08/10/2023 Nicolas Pfitzer

Computer Vision- Lab 2 Report: Local Features

1) Task 1: Harris Corner detection:

The Harris corner detection is performed in 5 steps:

a) **The image gradients** are computed using a Sobel filter. The derivation matrices are convolved with image data to produce the image gradients in X and Y.
b) **The squares (Ixx, Iyy and Ixy) of the image gradients are blurred** using a Gaussian filter to suppress noise and enhance the corner detection.
c) **The Harris response function** is calculated from:

$$R = \det(M) - k * trace^2(M)$$

Where "M" is the local auto-correlation Matrix expressed as:

$$M = \begin{pmatrix} Ixx & Ixy \\ Ixy & Iyy \end{pmatrix}$$

Therefore, without explicitly computing "M", we can calculate the Harris response function as:

$$R = (Ixx * Iyy - Ixy^2) - k * (Ixx + Iyy)^2$$

d) **The non-maximum suppression** is computed to isolate each corner, so that it's only detected once. The built-in *ndimage.maximum_filter()* function detects the maximum Harris response value in a $[20p \times 20p]$ window and all other values are suppressed using *numpy.where()*.
e) **The corner coordinates are extracted** by finding the elements of the suppressed Harris response matrix that are higher than a threshold of 2e-4.

2) Task 2: Description & Matching:

The second task of this assignment essentially required to implement three functions:

1. *Filter_keypoints*
2. *Ssd (Sum of squared differences)*
3. *Match_descriptors*

The first step of the second task was to implement the *filter_keypoints* function to remove the corners that are too close to the borders of the image. Without going too much into detail of this tedious process, the function works by removing all the key points within 4 pixels of the borders.

The purpose of the *ssd()* function is to calculate the feature "distances" between descriptors from I1 and I2. Each descriptor contains a 81-element long array containing information about the intensity of each pixel within the 9x9 patch. The sum of square differences between each pixel of each descriptor is computed into a q1xq2 matrix (q1/2: number of descriptors in image 1 / 2). For example, the ‘distance’ between the *i*’th descriptor of the first image and the *j*’th descriptor of the second image would be represented by the [*i,j*] element of the return matrix.

Finally, the *match_descriptors* function allows us to bring everything together. Using the descriptor distances computed in the *ssd()* function, we can match descriptors from both images together. Three methods were implemented:

a) “One way” matching:

Descriptors are matched together by finding the *j*’th descriptor in image n°2, that is closest to the *i*’th descriptor of image n°1.
The matching process is said to be “One way”, because the distance matrix is not always symmetric. Meaning that, whilst the match is true from image n°1 to image n°2, it is not necessarily the best match the other way around. In fact, “One way” matching tends to result in higher cases of miss-matching, where supposedly matched descriptors have very different locations on the image.

b) “Mutual” matching:

“Mutual” matching is like the “One way” method, however, only the symmetric minima of the distance matrix are retained, thus resulting in higher accuracy.

c) “Ratio” matching:

Instead of simply comparing one key point’s descriptor to another's, we calculate the ratio of the distances between the best and second-best matching descriptors. If the ratio is above a certain threshold, the match is considered secure, otherwise it’s ignored.