Computer Vision
Lab Assignment Report - Local Features

263-5902-00L Computer Vision HS2022
ETH Zurich

Anisha Mohamed Sahabdeen

October 15, 2022

Abstract

In order to perform computer vision tasks such as image stitching or video stabilization, different images need to be aligned so that they can be seamlessly stitched into a composite picture. This can be achieved by first identifying a set of feature points for each image under consideration (*feature detection*), and then establishing an appropriate correspondence between them (*feature matching*), so that an in-between view can be generated. The purpose of this report is to present a Python implementation of a complete feature detection and matching pipeline, based upon the Harris corner detector followed by different feature matching techniques.

1 Detection

During the *feature detection* stage, each image is searched for interest points. In particular, corners can be viewed as salient features of an image as they present large gradients in at least two significantly different orientations, making them the easiest to localize.

One of the most commonly used corner detection operators is the Harris corner detector [1], which makes use of the *auto-correlation function* to discriminate whether or not an image patch contains any corners. Generally, the Harris corner detector algorithm can be divided into the following steps:

1. Calculation of image gradients
2. Definition of the auto-correlation matrix
3. Calculation of the Harris response function
4. Application of detection criteria

Details of the Python implementation of the Harris corner detector and experimental results are provided below.

1.1 Image gradients

The gradients of a 2-dimensional grayscale image I with respect to x and y can be computed as follows:

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

It is possible to easily perform these calculations by convolving the image with the derivative filter D_x and D_y.

$$I_x = I * D_x, \quad D_x = \frac{1}{2} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

$$I_y = I * D_y, \quad D_y = \frac{1}{2} \cdot \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

The image gradients can be readily obtained using the `scipy.signal.convolve2d` function.

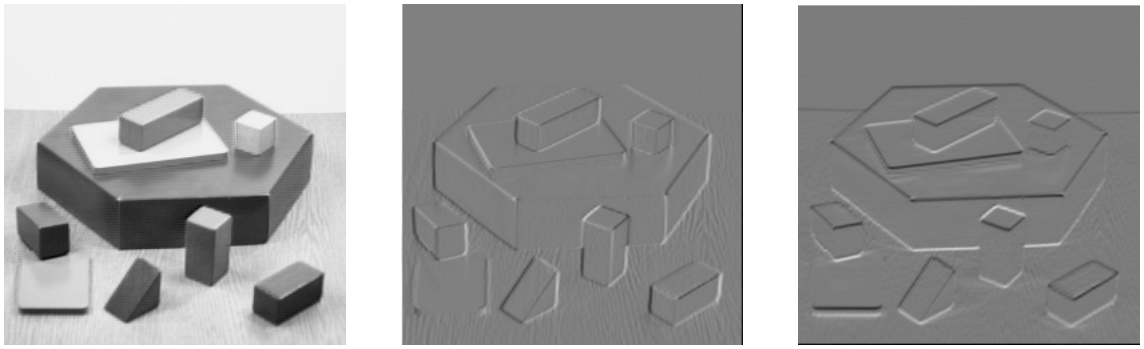


Figure 1: Original image and respective image gradients with respect to x and y.

```
Ix = convolve2d(img, np.array([1, 0, -1]).reshape(1, -1) * 0.5, mode="same")
Iy = convolve2d(img, np.array([1, 0, -1]).reshape(-1, 1) * 0.5, mode="same")
```

1.2 Local auto-correlation matrix

Given an image I, the auto-correlation function E_{AC} measures the weighted sum of squared difference (SSD) of an image patch of centre $\mathbf{p}$ and another patch whose centre is displaced by vector $\mathbf{u}$:

$$E_{AC}(\mathbf{u}) = \sum_{\mathbf{p}' \in \mathcal{N}(\mathbf{p})} w_{p'} \cdot [I(\mathbf{p}' + \mathbf{u}) - I(\mathbf{p}')]^2,$$

where $\mathcal{N}(\mathbf{p})$ is a neighbourhood of point $\mathbf{p}$ and w is a local weighing function. Note that by weighing each value based on its distance from the centre pixel $\mathbf{p}$ the result is rotation invariant.

It can be shown ([2]) that E_{AC} can be locally approximated by a quadratic error function:

$$\begin{aligned} E_{AC}(\mathbf{u}) &\approx \sum_{\mathbf{p}' \in \mathcal{N}(\mathbf{p})} w_{p'} \cdot [\ (I_x, I_y)(\mathbf{p})\ \cdot \mathbf{u}]^2 \\ &= \mathbf{u}^T \cdot M_p \cdot \mathbf{u}, \end{aligned}$$

with

$$M_p = \sum_{\mathbf{p}' \in \mathcal{N}(\mathbf{p})} w_{p'} \begin{bmatrix} I_x(\mathbf{p}')^2 & I_x(\mathbf{p}') \cdot I_y(\mathbf{p}') \\ I_x(\mathbf{p}') \cdot I_y(\mathbf{p}') & I_y(\mathbf{p}')^2 \end{bmatrix},$$

where M_p is the local auto-correlation matrix for pixel $\mathbf{p}$.

By using a Gaussian window function $\mathrm{g}(\sigma)$, the auto-correlation matrix M becomes:

$$M = \mathrm{g}(\sigma) * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \mathrm{g}(\sigma) * I_x^2 & \mathrm{g}(\sigma) * I_x I_y \\ \mathrm{g}(\sigma) * I_x I_y & \mathrm{g}(\sigma) * I_y^2 \end{bmatrix} = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix},$$

where S_{xx}, S_{xy} and S_{yy} are 2-dimensional matrices with the same shape as I, and where each element $s_{i,j}$ corresponds to the sum of the values of the Gaussian window applied to the product of image gradients in the (i, j)-th position.

The sum matrices S_{xx}, S_{xy} and S_{yy} can be thus computed by convolving the products of the image gradients with a Gaussian filter, by means of the function `cv.GaussianBlur`. Note that the standard deviation σ of the Gaussian filter is to be considered a hyperparameter.

```
Ixx = Ix ** 2
Iyy = Iy ** 2
Ixy = Ix * Iy

Sxx = cv.GaussianBlur(Ixx, (5, 5), sigma, borderType=cv.BORDER_REPLICATE)
Syy = cv.GaussianBlur(Iyy, (5, 5), sigma, borderType=cv.BORDER_REPLICATE)
Sxy = cv.GaussianBlur(Ixy, (5, 5), sigma, borderType=cv.BORDER_REPLICATE)
```

Hence, the locally defined 2x2 matrix M_p for each pixel $\mathbf{p}$ can be written as

$$M_p = \begin{bmatrix} S_{xx}(\mathbf{p}) & S_{xy}(\mathbf{p}) \\ S_{xy}(\mathbf{p}) & S_{yy}(\mathbf{p}) \end{bmatrix}.$$

Note that M_p is symmetric, and produces two eigenvalues $(\lambda_{p,0}, \lambda_{p,1})$ and two eigenvector directions.

1.3 Harris response function

The Harris response function or *cornerness* function $\mathcal{C}(i,j)$ can be defined as

$$\begin{aligned} \mathcal{C}(i,j) &= \Delta(M_{(i,j)}) - k \cdot [tr(M_{(i,j)})]^2 \\ &= \lambda_{(i,j),0} \cdot \lambda_{(i,j),1} - k \cdot (\lambda_{(i,j),0} + \lambda_{(i,j),1})^2, \end{aligned}$$

where k is an empirically determined hyperparameter ($k \in [0.04, 0.06]$).
Using closed-form formulas, it is possible to compute the cornerness function as follows:

$$\mathcal{C} = (S_{xx} \cdot S_{yy} - S_{xy}^2) - k \cdot (S_{xx} + S_{yy})^2.$$

1.4 Detection criteria

For a pixel $\mathbf{p}$ to be considered a corner, its response $\mathcal{C}(\mathbf{p})$ must suffice two conditions:

1. $\mathcal{C}(\mathbf{p}) \geq T, T > 0$ (threshold)
2. $\mathcal{C}(\mathbf{p}) = \max_{\mathbf{p}' \in \mathcal{N}(\mathbf{p})} \mathcal{C}(\mathbf{p}')$ (local maximality)

For the sake of code efficiency, both conditions can be checked by applying boolean masks on `numpy.ndarray` objects. The function `numpy.ndarray.maximum_filter` was used to verify condition (2).

```
condition1 = C > thresh
condition2 = (C == scipy.ndimage.maximum_filter(C, 3))

corners = np.argwhere((condition1 & condition2))
```

1.5 Hyperparameter selection

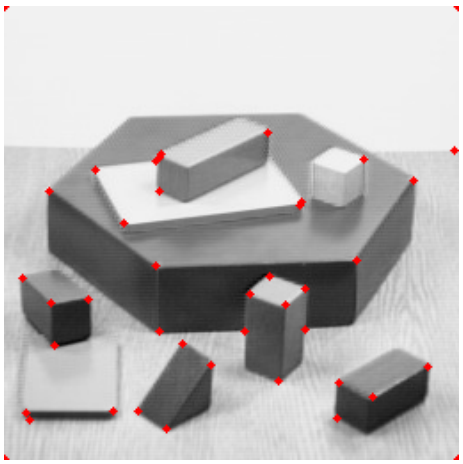
In order to set the value of the hyperparameters for the test images of the assignment, I ran different simulations on the given images varying:

- the parameter σ of the Gaussian kernel used during the computation of the auto-correlation matrix;
- the parameter k of the Harris response function;
- the threshold parameter T used in condition (1) of the corner detection criteria.

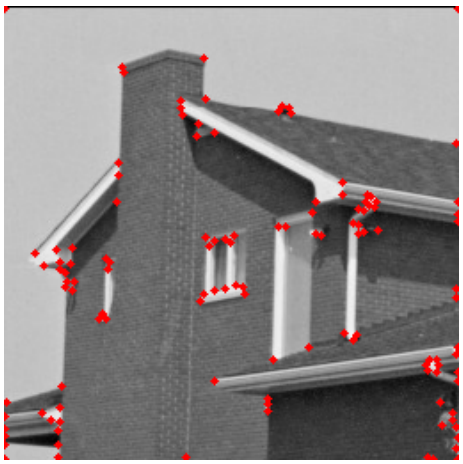
I observed that parameter T had the most influence on the number of detected corners: for values of $T \geq 10^{-4}$, most of the corners were not recognized, thus indicating a clear underestimation of feature points, whereas for $T \leq 10^{-6}$ a great number of non-corners were identified as feature points, thus suggesting an overestimation of such points.

Varying the value of k in the range $[0.04, 0.06]$ had little effect on the number of detected corners, yet an inverse proportion trend was noticeable. As a matter of fact, increasing k leads to a decreasing Harris response.

Overall, the best outcome was obtained with the following parameters: $\sigma = 1.5,\ k = 0.04,\ T = 10^{-5}$.



$\sigma = 1.5,\ k = 0.04,\ T = 10^{-5}$
52 points detected



$\sigma = 1.5,\ k = 0.04,\ T = 10^{-5}$
111 points detected

2 Description & Matching

After detecting the feature points of the images, in the *feature matching* stage it is required to determine which features come from corresponding locations in different images. This can be achieved by first converting the region around each detected keypoint into a more compact and stable descriptor that can be matched against other descriptors, and then searching for likely matching candidates in other images according to different criteria.

2.1 Local descriptors

Before extracting the descriptors out of the region around the detected corners, a filtering operation is performed on the corners so as to leave out the ones that are too close to the image borders, in order to avoid out-of-bound issues.

```
def filter_keypoints(img, keypoints, patch_size=9):

    img_h, img_w = img.shape
    d = patch_size // 2

    xcoords, ycoords = keypoints[:,0].flatten(), keypoints[:,1].flatten()
    filtered_xcoords_indices = np.logical_and(xcoords >= d, xcoords <= (img_w - d))
    filtered_ycoords_indices = np.logical_and(ycoords >= d, ycoords <= (img_h - d))

    return keypoints[filtered_xcoords_indices & filtered_ycoords_indices]
```

A possible function for computing keypoint descriptors is given below.

```
def extract_patches(img, keypoints, patch_size = 9):
    h, w = img.shape[0], img.shape[1]
    img = img.astype(float) / 255.0
    offset = int(np.floor(patch_size / 2.0))
    ranges = np.arange(-offset, offset + 1)
    desc = np.take(img, ranges[:, None] * w + ranges +
           (keypoints[:, 1] * w + keypoints[:, 0])[:, None, None])
    desc = desc.reshape(keypoints.shape[0], -1)
    return desc
```

2.2 SSD one-way nearest neighbours matching

The sum of squared differences(SSD) between descriptor p and descriptor q is defined as:

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

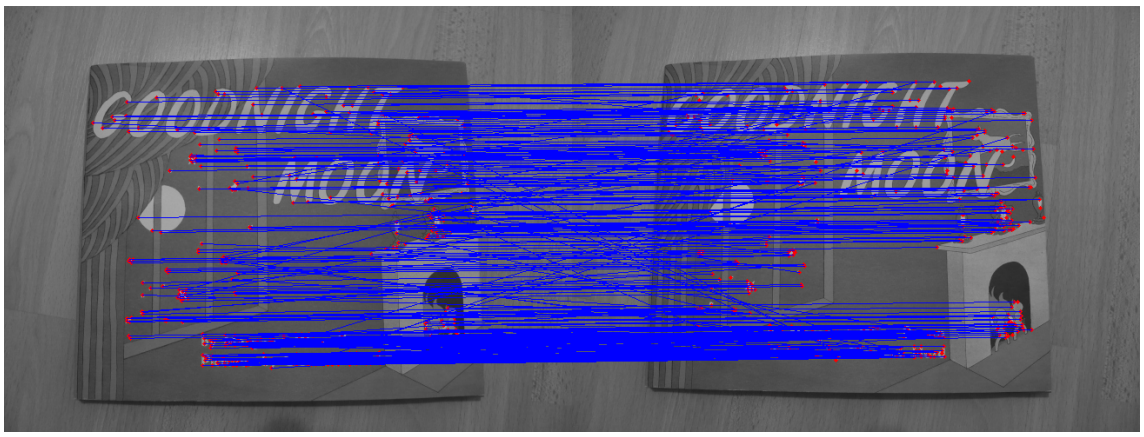


Figure 2: One-way nearest neighbour (382 points matching).

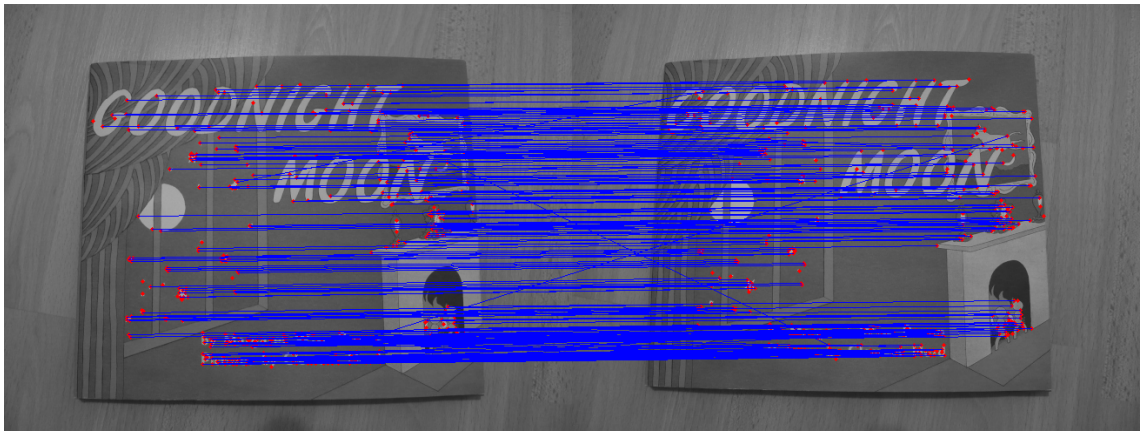


Figure 3: Mutual nearest neighbour (278 points matching).

In order to perform one-way nearest neighbours matching, for each feature descriptor of the first image, it is required to select the feature descriptor of the second image with the smallest distance from the first one. This is achieved by means of the `numpy.min` function.

```
distances = ssd(desc1, desc2)
nearest_neighbour = distances == np.min(distances, axis=-1).reshape(-1, 1)
matches = np.argwhere(nearest_neighbour)
```

Again, for the sake of code efficiency Python for loops were avoided by using vectorized computation trough the use of the numpy library.

2.3 Mutual nearest neighbours / Ratio test

For the implementation of the mutual nearest neighbours, in addition to the nearest neighbour requirement, it is also asked that the same matching is obtained when swapping the images.

```
nearest_neighbour_opposite = distances == np.min(distances, axis=-2).reshape(1, -1)
matches = np.argwhere(np.logical_and(nearest_neighbour, nearest_neighbour_opposite))
```

Lastly, for the ratio test, it is required that the ratio between the first and second nearest neighbour is below a certain threshold.

```
two_least_distant = np.partition(distances, 2)[:,:2]
ratio_check = two_least_distant[:,0] / two_least_distant[:,1] < ratio_thresh
matches = np.argwhere(nearest_neighbour)[ratio_check]
```

Overall, mutual matching performed better than nearest neighbour, as the number of crossings greatly decreased, but the best performing feature matching technique was the ratio test, for which no crossing were generated.

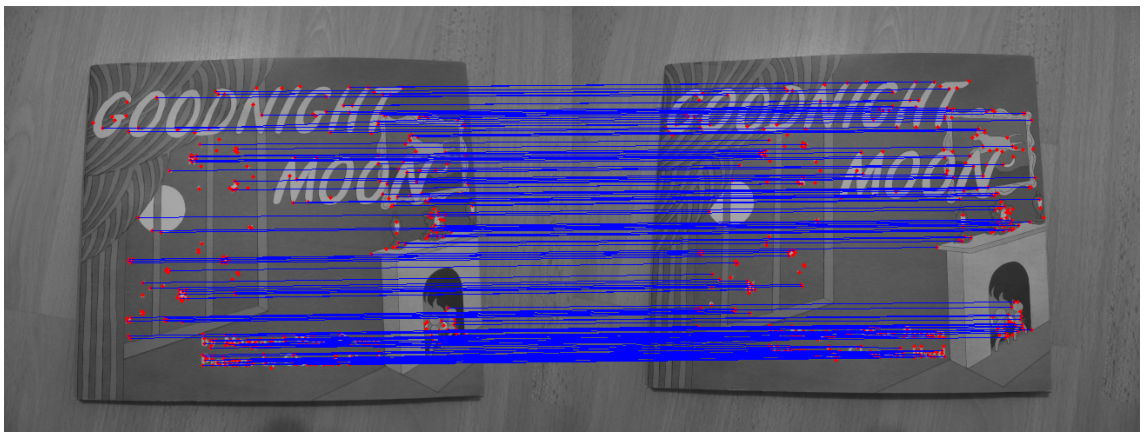


Figure 4: Mutual nearest neighbour (207 points matching).

References

[1] Christopher G. Harris and M. J. Stephens. A combined corner and edge detector. In *Alvey Vision Conference*, 1988.

[2] Richard Szeliski. Computer vision algorithms and applications, 2011.