Computer Vision

Assignment 2: Local features

Student: Ivan Alberico

1. Harris corner detector

The first part of the assignment requires you to implement a Harris corner detector in order to find the points of major interest in an image. The first step was about computing the gradients of the image along the x and y directions, which have the following form:

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

Instead of computing them pixel-wise, according to the previous formula, gradients were obtained by convolving the original image with some specific filters, as in the following way:

$$I_x(i,j) = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * I(i,j) \qquad I_y(i,j) = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}^T * I(i,j)$$

Once the gradients were obtained, next step was computing the auto-correlation matrix, which is defined by the following expression:

$$M = \sum_{(x,y)\in W} w(x,y) \begin{bmatrix} I_x{}^2 & I_x I_y \\ I_x I_y & I_y{}^2 \end{bmatrix}$$

In order to make our results to be rotationally invariant, Gaussian filters were used as local weighting w of the matrix. By introducing Gaussian filters, a hyperparameter is introduced, which is the standard deviation σ. Hence, the auto-correlation matrix becomes:

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

What needs to be done at this point is to compute the Harris response function $C(i,j)$ that contains the information about whether each pixel is a possible corner, or it belongs to a flat region or an edge. The response function must be computed for all pixels and it has the following form:

$$C(i,j) = \det(M_{i,j}) - k\,Tr^2(M_{i,j}) \qquad k \in [0.04,\ 0.06]$$

where k is a hyperparameter to be chosen. The Harris response function can be computed in two different ways: it can be either obtained by actually computing the determinant and the trace with matrix multiplications (element-wise multiplications) or it can be derived from the eigenvalues of M, knowing that $\det(M) = \lambda_1 \lambda_2$ and $Tr(M) = \lambda_1 + \lambda_2$.

$$C = g\left(I_x^{\,2}\right)g\left(I_y^{\,2}\right) - \left[g\left(I_x I_y\right)\right]^2 - k\left[g\left(I_x^{\,2}\right) + g\left(I_y^{\,2}\right)\right]^2$$

$$C = \lambda_1 \lambda_2 - k\left(\lambda_1 + \lambda_2\right)^2$$

Both ways were tested for the assignment, in order to check the correctness of the results.

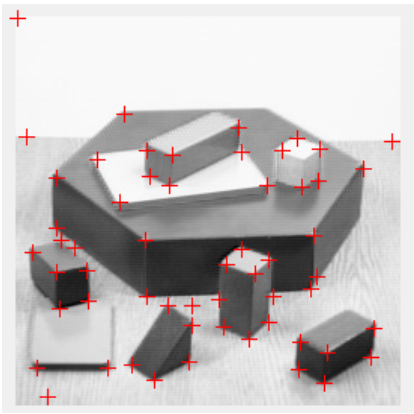
The very last step to obtain the Harris corner detector was implementing non-maximum suppression, since the two conditions for a pixel to be a corner are that the corresponding response function $C(i,j)$ is above a certain threshold (hyperparameter) and that it is a local maxima in its 3x3 neighbourhood. The algorithm I used to implement non-maximum suppression is the following:

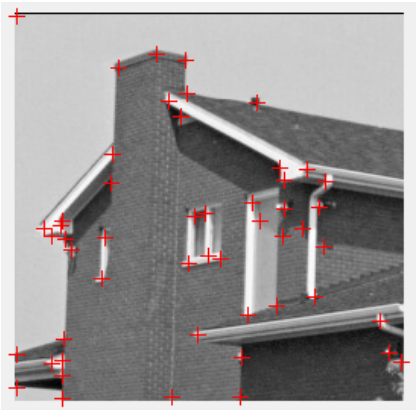
- For each pixel (i,j) of C, if its value is higher than a certain specified threshold, I consider the relative 3x3 window around it, which is the matrix $W = C(i-1:i+1, j-1:j+1)$.

- In order to check local maxima also on the border pixels of the image, I added a padding of zeros all around the image (a single layer).

- I computed *imregionalmax(W)* which gives as output a Boolean matrix having the value 1 in the position of the maximum element of the 3x3 window and 0 elsewhere. By multiplying this "mask" with the actual window, I obtain a filtered matrix showing only the maximum value and 0 in the other slots. At this point I check the central element (the one in position $(2,2)$) of this last matrix: if it is different from zero, it means that it actually is the maximum value of the 3x3 window, and in this case I should consider the pixel to be a corner.

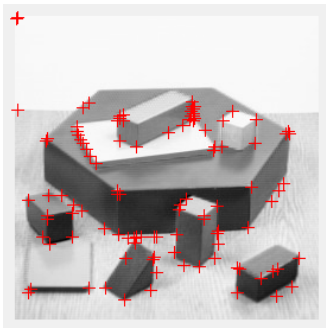
By performing different simulations with varying values of σ, k and T, I found the best detection performances by setting $\sigma = 3$ (even though the suggest values were 0.5, 1 and 2, by setting $\sigma = 3$ I observed that the Harris corner detector was performing relatively less misclassifications than in the previous cases), $k = 0.05$ and $T = 10^{-6}$.

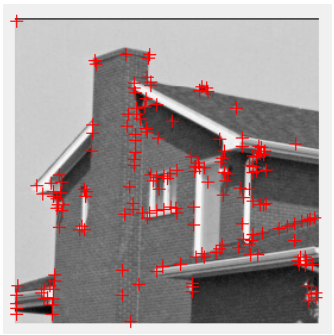




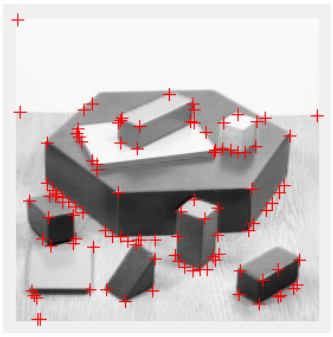
I will now show the performance of the detector under different values of the hyperparameters, which in some cases led to an underestimation of the corners, while in most cases to an overestimation.

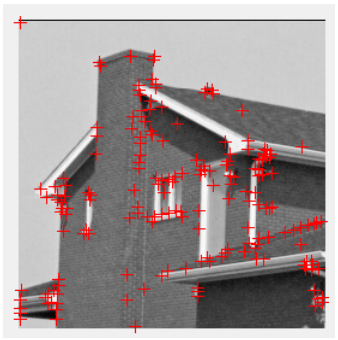
- $\sigma = 0.5 \quad k = 0.05 \quad T = 10^{-6}$



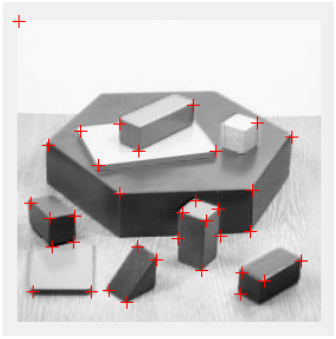


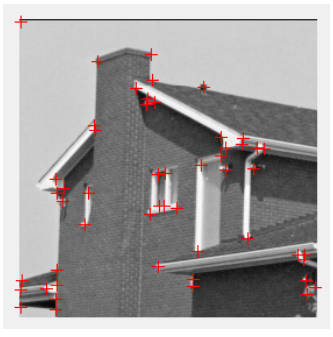
- $\sigma = 1 \quad k = 0.05 \quad T = 10^{-6}$



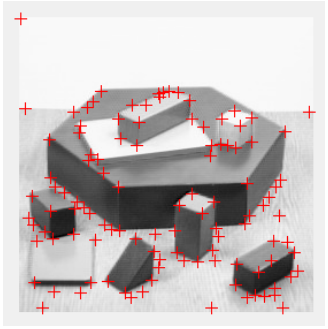


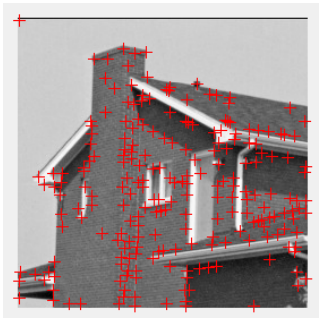
- $\sigma = 2 \quad k = 0.05 \quad T = 10^{-5}$





- $\sigma = 2 \quad k = 0.05 \quad T = 10^{-7}$





Overall, the implemented Harris corner detector seems to correctly detect the most relevant points in the image. However, the main issue is that there are still cases in which the detector inevitably classifies points as corners, even though, for example, they are part of a flat region. This is reasonable since thresholding between the two different cases is not deterministically done in a closed-loop way, but rather it is the result of tuning a hyperparameter.

2. Description & Matching

The aim of the second part of the assignment is to match features among different images. To achieve such a result, the first step is to extract local descriptors out of each previously detected keypoint. This is done by means of the *extractPatches* function, which extracts 9x9 patches around each detected keypoint. However, the keypoints must be first filtered so that we do not consider those too close to the edges. Hence, in order to be able to extract 9x9 patches out of each point, I selected only those corners that had a margin of at least 4 pixels from the borders.

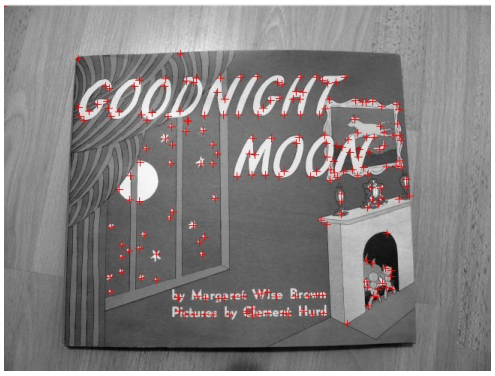


Figure 1 — Harris corner detector applied to the first image

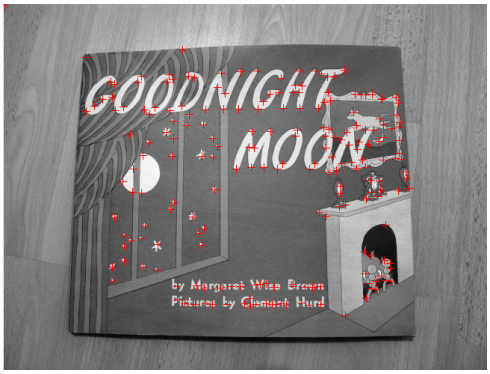


Figure 2 — Harris corner detector applied to the second image

2.1. SSD one-way nearest neighbours matching

To compute the SSD one-way nearest neighbours matching the first thing to do is to compute the SSD distance between the descriptors of the features of the first and the second image. This is done by means of the *pdist2* function, which then needs to be squared since the SSD has the following form:

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

Once the distance between the descriptors is obtained, in order to perform the one-way nearest neighbours matching, for each feature descriptor of the first image we have to select the feature descriptor of the second image which has the smallest distance. This is done by means of the min function in Matlab applied to each vector $distances(i,:)$ (which, for each feature descriptor of the first image, contains the distances from all the feature descriptors of the second image).

Once the minimum distance is found, I extract the coordinates of that specific element, which basically maps the corresponding feature descriptors between the two images, and then I store them in a matrix.

The following code was used to implement this type of matching:

```
for i = 1 : num_descr1
      min_val = min(distances(i,:));
      [x_min, y_min] = find(distances == min_val);
      matches = [matches; x_min y_min];
end
```

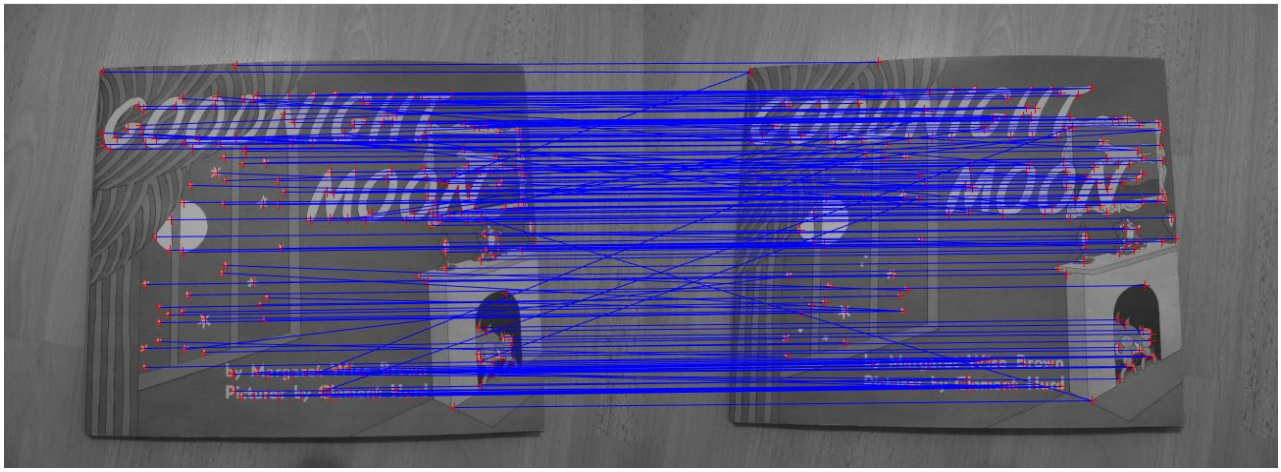


Figure 3 — SSD one-way nearest neighbours matching between two similar images.

2.2. Mutual nearest neighbours matching

For the implementation of the mutual nearest neighbours matching, instead, I should check that for each feature matching obtained from one-way NN, the same matching is obtained also when swapping the images. Hence, I performed two *for* loops, one scrolling over the feature descriptors of the first image, and one over the ones of the second image. In both cases, I stored the matchings in a vector, and in the end I considered only those matchings which were the same in both cases.

The following code was used to implement this type of matching:

```
for i = 1 : num_descr1

   min_val = min(distances(i,:));
   [x_min, y_min] = find(distances == min_val);

    for j = 1 : num_descr2
         min_val_mut = min(distances(:,j));
         [x_min_m, y_min_m] = find(distances == min_val_mut);

         if [x_min, y_min] == [x_min_m, y_min_m]
               matches = [matches; x_min y_min];
         end
    end
end
```

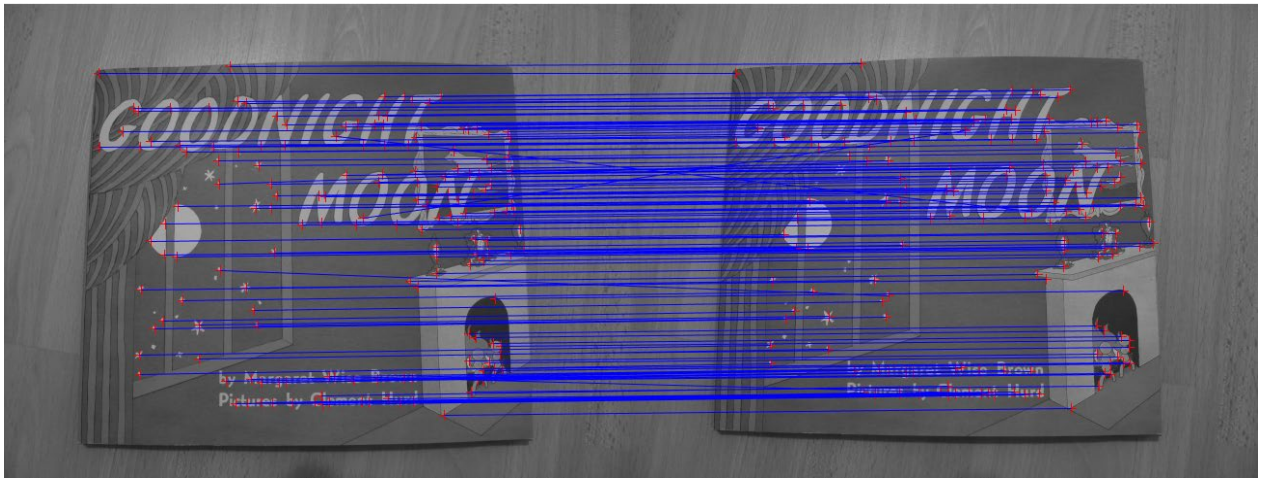


Figure 4 — Mutual nearest neighbours matching between two similar images.

2.3. Ratio test matching

To implement Ratio test matching, instead, for each feature descriptor of the first image, I have to select the first and the second nearest neighbours (such as the two descriptors that have shortest distance from the specific feature) and check whether their ratio is below a certain chosen threshold. The two nearest neighbours' distances are obtained by means of the mink function, applied to the each row vector $distances(i,:)$, which returns the two lowest values of the vector.

The threshold was set to 0.5. The following code was used to implement this type of matching:

```
for i = 1 : num_descr1
      min_two = mink(distances(i,:),2);
      min_val = min_two(1,1);
      [x_min, y_min] = find(distances == min_val);

      if (min_two(1,1)/min_two(1,2)) < 0.5
            matches = [matches; x_min y_min];
      end
end
```

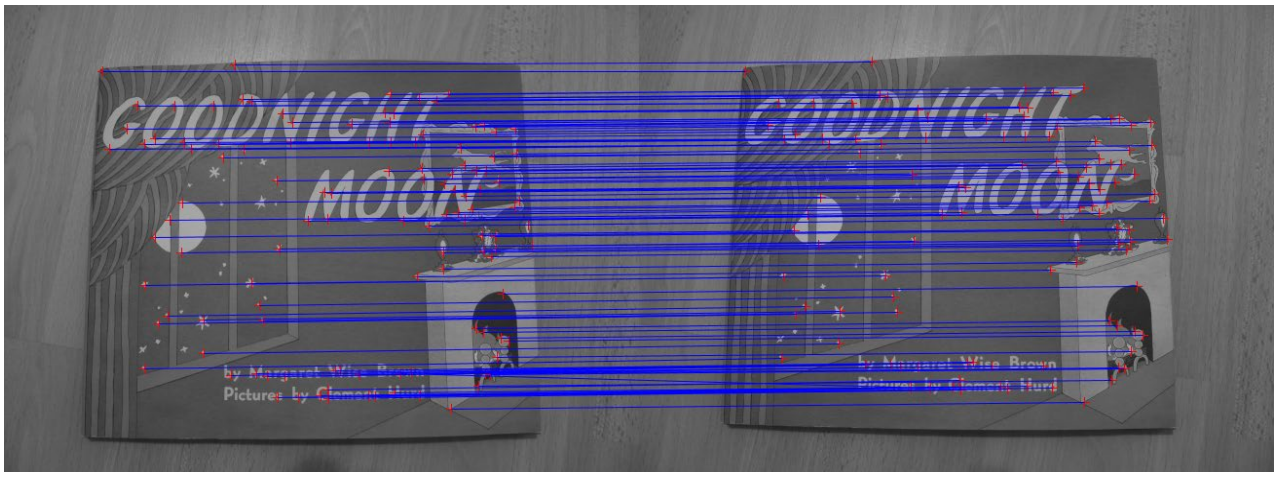


Figure 5 — Ratio test matching between two similar images.

Overall, a better performance was observed with the last two matching methods. In fact, in the one-way NN matching image, it is possible to see that there are several crossings between the segments connecting the corners of the two images, meaning that some features are mismatched. As far as the ratio test and the mutual NN matchings are concerned, it is possible to see that both methods are performing pretty well, however it seems that there is no major difference between them and both performances are comparable, as the respective images show similar results.