# Lab Assignment Report - Local Features

Yaqi Qin*

October 2022

## 1 Detection

### 1.1 Image gradients

The image gradients $I_x$ and $I_y$ in the $x$ and $y$ directions in this case can be represented by 2D convolution with a $1 \times 3$ kernel $K_x$ and a $3 \times 1$ kernel $K_y$ respectively, i.e.,

$$I_x(i,j) = \sum_{m=0}^{0}\sum_{n=-1}^{1} K_x(-m,-n)I(i+m,j+n),\ where\, K_x = \begin{bmatrix}0.5 & 0 & -0.5\end{bmatrix}$$

$$I_y(i,j) = \sum_{m=-1}^{1}\sum_{n=0}^{0} K_x(-m,-n)I(i+m,j+n),\ where\, K_y = \begin{bmatrix}0.5\\ 0\\ -0.5\end{bmatrix}$$

Accordingly, the convolution is realized using the scipy.signal.convolve2d function. Here the 'boundary' is set to be 'symm' to reproduce boundary conditions and avoid getting huge gradients at boundaries due to 0 padding.

```
Ix = signal.convolve2d(img, [[0.5, 0, -0.5]], boundary = 'symm', mode = 'same')
Iy = signal.convolve2d(img, [[0.5], [0], [-0.5]], boundary = 'symm', mode = 'same')
```

### 1.2 Local auto-correlation matrix

Instead of separately computing the local correlation matrix for each pixel, each of the 3 elements of the matrix (weighted sum of square of $I_x$ and $I_y$, as well as the product of $I_x$ and $I_y$) is computed for all pixels at once. The weighting of the neighborhood is realized by the GaussianBlur function as follows. The kernel size is set as (0 0), so that it could be computed from the given sigma values.

```
Ixx = cv2.GaussianBlur(Ix * Ix, ksize = (0, 0), sigmaX = sigma,
                       sigmaY = sigma, borderType=cv2.BORDER_REPLICATE)
Iyy = cv2.GaussianBlur(Iy * Iy, ksize = (0, 0), sigmaX = sigma,
                       sigmaY = sigma, borderType=cv2.BORDER_REPLICATE)
Ixy = cv2.GaussianBlur(Ix * Iy, ksize = (0, 0), sigmaX = sigma,
                       sigmaY = sigma, borderType=cv2.BORDER_REPLICATE)
```

With these 3 matrices computed, we can reconstruct the local auto-correlation matrix $M_p$ for any given pixel $p_{ij}$ as:

$$M_p = \begin{bmatrix}I_{xx}(i,j) & I_{xy}(i,j)\\ I_{xy}(i,j) & I_{yy}(i,j)\end{bmatrix}$$

*Student ID: 21-946-819

### 1.3 Harris response function

For each pixel $p_{ij}$, the Harris response function $C_{ij}$ has a closed form solution:

$$\begin{aligned} C_{ij} &= det(M_p) - kTr^2(M_p) \\ &= I_{xx}(i,j)I_{yy}(i,j) - I_{xy}(i,j)^2 - k\left[I_{xx}(i,j) + I_{yy}(i,j)\right]^2 \end{aligned}$$

Since the related operations are all calculated element wise, so we can simply utilize element wise matrix operation to get the result for all pixels, i.e.,

```
det = Ixx * Iyy - Ixy ** 2
trace = Ixx + Iyy
C = det - k * trace ** 2
```

### 1.4 Detection criteria

For picking corner points, I first calculated the maximum C value in each $3 \times 3$ neighborhood using the ndimage.maximum_filter function. And then, a corner point is picked if its Harris response is greater than the predefined threshold and at the same time, is the local maximum. Here the output indices are reversed for later visualization using openCV.

```
local_max = ndimage.maximum_filter(C, size = 3)
corners = np.argwhere(np.logical_and(C > thresh, C == local_max))
return np.stack((corners[:, 1], corners[:, 0]), axis = -1), C
```

### 1.5 Result analysis

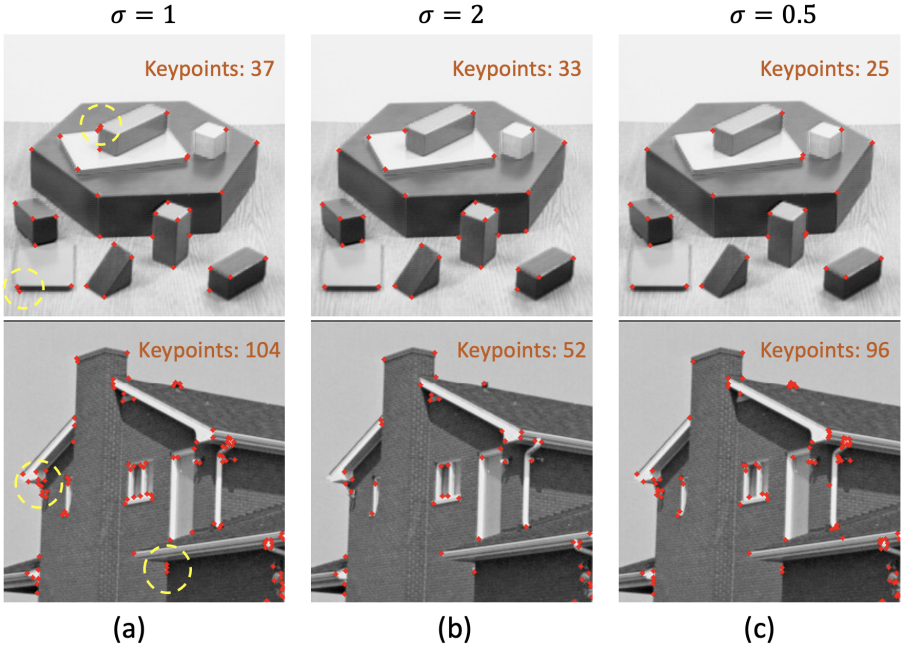


Figure 1: The Harris detection results on the block image (the first row) and the house image (the second row) with $\sigma = 1$ (a), $\sigma = 2$ (b) and $\sigma = 0.5$ (c). In all 6 images, $k = 0.05$ and $T = 1e - 5$ are fixed for fair comparison.

**Varying the standard deviation $\sigma$ of the Guassian window:** To analyze the influence of $\sigma$ on the final detection result, I first fix the $k = 0.05$ and $T = 1e - 5$, and only vary $\sigma$ for comparison. The results are shown in Figure 1.

We can see that in the cases where $\sigma$ is relatively small, closed points within the same corner region (like the yellow mark regions in Figure 1-a) might be redundantly selected as key points. The same holds true for $\sigma = 0.5$ (Figure 1-c). When $\sigma$ increases to 2, the detection results are more clean in that the number of redundant points in proximity is diminished. The reason could be that when $\sigma$ increases, we use broader neighborhood to account for the image intensity change of one pixel, so that some local peaks of the gradients can be smoothed out.

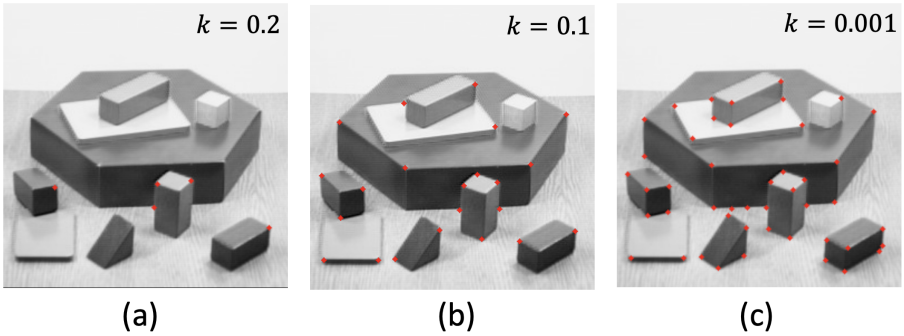


Figure 2: The Harris detection results on the block image with $k = 0.2$ (a), $k = 0.1$ (b) and $k = 0.001$ (c). In all images, $\sigma = 2$ and $T = 1e - 5$ are fixed for fair comparison.

**Varying the Harris constant $k$:** In this case, I keep $\sigma = 2$ and $T = 1e - 5$ fixed to see the effect of varying $k$. Since varying $k$ within the suggested range (0.06 to 0.04) yields pretty similar results, I tuned it to lower (0.001) and higher magnitude (0.1, 0.2) to allow for more significant changes. The results are shown in Figure 2. As $k$ grows larger, more corners are lost. As it decreases drastically to 0.001, many edge points are detected as corners. Generally, $k = 0.05$ (Figure 1-b) yields more balanced results.

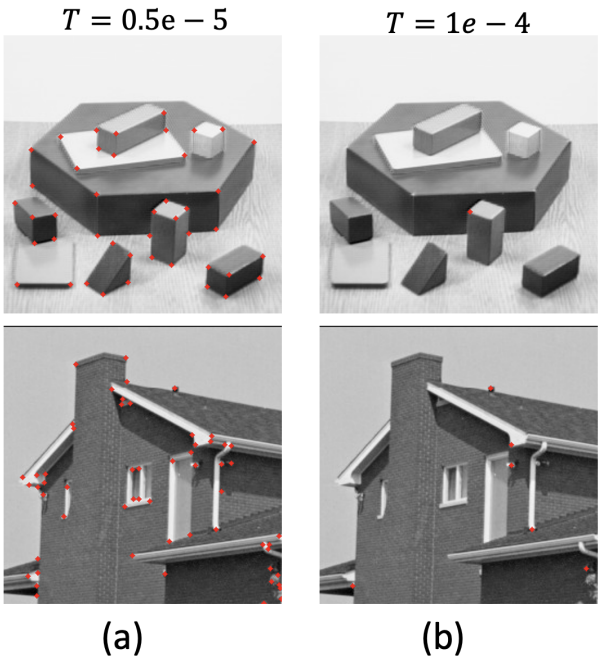


Figure 3: The Harris detection results with $T = 0.5e - 5$ (a) and $T = 1e - 4$ (b). In all images, $\sigma = 2$ and $k = 0.05$ are fixed for fair comparison.

**Varying the threshold $T$:** As illustrated in the first two cases, $\sigma = 2$ and $k = 0.05$ yields a relatively better results. So I keep them fixed to find the optimal $T$. As shown in Figure 3, increasing T in a small range can cause severe loss of corners (Figure 3-b). In this case, subtle

decreasing T to $0.5e-5$ yields the optimal results, where most corners can be detected.

**Discussion** In general, most corners can be correctly detected with $\sigma = 2$, $k = 0.05$ and $T = 0.5e-5$. However, there are still some corners, like the ones from the hexagonal pyramid at the backward side in the block image, failed to be detected in all tests. Besides, the performance of Harris detection relies heavily on hyperparameters which could be problematic.

# 2 Description & Matching

## 2.1 Local descriptors

To filter out the keypoints that are too close to the image boundaries, the 2D indices of key points are transformed as 1D flattened indices, where for key point $(i, j)$, the flattened indices can be constructed as:

$$I(i, j) = w * j + i$$

For a $9 \times 9$ patch size, the largest offset (the bottom right corner and the upper left corner) in the flattened array are $\pm w * 4 + 4$. Thus, to avoid out-of-bounds issues, the flattened indices of key points should still within the range of $[0, hw - 1]$ when the largest offset is added.

```
def filter_keypoints(img, keypoints, patch_size = 9):
    # Filter out keypoints that are too close to the edges
    h, w = img.shape
    offset = int(np.floor(patch_size / 2.0))
    flattened_kp = keypoints[:, 1] * w + keypoints[:, 0]
    up_bound = flattened_kp + offset * w + offset < h * w
    low_bound = flattened_kp - offset * w - offset >= 0
    return keypoints[np.logical_and(up_bound, low_bound)]
```

## 2.2 SSD one-way nearest neighbors matching

To compute the SSD between the descriptors of all features from the first image and the second image, matrix subtraction using broadcast can be utlized to fasten computation, by augmenting the first descriptors matrix $desc1$ to be the size $q1 \times 1 \times 81$, and the second descriptors matrix $desc2$ to be the size $1 \times q2 \times 81$. Then by applying matrix subtraction, each descriptor from $desc1$ can be broadcasted to all descriptors from $desc2$. Then SSD is just the square of the l2 norm of the resulting matrix.

```
def ssd(desc1, desc2):
    assert desc1.shape[1] == desc2.shape[1]
    pair_wise_dist = np.linalg.norm(desc1[:, None, :] - desc2[None, :, :], axis=-1)
    return pair_wise_dist ** 2
```

After getting the distance matrix, one-way nearest neighbor matching is simply finding the column index of the minimum distance of each row.

```
if method == "one_way": # Query the nearest neighbor for each keypoint in image 1
        matches = np.stack((np.arange(q1), np.argmin(distances, axis = 1)), axis = -1)
```

The results are shown in Figure 4, where most of the 218 matching pairs seem to be correct, except for a few crossing lines that mistakenly match points with similar patch features at very different locations. Thus, this matching method might be not so accurate in that it focuses more on the local pattern without taking the relative layout into account.

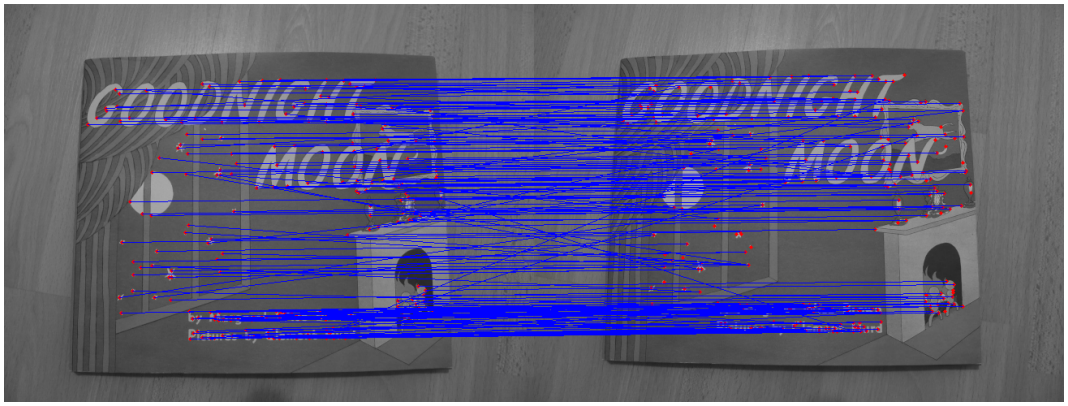


Figure 4: The one-way nearest neighbors matching results with 218 matching pairs.

### 2.3 Mutual nearest neighbors / Ratio test

For the mutual nearest neighbors matching, one more one-way matching is conducted for the second image. The matching pairs are just the intersection of the two matching results. For the ratio test matching, just use the np.partition function to get the first and second nearest neighbor and then compare their value ratio.

```
matches = np.stack((np.arange(q1), np.argmin(distances, axis = 1)), axis = -1)
elif method == "mutual":#implement the mutual nearest neighbor matching
        mutual_mask = np.argmin(distances, axis = 0)[matches[:, 1]] == matches[:, 0]
        matches = matches[mutual_mask]
elif method == "ratio":# implement the ratio test matching
    # get all the first and second neighbors for each kp in img1
    NN_2 = np.partition(distances, (0, 1), axis = 1)[:, : 2]
    valid_mask = NN_2[:, 0] / NN_2[:, 1] < ratio_thresh
    matches = matches[valid_mask]
```

The matching results are as follows. From the number of matching pairs, ratio test (118 matching pairs) is apparently more strict than the mutual test (154 matching pairs). With a double check in the mutual matching, mismatch with points at very different locations (non-horizontal lines) can be almost avoided. The same holds true for the ratio test case. Most mismatches are among those tiny stars in the left region with almost the same surroundings (nothing but gray blocks). Such cases are hard to avoid as long as the matching algorithms focus only on the tiny patch around the pixel, and do not care about the global layout.

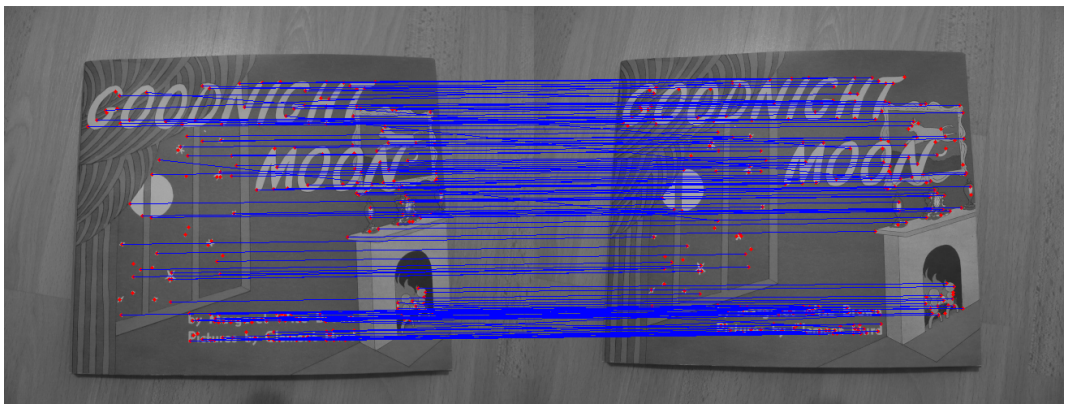


Figure 5: The mutual nearest neighbors matching results with 154 matching pairs left.

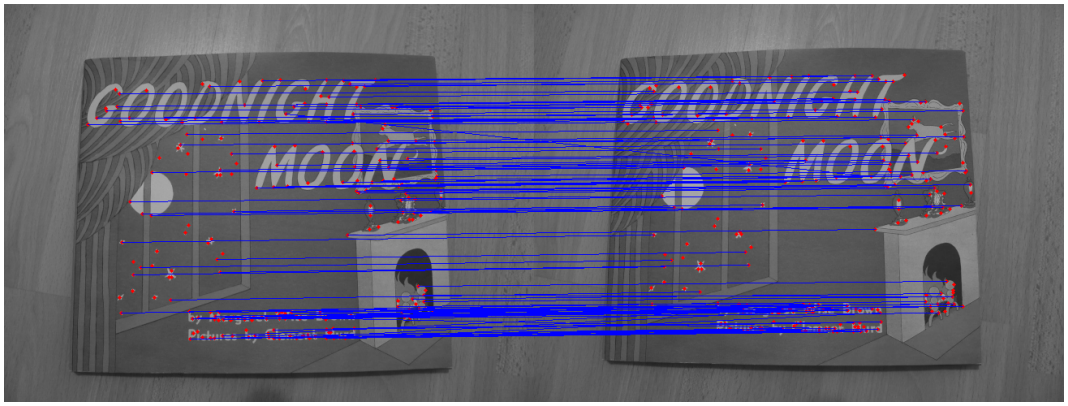


Figure 6: The ratio test matching results with 118 matching pairs left.