Local Features
Computer Vision Lab 02 Report
HS2023

Christopher Zanoli

czanoli@ethz.ch
Student Number: 23-942-394

Abstract

The report is divided into two main sections: **Harris Corner Detection** and **Description & Matching**. Within each section, I provide a comprehensive analysis of the implementation decisions and clarify the results both at the level of theoretical foundations and empirical observations.

1 Harris Corner Detection

The Harris Corner Detection is a popular method for detecting corners in images. The results presented from here on were obtained with a specific combination of parameters. The choice of the best combination was made by visual inspection of the results obtained on all possible combinations of the parameters. For more information on this step see section 3.

The following are the main steps of Harris Corner Detection:

1.1 Gradient Computation

In the domain of Image Processing, computing gradients is an integral step for edge detection and feature extraction. The gradient describes the direction and rate of change in intensity across an image. For a 2D function, i.e. the input image, the gradient is a vector comprising partial derivatives in the x and y directions. This two-element vector indicates the direction of increasing luminosity and its magnitude is proportional to the rate at which the luminosity is changing. A substantial change in luminosity results in a longer gradient vector, pointing in the direction of increased brightness.

The significance of the gradient is that the contour of an object within an image generally follows a vector that's perpendicular to the gradient. Thus, by determining gradients and then evaluating the perpendicular vectors, edges within the image can be discerned. The length of the gradient vector, often termed its "magnitude", provides insight into the strength of the edge. A high magnitude indicates a sharp or strong edge, stemming from a significant change in luminosity. Conversely, a short gradient signifies a subtle transition in luminosity, indicating a weaker edge.

A critical question arises: how do we compute these partial derivatives in the x and y directions for an image? Recalling the 1D context, it's evident that the notion of "Central Differences" bears a striking resemblance to convolution, primarily because both involve linear manipulation of pixel values. In the context of discrete functions, such as digital images, derivatives are computed using the method of central differences. This method effectively measures the difference between adjacent values, and in our case, this means the difference in pixel values. By convolving the image with specific filters designed to compute these differences, we can obtain the desired gradients. This convolution approach is particularly apt as it corresponds to the central difference method, crucial when deriving a discrete function such as an image.

$$I_x = \frac{I(x+1,y) - I(x-1,y)}{2}$$
$$I_y = \frac{I(x,y+1) - I(x,y-1)}{2}$$

Figure 1: Pixel-wise Gradient Formulas

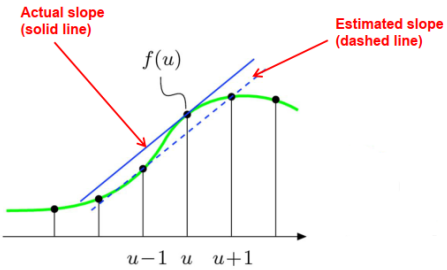


Figure 2: Central Differences

In this regard, the convolution between the original image and the horizontal kernel was realized by the function `scipy.signal.convolve2d()`. In addition to `img` and `dx` (or `dy`) as input parameters, `mode` = '`same`' was used, the parameter that controls the size of the output after the convolution operation. Therefore, the output will have the same size as the input image `img`, ensuring that subsequent operations on the output have sizes consistent with the original image. Finally, the `boundary` = '`symm`' parameter was used. This parameter controls how the convolution operation handles the boundaries of the image. Symmetric padding mirrors the image at its boundaries. This is useful to reduce boundary artifacts that can arise from the specific convolution operation performed and to avoid "natural" image corners being detected as keypoints.

```
1  dx = (1/2) * np.array([1, 0, -1]).reshape(1,3)
2  dy = (1/2) * np.array([1, 0, -1]).reshape(3, 1)
3  Ix = signal.convolve2d(img, dx, mode='same', boundary='symm')
4  Iy = signal.convolve2d(img, dy, mode='same', boundary='symm')
```

Figure 3: Step 1 Implementation

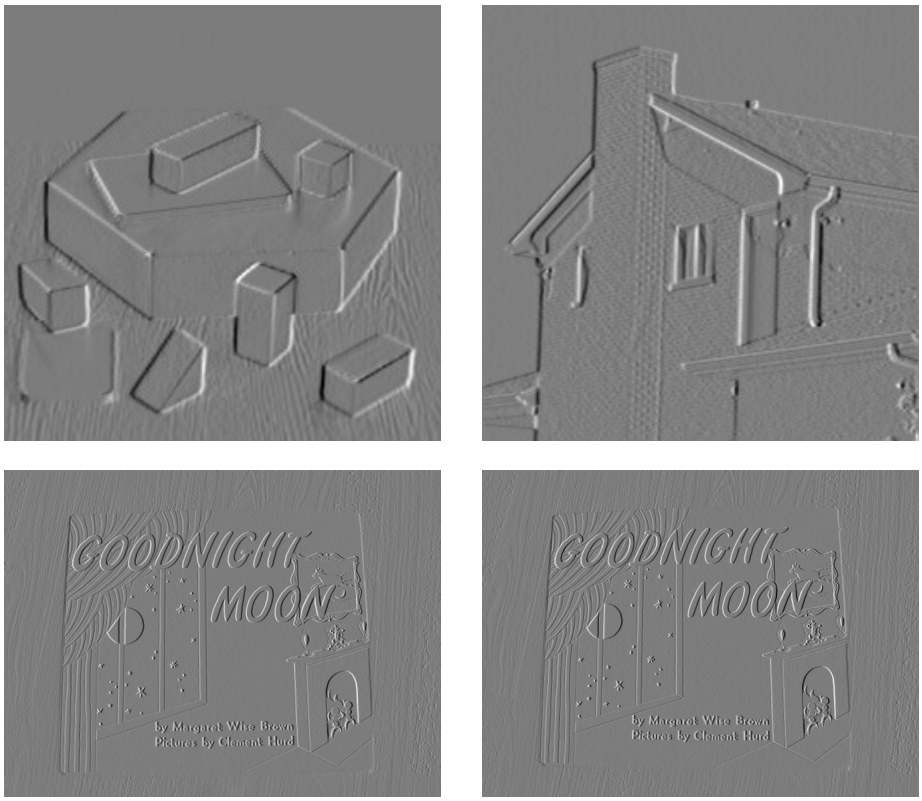


Figure 4: Gradients along x-axis

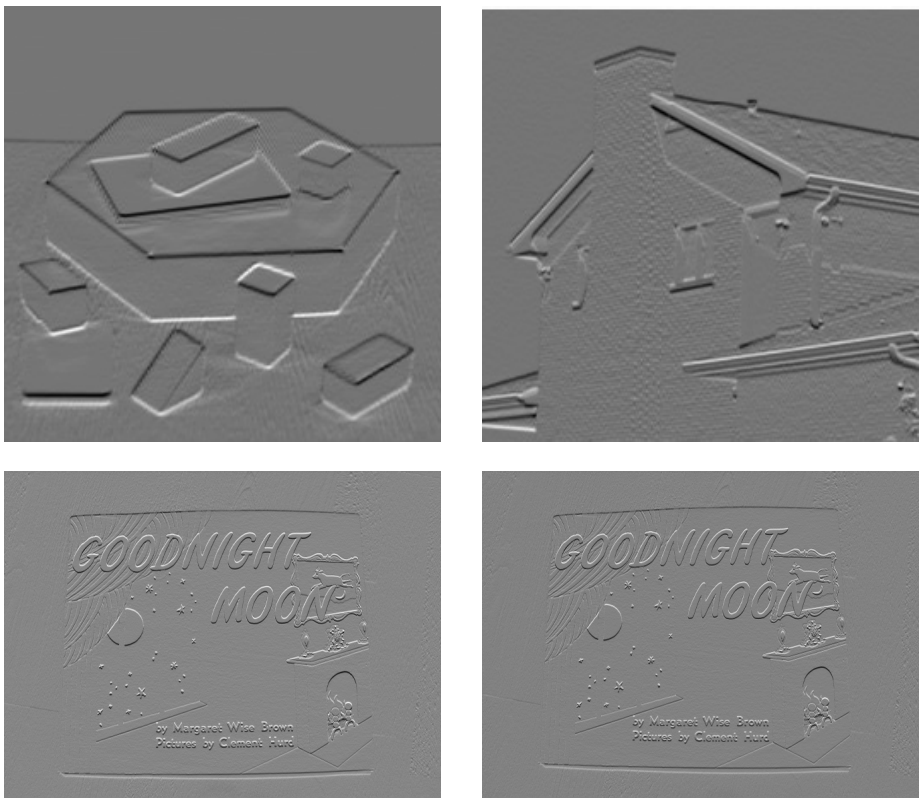


Figure 5: Gradients along y-axis

1.2 Blurring

The second step is to blur the computed gradients. This process helps in reducing noise due to the previous computation and enhancing the main features.

For this purpose, the "second moment" matrix M is computed. In this matrix, the window function is represented by a Gaussian function with a standard deviation of σ. This choice allows for a weighted sum in which more importance is given to the pixels at the center, making the result rotation invariant and eliminating the noise generated at the previous step. The computation followed the following formulation:

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

Figure 6: M Matrix Formula

```python
Ix_squared = Ix * Ix
Iy_squared = Iy * Iy
Ixy = Ix * Iy

Ix_squared_smoothed = cv2.GaussianBlur(Ix_squared, (0, 0), sigma, borderType=cv2.BORDER_REPLICATE)
Iy_squared_smoothed = cv2.GaussianBlur(Iy_squared, (0, 0), sigma, borderType=cv2.BORDER_REPLICATE)
Ixy_smoothed = cv2.GaussianBlur(Ixy, (0, 0), sigma, borderType=cv2.BORDER_REPLICATE)
```

Figure 7: Step 2 Implementation

It was decided to use the function `cv2.GaussianBlur()` with `ksize = (0,0)` so that the kernel size is computed from σ. The second parameter is σ, which indicates the standard deviation of the kernel along both the x-axis and y-axis. The suggested values for σ are 0.5, 1.0 and 2.0. A larger value produces a blurrier image, whereas a smaller value produces a less blurry image.

Since the algorithm has to take into account the boundary pixels of the image, The `borderType` parameter was set to `cv2.BORDER_REPLICATE`. This indicates that, if necessary, the pixels at the edge of the image are replicated, ensuring that the dimensions of the output image remain the same as those of the input image.

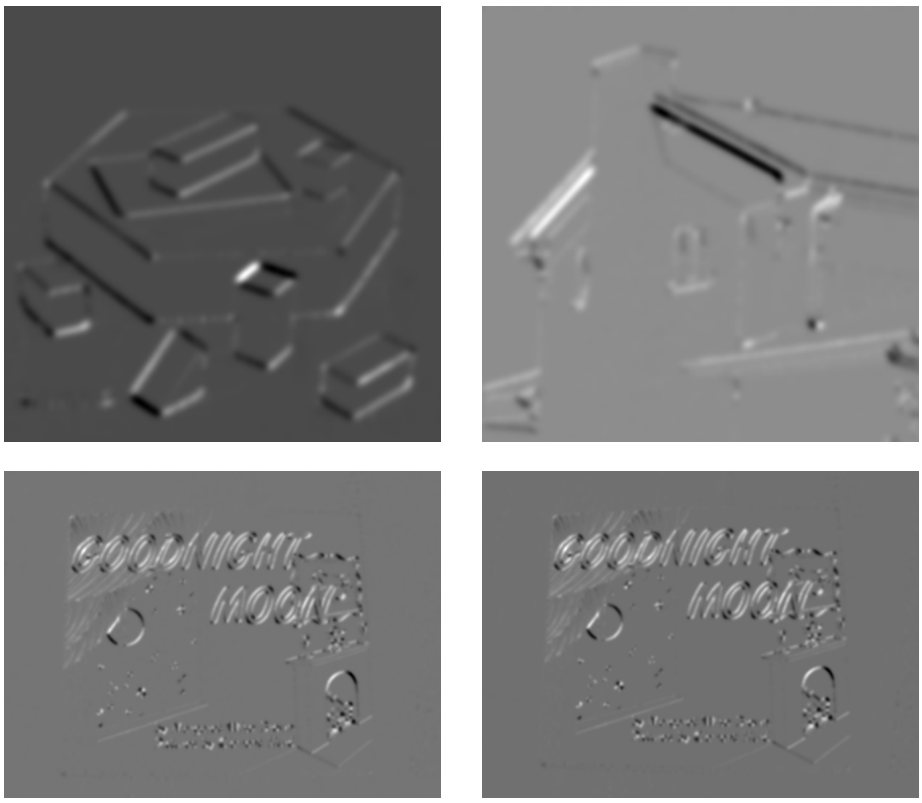


Figure 8: Blurred I_{xy}

1.3 Harris Response Function

At this point, it is necessary to calculate the Harris Response Function, which determines whether a given region of the image is a corner, an edge, or a flat one. The following formulation was followed for its calculation:

$$C = \det(M) - k \times \text{trace}(M)^2 = \left(\sum g(I_x^2) \times \sum g(I_y^2) - \left(\sum g(I_x I_y)\right)^2\right) - k \times \left(\sum g(I_x^2) + \sum g(I_y^2)\right)^2$$

where:

- $det(M)$ represents the determinant of M
- $trace(M)$ represents the sum of the diagonal elements of M
- k is a sensitivity factor, empirically chosen in the range [0.04, 0.06]

```
M_determinant = Ix_squared_smoothed * Iy_squared_smoothed - pow(Ixy_smoothed, 2)
M_trace = Ix_squared_smoothed + Iy_squared_smoothed
C = M_determinant - k * pow(M_trace, 2)
```

Figure 9: Step 3 and 4 Implementation

Based on the above definition, it can be concluded that:

- If C is a large positive value, it indicates a corner. This implies that both the eigenvalues of M are large.
- If C is around zero, it indicates a flat region.
- If C is a large negative value, it suggests an edge. This would indicate that one eigenvalue is much larger than the other.

The parameter k is an adjustment factor that affects angle detection: when k is close to zero, the response C will be dominated by the determinant of M. This tends to produce a strong response even at edges, not only at corners. With higher values of k, the trace component in C becomes dominant. This can make the algorithm less sensitive in angle detection. In this regard, see 3

1.4 Corners Detection

Subsequently, the following steps are undertaken to detect corners:

1. **Thresholding the Harris Response:** A threshold is applied to the computed Harris response values. Only pixels with a Harris response greater than this threshold *thresh* are considered potential corner candidates. This operation reduces the likelihood of weak corners and diminishes the number of false positives. The suggested range of values is $[10^{-6}$ - $10^{-4}]$.
2. **Local Maxima Identification:** The step applies a 3x3 maximum filter from the `scipy.ndimage` module to discern local maxima in the Harris response. It checks if the center pixel in each window has the highest value compared to its neighbors. This process results in a binary mask, with `True` values marking local maxima.
3. **Combination of Thresholding and Local Maxima:** Both the above conditions must be met for a pixel to be deemed a corner: the pixel must possess a Harris response exceeding the threshold and also be a local maximum.
4. **Extraction of Corner Coordinates:** Using the `np.where()` function, this step extracts the x and y coordinates of all the `True` values (i.e., candidate corners) in the `corners_mask`
5. **Storing Coordinates:** Finally, the x and y coordinates are stacked into a single 2D array. Each row of this array represents the (x, y) coordinates of a detected corner in the image.

```
C_thresholded = C > thresh
local_maxima = ndimage.maximum_filter(C, size=3) == C
corners_mask = C_thresholded & local_maxima
y_coords, x_coords = np.where(corners_mask)
corners = np.stack((x_coords, y_coords), axis=-1)
```

Figure 10: Step 5 Implementation

House Image

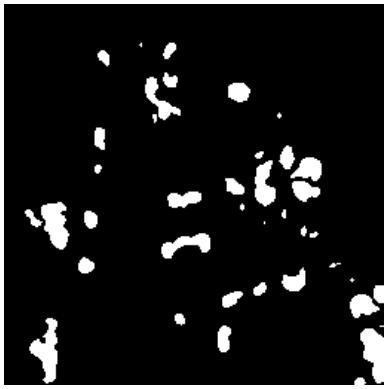


Figure 11: C Thresholded

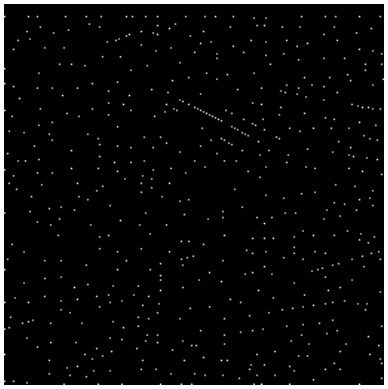


Figure 12: Local Maxima

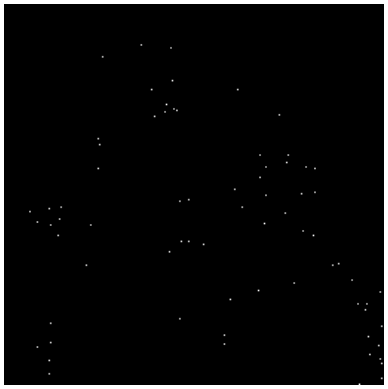


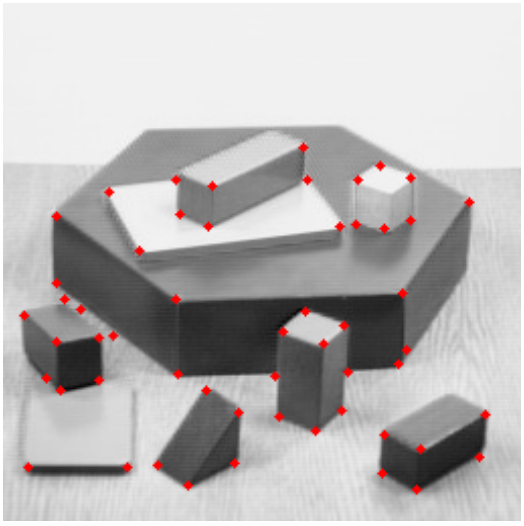
Figure 13: Corners Mask

1.5 Results

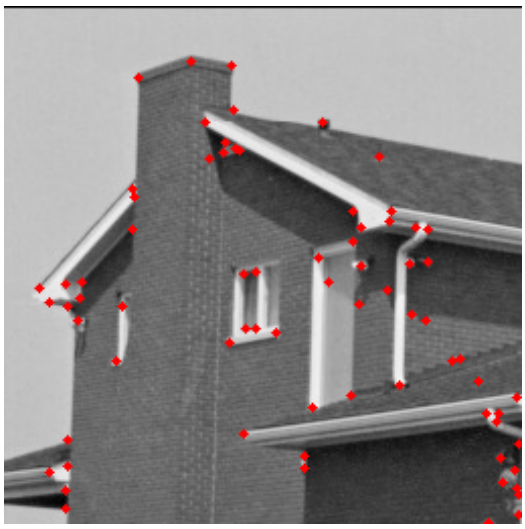
The following are the images results from the Harris Corners Detection. The values of the parameters used are given in Table 1.

sigma	k	thresh
2.0	0.06	10^{-6}

Table 1: chosen parameter values after all possible combinations (see 3)

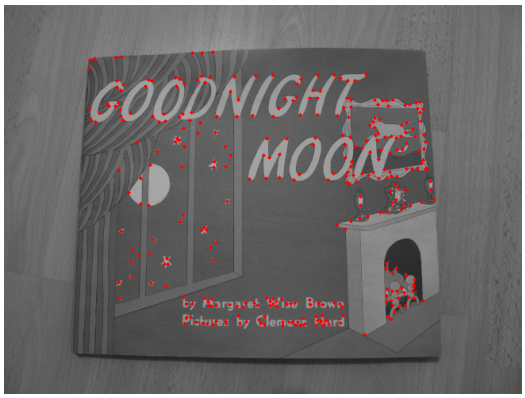


(a) 54 keypoints



(b) 72 keypoints

(c) 304 keypoints



(d) 328 keypoints

Figure 14: Harris Corners Detection Results

2 Description & Matching

The second part of the assignment involves the implementation of image matching, which aims to determine which parts of one image match which parts of another.

2.1 Filter Keypoints & Patch Extraction

If a keypoint is too close to the boundary of the image, we cannot extract a full patch around it, leading to boundary effects that may introduce artifacts or undesirable behaviors. Since the provided `extract_patches()` function is based on a 9x9 patch, I decided to implement the `filter_keypoints()` function in such a way as to exclude keypoints that were in the outer 4-pixel frame of the image, as follows:

2.2 Sum of Squared Differences (SSD)

The implementation of matching, whose goal is to find the closest descriptor in one image to a given descriptor in another image, relies on the ssd() function:

Figure 15: Filtering Mask

```python
half_patch = patch_size // 2
mask = (
    (keypoints[:, 0] >= half_patch) &
    (keypoints[:, 0] < w - half_patch) &
    (keypoints[:, 1] >= half_patch) &
    (keypoints[:, 1] < h - half_patch)
)
filtered_keypoints = keypoints[mask]
```

Figure 16: Step 1 Implementation

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

Its purpose is to compute the squared difference between descriptors of two images, given the following two inputs:

- `desc1`: descriptors from the first image of shape (q1, feature_dim). In my case: (304, 81).
- `desc2`: descriptors from the second image of shape (q2, feature_dim). In my case: (328, 81).

The output is `distances`: A matrix of size (q1, q2) storing the squared differences. In my case: (304, 328)

For visualization purposes, here are 3 sample descriptors for image I1 (`desc1`), where `q1 = 304` (i.e. the number of keypoints detected) and `feature_dim` is the number of elements per patch. Since the patch is 9x9 then `feature_dim = 81`:

3 samples of `desc1`

I decided to leverage the `NumPy` broadcasting to perform the operations on the two arrays of different shapes. Therefore, squared differences between descriptors are computed and subsequently summed up along the feature dimensions to get the SSD. Being a similarity measure, if the distance between two descriptors is small, then the descriptors are very similar; if it's large, they're very different.

```python
desc1 = desc1[:, np.newaxis, :]
desc2 = desc2[np.newaxis, :, :]
squared_diffs = pow((desc1 - desc2), 2)
distances = np.sum(squared_diffs, axis=2)
```

Figure 17: Step 2 Implementation

2.3 One Way Matching

The primary advantage of the "one_way" method is its simplicity and efficiency, at the cost, however, of its lack of robustness. For each descriptor in `desc1`, this method finds the descriptor in `desc2` that has the smallest distance (i.e., the nearest neighbor). This is done using the `np.argmin()` function on the distance matrix. The result is an array of indices that point to the best match in `desc2` for each descriptor in `desc1`. The following step involves creating matches, i.e. pairing up each descriptor in `desc1` with its best match in `desc2`. This creates an array of matches, where each match is represented by a pair of indices, one from `desc1` and one from `desc2`.

```
1  if method == "one_way":
2      min_indices = np.argmin(distances, axis=1)
3      matches = np.column_stack((np.arange(q1), min_indices))
```

Figure 18: One Way Matching Implementation

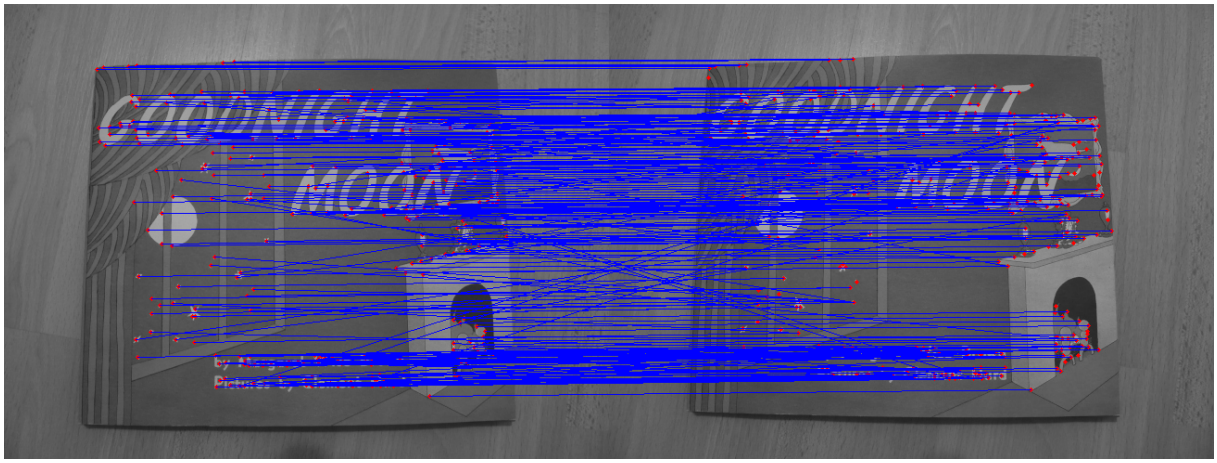


Figure 19: One Way Matching results (304 matches)

2.4 Mutual Matching

The "mutual" method is a refinement of the previous approach, and it aims to provide more reliable matches by considering mutual consistency. In addition to what was done with `desc1` for the "one-way" metho, for each descriptor in `desc2` it finds its nearest neighbor in `desc1`. At this point, for a descriptor *d1* in `desc1` and a descriptor *d2* in `desc2` to be considered a match, *d2* should be the best match (nearest neighbor) for *d1* when querying in `desc2`. Similarly, *d1* should be the best match for *d2* when querying in `desc1`. If both these conditions hold, then *d1* and *d2* are mutually best matches and are thus paired as a *true* match.

The implementation (see Figure 20) is as follows:

- *Line*2: For every descriptor in `desc1`, find the index of the descriptor in `desc2` that is the closest match.
- *Line*3: For every descriptor in `desc2`, find the index of the descriptor in `desc1` that is the closest match.
- *Line*4: It checks mutual matches between `desc1` and `desc2`, after reshaping `min_indices_1` to be a column vector. After having reshaped `min_indices_1` to be a column vector, the "==" operation checks if for every i_{th} descriptor in `desc1`, its closest descriptor in `desc2` also has `i` as its closest descriptor in `desc1`. The matrix `mutual_mask` is obtained, where each row i and column j is *true* if the descriptor i in `desc1` and descriptor j in `desc2` are mutual best matches.
- *Line*5: It extracts the row and column indices where mutual matches are *true*.

```python
elif method == "mutual":
    min_indices_1 = np.min(distances, axis=1)
    min_indices_2 = np.min(distances, axis=0)
    mutual_mask = (min_indices_1[:, np.newaxis] == min_indices_2)
    matches = np.column_stack(np.where(mutual_mask))
```

Figure 20: Mutual Matching Implementation

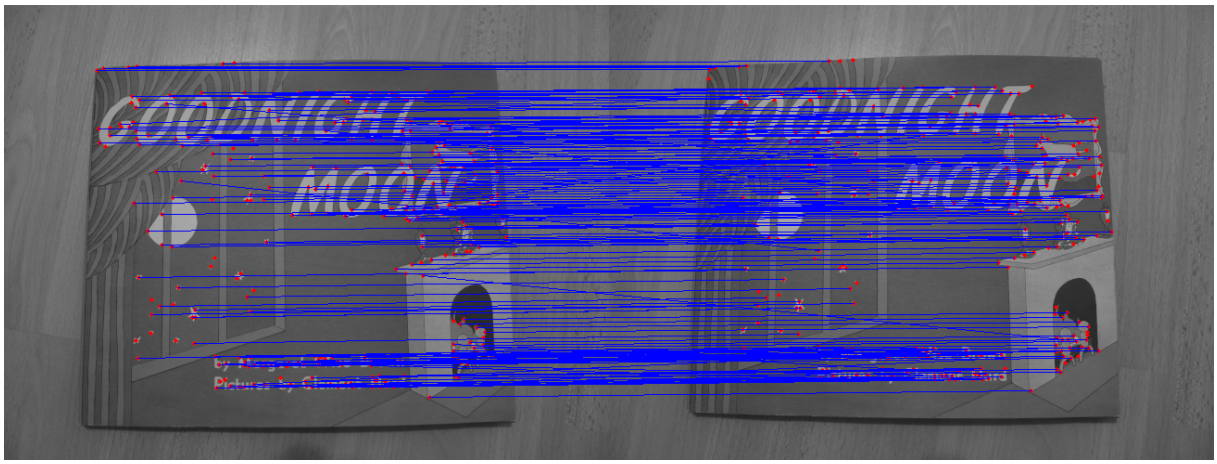


Figure 21: Mutual Matching results (227 matches)

2.5 Ratio Matching

The "ratio" method leverages the ratio of the distances to the nearest and the second-nearest descriptors. The intuition is to ensure that the closest match is distinctively closer than the second-best match.

For each descriptor in `desc1`, it computes the distance to every descriptor in `desc2` (matrix of distances). Then for each descriptor d1 in `desc1`, it determines:

- Its nearest neighbor (i.e. the most similar descriptor) in `desc2`
- The second nearest neighbor (i.e. the second most similar descriptor) in `desc2`.

After that, for each descriptor *d1* it computes the ratio of the distance of *d1* to the nearest neighbor over the distance of *d1* to the second nearest neighbor. Finally, for a descriptor *d1* in `desc1` and its nearest neighbor in `desc2` to be considered a valid match, the ratio computed in the previous step should be below a certain threshold, `MATCHING_RATIO_TEST_THRESHOLD`. Its default value is 0.5.

The implementation (see Figure 22) is as follows:

- *Line*2: For each descriptor in `desc1`, it finds the distance to its second-closest match in `desc2`. Using `np.partition()` is it possible to get the k smallest elements in the array without fully sorting it. Here, $k = 2$, so it arranges the smallest and second smallest elements in the first two positions. Here, `[:, 1]` extracts the second smallest element (second-closest match) for each descriptor in `desc1`.
- *Line*3: It finds the distance of the best match for each descriptor in `desc1`. The best match's distance is given by `distances[np.arange(q1), np.argmin(distances, axis=1)]`. Then, this distance is divided by the distance of the second-best match for each descriptor (from the previous step). This gives the ratio of the best match's distance to the second-best match's distance.
- *Line*4: For each descriptor in `desc1`, it checks if the distance ratio computed in the previous step is less than the given threshold. This results in a boolean array, `ratio_mask`.
- *Line*5: Finally, for each descriptor in `desc1` for which `ratio_mask` is *true*, it finds the index of its best match in `desc2`. This is given by `np.argmin(distances, axis=1)[ratio_mask]`. Then it combines the indices of these descriptors in `desc1` and their best match indices in `desc2` to produce the final matches.

```python
elif method == "ratio":
    second_min_distances = np.partition(distances, 2, axis=1)[:, 1]
    distance_ratio = distances[np.arange(q1), np.argmin(distances, axis=1)] / second_min_distances
    ratio_mask = (distance_ratio < ratio_thresh)
    matches = np.column_stack((np.where(ratio_mask)[0], np.argmin(distances, axis=1)[ratio_mask]))
```

Figure 22: Ratio Matching Implementation

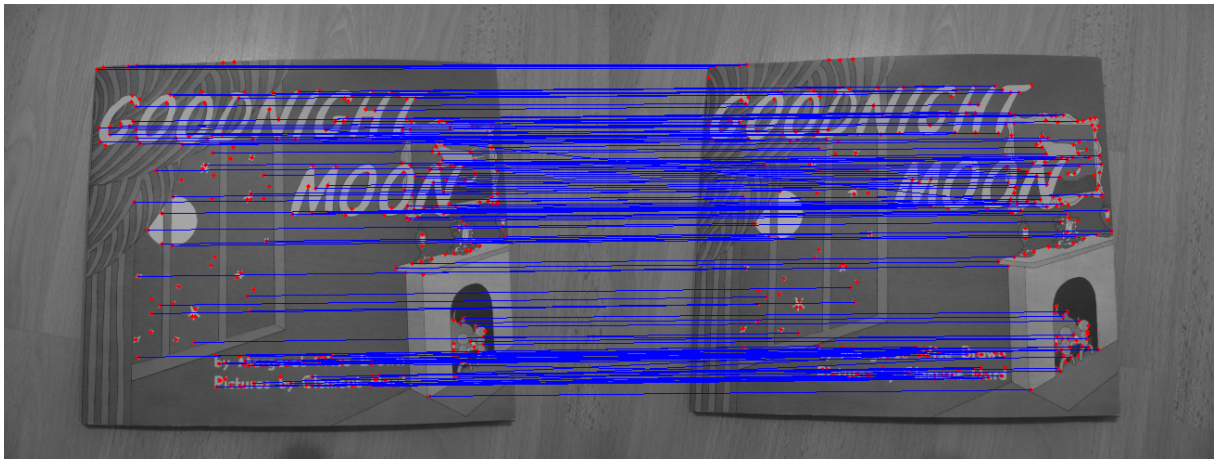


Figure 23: Ratio Matching results (166 matches)

3 Interesting Remarks

As mentioned in the previous sections, before defining the parameter values with which the results shown were obtained, I decided to try different combinations of parameters available. Regarding the Harris Corner Detection, the parameters whose values may vary are as follows:

- $\sigma = [0.5, 1.0, 2.0]$
- $k = [0.04 - 0.06]$
- $thresh = [10^{-6} - 10^{-4}]$

All the various combinations of these parameters were calculated as shown in Figure 24, and the results were plotted for each combination. By visual inspection, it was found that the best values of these parameters were those for which between 47 and 59 keypoints were obtained in the Harris Corner Detection of the "blocks" image.

```python
# Parameters range
sigmas = [0.5, 1.0, 2.0]
ks = [0.04, 0.045, 0.05, 0.055, 0.06]
threshs = [1e-6, 1e-5, 1e-4]
parameters_combinations = product(sigmas, ks, threshs)
```

Figure 24: Parameters Combinations Computation

In particular, I noticed that:

- A smaller σ highlights finer details, whereas a larger σ makes the detector more robust against small noise but may miss finer corners.
- The parameter k determines the balance between edge detection and corner detection. A small value of k results in the detection of sharp corners, while a larger value might detect a mix of edges and corners.

- A high *thresh* results in fewer but more prominent corners being detected (potentially missing some true corners), whereas a low *thresh* detects more corners, including those with a weaker Harris response (potentially introducing some false corners)

Below are some examples where there was an **overestimation** of the keypoints found:

σ=0.5, k=0.04, $thresh$=10^{-6}

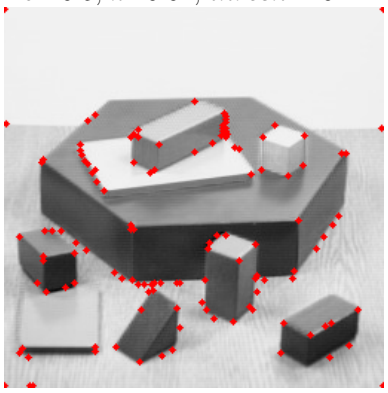


Figure 25: 127 keypoints

σ=0.5, k=0.04, $thresh$=10^{-6}

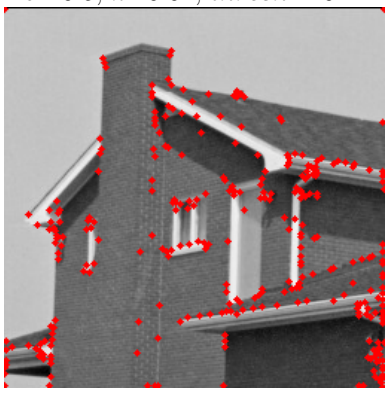


Figure 26: 294 keypoints

σ=0.5, k=0.06, $thresh$=10^{-6}

Figure 27: 929 keypoints

Below are some examples where **underestimation** occurred:

σ=1.0, k=0.04, $thresh$=10^{-4}

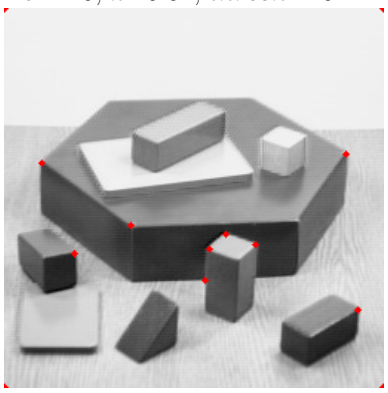


Figure 28: 13 keypoints

σ=0.5, k=0.04, $thresh$=10^{-4}

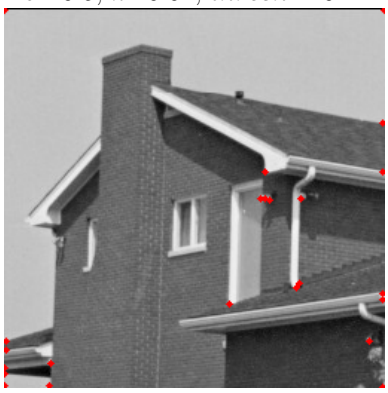


Figure 29: 23 keypoints

σ=2.0, k=0.06, $thresh$=10^{-5}

Figure 30: 134 keypoints

In light of the results and observations found, I decided to use the parameter values given in table 1.

Regarding matching, the parameter that can vary is `MATCHING_RATIO_TEST_THRESHOLD`.
In particular, I noticed that:

- A high `MATCHING_RATIO_TEST_THRESHOLD`, such as 0.9, means being more permissive. With a threshold of 0.9, we're accepting matches even if the best matching descriptor is only slightly better than the second best. This means many more matches will pass the threshold check, resulting in a larger number of matches (272 matches in my case).
- A low `MATCHING_RATIO_TEST_THRESHOLD`, such as 0.1, is much more stringent. With a threshold of 0.1, only those descriptors with a distinct difference in distance between the best match and the second-best match are deemed valid. This dramatically reduces the number of matches that pass the threshold, resulting in far fewer matches (31 matches in my case).

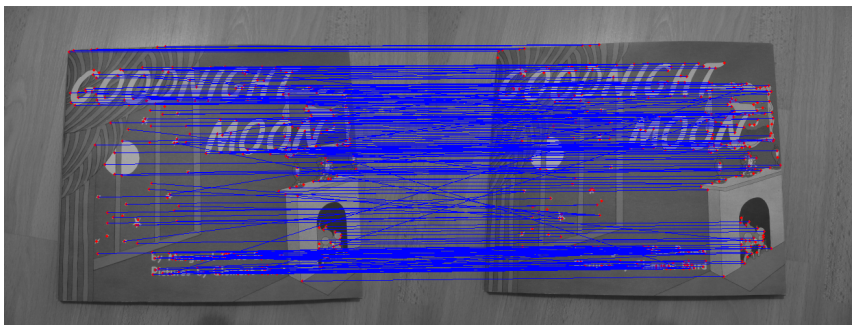


Figure 31: Threshold = 0.9, 272 matches

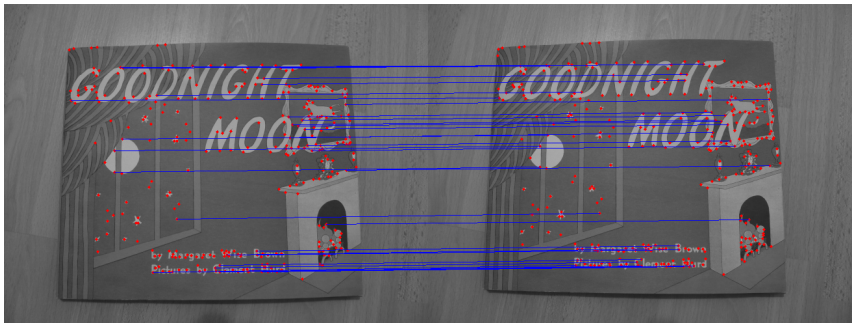


Figure 32: Threshold = 0.1, 31 matches