Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lab Assignment - Local Features

Computer Vision

Orestis Oikonomou

ooikonomou@student.ethz.ch

Computer Vision and Geometry Lab
Computer Science Department
ETH Zürich

Contents

Chapter 1

Detection

In this chapter, we implemented a Harris corner detector to find interest points in an image. For this, you will start by computing the Harris response function C for all pixels in an image. Afterwards, a pixel (i, j) is selected as a keypoint in an image if the following two conditions are satisfied: C(i, j) is above a certain detection threshold T and C(i, j) is a local maxima in its 3×3 neighborhood. For more details, please refer to [1]

1.1 Image gradients

In order to achieve that we started by computing first the image gradients I_x and I_y in the x and y directions respectively.

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

In particular, firstly we computed the kernels for gradient detection at both sides (Code 1.1 lines 2-3). Consecutively we used the function signal.convolve2d() to get the final gradients of image I with the right convolutional filters (Code 1.1 lines 6-7).

```
1     #compute kernels for gradient detection
2     xKernel = (1/2)*np.array([[1. ,0. ,-1.]])
3     yKernel = xKernel.T
4
5     #compute gradients of image I
6     Ix = signal.convolve2d(img, xKernel, 'same')
7     Iy = signal.convolve2d(img, yKernel, 'same')
```

Code 1.1: Image gradients

1.2 Local auto-correlation matrix

By obtaining the image gradients we continued with the next important step which is the computing of the auto correlation matrix M. It is important to mention that that in order to smooth any noise at our results we used Gaussian filters for the weighting of the matrix M, which is described by the following formula (where σ is the standard deviation of the Gaussian function):

$$M = g(\sigma) * \begin{bmatrix} {I_x}^2 & I_{xy} \\ I_{xy} & {I_y}^2 \end{bmatrix}$$

The aforementioned computation was implemented with the use of cv2.GaussianBlur() function (Code 1.2 lines 9-11) of OpenCV package. Particularly, we pass as inputs to the function the multiplication of all the potential gradient pairs (Code 1.2 lines 2-4). Also, we used as border type the BORDER REPLICATE in order to define the extra padding for an image, by replicating the row or column at the very edge of the original image to the extra border.

```
1  #squares of derivatives for later use
2  Ix_2 = Ix * Ix
3  Iy_2 = Iy * Iy
4  Ixy  = Ix * Iy
5
6  # Compute local auto-correlation matrix
7  # TODO: compute the auto-correlation matrix here
8  # You may refer to cv2.GaussianBlur for the gaussian filtering (
       border_type=cv2.BORDER_REPLICATE)
9  gw_Ixx = cv2.GaussianBlur(Ix_2, ksize = (5, 5), sigmaX = sigma,
       borderType = cv2.BORDER_REPLICATE)
10 gw_Iyy = cv2.GaussianBlur(Iy_2, ksize = (5, 5), sigmaX = sigma,
       borderType = cv2.BORDER_REPLICATE)
11 gw_Ixy = cv2.GaussianBlur(Ixy, ksize = (5, 5), sigmaX = sigma,
       borderType = cv2.BORDER_REPLICATE)
12
13
```

Code 1.2: Computation of Local auto-correlation matrix

1.3 Harris response function

Then in order to determine if a point is corner or not we constructed the Harris response function by using the previously computed auto-correlation matrix:

$$C(i,j) = det(M_{i,j}) - kTr^2 M_{i,j}$$

Sequentially, we calculated the Harris response C for each window at the image (Code 1.3 line 6). It worth's mentioning that depending on the value of C score we can extract the information about the 'identity' of a point (where identity: flat region or corner or edge).

```
1  #compute det and trace of M
2  det_M = gw_Ixx*gw_Iyy - gw_Ixy**2
3  trace_M =  gw_Ixx + gw_Iyy
4
5  #response function C
6  C = det_M - k * (trace_M**2)
7
8
```

Code 1.3: Harris response for images points

1.4 Detection criteria

Finally, by obtaining the C for all pixels we can detect the corners by checking the following:

1. Filter - out the C values that are below or equal to the threshold
2. Perform a local maximum check on those C values

For the local maximum we used the ndimage.maximum filter() function from scipy library, while the whole implementation is the following:

```
#find points whose surrounding window gives large corner response (C >
   threshold)
strength = C > thresh
#take the points of local maxima
local_maximality = (C == ndimage.maximum_filter(C, size=3))

#take the points that fulfill the above criteria
col,row = np.asarray(strength & local_maximality).nonzero()
corners = np.array([row, col]).T

```

Code 1.4: Detection criteria

1.5 Results from detection

As we can observe at the following pictures the least misplacements of pixels as corners were given by the combination of:

1. Harris sigma s = 2
2. Harris kappa k = 0.04
3. Threshold thresh = 1e - 5

In particular, even tough pair figures (1.3 - 1.5 , 1.4 - 1.6) seemingly do not show major differences our combination of hypermarameters gives us less number of keypoints but also better positioned on the actual corners. However, there are still cases where our Harris detector classifies points as corners, even though, they are edges or flat regions. The latter observation occurs mainly in the case of the house image. Moreover, it worth's mentioning that as we reducing the threshold the number of keypoints can be increased significantly.

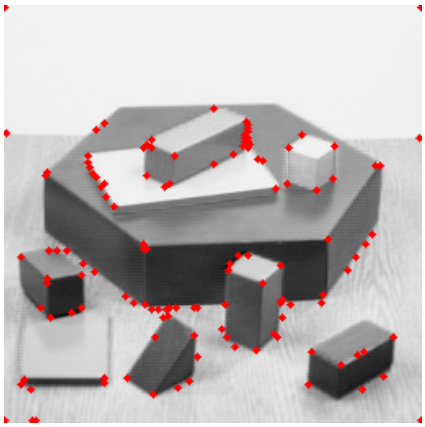

Figure 1.1: s = 0.5, k= 0.04,thresh = 1e-6

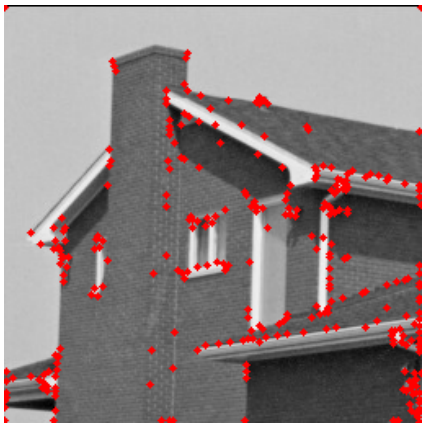

Figure 1.2: s = 0.5, k= 0.04, thresh = 1e-6

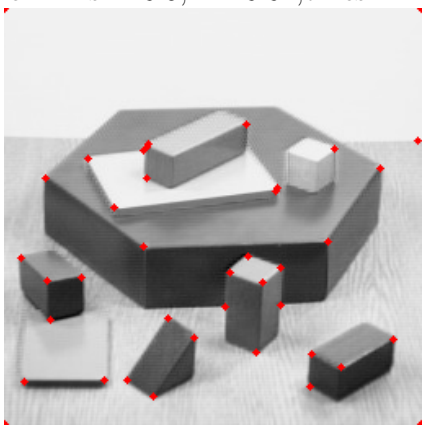

Figure 1.3: s = 1, k= 0.06,thresh = 1e-5

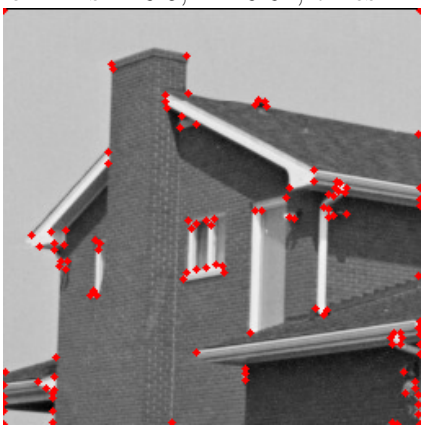

Figure 1.4: s = 1, k= 0.06,thresh = 1e-5

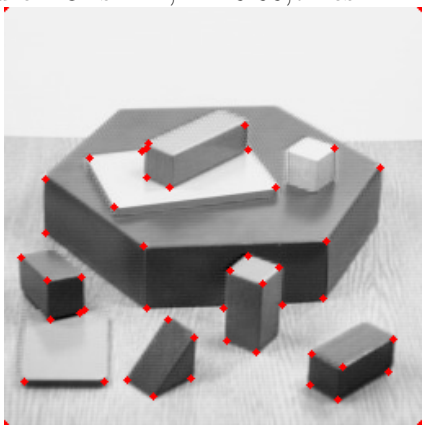

Figure 1.5: s = 2, k= 0.04,thresh = 1e-5

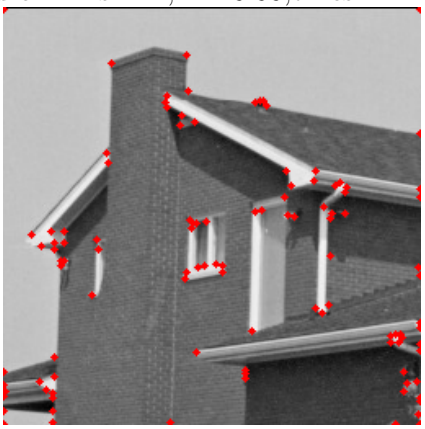

Figure 1.6: s = 2, k= 0.04,thresh = 1e-5

Chapter 2

Description Matching

In this chapter, we implemented a matching protocol for image patches and tested it out on the provided image pair.

2.1 Local descriptors

First of all we filter out the keypoints that are too close to the image boundaries so as to avoid out-of-bounds issues (Code . lines). To achieve that we selected only those corners that had a a distance of at least 9 pixels (patch size) from the borders (Code . lines).

```
h, w = img.shape[0], img.shape[1]
# calcuate the threshold for the padding in all directions
top_bound = patch_size #top
bottom_bound = h - patch_size #bottom
left_bound = patch_size #left
right_bound = w - patch_size #right

# find the in which indexes corners do not 'violate' the setted
   threshold for image boundaries
indexes = np.logical_and(np.logical_and(top_bound < keypoints[:, 1],
   keypoints[:, 1] < bottom_bound),
                         np.logical_and(left_bound < keypoints[:, 0],
   keypoints[:, 0] < right_bound))
keypointsFLT = keypoints[indexes]

```

Code 2.1: Filtering of keypoints

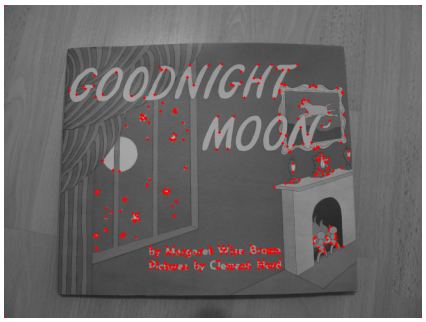

Figure 2.1: Filtered keypoints representing corners of I1 image

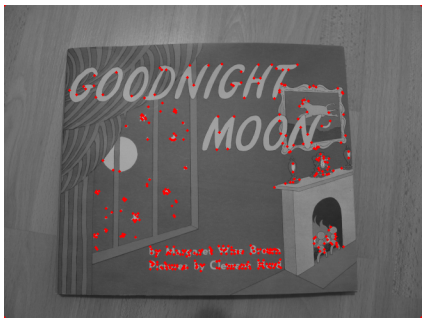

Figure 2.2: Filtered keypoints representing corners of I2 image

2.2 SSD one-way nearest neighbors matching

Continuing now with the matching process we are going to try different protocols. However, in order to implement those protocols it is important to compute the sum-of-squared-differences (SSD):

$$SSD(p, q) = \sum_i (p_i - q_i)^2,$$

between the the descriptors of all features from the first image and the second image. The implementation is given below:

```
# TODO: implement this function please
# desc1 is of shape 1999x81 while desc2 is of shape 2300x81 so we create
# a big matrix that contains of all the differences  for the pairs of
   descriptors
differences = []
differences = desc1[:,np.newaxis,:]-desc2[np.newaxis]

# find the square for each potential difference
differences_2 = np.square(differences)

return(np.sum(differences_2 ,axis=-1))

```

Code 2.2: Sum-of-squared-differences (SSD)

Being able to compute the SSD between descriptors we performed our first matching protocol. In particular, with one-way nearest neighbors protocol we have to select the feature from the first image which is associated to its closest feature from the second image. That basically is the feature with the smallest calculated distance SSD function produced. To achieve that we used the following implementation:

```
min_feature_dist = np.argmin(distances, 1)
#array that contains the index of the descriptor and its chosen feature
matches = np . array ([[ i , dist ] for i , dist in zip(range(q1),
   min_feature_dist)])
```

Code 2.3: One-way matching

All in all, our matching implementation computed 285 mathcings that are shown below:

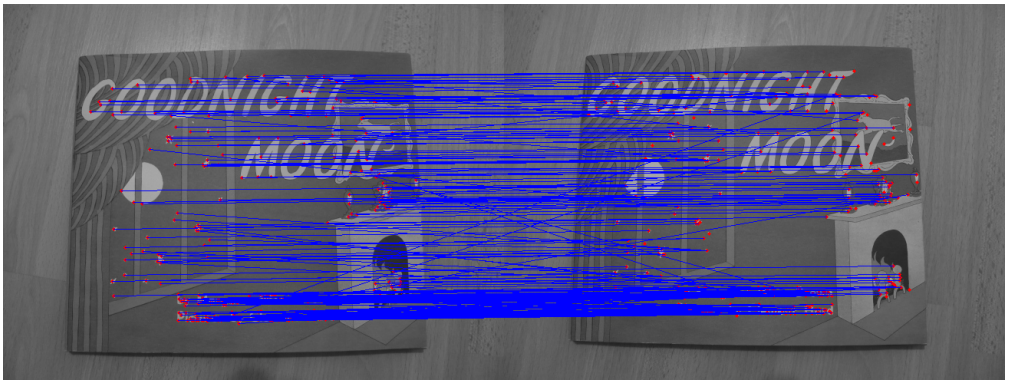

Figure 2.3: One-way nearest neighbor matchingrepresenting corners of I1 image

2.3 Mutual nearest neighbors / Ratio test

2.3.1 Mutual nearest neighbors

The second matching policy that we tested was that of mutual nearest neighbours. In particular, we performed the one-way nearest neighbour matching policy from the first to second image and vice versa and sequentially we checked for each feature if the matching was the same in both cases. By using the following implementation for this policy we were able to achieve 32% less matchings this time specifically 196:

```
        #extract the feature with that has the smallest distance between
    the two images I1 to I2
        min_feature_dist12 = np.argmin(distances, 1)
        #array that contains the index of the descriptor and its chosen
    feature
        matchesI1I2 = np . array ([[ i , dist ] for i , dist in zip(range
    (q1), min_feature_dist12)])

        #extract the feature with that has the smallest distance between
    the two images I2 to I1
        min_feature_dist21 = np.argmin(distances,0)
        #array that contains chosen feature of the descriptor and its
    index
        matchesI2I1 = np . array ([[ dist , i ] for i , dist in zip(range
    (q2), min_feature_dist21)])

        # chosen descriptor feature I2 to I1
        chDF = matchesI2I1[:, 0]

        # boolean array that contains all the cases that matches from I1
    to I2
        # are matches from I2 to I1 too
        mutual = matchesI1I2[chDF, 1] == matchesI2I1[:, 1]

        # as in this case each matching might give different number of
    matches
        # we ensure that the array of mutual matches is the same size
    with
        # the array of matches that we will use to extract the final info
        if (len(matchesI1I2)< len(mutual)):
            mutual = mutual[:len(matchesI1I2)]
```

```
            matches = matchesI1I2[mutual]
        else:
            matches = matchesI1I2[mutual]
```

Code 2.4: Mutual nearest neighbors matching

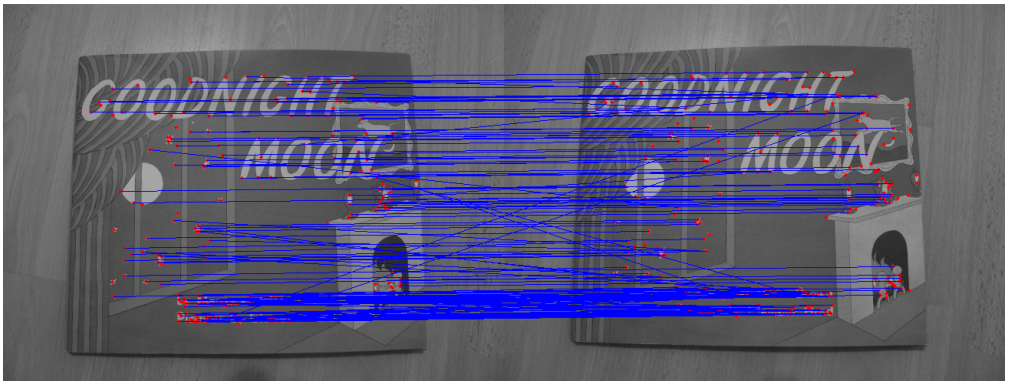

Figure 2.4: Mutual nearest neighbors matching representing corners of I1 image

2.3.2 Ratio test matching

Finally we implemented the ratio test matching method. In particular, we performed the one-way match but it was considered valid if the ratio between the first and the second nearest neighbor was lower than a threshold of 0.5. By using the following implementation for this policy we were able to achieve 17% less matchings this time specifically 163:

```
        # use of np.partition to obtain the first and second nearest
    neighbor
        partition = np.partition (distances,(0,1),1 )

        #apply ratio threshold
        partition_ratio = partition[:,0]/partition[:,1] < ratio_thresh

        #find min distances for all the feature pairs
        match = np.arange(q1)
        fea_min_dist = np.argmin(distances, 1)

        #initialization of array and assign zeros for ratios that
        #did not make through the threshold
        matches = np.zeros((q1,2))

        #use found indexes to the descriptors that we are going to keep
        col = match[partition_ratio]
        row = fea_min_dist[partition_ratio]
        matches = np.array([col,row]).T
```

Code 2.5: Mutual nearest neighbors matching

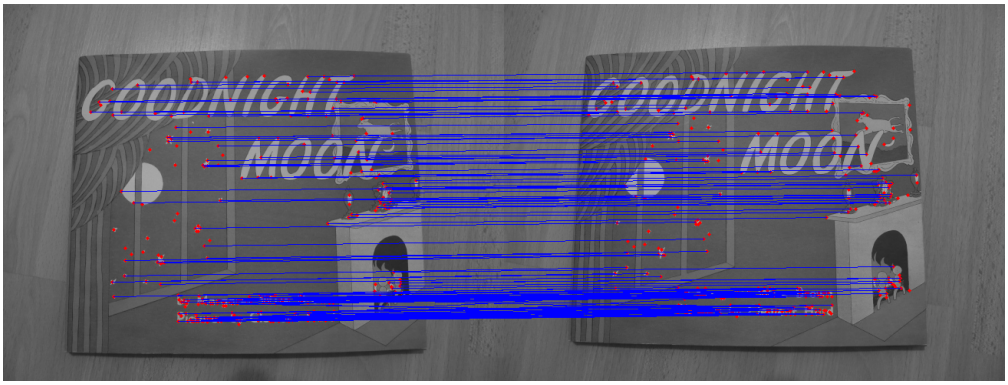

Figure 2.5: Ratio test matching representing corners of I1 image
All in all, ratio test matching gives the least number of matchings mainly because it is more picky than the other matching policies, via its imposing threshold.