CV: Lab 02 Writeup

Benedict Armstrong
benedict.armstrong@inf.ethz.ch

Assignment

This weeks assignment was split into two parts:
- Implementing Harris corner detection
- Implementing three matching algorithms to match Images descriptors

Harris corner detection

The Harris corner detection algorithm can be split up into four steps.
- Compute intensity gradients in x and y direction
- Blur Images to get rid of noise
- Compute Harris response
- Thresholding and non-maximum suppression

Computing Image Gradiant

We start by computing the Image gradiants I_x in the x and I_y int the y direction. For this we convolve the images using a Sobel filter

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and it's transpose for the x direction.

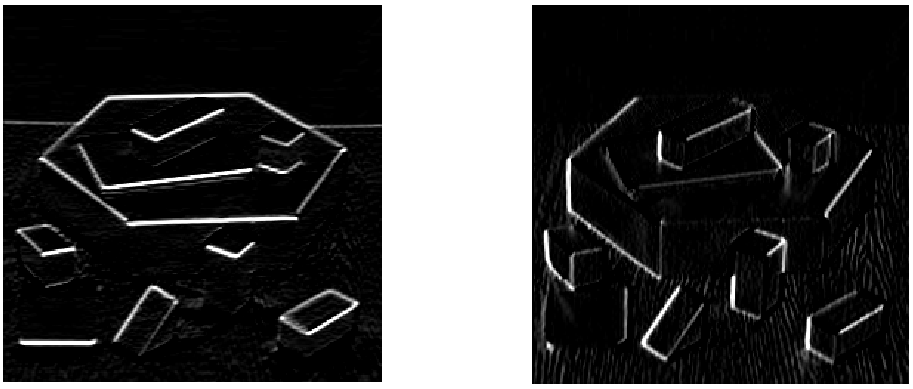


Figure 1: Image gradiants for the example image `blocks.jpg` in x and y direction

The gradients are then blurred using `cv2.GaussianBlur` with a 5 by 5 kernel. Next we define a matrix M:

$$M_{i,j} = \begin{bmatrix} (I_x^2)_{i,j} & \left(I_x I_y\right)_{i,j} \\ \left(I_x I_y\right)_{i,j} & (I_y^2)_{i,j} \end{bmatrix}$$

This allows us make use of numpy's broadcasting rules to calculate the harris response in a single line of code.

```
C = np.linalg.det(M) - k * np.square(np.trace(M, axis1=2, axis2=3))
```

The last step is to select the maximum from 3 by 3 neighbourhoods, then get the indecies of all values greater than a given threshold and lastly to swap the x and y coordinates.

```
C = C * (C == ndimage.maximum_filter(C, size=3))
corners = np.argwhere(C > thresh)
corners[:, [1, 0]] = corners[:, [0, 1]]
```

Playing around a bit with the Harris k value and the threshold t we arrive at values $k = 0.04$ and $t = 0.01$

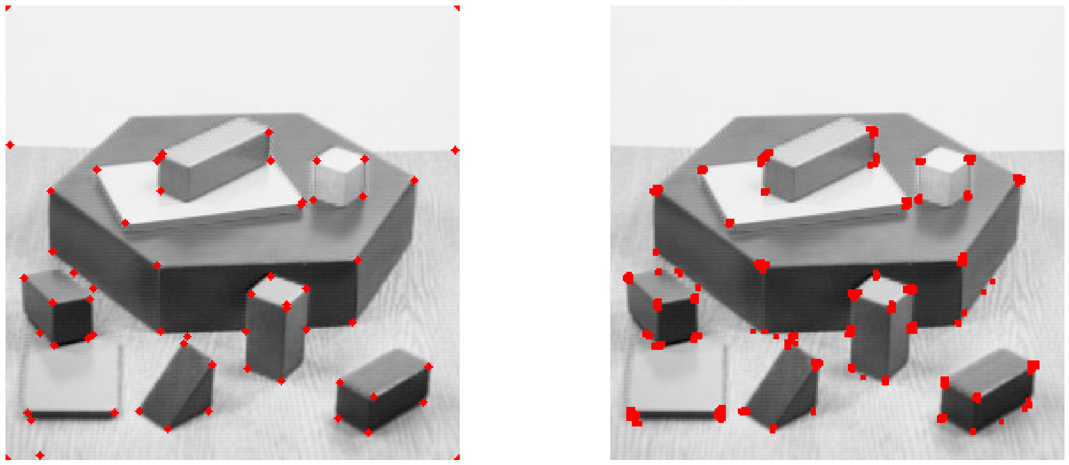


Figure 2: Results of running harris corner detection for `blocks.jpg` with values $k = 0.04$ and $t = 0.01$ (left) and the result using the `cv2.cornerHarris` implementation (right) with the same parameters

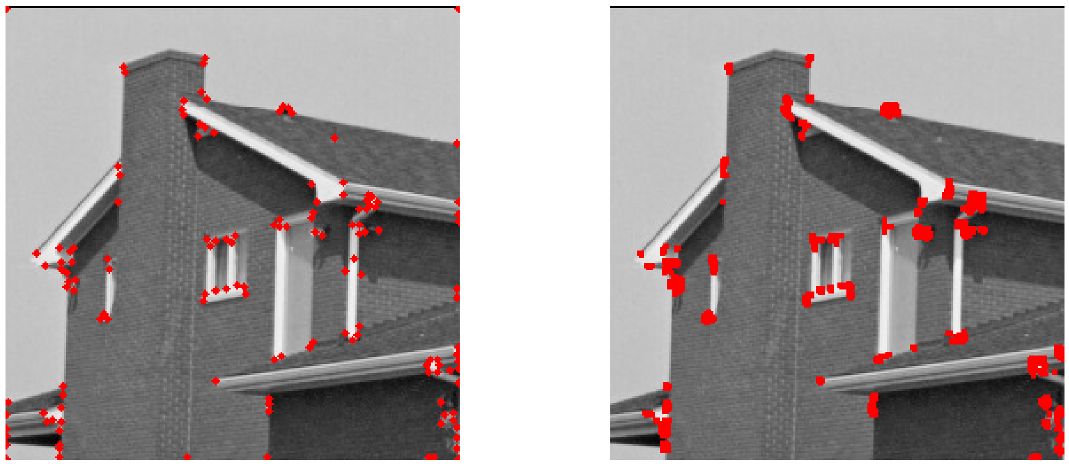


Figure 3: Results of running harris corner detection for `house.jpg` with values $k = 0.04$ and $t = 0.01$ (left) and the result using the `cv2.cornerHarris` implementation (right) with the same parameters

Description & matching

The goal of the second task was to implement three different strategies for matching keypoints from one image to keypoints in another image. As keypoints we will use the corners found by the harris detector described above.

Implementation

The first step was removing any keypoints which were too close to the edge of the image. Next we use the provided `extract_patches()` function to extract the 3 by 3 neigbourhood around each keypoint which we will use for our matching. All three strategies rely on a distance metric (SSD) calculated between all 3 by 3 neigbourhoods using the following equation.

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

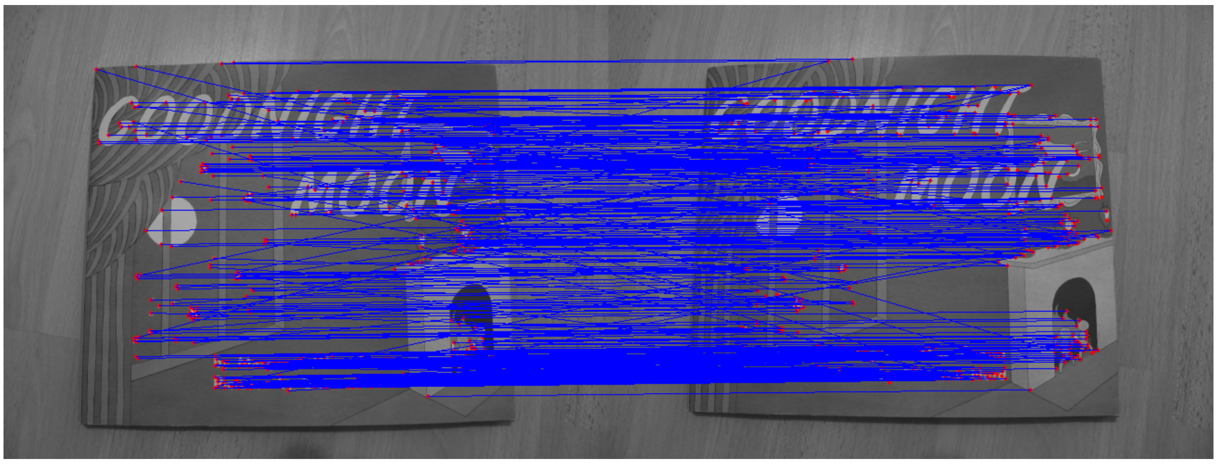
Where p is a neighbourhood from image 1 and q is from image 2.

One way matching

The first strategy implemented is as the name suggest to find the nearest neigbour for each keypoint in the first image acording to the distance metric outlined above.

```
# distances contains all dist. from descriptors of img1 to descriptors of img2
matches = np.argmin(distances, axis=1)
matches = np.vstack((np.arange(q1), matches)).T
```

This naive approach results in quite a few obviously wrong matches (lines going diagonally across the two images).



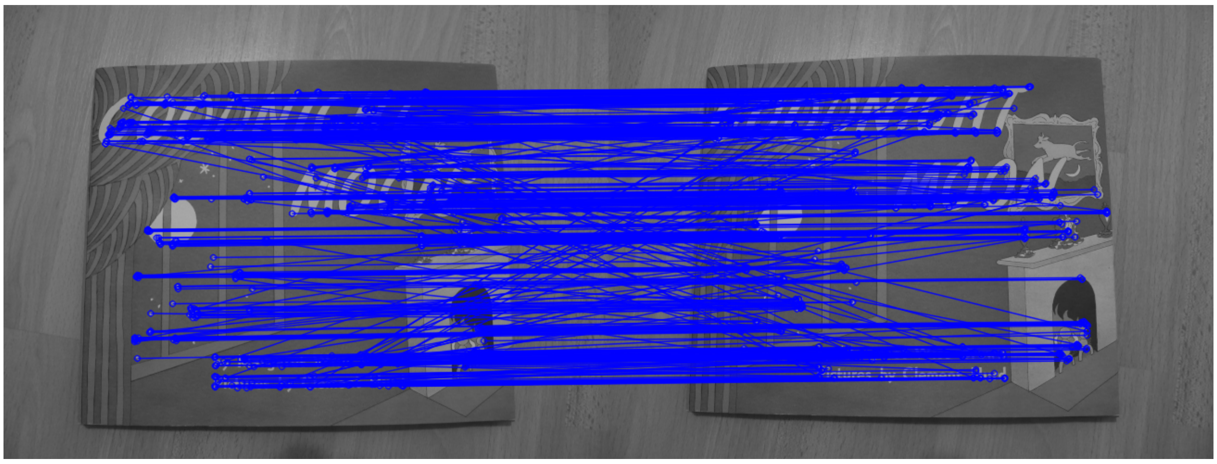


Figure 4: Result of one way matching implemented manually (above) vs. reference implementation found online (https://www.geeksforgeeks.org/feature-matching-using-brute-force-in-opencv/) which uses OpenCV's `cv2.BFMatcher` algorithm with `crossCheck=False`.

Mutual nearest neigbour

We start the same as with one way matching except that for each match we check if it goes the other way too if we switch the two images.

The python code is again quite simple:

```
matches_1 = np.argmin(distances, axis=1)
matches_2 = np.argmin(distances, axis=0)

matches = []
for i in range(q1):
    if matches_2[matches_1[i]] == i:
        matches.append([i, matches_1[i]])

matches = np.array(matches)
```

In the result we can clearly see that there are less erronius matches that go diagonally across the images.

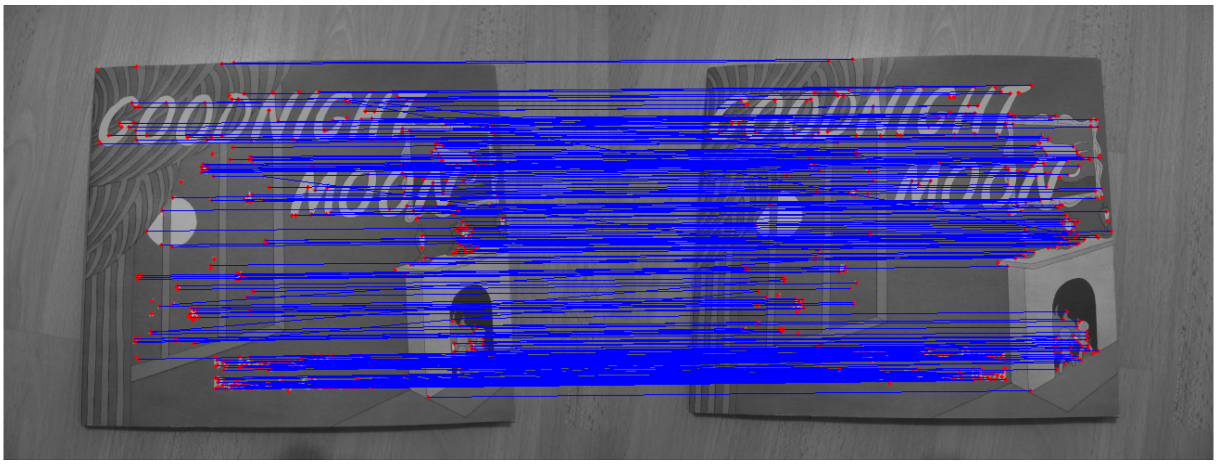
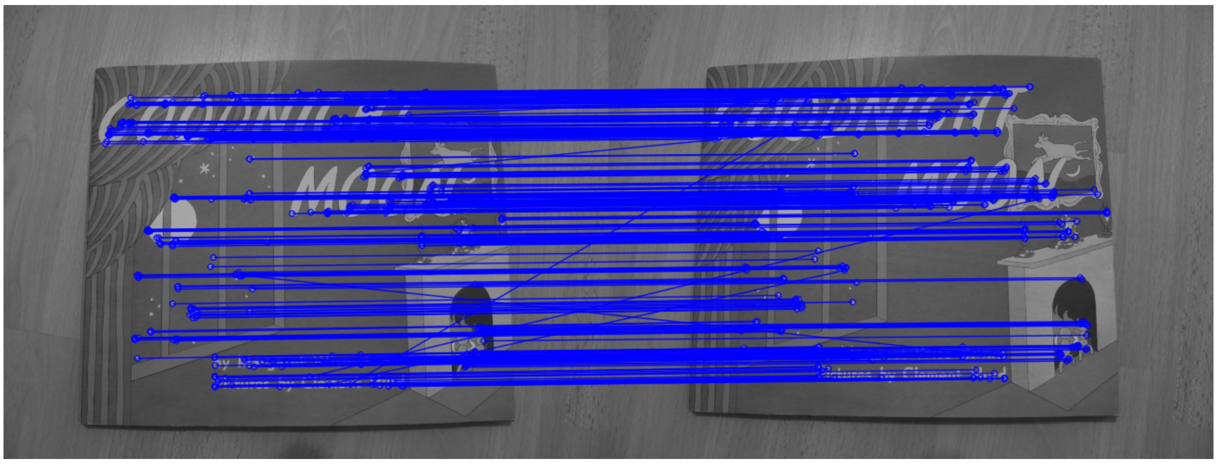



Figure 5: Result of mutual matching implemented manually (above) vs. reference implementation found online (https://www.geeksforgeeks.org/feature-matching-using-brute-force-in-opencv/) which uses OpenCV's `cv2.BFMatcher` algorithm with `crossCheck=True`.

Ratio test matching

For this strategy we calculate the ratio between the best and the second best match and only accept if the ratio is below a certain threshold. This should exclude ambigious matches where there are multiple good candidates.

Python implementation of ratio test matching strategy:

```
two_closest = np.partition(distances, 2, axis=1)[:, 0:2]
ratio = two_closest[:, 0] / two_closest[:, 1]

matches = np.argmin(distances, axis=1) * (ratio < ratio_thresh)
matches = np.vstack((np.arange(q1), matches)).T
matches = matches[matches[:, 1] != 0]
```

The results look very promising although there are a number of points with no match (the points at the very top of the books edge for example). Even still there are a lot of matches that look to be correct.

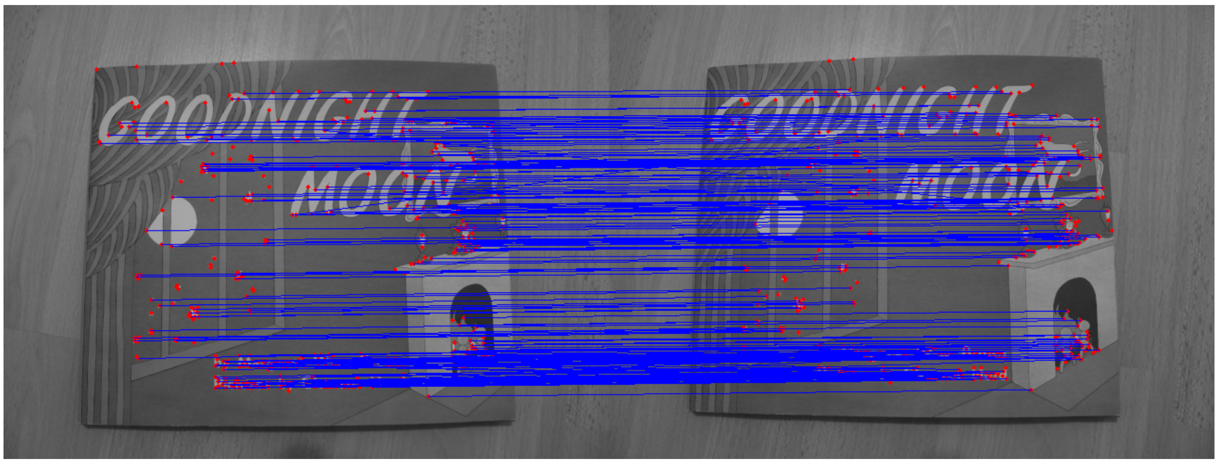


Figure 6: Result of ratio test matching with a threshold value of 0.5.