Computer Vision - Local Features

Omenetti Matteo

ETH Zurich - momenetti@ethz.ch

1 Summary

This report is part of the first project for the course "Computer Vision" hosted at ETH Zürich. The project's scope was to build a feature detector and a basic matching protocol to establish pixel-wise correspondences between images. This report will explain each section of the assignment and its corresponding code.

2 Detection

In this section, the implementation of a Harris corner detector to find interest points in an image is discussed.

2.1 Image Gradients

First, the derivatives of the images had to be taken. To do so, two special kernels were defined, one for the derivative on the x-axis and one for the y-axis. These two kernels are very simple and they allow to take the difference between the right and left (and top and bottom) elements and divide it by two (this comes from the definition of derivative also reported in the project's PDF). Mode 'same' allows keeping the output dimension the same as the input dimension. Boundary 'symm' allows to mirror the image so that the four corners of the image will not be detected as corners. The corresponding code can be found at 1

```
27      kernel_x = 0.5 * np.array([[0, 0, 0], [1, 0, -1], [0, 0, 0]])
28      kernel_y = 0.5 * np.array([[0, 1, 0], [0, 0, 0], [0, -1, 0]])
29      Ix = scipy.signal.convolve2d(img, kernel_x, mode='same', boundary='symm')
30      Iy = scipy.signal.convolve2d(img, kernel_y, mode='same', boundary='symm')
```

Figure 1: Derivatives of the image

2.2 Local auto-correlation matrix

Now the relevant terms of the auto-correlation matrix are computed. Therefore, the x-derivative and y-derivative are multiplied together for the elements on the negative diagonal and squared for the elements on the positive diagonal. A Gaussian filter is applied to each term to perform the summation over the neighbors (that in the formula reported in the PDF is expressed with the Σ symbol). The corresponding code can be found at 2

```
35      Ix2 = cv2.GaussianBlur(np.square(Ix), (3, 3), sigma, borderType=cv2.BORDER_REPLICATE)
36      Iy2 = cv2.GaussianBlur(np.square(Iy), (3, 3), sigma, borderType=cv2.BORDER_REPLICATE)
37      Ixy = cv2.GaussianBlur(Ix * Iy, (3, 3), sigma, borderType=cv2.BORDER_REPLICATE)
```

Figure 2: The Local auto-correlation matrix

2.3 Harris response function

The Harris response function is derived using the formula reported in the lecture slides. The corresponding code can be found at 3

```
41      C = Ix2 * Iy2 - np.square(Ixy) - k * np.square(Ix2 + Iy2)
```

Figure 3: The Harris response function

2.4 Detection criteria

Now, each pixel is classified as being (or not being) a corner. First, the maximum of the response function in a 3x3 patch is recorded using the suggested function 'scipy.ndimage.maximum filter'. Finally, a mask array is created to keep only the points that are greater than the given threshold and that are also the maximum in their neighborhood. The corresponding code can be found at 4.

```
48      max_filter = scipy.ndimage.maximum_filter(C, size=3)
49      suppressed_response = C == max_filter
50      thresholded_response = C > thresh
51      mask = np.logical_and(suppressed_response, thresholded_response)
52      corners = np.argwhere(mask.transpose())
```

Figure 4: Detection criteria

2.5 Results

Different values of the threshold T, the constant k, and the standard deviation σ have been used to fine-tune the corners detection for the provided images. Increasing σ allows more points to be picked up as corners by the algorithm. The same thing happens when increasing k and decreasing the threshold. These parameters allow deciding a trade-off between false positives and false negatives. The following are the values that allow to obtain the best trade-off: σ = 1, k = 0.05, T = 1e-5. In 5 and 6, it is possible to notice (especially in the simpler block image) that some random points are detected as corners while some corners are not detected at all. However, using the hyper-parameters reported above, the number of false positives is low. Choosing the parameters such that also the missing corners would be detected introduces too many false positives.

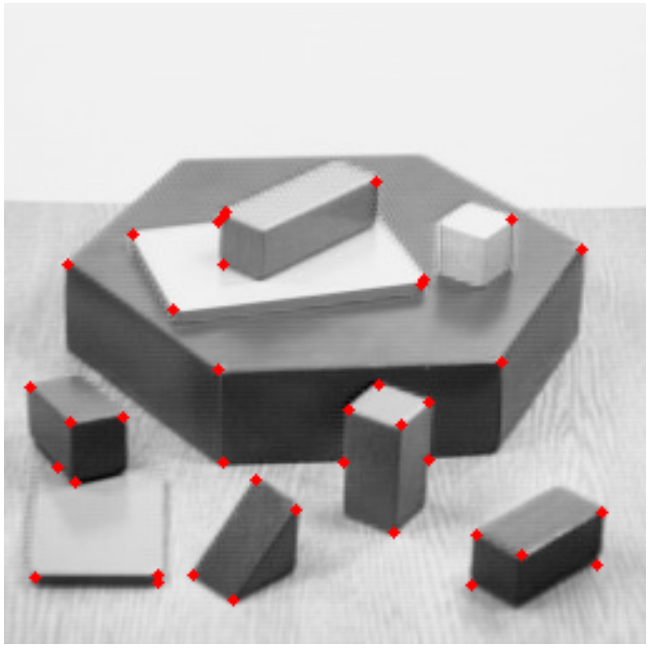

Figure 5: An image of blocks in which corners have been detected using Harris Detector. Some corners are missing, but the number of false positives is very low.

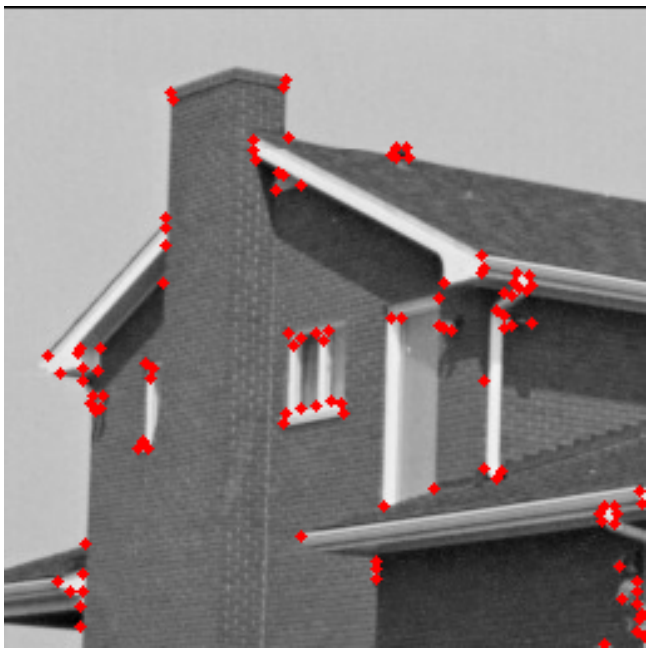

Figure 6: An image of a house in which corners have been detected using Harris Detector. Some corners are missing, but the number of false positives is very low.

3 Description & Matching

In this section, the implementation of a matching protocol for image patches is discussed.

3.1 Local descriptors

First a bound is derived diving the given patch size by 2. This bound is going to be used to check whether a key point is too close to the boundary and therefore must be removed. The condition that can be found in 7 allows keeping only the points that are at least 'bound' pixels away from the 4 edges of the picture. This is done because the extraction of 9x9 patches can be done only if the patch is entirely inside the image.

```
5    bound = int(patch_size / 2)
6    mask = (keypoints[:, 0] >= bound) & (keypoints[:, 0] < img.shape[0] - bound) & (keypoints[:, 1] >= bound) & (keypoints[:, 1] < img.shape[1] - bound)
7
8    return keypoints[mask]
```

Figure 7: Local descriptors

3.2 SSD

For this task, the squared euclidean distance had to be computed from the local descriptors of two images. To carry out the computation efficiently (therefore avoiding Python for loops) the 'scipy.spatial.distance.cdist' function has been used. This function allows performing such computation in a vectorized manner. The corresponding code can be found at 8

```
13       assert desc1.shape[1] == desc2.shape[1]
14       # TODO: implement this function
15       distances = cdist(desc1, desc2, metric='sqeuclidean')
```

Figure 8: SSD

3.3 Nearest neighbors matching

For the nearest neighbor matching, it was sufficient to pick the minimum for each row of the matrix returned by 'ssd'. The corresponding code can be found at 9.

```
32      if method == "one_way": # Query the nearest neighbor for each keypoint in image 1
33          nn = np.argmin(distances, axis=1)
34          matches = np.array([[i, nn[i]] for i in range(len(nn))])
```

Figure 9: Nearest neighbors matching

3.4 Mutual nearest neighbors

For mutual nearest neighbors, a for loop is carried out to check whether the found one-way match is also valid when swapping the images. The minimum for each point is computed twice, for the two different axis (0 and 1). Using the for loop at 10 the two way matching condition is checked. The corresponding code can be found at 10.

```
38      elif method == "mutual":
39          # TODO: implement the mutual nearest neighbor matching here
40          nn = np.argmin(distances, axis=0)
41          nn_inversed = np.argmin(distances, axis=1)
42          matches = np.array([[nn[i], i] for i in range(len(nn))])
43          matches_inversed = np.array([[i, nn_inversed[i]] for i in range(len(nn_invers
44
45          matches_to_return = []
46          for i in range(len(matches)):
47              if matches[i][1] == matches_inversed[matches[i][0]][1]:
48                  matches_to_return.append(matches[i])
```

Figure 10: Nearest neighbors matching

3.5 Ratio Test

To implement the ratio test, the suggested 'np.partion' function has been used to order the matrix of squared euclidean distances such that the first and second minimum are correspondingly in the first and second positions of each row. Finally, a mask array has been used to keep only the entries in which the ratio is less than the given threshold. This allows us to check that the second minimum is far away from the first minimum. The corresponding code can be found at 11.

```
52      elif method == "ratio":
53          # TODO: implement the ratio test matching here
54          nn = np.argmin(distances, axis=1)
55          matches = np.array([[i, nn[i]] for i in range(len(nn))])
56
57          partitioned = np.partition(distances, (0,1), axis=1)
58          ratio = partitioned[:,0] / partitioned[:,1]
59          print(ratio)
60          mask = ratio < ratio_thresh
61          matches = matches[mask]
```

Figure 11: Ratio test

3.6 Results

In figure 12 13 14 it is possible to observe the matching performed by the 3 respective algorithms. However, there are too many connections, therefore also 3 other images with only a subset of the total connections are shown in the report. They can be found at 15 16 17. The nearest neighbor connection test draws some diagonal lines that are clearly wrong, mutual and ratio tests make this wrong behavior disappear. Furthermore, it is possible to notice that the ratio test significantly reduces the number of connections.

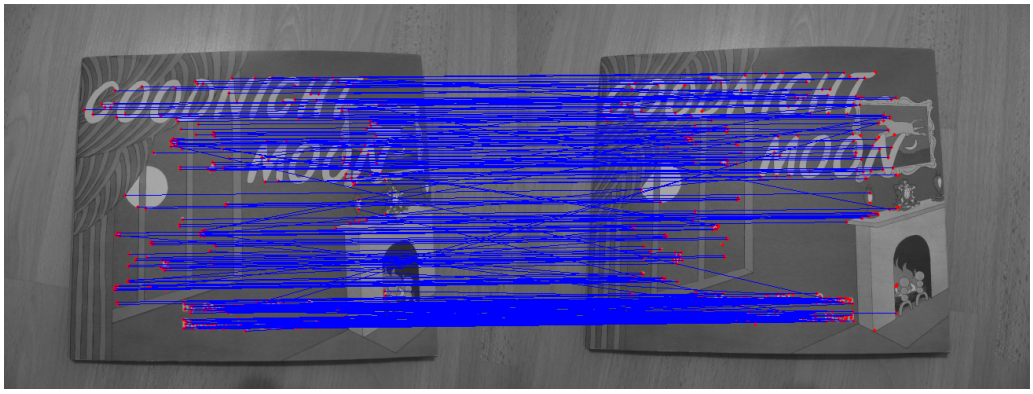

Figure 12: The matching performed by Nearest neighbors

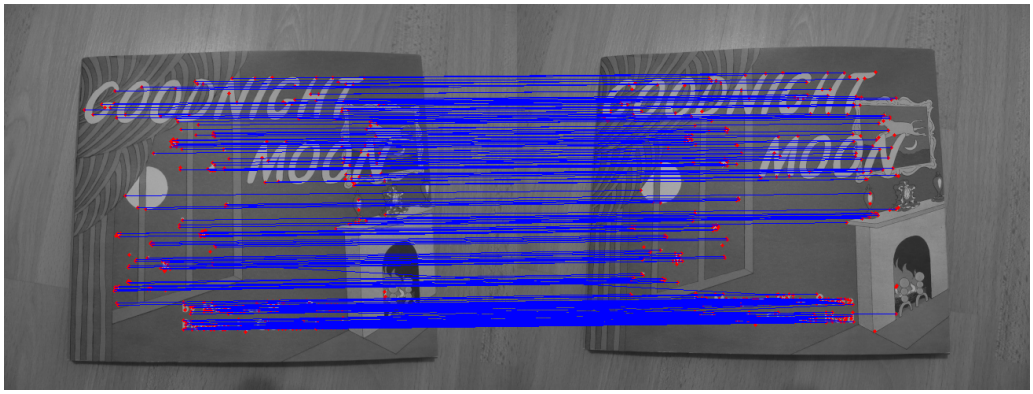

Figure 13: The matching performed by Mutual neighbors

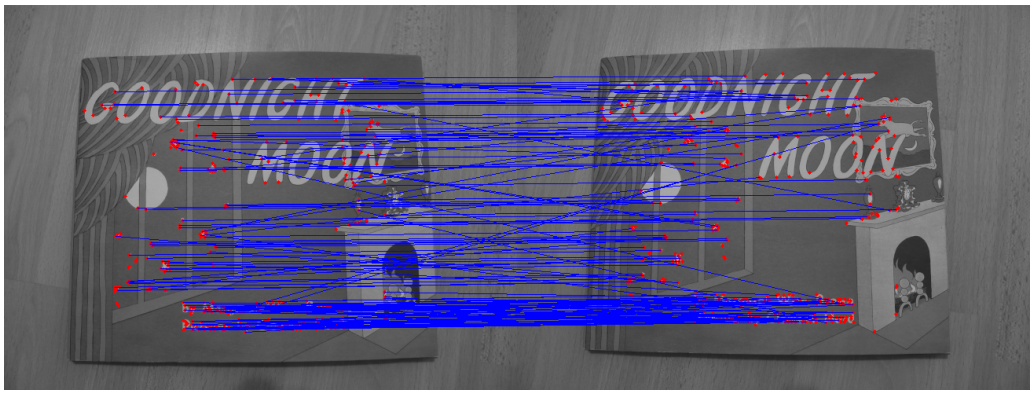

Figure 14: The matching performed by Nearest neighbors

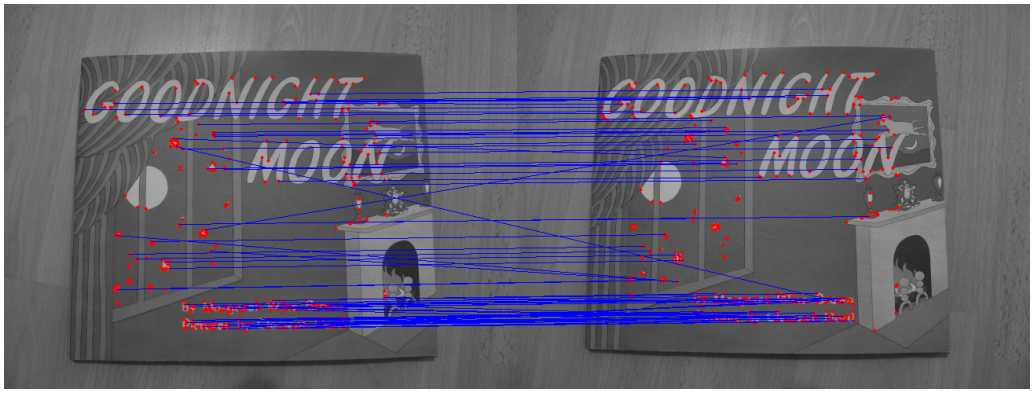

Figure 15: The matching performed by Nearest neighbors with fewer connections displayed

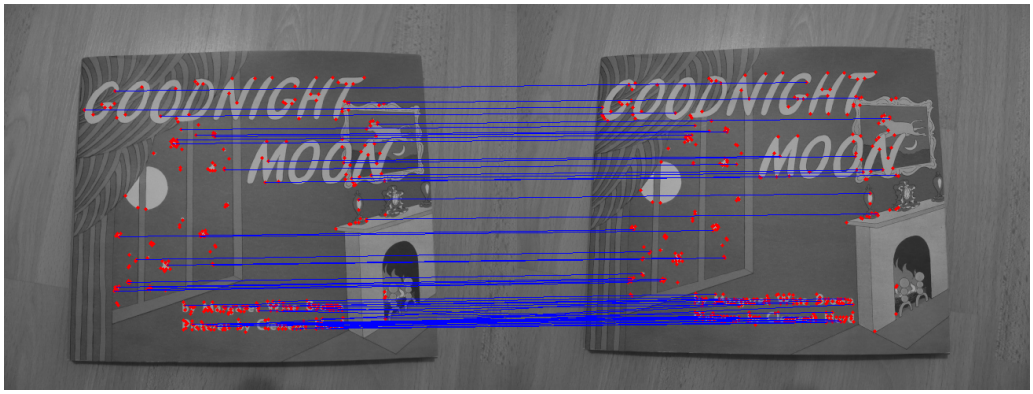

Figure 16: The matching performed by Mutual neighbors with fewer connections displayed

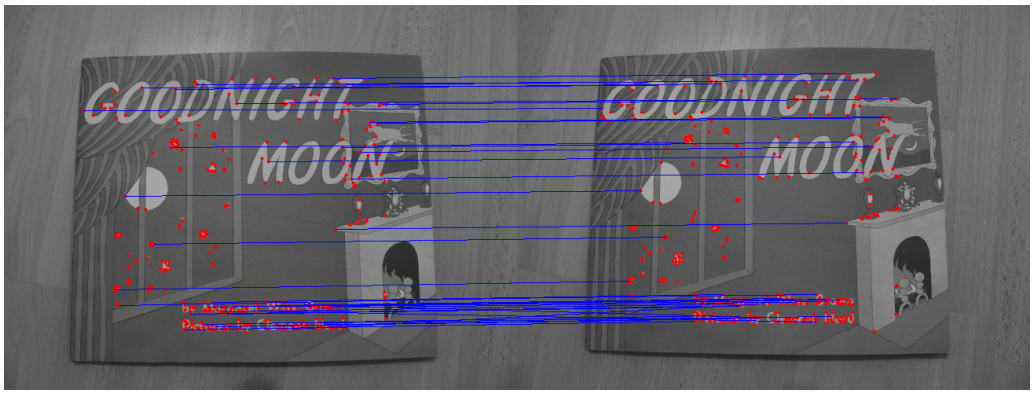

Figure 17: The matching performed by Nearest neighbors with fewer connections displayed