ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision
Fall 2023

Prof. Marc Pollefeys

Eric Tillmann Bill: Assignment 1

erbill@student.ethz.ch

Task 1: Harris Corner Detection

In the following, I briefly describe the idea behind my implementation for each step in the function `extract_harris()`.

In the initial step, I computed image gradients along each axis by utilizing convolution. Specifically, I employed the `signal.convolve2d()` function with the kernels k_x and k_y to calculate gradients along the x and y axes, respectively. The kernel is defined as follows:

$$k_x := \begin{bmatrix} 0 & 0 & 0 \\ -0.5 & 0 & 0.5 \\ 0 & 0 & 0 \end{bmatrix}, \quad \text{and} \quad k_y := \begin{bmatrix} 0 & -0.5 & 0 \\ 0 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$$

For the second step, where it is recommended to smooth the gradients with a Gaussian, I found that my solution yields better results if I do not smooth the gradients. A possible explanation for this could be that since I also use a Gaussian-Convolution as the window function (as recommended in the lecture), the gradients would be smoothed twice, losing potential information.

For the third step, I computed I_x, I_y, and I_{xy} for the entire image using `ndimage.gaussian_filter()` as my window function. It's worth noting that, since $I_{xy} = I_{yx}$, I computed this term only once. Subsequently, I reshaped all these matrices into a larger matrix, denoted as M, where each pixel's coordinates correspond to the matrix $M_{i,j}$ of the subsequent pixel. In Python:

```
I_xx = ndimage.gaussian_filter(gradient_x**2, sigma)
I_yy = ndimage.gaussian_filter(gradient_y**2, sigma)
I_xy = ndimage.gaussian_filter(gradient_x * gradient_y, sigma)

M = np.c_[I_xx.flatten(),
          I_xy.flatten(),
          I_xy.flatten(),
          I_yy.flatten()].reshape(img.shape[0], img.shape[1], 2, 2)
```

In the following step, I utilized the `np.linalg.det()` function to compute the determinant of each matrix $M_{i,j}$. Similarly, I calculated the trace for each pixel using `np.trace()`. Subsequently, I calculated the Harris response C as `C = det_M - k * trace_M**2`.

In the final step, as recommended, I identified the maximum region within a 3×3 neighborhood by utilizing `ndimage.maximum_filter`. Afterward, I extracted all pixels that were the largest within their respective 3×3 neighborhoods and exceeded the given threshold. To accomplish this, I run the following code:

```
C_max_region = ndimage.maximum_filter(C, size=(3,3))
corners = np.argwhere((C >= C_max_region) & (C > thresh))
```

Afterwards, I simply swap the x and y coordinates for each corner and return them.

Sample results of this function can be found at Fig. 1, where the detected corners are plotted. In conclusion, I am confident that my implementation works well and reliably detects corners.

Figure 1: Visualization of the corners detected by `extract_harris()` on four sample images.

Task 2: Match Descriptor

Before I begin to explain my implementation of the match descriptors, I will provide a brief overview of my implementations for the `ssd()` metric and the `filter_keypoints()`

function.

For the SSD metric, I utilized the `scipy.spatial.distance.cdist()` function, which efficiently computes pairwise distances between elements in two input sequences. Since our goal is to get the total squared distance, I defined this metric and then called the `cdist()` function. The SSD between input sequences `desc1` and `desc2` was calculated as follows:

```
def ssd_metric(A, B):
    return np.sum((A - B)**2)
result = cdist(desc1, desc2, metric=ssd_metric)
```

For the `filter_keypoints()` function, I used `np.argwhere` and constructed a boolean expression that returns only the indices of keypoints that are distant enough from the edges of the image. In detail:

```
dim1, dim2 = img.shape

index = np.argwhere(
    (keypoints[:, 0] > patch_size)
    & (keypoints[:, 0] < dim1 - patch_size)
    & (keypoints[:, 1] > patch_size)
    & (keypoints[:, 1] < dim2 - patch_size)
    ).flatten()

return keypoints[index]
```

Moving on to the `match_descriptors()`. Starting with the one-way matching, I used the `distance` matrix I obtained after calling the SSD function. This matrix has the dimension of the number of corners of the first image times the number of corners of the second image. Therefore, for the one-way matching, I calculated the minimum along the second axis. Consequently, each corner of the first image is assigned to the descriptor with the smallest distance to it. In Python code:

```
matches = np.c_[np.arange(q1), np.argmin(distances, axis=1)]
```

For the mutual mapping, I essentially repeated the first process for both images, resulting in two lists, each containing tuples in the form (descriptor of image 1, descriptor of image 2). I then calculated the intersection of these two lists. In Python code:

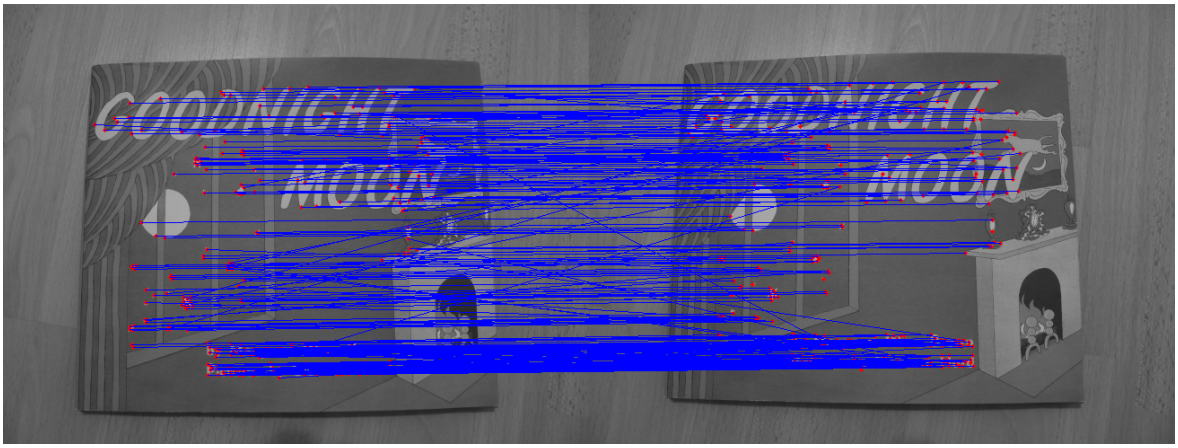
```
matches1 = np.c_[np.arange(q1), np.argmin(distances, axis=1)].tolist()
matches2 = np.c_[np.argmin(distances, axis=0), np.arange(q2)].tolist()

matches = np.array([m for m in matches1 if m in matches2])
```

Lastly, for the ratio mapping, I repeated the one-sided matching procedure twice. First, I calculated the one-sided match, then I set the distances between these pairs to infinity, and finally, I calculated a second one-sided match. I then checked to see if the ratio for distances of each pair was below the specified threshold. In Python:

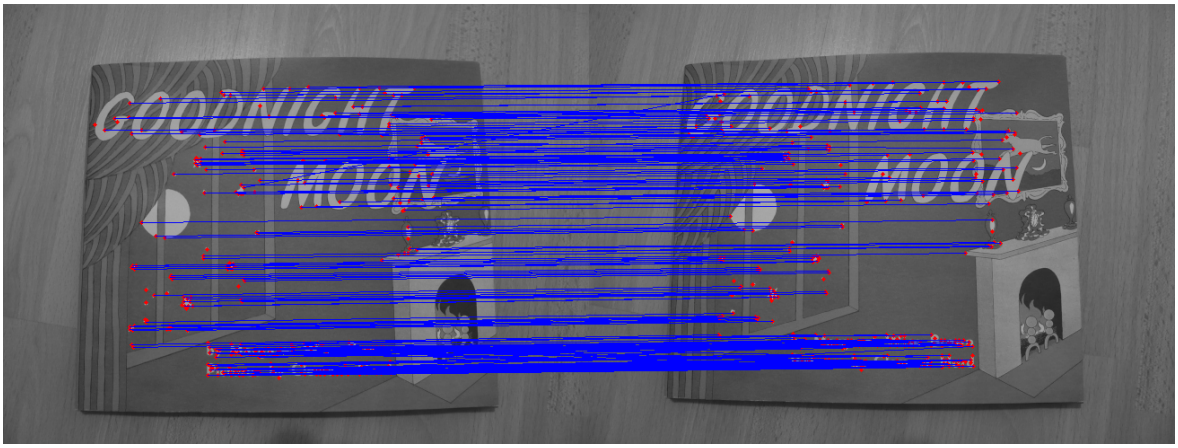
```
first_nn = np.argmin(distances, axis=1)
first_nn_distances = np.min(distances, axis=1)
```

```
matches = np.c_[np.arange(q1), first_nn]

distances[matches[:, 0], matches[:, 1]] = np.inf

second_nn = np.argmin(distances, axis=1)
second_nn_distances = np.min(distances, axis=1)

ratio = first_nn_distances / second_nn_distances
index = np.argwhere(ratio < ratio_thresh).flatten()

matches = np.c_[np.arange(q1), first_nn][index]
```

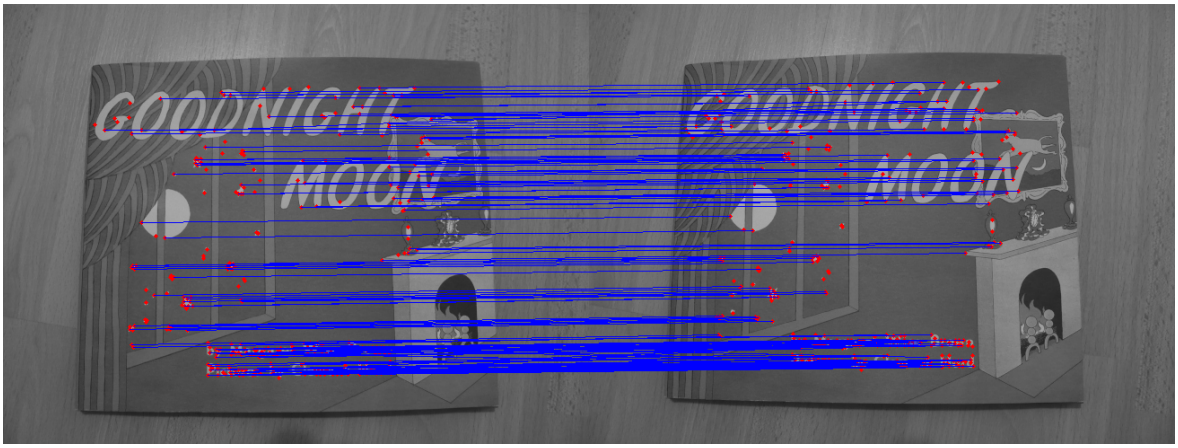
Sample results for each mapping can be found in Fig. 2, where for two images, each corner in one image is connected to its mapped corner of the other image by a blue line. Ratio matching gives the best result, while one-way matching is the worst of the three methods. How good the matching is can be judged visually by seeing how many lines are "parallel". In this example, if some lines intersect, it means that some corners are not matched properly. The number of lines that are not "parallel" to the rest is very pronounced in one-way matching, less so in mutual matching, and nonexistent in ratio matching.



(a) One-way Matching



(b) Mutual Matching



(c) Ratio Matching

Figure 2: Visualization of the corners detected by `extract_harris()` and than matched using one of the three methods.