Local Features Report

Nicole Damblon

Computer Vision

1 Feature Detection

The goal of this assignment is to detect *corners* in an image by computing the Harris Response C for each pixel ij of a given image. A pixel is considered a *corner*, when the following two conditions hold: The Harris Response $C(i, j)$

1. is greater than a given threshold
2. has the the maximal response in the center of a $3x3$ patch. This process is called non-maximum suppression.

First, the corners are detected which represent the points of interest in an image. Second, the retrieved corners are matched to the independently detected corners of a second image, according to 3 different protocols.

Image Gradients. To obtain the Auto-Correlation Matrix, first the image gradients I_x, I_y with respect to x and y are computed using a convolution:

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

Auto-Correlation Matrix. The gradients I_x, I_y are smoothed by applying a gaussian filter with standard deviation σ, yielding the elements of the local auto-correlation matrix M:

$$M = \sum \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix} \tag{2}$$

Harris Response Function. For each pixel of the image the Harris Response C is computed, where k is a hyperparameter:

$$C(i,j) = det(M_{ij}) - k * trace(M_{ij})^2 \tag{3}$$

Corner Detection. The value of the Harris response C represents the corner strength of each pixel. If the value is above a certain threshold T and is the maximum value in its local neigborhood, it is considered a corner. Fig. 1 shows the implementation of the corner detection. The function

`scipy.ndimage.maximum_filter` slides a $3x3$-window over the response image C and overwrites the non-maximum values in each patch with the local maximum. The maximum values are saved in $CMax$. If $C(i, j)$ is greater than a threshold, which is a hyperparameter, and corresponds to the maximum value in $CMax$, the pixel position is saved as a corner.

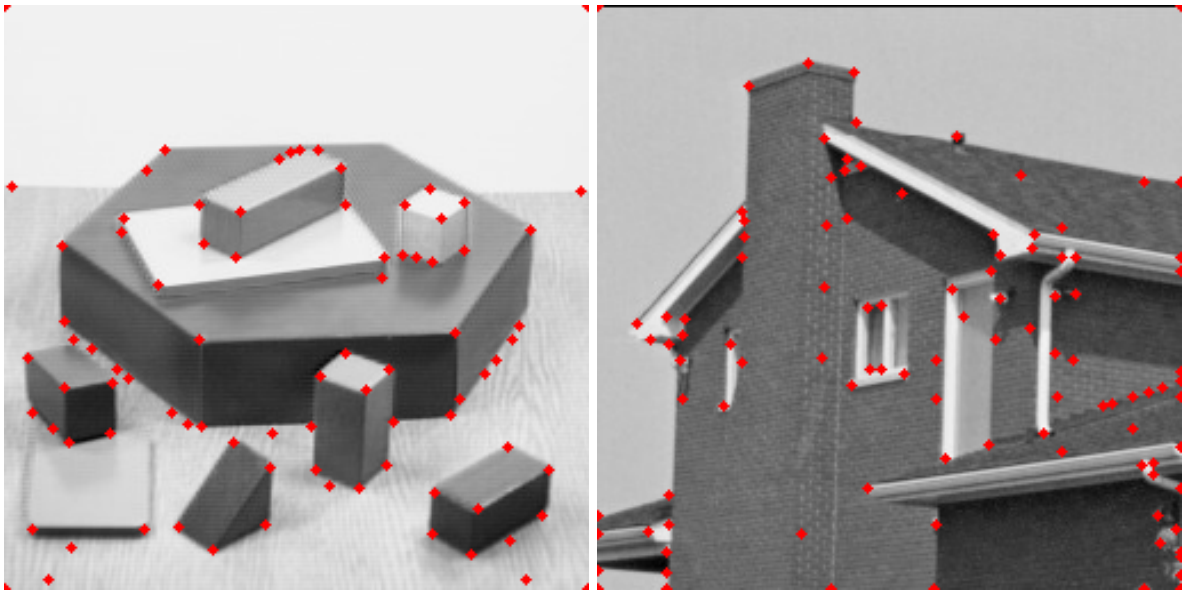
The following figures demonstrate the resulting corners when varying the hyperparameters. Fig. 2 shows an overestimation of the detector. If the threshold is too small $T = 1e - 5$ too many corners are detected, including keypoints along the edges.

The hyperparameters in fig.3 lead to an underestimation of the detector. In the left subfigure, no edges are misclassified as corners due to the greater threshold, but many corners are missing. However, the house image on the right side displays too many corners in small clusters, including a few edges.

Fig. 4 yields the best results. Both images are missing a few corners, but no edges are classified and most of the relevant corners are detected. This parameters will be used for the feature matching.

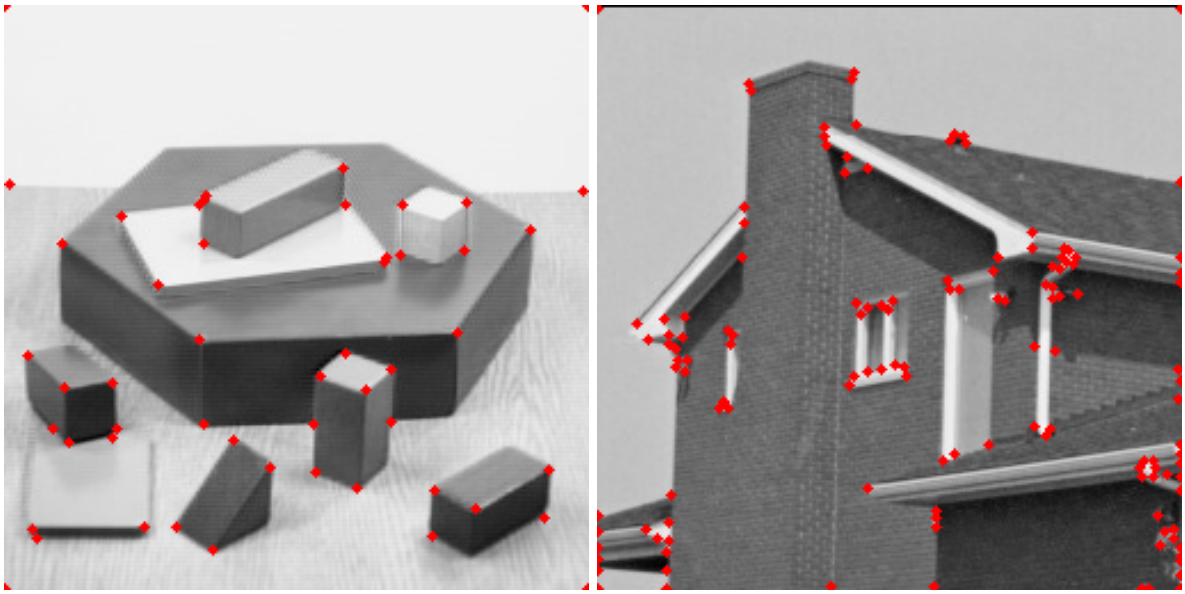
```
# slides 3x3 window over response img C and
# saves maximum pixels and overwrites non maximum pixel with max in CMax
CMax = scipy.ndimage.maximum_filter(C, size=(3,3))

corners = np.array([[]])
x = np.array([])
y = np.array([])
for rows in range(len(C)):
    for cols in range(len(C)):
        # check if response is greater than thresh and if it
        # corresponds to the max value in 3x3 window
        if C[rows,cols] > thresh and C[rows,cols] == CMax[rows,cols]:
            x = np.append(x,cols)
            y = np.append(y,rows)
x = x.reshape(len(x),1)
y = y.reshape(len(y),1)
corners = np.hstack((x,y))
corners = corners.astype(int)
return corners, C
```

Fig. 1. Corner Detection: Testing if the Harris Response C exceeds a threshold and has the maximum value in its neighborhood for every pixel.



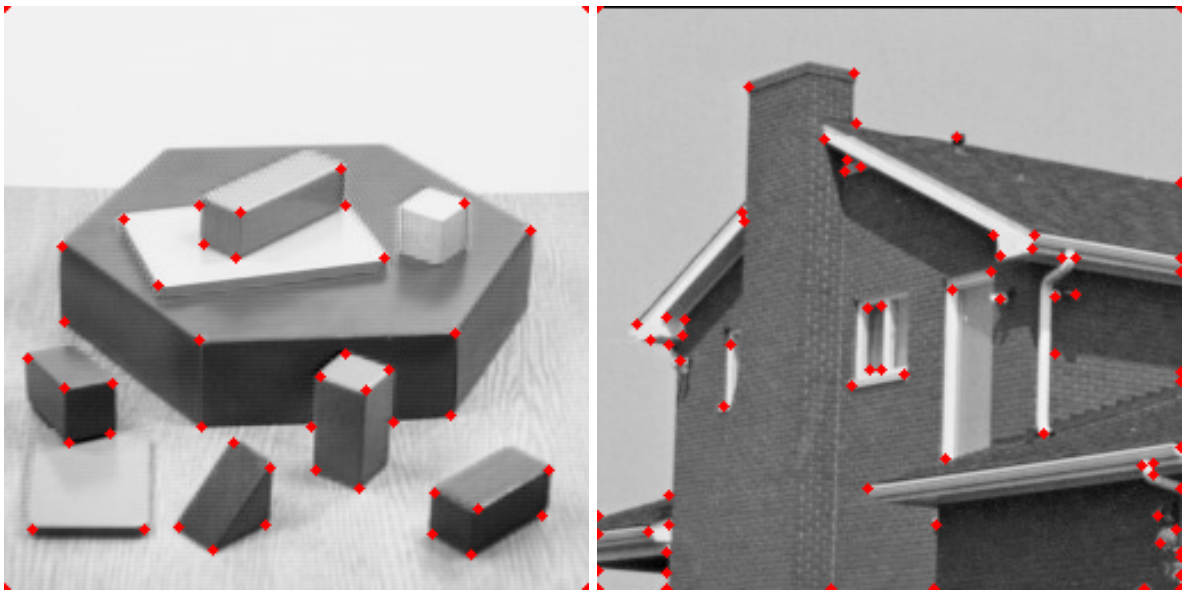
(a) $T = 1e-5$, $\sigma = 2.0$, $k = 0.04$ (b) $T = 1e-5$, $\sigma = 2.0$, $k = 0.04$

Fig. 2. Detected corners with varying hyperparameters.



(a) $T = 1e-4$, $\sigma = 1.0$, $k = 0.05$ (b) $T = 1e-4$, $\sigma = 1.0$, $k = 0.05$

Fig. 3. Detected corners with varying hyperparameters.



(a) $T = 1e-4$, $\sigma = 2.0$, $k = 0.04$ (b) $T = 1e-4$, $\sigma = 2.0$, $k = 0.04$

Fig. 4. Detected corners with varying hyperparameters.

2 Feature Description and Matching

Local descriptors. Keypoints lying too close to the border of the image are filtered out and $9x9$ patches are extracted for feature matching. The code snippet in fig. 2 filters out keypoints that lie within a 5 pixel margin at the image border, by applying a boolean mask to the pixel positions $[x, y]$, respectively.

```python
def filter_keypoints(img, keypoints, patch_size = 9):
    # TODO: Filter out keypoints that are too close to the edges

    # filter out keypoints that lie within a 5 pixel margin at the image edge

    h = img.shape[0] - 5 # (.,y) l1 and l2. Height corresponds to y position in keypoints
    w = img.shape[1] - 5 # (x,.) l1 and l2. Width corresponds to x position in keypoints

    # delete all [x,y] where x < w
    mask0 = keypoints[:,0] < w
    keypoints = np.compress(mask0, keypoints, axis=0)
    # delete all [x,y] where x < 5
    mask1 = keypoints[:,0] > 5
    keypoints = np.compress(mask1, keypoints, axis=0)
    # delete all [x,y] where y < w
    mask2 = keypoints[:,1] < h
    keypoints = np.compress(mask2, keypoints, axis=0)
    # delete all [x,y] where y < 5
    mask3 = keypoints[:,1] > 5
    keypoints = np.compress(mask3, keypoints, axis=0)
```

Fig. 5. Filter out keypoints.

SSD One-way nearest neighbour matching. To calculate the distance between the i-th and j-th feature of two given input images, the sum of squared distances is calculated for each pixel pair and stored in the matrix *distances*:

$$SSD(p, q) = \sum (p_i - q_i)^2. \tag{4}$$

The python function *scipy.spatial.distance.cdist* with parameter $'seuclidean'$ is used to calculate the differences between the two descriptors. The one-way feature matching protocol is implemented in fig. 6. First, the position of the minimum distance between the i-th feature of the first image to every feature of the second image is computed with the numpy function *argmin*. The indices are stored in the array *matches* that maps to every feature of image 1 the closest feature of image 2.

The resulting matches are depicted in fig. 7. It seems, that there are more lines than features, which indicates that some features are matched double and hence wrong.

Mutual nearest neighbors. According to the mutual nearest neighbors protocol, every match that is determined by one-way nearest neighbor matching, that is comparing the descriptors from image 1 to the descriptors of image 2, also has to be the nearest neighbour when comparing image 2 to image 1. Fig. 8 shows the main part of the implementation. To check whether the one-way match at position $[i, j]$ also is minimal when comparing $[j, i]$ first the mirrored index is calculated:

$$\sum [i, j] - [j, i] = 0. \tag{5}$$

If the distance is also minimal for features [j,i] the match is considered valid. The results are plotted in fig. 9, showing less matches than the one-way matching protocol. That indicates a better performance because significantly less mismatches occur.

```python
if method == "one_way": # Query the nearest neighbor for each keypoint in image 1
    # TODO: implement the one-way nearest neighbor matching here

    # get idx of minimum distance of i-th feature in each row
    Imin = np.argmin(distances, axis=1)
    # create index array with len = q1
    i = np.arange((q1))
    # matches = [[0,Imin[0]], [1, Imin[1]], ... ,[q1,Imin[q1]]
    Imin = Imin.reshape(len(Imin),1)
    i = i.reshape(len(i),1)
    matches = np.hstack((i,Imin))
```

Fig. 6. One-way nearest matching protocol with SSD.

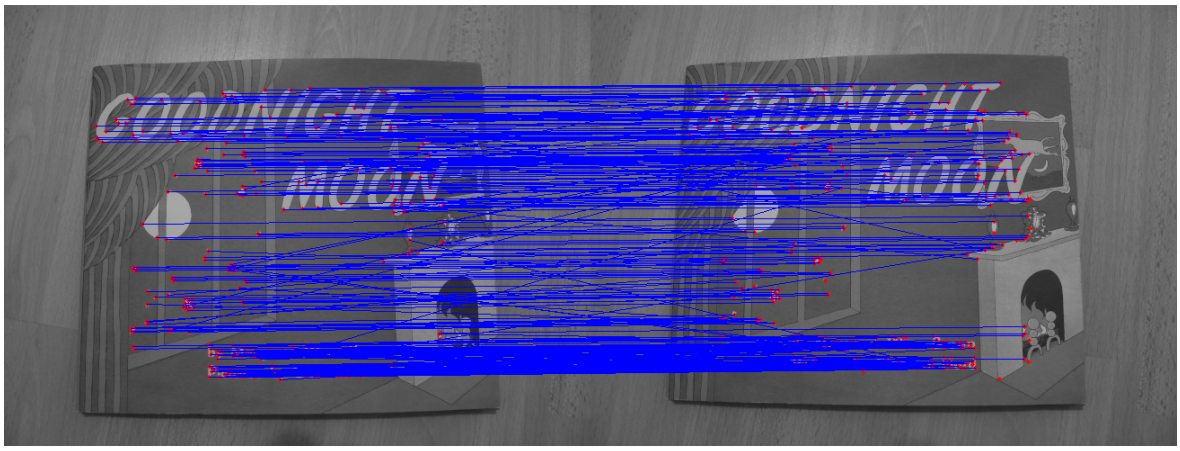


Fig. 7. One-way nearest neighbors feature matching of two images.

```python
# filter out mutual matches
matches = np.array([])
for i in range(q1):
    for j in range(q2):
        # find for every ow-match [0,1] the match [1,0]
        if np.sum(matches_ow[i] - matches_wo[j]) == 0:
            # check if for this matches min(desc1,desc2) == min(desc2,desc1)
            if min1[i] == min2[j]:
                matches = np.append(matches, matches_ow[i])
matches = matches.reshape((int(0.5*len(matches)),2))
matches = matches.astype(int)
```

Fig. 8. Mutual nearest neighbor matching protocol.

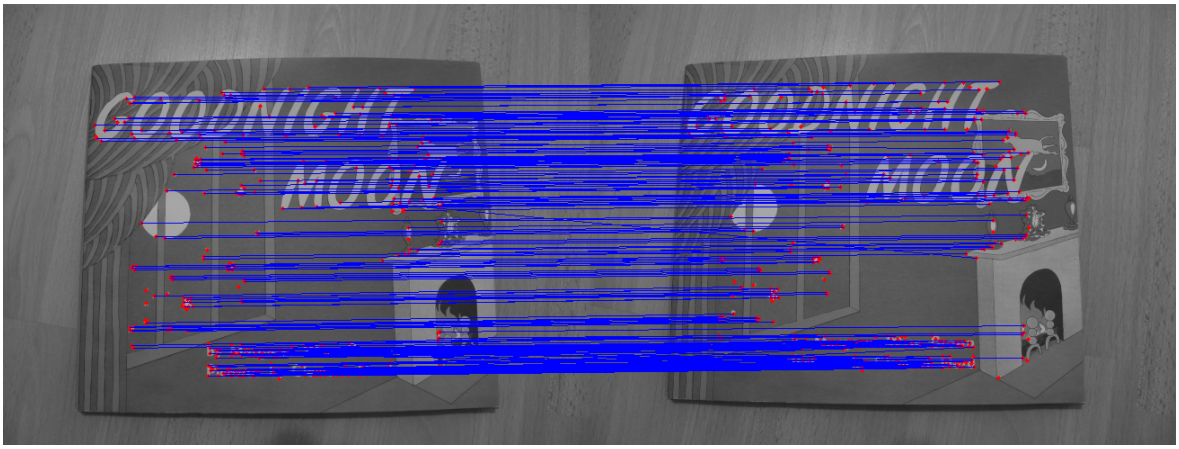


Fig. 9. Mutual nearest neighbor feature matching of two images.

Ratio test. The ratio test protocol finds the first and the second nearest neighbor of each image patch, computes the ratio and compares it to a threshold, here it is 0.5. If the ratio is lower than the threshold, it is considered a valid match. The implementation is shown in fig. 10. The function `np.partition` with parameter 2 is used to partially sort the array *distances* and output the two smallest distances. They are stored in the arrays $firstMin$ and $secondMin$. After calculating the ratio for every descriptor, only those smaller than the threshold are retained.

```
# find first and second minimum of each ow-match
part = np.partition(distances,2,axis=1)
# first element from partially sorted array 'distances is 1st minimum
firstMin = part[:,0]
# second element from partially sorted array 'distances' is 2nd minimum
secondMin = part[:,1]
# compute ratio between first and second minimum
ratio = firstMin / secondMin
# retain only ow-matches that have a ratio smaller than ratio_thresh
mask = ratio < ratio_thresh
matches = matches_ow[mask]
```

Fig. 10. Ratio test feature matching protocol.

Fig. 11 visualizes the resulting matches of the two images. Similar to the mutual nearest neighbor protocol, there seem to be little or no mismatches. These findings confirm, that the mutual nearest neighbors and the ratio test yield better results when matching feature descriptors of two different images. A possible explanation could be that those protocols introduce additional constraints for a match to be considered valid.

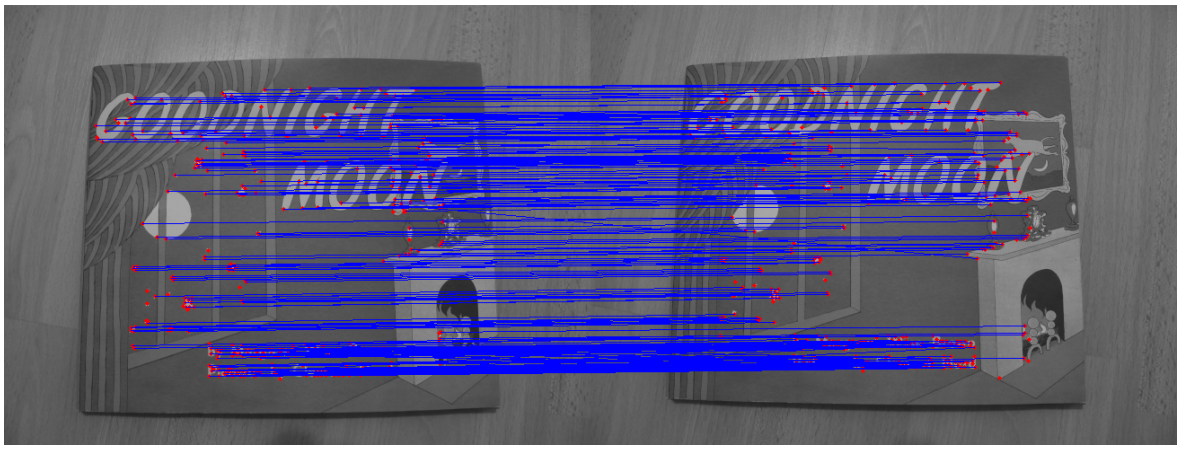


Fig. 11. Ratio test feature matching of two different images.