ETH Zürich
Eidgenössische Technische Hochschule Zürich

Computer Vision
Department of Computer Science

Lab 1:
Feature extraction and Matching

Christos Dimopoulos - 23-941-834
cdimopoulos@student.ethz.ch

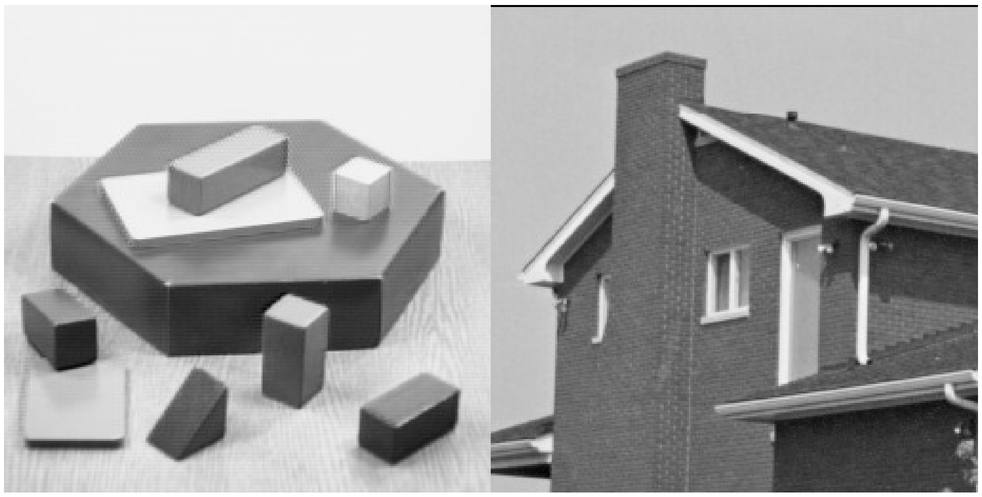
October, 2023

Contents

Task 1: Harris Corner Detection

In the first part of the lab exercise, we practice with the Harris corner detection algorithm in two different input images. The entire procedure can be summarised on the following steps:

- Compute intensity gradients in x and y direction
- Blur Images to get rid of noise
- Compute Harris response
- Thresholding and non-maximum suppression



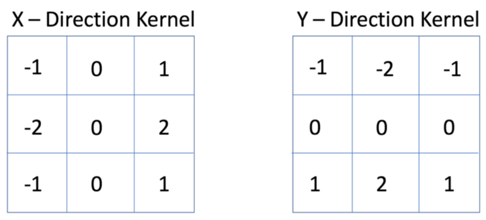
(a) (b)

Figure 1: The two input grayscale images (a) Blocks, (b) House.

We analyze each step in our code implementation below. The initial parameter values are *sigma=1*, *k=0.05* and *threshold=1e-5*.

1. Image Gradients Calculation

First of all, the gradients of each input image are computed. For this purpose, we define the Sobel operators as kernels, with which we then convolve the original images, using the command ***signal.convolve2d*** with parameters *mode='same', boundary='symm'*. This way, we get the image gradients for both the x and y direction.



X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

Figure 2: Sobel operators for Image Gradient Calculation.

2. Image Gradient Blurring

Next up, we use Gaussian kernels to blur the recently computed image gradients. Blurring reduces the "noise" (randomness) in the picture, and allows us to find edges and determine image gradients more accurately. In order to do so, we use the following orders:

```
blurred_gradient_x = cv2.GaussianBlur(gradient_x, (0, 0),
    sigmaX=sigma,sigmaY=sigma, borderType=cv2.BORDER_REPLICATE)
blurred_gradient_y = cv2.GaussianBlur(gradient_y, (0, 0),
    sigmaX=sigma,sigmaY=sigma, borderType=cv2.BORDER_REPLICATE)
```

3. Computation of auto-correlation matrix M

Up next, we compute the elements of the local auto-correlation matrix M as following:

$$M = \sum_{(x,y)\in W} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} (1)$$

For the windowing function W, we opt for a Gaussian Kernel of the same sigma defined in the beginning for both x and y directions, using once again the *cv2.GaussianBlur* command.

4. Computation of Harris response function R

Having computed the auto-correlation matrix, we then compute its eigenvalues using the following formula:

$$\lambda_+ = \frac{1}{2}(M_{xx} + M_{yy} + \sqrt{M_{xx} - M_{yy} + 4M_{xy}^2}) \tag{2}$$

$$\lambda_- = \frac{1}{2}(M_{xx} + M_{yy} - \sqrt{M_{xx} - M_{yy} + 4M_{xy}^2}) \tag{3}$$

Having computed the eigenvalues, we easily compute the determinant and trace of matrix M as the product and sum of eigenvalues respectively. Finally, the Harris response function is computed as following:

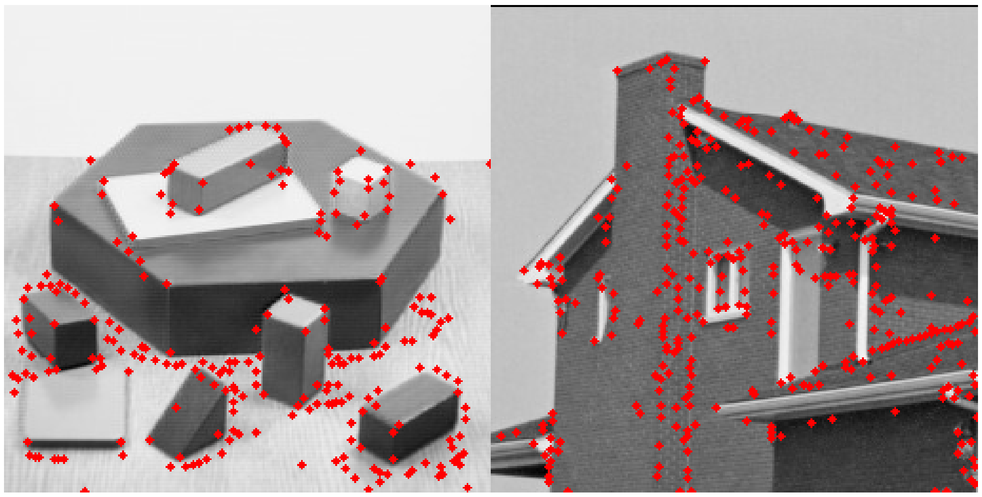
$$R = det(M) - ktrace^2(M) \tag{4}$$

where k is the constant defined in the beginning.

5. Detection with threshold and Non-maximum Suppression

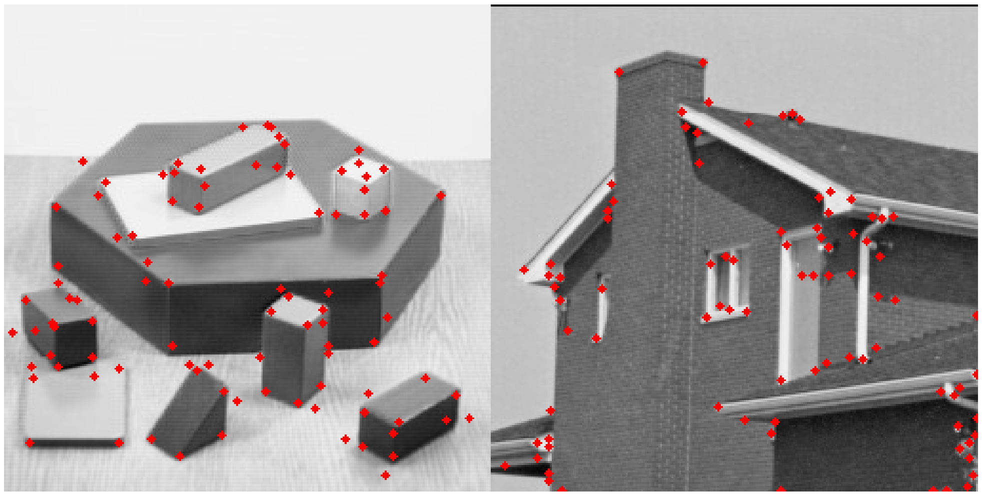
Finally, the last step involves thresholding the response function over a specific initial value (Condition 1) and at the same time applying nonmaximum suppression over a 3x3 pixel neighbourhood (Condition 2). For the second condition we use the command *scipy.ndimage.maximum_filter*.

Below, we visualize the detected edges for the initial values of our parameters sigma, threshold and k. One easily notices that although many detected points correspond to real edges, lots of them don't, since points in the floor and walls are also detected. To tackle this we problem we experiment by augmenting the sigma value to 2. The detected edges seem to be chosen much better after this change.



(a) (b)

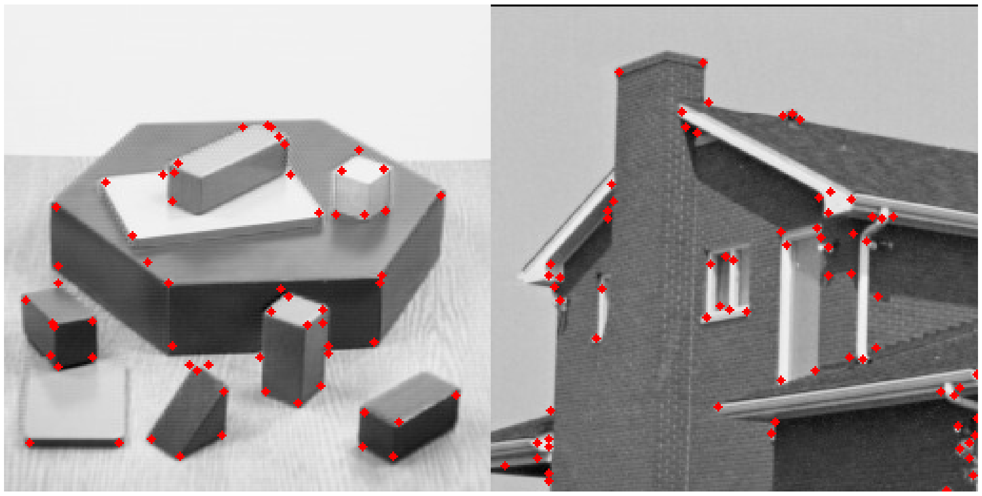
Figure 3: Detected edges for ***sigma=1, k=0.05* and *threshold=1e-5*** (a) Blocks, (b) House.



(a) (b)

Figure 4: Detected edges for ***sigma=2, k=0.05* and *threshold=1e-5*** (a) Blocks, (b) House.

To further enhance the result and get rid of noisy detections of non edges, we further increase the threshold value to **1e-4**. As a result, fewer points are kept as detected edges, thus discarding noisy results. This combination of parameter values gives the best results so far, detecting real edge points with minimum noise, and is thus preferred from now on.



(a) (b)

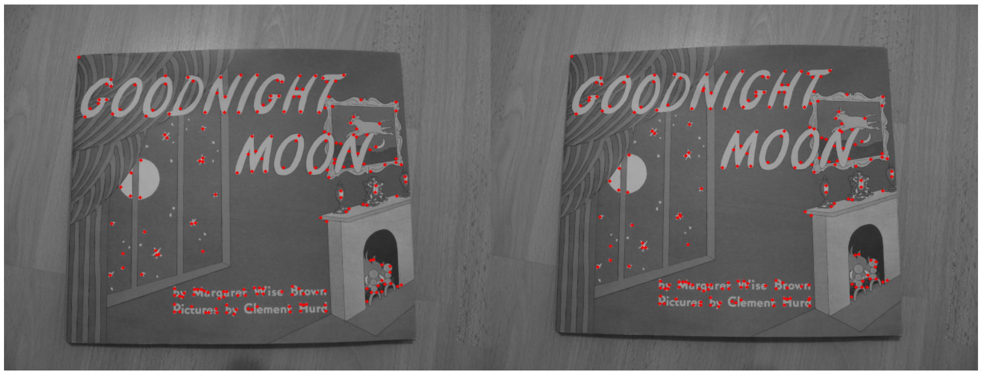
Figure 5: Detected edges for ***sigma=2, k=0.05*** **and** ***threshold=1e-4*** (a) Blocks, (b) House.

Description and Matching

In the second part of the lab exercise, having created the basis on feature extraction with the Harris detector, we experiment with several image matching techniques. As input images, we use two images, I1 and I2, of the same child book from two different viewpoints. The values of the parameters used are initialized as following: *sigma=2, k=0.05* , *harris_threshold=1e-4* and *ratio_threshold=0.5*.

In general, each matching algorithm can be summarized into the following steps:

1. **Feature extraction and Description**: Using the Harris detector that was developed in the first part, edges are extracted as features for both images. To avoid noisy detections, we implement the function *filter_keypoints* that discards detected edges that are too close to the image borders (by a patch of 9 pixels). Each detected feature is then described as a numerical representation, that encodes information about the region around the feature (descriptors). This is implemented by the given function *extract_patches*.



(a) (b)

Figure 6: Detected edges for ***sigma=2, k=0.05*** **and** ***threshold=1e-4*** (a) I1, (b) I2.

2. **Descriptor Comparison**: Next up, the sum of squared distances between the descriptors of the features of the first and second image respectively is computed as a measure of dissimilarity. For

this purpose, we could implement a function that utilizes two nested for loops, but this would deem to be computationally expensive. Instead, we utilize the power of numpy and opt for the command *np.expand_dims* to add singleton dimensions so that broadcasting can be applied.

```python
def ssd(desc1, desc2):
    '''
    Sum of squared differences
    Inputs:
    - desc1:    - (q1, feature_dim) descriptor for the first image
    - desc2:    - (q2, feature_dim) descriptor for the second image
    Returns:
    - distances: - (q1, q2) numpy array storing the squared distance
    '''
    assert desc1.shape[1] == desc2.shape[1]
    diff = desc1[:, np.newaxis, :] - desc2
    distances = np.sum(diff**2, axis=-1)
    return distances
```

3. **Corresponding Features**: The final step involves the core of image matching by corresponding features and depends on the algorithm that we ultimately choose to use. We experiment with three different approaches: One way Nearest Neighbours Matching, Mutual Nearest Neighbours Matching and Ratio Test Matching.

One way Nearest Neighbours Matching

In this approach, each feature from the first image is matched to the feature of the second image that is the closes (nearest neighbour), that is has the smallest sum of square distance. As such, we use *np.argmin* to find the indices of the smallest distance value for each feature descriptor and stack them together using *np.vstack*. The matching result seems satisfying enough, with only a few detected features being corresponded wrong.

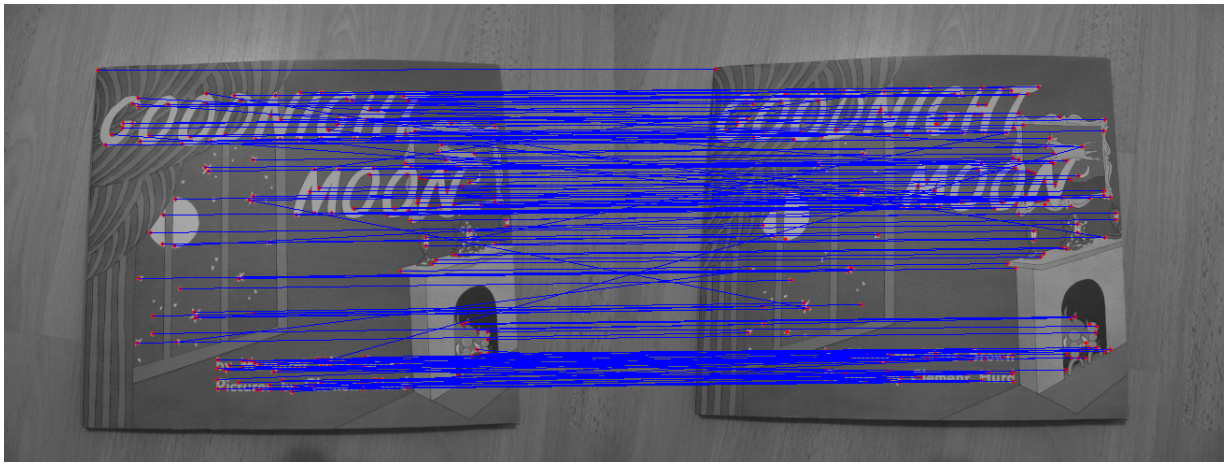


Figure 7: One way Nearest Neighbours Matching.

Mutual Nearest Neighbours Matching

In the second approach, we use the same algorithm with one way nearest neighbour matching, but only consider valid pairs those that remain if we were to switch the two images. To achieve that, we first compute the sum of squared distances for both permutations of images and then use a mask to keep the valid matching pairs.

```
min_idx1 = np.argmin(distances, axis=1)
min_idx2 = np.argmin(distances[:,min_idx1], axis=0)
valid_pairs = min_idx2==np.arange(q1) # use mask
matches = np.vstack((np.arange(q1)[valid_pairs], min_idx1[valid_pairs])).T
```

Observing the result, we see that the image matching is more successful with almost no wrong correspondences between feature descriptors of the two images, beating all other algorithms.

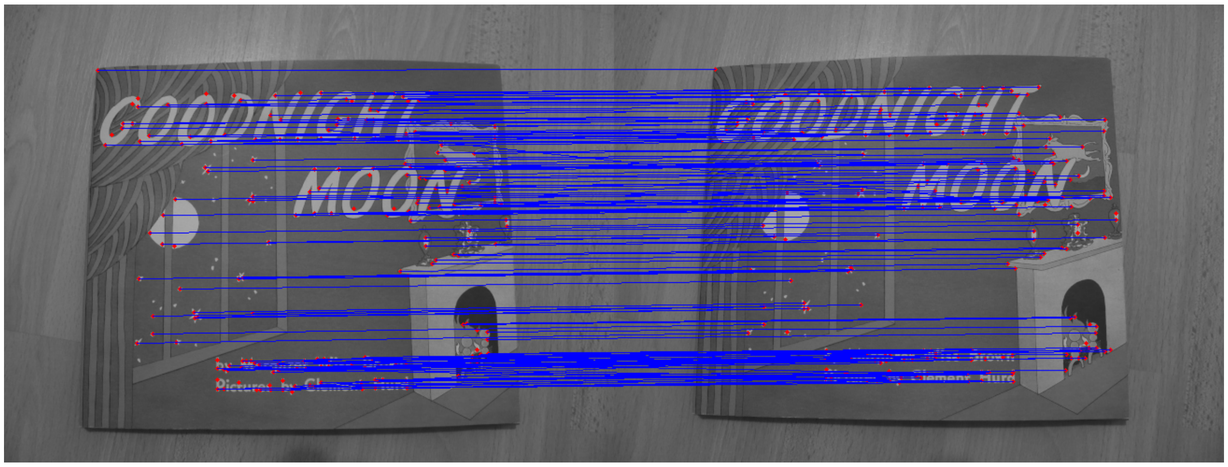


Figure 8: Mutual Nearest Neighbours Matching.

Ratio Test Matching

Lastly, in ratio test matching, we use one way nearest neighbour matching, but only keep valid matches if the ratio between the smallest and the second smallest sum of squared distances is below a specific threshold value (here 0.5). In order to calculate the indices of the two smallest distances we use the command *np.argpartition(distances, 2, axis=1)[:, :2]*. Although some wrong correspondences occur, it is still a pretty satisfying result, better than the simple one way nearest neighbour matching algorithm.

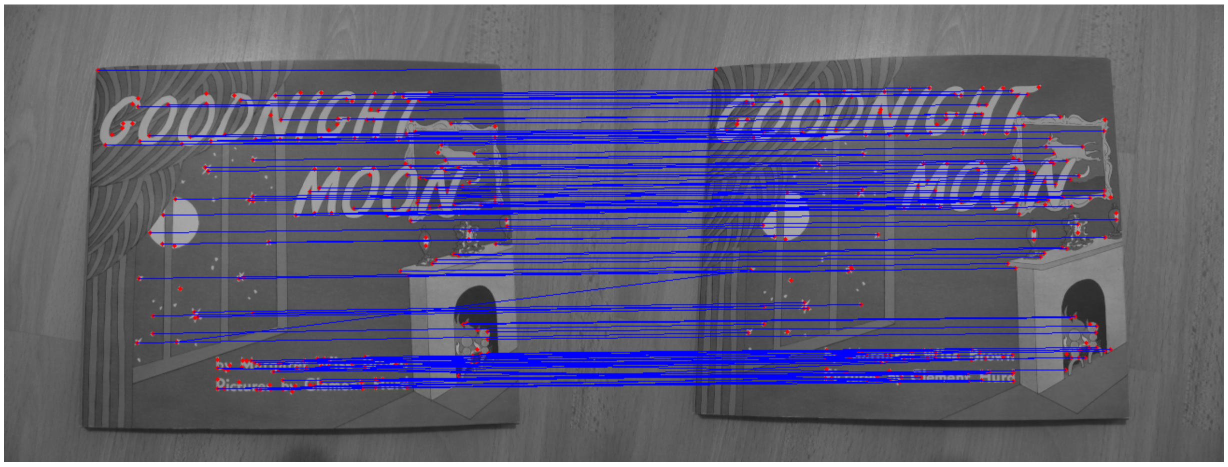


Figure 9: Ratio Test Matching.