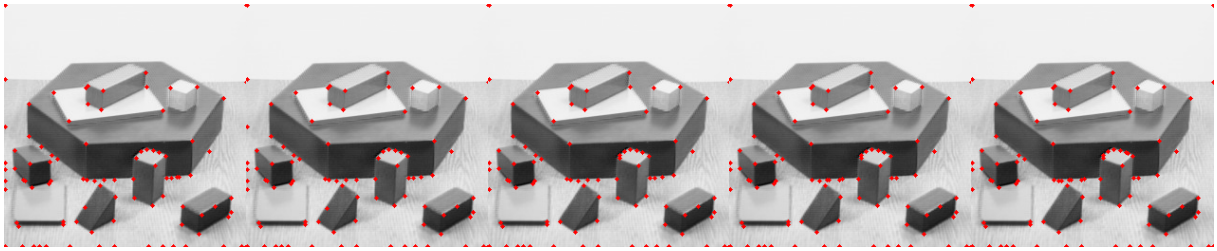
((a)) $T = 1e-06$ ((b)) $T = 1e-05$ ((c)) $T = 5e-05$ ((d)) $T = 1e-04$ ((e)) $T = 5e-04$

Figure 1: Result of running the program on blocks image for varying values of threshold T, $\sigma = 1.0$, $k = 0.05$.



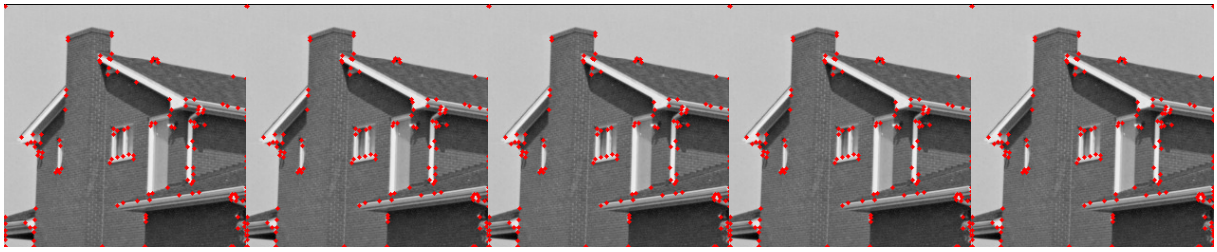
((a)) $T = 1e-06$ ((b)) $T = 1e-05$ ((c)) $T = 5e-05$ ((d)) $T = 1e-04$ ((e)) $T = 5e-04$

Figure 2: Result of running the program on house images for varying values of threshold T, $\sigma = 1.0$, $k = 0.05$.



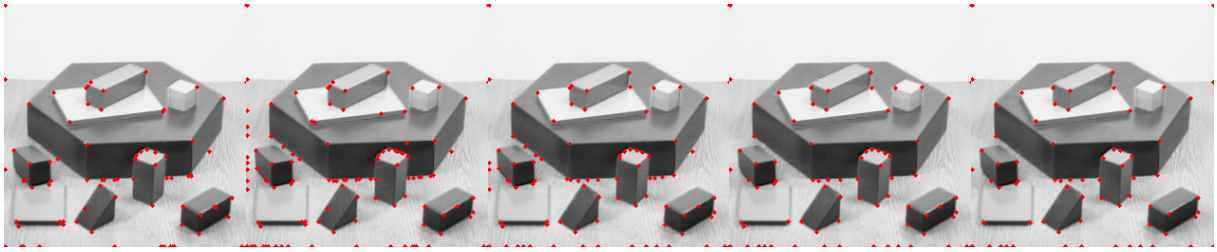
((a)) $k = 0.04$ ((b)) $k = 0.045$ ((c)) $k = 0.05$ ((d)) $k = 0.055$ ((e)) $k = 0.06$

Figure 3: Result of running the program on the blocks image for varying values of parameter k, $\sigma = 1.0$, $T = 1e-5$.



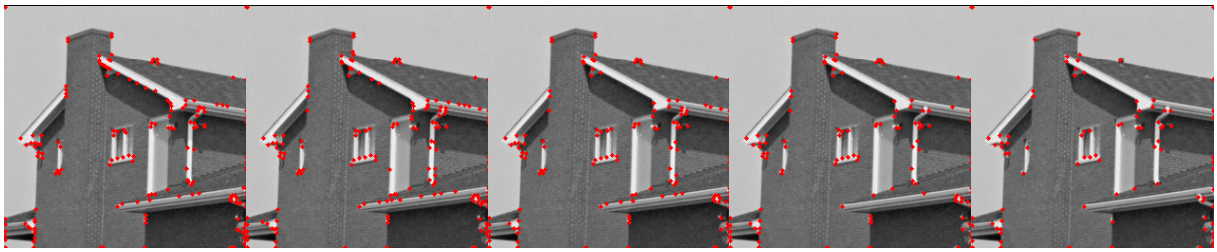
((a)) $k = 0.04$ ((b)) $k = 0.045$ ((c)) $k = 0.05$ ((d)) $k = 0.055$ ((e)) $k = 0.06$

Figure 4: Result of running the program on the house image for varying values of parameter k, $\sigma = 1.0$, $T = 1e-5$.



((a)) $\sigma = 0.5$ ((b)) $\sigma = 0.75$ ((c)) $\sigma = 1.0$ ((d)) $\sigma = 1.25$ ((e)) $\sigma = 1.5$

Figure 5: Result of running the program on the blocks image for varying values of parameter σ, $k = 0.05$, $T = 1e - 5$.



((a)) $\sigma = 0.5$ ((b)) $\sigma = 0.75$ ((c)) $\sigma = 1.0$ ((d)) $\sigma = 1.25$ ((e)) $\sigma = 1.5$

Figure 6: Result of running the program on the house image for varying values of parameter σ, $k = 0.05$, $T = 1e - 5$.

2 Description & Matching

2.1 Implementation

2.1.1 Local descriptors

The task was to filter out the keypoints that were too close to the edge. If we define our variable $patch = 9$, then we want out keypoint to be separated from the edge by at least four other pixels in all directions. On this basis, we can say that the x coordinate needs to be at least equal to $patchsize//2$ and smaller or equal to $width - 1 - patchsize//2$. The similar condition applies to y, only with the replacement of width with height.

The code implementation of this part is presented below:

```
filtered = []
height = img.shape[0]
width  = img.shape[1]
for [x, y] in keypoints:
    if (x >= patch_size//2) and (x <= (width - 1 - patch_size//2)) and (y >=
        patch_size//2) and (y <= (height - 1 - patch_size//2)):
         filtered.append([x, y])

filtered = np.array(filtered)
```

2.1.2 SSD one-way nearest neighbors matching

The next step was to compute the sum-of-squared-differences (7). To avoid utilizing for-loop for vectorized computations, I create two auxiliary matrices $Q1$, $Q2$. I perform operations on them so that $Q1$ contains respective elements $q1[i]$ of descriptor nr 1 with i equal to the current index of row. $Q2$ is created on a similar idea, however the index i of $q2[i]$ depends on the index of the column. Once I have these two matrices, I perform subtraction and then multiplication to perform the computation of $(q1[i][j] - q2[i][j])^2$ for each element (i, j). Once it is done, I sum up the elements of each 81-element-long vector in matrix element on position (i, j). This is how the final distances matrix is obtained.

$$SSD(p, q) = \sum_i (p_i - q_i)^2 \tag{7}$$

The code implementation of this part is presented below:

```
Q1 = np.repeat(desc1, desc2.shape[0], axis=0)
Q1 = np.reshape(Q1, (desc1.shape[0], desc2.shape[0], desc2.shape[1]))

Q2 = np.repeat(desc2, desc1.shape[0], axis=0)
Q2 = np.reshape(Q2, (desc2.shape[0], desc1.shape[0], desc2.shape[1]))
Q2 = np.transpose(Q22, (1, 0, 2))

diff = Q1–Q2
mult = np.multiply(diff, diff)
distances = np.sum(mult, axis=2)
```

To find the closest neighbor one-way, for each row of the distances matrix I find the column index of the smallest value in the row. The pair defines the matched keypoints.

The code implementation of this part is presented below:

```
distances = ssd(desc1, desc2)
q1, q2 = desc1.shape[0], desc2.shape[0]
matches = []
if method == "one_way":
    for i in range(0, q1):
        j = np.argmin(distances[i])
        matches.append([i, j])
...
matches = np.array(matches)
```

2.1.3 Mutual nearest neighbors / Ratio tests

To find mutual neighbors, I first perform the one-way closest neighbor. Then, I take the found column index and look for the row index corresponding to the minimum value of the aforementioned column. If it is equal to the row index we initially performed the search for, then we know it is a mutual neighborhood.

When the one-way closest neighbor is found, we can additionally select the column corresponding to the found column index, sort it and check the ratio between the first two values. If the ratio is smaller than the predefined threshold, we add the initial indices to the final matching keypoints list.

The code implementation of this part is presented below:

```
distances = ssd(desc1, desc2)
q1, q2 = desc1.shape[0], desc2.shape[0]
matches = []
if method == "one_way":
    ...
elif method == "mutual":
    for i in range(0, q1):
        j = np.argmin(distances[i])
        x = np.argmin(distances[:,j])
        if x == i:
            matches.append([i, j])
elif method == "ratio":
    for i in range(0, q1):
        j = np.argmin(distances[i])
        sorted = np.sort(distances[i])
        if sorted[0] / sorted[1] < ratio_thresh:
            matches.append([i, j])
matches = np.array(matches)
```

2.2 Results

The corner detections on both images after the keypoint filtering are presented in Fig. **??**.

((a)) First image ((b)) Second image

Figure 7: Harris detector keypoints after edge filtering.

On that basis, the neighbor matching was performed. We can see differences between one-way (Fig. 8), mutual (Fig. 9) and one-way with ratio check (Fig. 10) approaches. Mutual matching gives more certain results than one-way one – fewer keypoints are incorrectly matched, because we make sure that the minimal distance from keypoint a to b is also a minimal distance from b point perspective. However, if we want to perform only one-way check, then ratio test can help us achieve a better accuracy, as we check if the second closest neighbor can be easily mistaken with the first one (we compare the ratio between those two values to a predefined threshold). If however the threshold is too high or too low, we can incorrectly discard or accept keypoint matches.

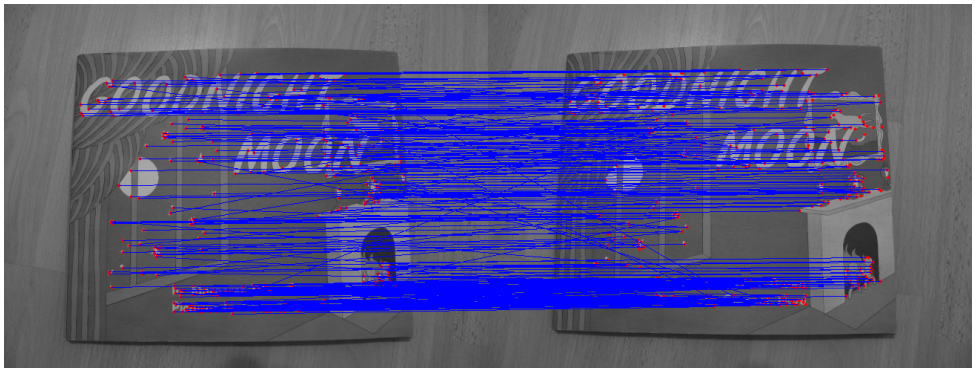


Figure 8: One-way closest neighbor matching.

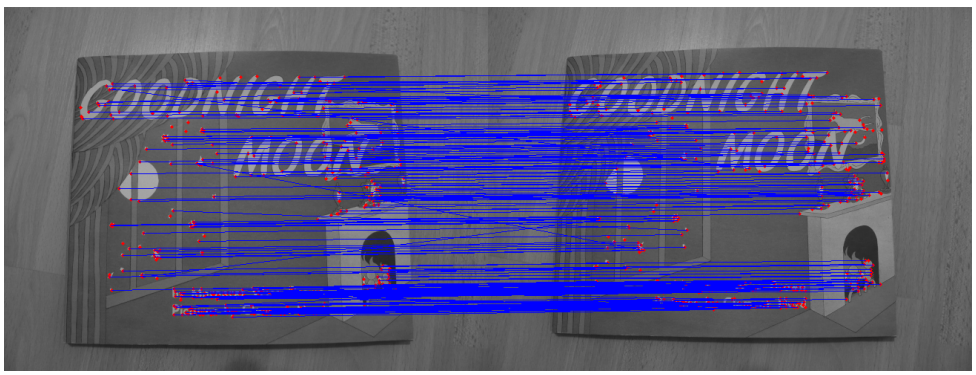


Figure 9: Mutual closest neighbor matching.

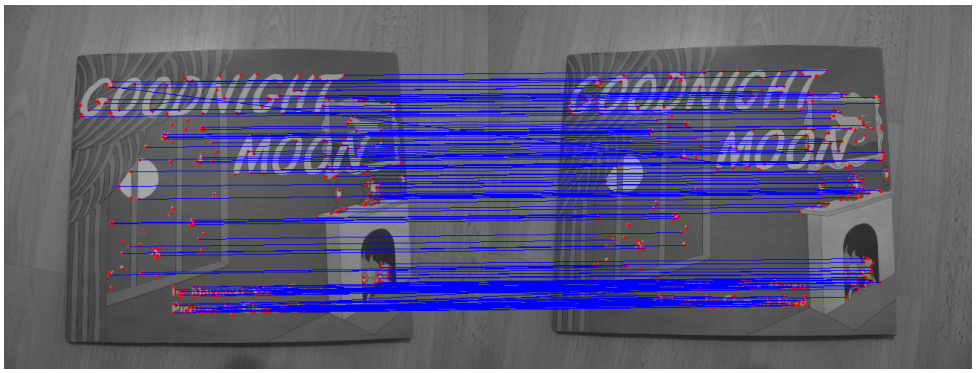


Figure 10: One-way closest neighbor matching with a ratio test.