Local Features
Computer Vision Lab 2

Álvaro Belmonte Baeza
Student ID 19-940-386

October 9, 2020

Abstract

In this second assignment of the Computer Vision course, an implementation of the Harris corner detector and a simple feature matching protocol are presented. First, an step by step description on the Harris corner detection algorithm and its implementation is discussed, and the obtained results analysed, as well as the effects on changing the value of the several hyperparameters that appear in the algorithm. Then, the implementation of a basic description and matching protocol for finding correspondences between keypoints is shown. Here, three different approaches to find the correspondences will be presented, and then their results are compared. Finally, some conclusion after the completion of the whole assignment are presented.

1 Detection

In this first section, it is requested to perform the detection of interest points in an image. To do so, there exist several approaches in the Computer Vision world, and one of the most well-known is the Harris corner detection algorithm, which has been implemented in this assignment.

In the following subsections, the different steps that form the algorithm are presented, as well as the proposed implementation for each of them, coded in MATLAB as requested. In addition, the image shown in Figure 1 is the one used in this first section of the assignment.

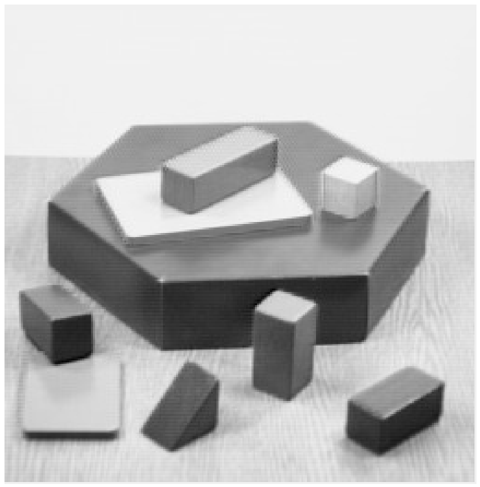


Figure 1: Image used for corner detection

1.1 Image Gradients

The first step in the algorithm is to compute the image gradients in both x and y directions. Talking about images, the gradient of an image is understood as the difference of intensity values of consecutive pixels. There exist several ways of computing this as a convolution operation through the image, being two of the most used the Sobel and Prewitt kernels. However, in this assignment it is asked to use the expressions shown in equation (1):

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

with $I(i,j)$ being the intensity value of a grayscale image at pixel (i,j), and I_x, I_y being the gradients of the image in each direction.

To apply these operations as a convolutional operation, it is needed to use the adequate convolutional filters. To replicate the expressions shown in equation (1), the following 3x3 kernels are used:

$$I_x filter = \frac{1}{2}\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, I_y filter = \frac{1}{2}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

As it can be seen, applying the convolutional operation to every pixel of the image using each of these kernels will result in obtaining the necessary image gradients.

The implementation of this step is pretty straightforward, and can be checked in the following code block:

```
% Define the convolution filters for X and Y directions to replicate
% the formulas specified in the assignment
Ix_filter = [0 -0.5 0;0 0 0;0 0.5 0];
Iy_filter = [0 0 0;-0.5 0 0.5;0 0 0];

% Compute the Image gradients using the 2d convolution function. the
% 'same' parameter enforces the gradient image to have the same size as
% the original image
Ix = conv2(img, Ix_filter, 'same');
Iy = conv2(img, Iy_filter, 'same');
```

As it is shown, first the convolutional kernels are created, and then they are applied to obtain the image gradients in each direction.

After executing this code, the obtained image gradients can be checked in Figure 2.

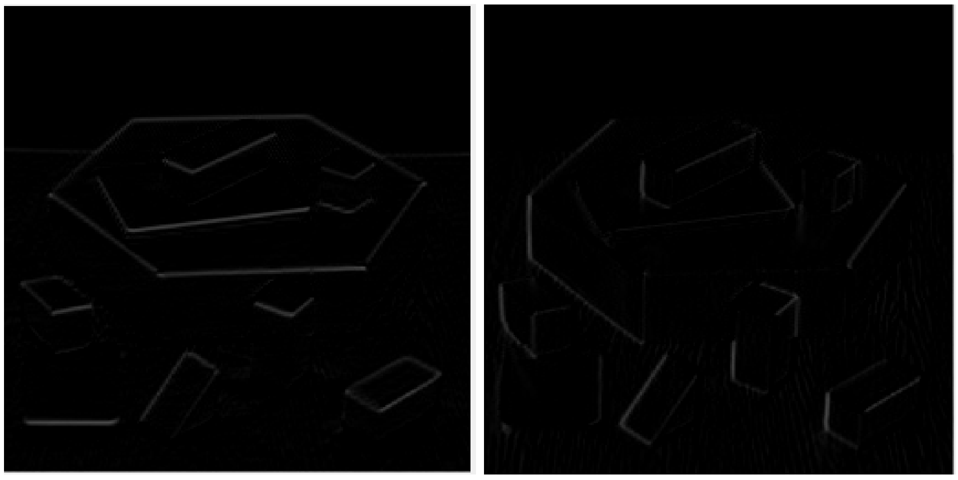

(a) I_x (b) I_y

Figure 2: Image gradients obtained with equation (1) kernels

1.2 Local auto-correlation matrix

Once the image gradients are obtained, the second step is to compute the elements of the local auto-correlation matrix, which is defined for each pixel position p in equation (2).

$$M_p = \sum_{p'} w_{p'-p} \begin{bmatrix} I_x(p')^2 & I_x(p')I_y(p') \\ I_y(p')I_x(p') & I_y(p')^2 \end{bmatrix}. \tag{2}$$

In equation (2), the weight w is generally Gaussian, and it will be as well in this assignment. The use of a Gaussian function is a nice way to blur the image gradients and therefore retrieve a more homogeneous image, where some of the noise that it could have is erased by the Gaussian kernel.

To implement the above mentioned operation, several steps are needed. First of all, the elements of the M matrix are computed (i.e. I_x^2, I_y^2, I_xI_y). Then the Gaussian filtering is applied to each of these components, obtaining the elements of the auto-correlation matrix. The implementation of this part of the algorithm can be checked here:

```
% Compute auto-correlation
% First compute square of derivatives
Ixx = Ix .* Ix;
Iyy = Iy .* Iy;
Ixy = Ix .* Iy;

% Secondly Apply Gausian Filtering
GIxx = imgaussfilt(Ixx,sigma);
GIyy = imgaussfilt(Iyy,sigma);
GIxy = imgaussfilt(Ixy,sigma);
```

Note from the code above that the multiplication between image gradients is a element-wise multiplication. Consider as well the parameter *sigma*, which represents the standard deviation of the applied Gaussian kernel. In this case, a value $\sigma = 1$ has been chosen after testing with all recommended values.

A visualization of the Gaussian filtered square derivatives won't be shown here, as it is a rather black image which in this case doesn't add much to the report.

1.3 Harris response function

In this third step, once the elements of the M matrix have been obtained, the Harris response function can be computed. This function has the expression shown in equation (3).

$$C(i,j) = |M_{i,j}| - kTr^2(M_{i,j} \tag{3}$$

with $k \in [0.04, 0.06]$ being an empirically determined constant. In this case, a value $k = 0.05$ has been chosen after different tests.

It is important to notice what the elements in equation (3) mean. The determinant of M corresponds to an approximation of the product of the eigenvalues, and the trace is the sum of these values, and it has been studied that when both eigenvalues are large, it means that a corner exists in that region of the image.

The implementation of this part of the algorithm is shown here:

```
    % Compute Harris response
    C = GIxx.*GIyy - GIxy.*GIxy - k*(GIxx + GIyy).^2;
```

As it can be seen, this is an equivalent computation to equation (3), which results in the so called *cornerness* of each pixel in the image.

1.4 Detection criteria

Last but not least, once the Harris response has been computed, the final step is to apply a detection criteria, and then apply a local maximum filtering as well.

First of all, a *cornerness* threshold is applied to each pixel, taking into account the value that each pixel obtained after computing the Harris response function. To do so, a simple iteration through the image is implemented, and if the currently evaluated pixel has less *cornerness* value that the selected threshold, its value is considered to be zero.

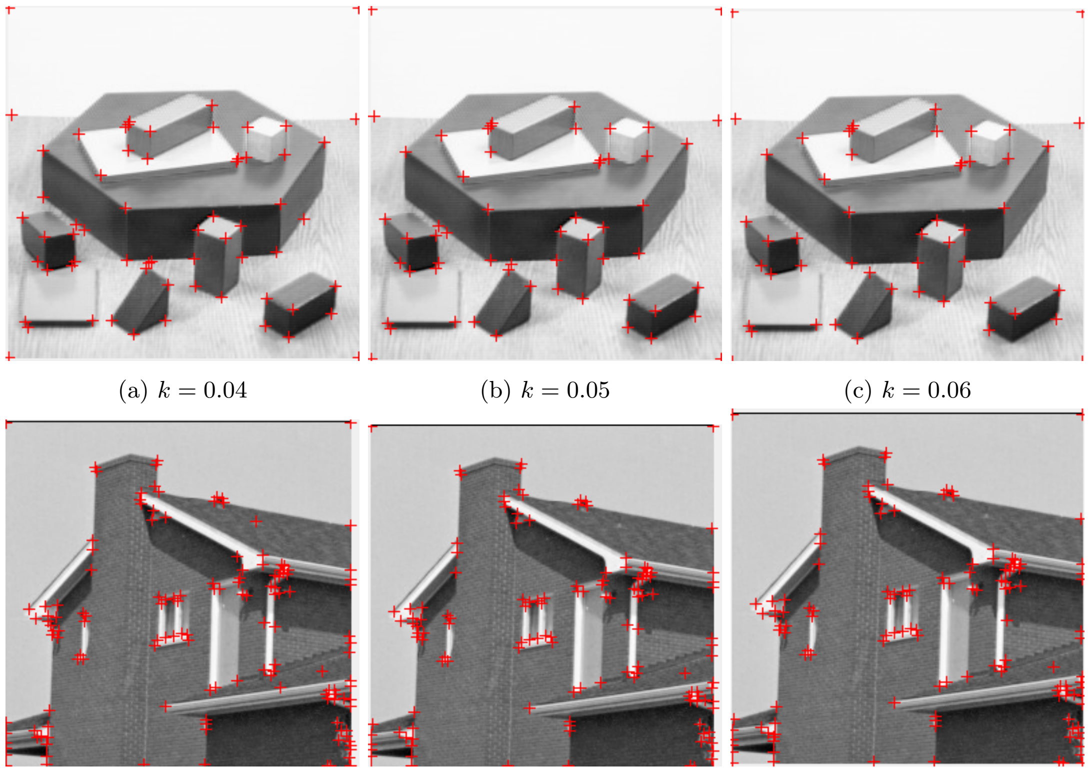
After computing this first filtering based on the threshold, a Non-Maximal Supression is applied in order to consider only the highest values in each local region of the image.

The implementation of both detection criteria above mentioned is as follows:

```
    % Apply detection threshold
    thresholdedC = C;
    for i = 1:size(C,1)
        for j = 1:size(C,2)
            if(C(i,j) < thresh)
                thresholdedC(i,j) = 0;
            end
        end
    end

    % Perform Non Maximal Supression
    NMSCorners = imregionalmax(thresholdedC);
```

At this point, the whole Harris corner detection algorithm has been implemented. To check the results and tune the hyperparameters, two test images were provided. In Figures 3, 4 and 5, the differences between several values of the constant k of the Harris response function, the standard deviation σ of the gaussian filters, and the final threshold respectively, in order to analyse the actual effect of these hyperparameters in the final output.

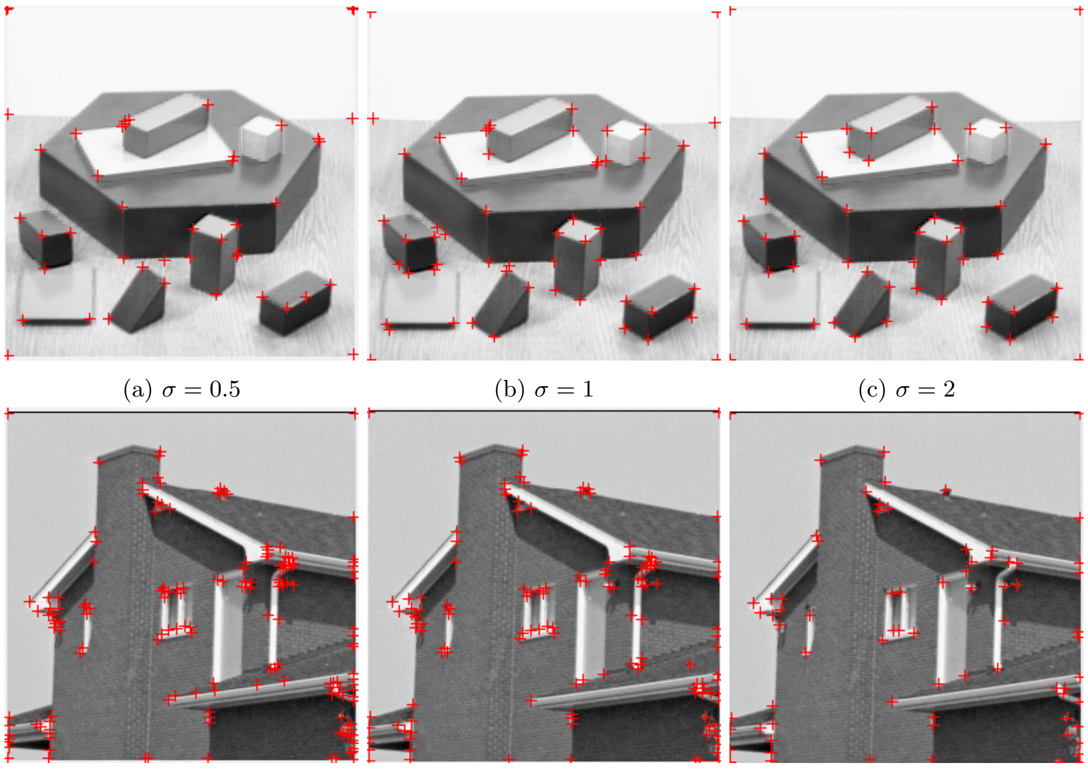


(a) $k = 0.04$ (b) $k = 0.05$ (c) $k = 0.06$

(d) $k = 0.04$ (e) $k = 0.05$ (f) $k = 0.06$

Figure 3: Output for different values of k

From Figure 3, it can be seen how increasing the value of k slightly less corners are detected, as it was expected by just looking into the Harris response function equation. However, at least for the tested values, there is not a huge difference.

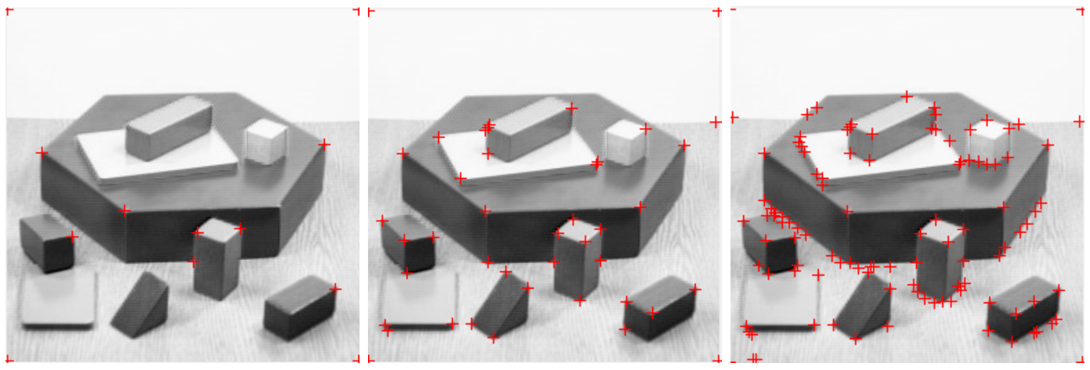


(a) $\sigma = 0.5$ (b) $\sigma = 1$ (c) $\sigma = 2$

(d) $\sigma = 0.5$ (e) $\sigma = 1$ (f) $\sigma = 2$

Figure 4: Output for different values of σ

This time, a more clear difference between the detected keypoints can be seen when varying the gaussian filter's standard deviation, specially when changing from $\sigma = 1$ to $\sigma = 2$. This is also expected, because when σ increases, the image becomes more homogeneous, and thus the non clear huge gradients become smoother and may not pass the detection criteria anymore.


(a) $T = 1e^{-4}$ (b) $T = 1e^{-5}$ (c) $T = 1e^{-6}$

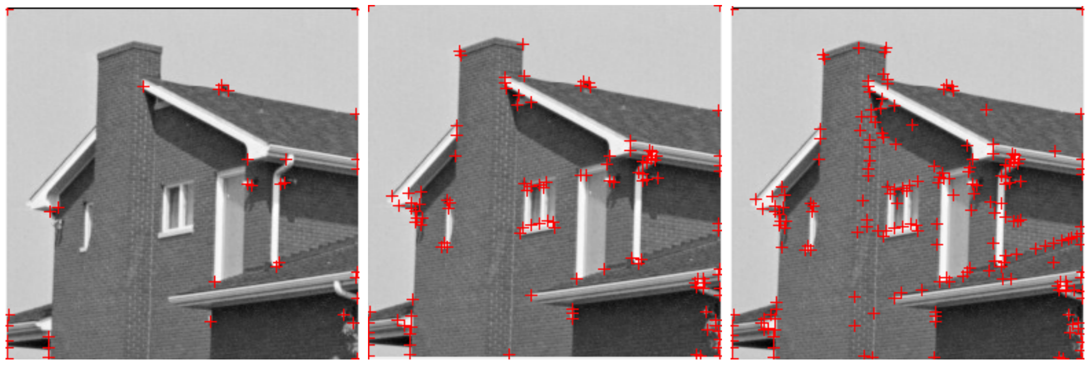
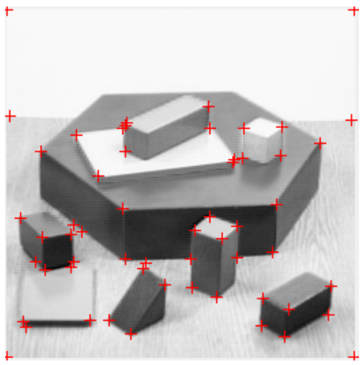

(d) $T = 1e^{-4}$ (e) $T = 1e^{-5}$ (f) $T = 1e^{-6}$

Figure 5: Output for different values of detection threshold

In this last case, the increasing amount of corners detected when decreasing the threshold value is evident.

Thus, when some hyperparameters have to be tuned, in most cases it is necessary to find a compromise between number of detections and *certainty* that the detection is correct. With these in mind, after several different configurations, the chosen values for the hyperparameters are: $k = 0.05$, $\sigma = 1$ and $T = 5e - 6$. The resulting detection using these value can be seen in Figure 6.


(a) Results for image *blocks.jpg*

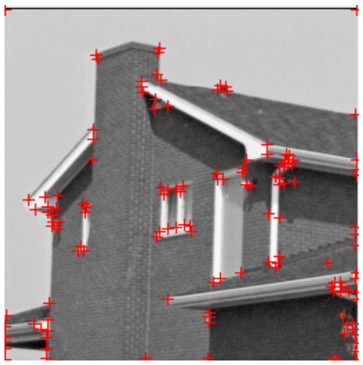

(b) Results for image *house.jpg*

Figure 6: Results with final selection of parameters

In this case, it has been preferred to have a bigger amount of keypoints, in order to test how the description and matching of these features works when having a more dense set of interest points.However, some issues may arise with the detected keypoints. For example, the pixels at the image corners are detected, as the convolution operator does not take the image borders into account (this problem is addressed in the next section). Another possible issue is that some corners that are obvious to the human eye are not captured by the algorithm, for example the two back corners of the big hexagon in figure 6a. This could be due to a very smooth transition in the intensity of the image in those points, due to low resolution or simply the actual light conditions when the picture was taken.

This way, the section corresponding to the detection procedure comes to an end, having implemented succesfully all the requested parts, and obtaining reasonably good results.

2 Description and Matching

In this second section, a simple implementation of a description and matching protocol for image features is presented. To look for these correspondences between image interest points, the image pair shown in Figure 7 is provided.

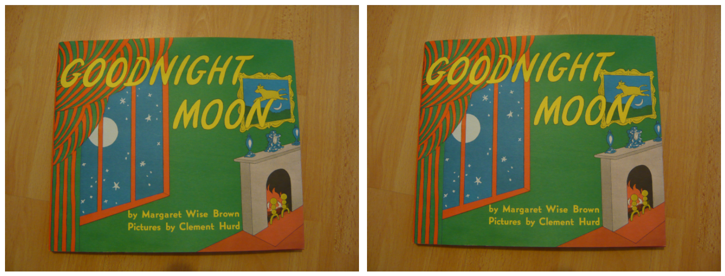


Figure 7: Provided image pair

Next, the different steps necessary to perform this feature description and matching are presented. The descriptors will be obtained only as the pixel values at a defined region of an image. On the other hand, several matching techniques will be presented. Concretely, the one-way nearest neighbors matching, Mutual nearest neighbors, and Ratio test protocols are discussed.

2.1 Local Descriptors

The first step is to compute a feature descriptor in order to extract some relevant characteristic of the surroundings of a keypoint. There exist many types of descriptors, such as SIFT, SURF or Texture descriptors. However, in this assignment it is asked to compute a very simple descriptor, consisting of the pixel values that surround each keypoint in a 9x9 window.

However, before computing the descriptor, it is necessary to get rid of the *false* keypoints detected at the corners of the image. To do so, the image coordinates of each keypoints are checked, and if they are less than 5 pixels (as a 9x9 window is being used) away from a border, then the keypoint is discarded. This consideration can be implemented as follows:

```
    % Filter keypoints that are too close to the image edges (<5 pixels as
    % we are using 9x9 patches)
    invalid_keypoints = [];
    for j = 1:size(keypoints,2)
        if(keypoints(1,j)≤ 5 || keypoints(1,j) ≥ (size(img,1)-5) || ...
            keypoints(2,j)≤10 || keypoints(2,j) ≥ (size(img,2)-5))
            invalid_keypoints = [invalid_keypoints,j];
        end
    end
    keypoints(:,invalid_keypoints) = []; %Remove border keypoints from set
```

This way, it is avoided to include non-real pixel values obtained trespassing the image borders.

Once the invalid keypoints have been deleted, then the 9x9 feature descriptors can be obtained. Note that the pixel values inside the 9x9 window will be stored as a column vector of dimension $81x1$. To do so, the provided function *extractPatches* is used, resulting in the following straightforward implementation.

```
    % Obtain keypoint descriptors as flattened vectors containing the pixel values of
    % the specified patch size
    descriptors = extractPatches(img,keypoints,9);
```

At this point, the feature descriptor for each feature has been computed, being the matching step the next one.

2.2 SSD one-way nearest neighbors matching

In this second step, once the descriptors are already computed, a measure of the similarity between the descriptors in each image is needed. In this case, the sum-of-squared-differences (SSD) is used. The SSD distance is defined in equation (4).

$$SSD(p,q) = \sum_i (p_i - q_i)^2. \tag{4}$$

where p, q are the descriptors of a keypoint in each image.

To compute these distances from each descriptor in image 1 to each descriptor in image 2, the *pdist2* function is used. However, it is necessary to choose between the several types of distances that this function allows. In this case, the parameter *'squaredeuclidean'* is used, as the Squared Euclidean distance has the expression:

$$d^2(p,r) = (p_1 - r_1)^2 + (p_2 - r_2)^2 + \ldots + (p_n - r_n)^2$$

which is equivalent to the SSD distance shown in equation (4).

In the end, the implementation of the SSD distance is rather simple, and it is as follows:

```
function distances = ssd(descr1, descr2)
    % We use the squared euclidean distance, which is equivalent to SSD
    distances = pdist2(descr1',descr2','squaredeuclidean');
end
```

Then, the one-way nearest neighbors matching protocol is implemented. In this protocol, for each feature in the first image, it is checked which is the feature in the second image with the less SSD distance.

The proposed implementation for this method is as follows:

```
    if strcmp(matching, 'one-way')
        % Obtain closest feature in image 2 for each feature in the first image
        matches = zeros(2,size(distances,1));
        for i=1:size(distances,1)
            % For each keypoint in image 1, retrieve the keypoint of image
            % 2 with minimal distance and store its correspondence
            [minVal, minIndex] = min(distances(i,:));
            matches(1,i) = i;
            matches(2,i) = minIndex;
        end
```

It can be seen that a *for* loop through the first image's keypoints is implemented, and then for each keypoint the closest one in image 2 is associated and stored in a 2-row vector.

This way, the correspondences between image 1 keypoints and their closest ones in image 2 is already computed. The resulting correspondences can be checked in Figure 8.

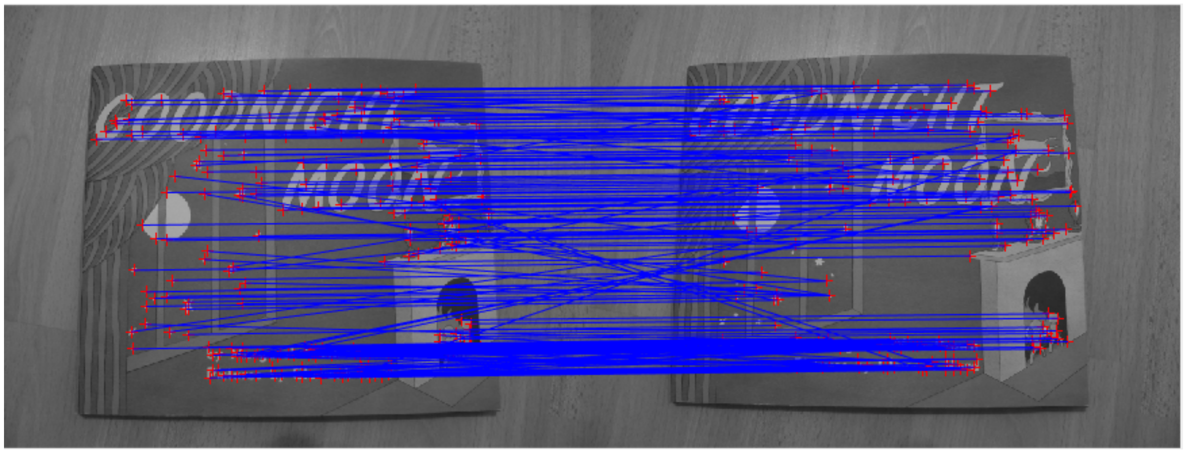


Figure 8: One-way matching correspondences

As it is shown in Figure 8, in this case every keypoint in image 1 finds a correspondence in image 2, as the protocol only looks for the closest distance for each keypoint separately.

However, this simple approach has its drawbacks. One of the most importants is that some keypoints in image 1 may end with more than one correspondence in image 2, as the protocol doesn't have any control on duplicates.

There are several alternative techniques that are more restrictive, providing a more reliable matching between keypoints. Two of them are going to be discussed in the following paragraphs.

2.3 Mutual Nearest Neighbors

By using the Mutual Nearest Neighbors protocol, the correspondences for the keypoints in image 2 looking at image 1 are also considered, in addition to the ones for image 1 in image 2. This way, it is checked that for the one-way correspondences obtained by looking at the closest features for each keypoint in image 1, it outputs the same pair of correspondences when looping through every keypoint in image 2. Then, the correspondences which are consistent in both directions are considered, and the other ones are deleted.

There are several ways to compute this mutual correspondences checking. In this case, a explicit, step-by-step implementation has been selected for the sake of clearness, although it might not be the most efficient. The proposed implementation is as follows:

```
    elseif strcmp(matching, 'mutual')
        % Obtain closest feature from I1 in I2
        matches1 = zeros(2,size(distances,1));
        for i=1:size(distances,1)
            % For each keypoint in image 1, retrieve the keypoint of image
            % 2 with minimal distance and store its correspondence
            [minVal, minIndex] = min(distances(i,:));
            matches1(1,i) = i;
            matches1(2,i) = minIndex;
        end
        %Obtain closest feature from I2 in I1
        matches2 = zeros(2,size(distances,2));
        for j=1:size(distances,2)
            % For each keypoint in image 2, retrieve the keypoint of image
            % 1 with minimal distance and store its correspondence
            [minVal, minIndex] = min(distances(:,j));
            matches2(1,j) = minIndex;
            matches2(2,j) = j;
        end

        %Check the mutual correspondences
        matches=[];
        if size(matches1,2) < size(matches2,2) % If there are more keypoints in ...
            Image 2
            for i = 1:size(matches1,2)
                if matches1(2,i) == matches2(2,find(matches2(1,:)==i))
                    matches = [matches, matches1(:,i)];
                end
            end
        else % If there are more keypoints in Image 1
            for j = 1:size(matches2,2)
                if matches2(1,j) == matches1(1,find(matches1(2,:)==j))
                    matches = [matches, matches2(:,j)];
                end
            end
        end
```

From the code above, it can be seen that first both one-way correspondences are computed, as they were in the one-way protocol. Next, it is checked if the correspondences found in one way have been also found on the other way. Note that, as it is unknown which image will have more keypoints, an *if-else* statement is included in order to do the *for* loop through the smaller set of keypoints.

After this filtering, the obtained correspondences are reduced, as it can be appreciated in Figure 9

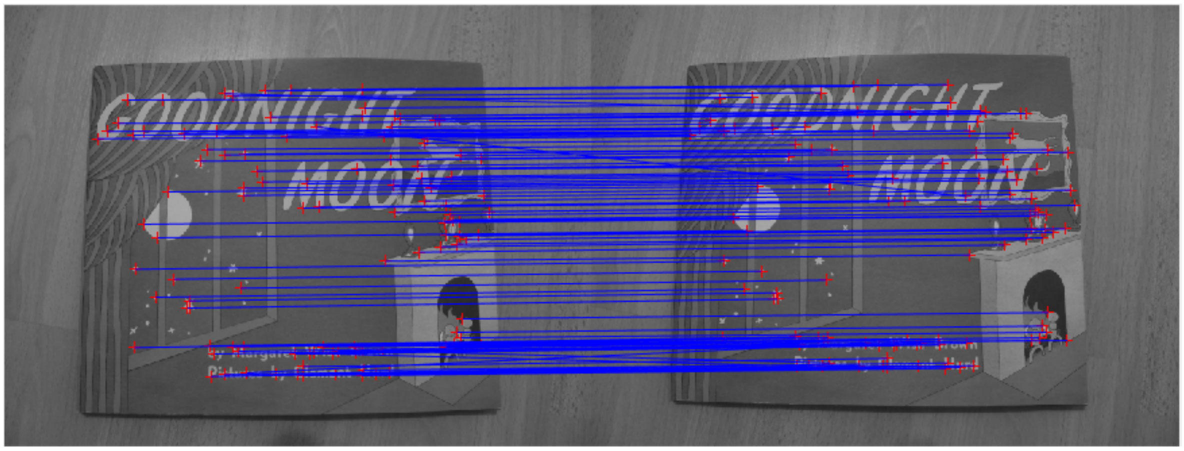


Figure 9: Mutual matching correspondences

2.4 Ratio test

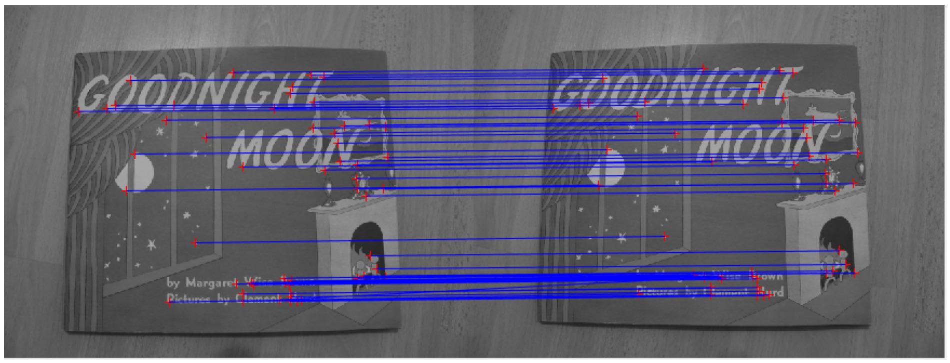
Last but not least, another approach to improve the one-way matching results by considering a correspondence as valid if the ratio between the closest feature in image 2 and the second closest feature is lower than a specified threshold.

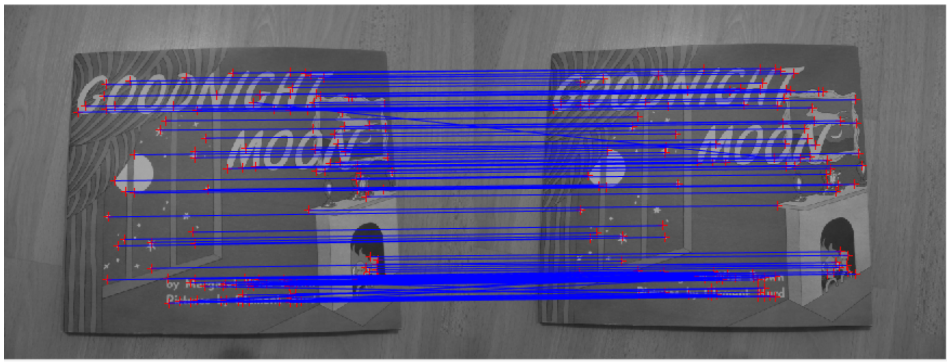
The implementation of this method is pretty straigthforward, as it can be seen in the following code block:

```
    elseif strcmp(matching, 'ratio')
        % Obtain closest 2 features in image 2 for each feature in the first
        % image and then compute the ratio between them. If it's lower
        % than the specified threshold, consider the match valid.
        matches = [];
        ratio_thresh = 0.5;
        for i=1:size(distances,1)
            % Retrieve 2 minimal distances and corresponding indexs
            [minVals, minIndexs] = mink(distances(i,:),2);
            if ((minVals(1)/minVals(2)) < ratio_thresh) %Check if ratio is satisfied
                matches = [matches, [i;minIndexs(1)]];
            end
        end
```

As it is shown, the correspondence is only included in the *matches* set if it has a ratio bigger than the threshold. Obviously, the value of the ratio is an hyperparameter that should be tuned depending on our preference, whether it is having only the strongest correspondences or there is more flexibility in that regard.

In Figure 10, the results for several values of the ratio threshold are presented.


(a) Threshold = 0.2


(b) Threshold = 0.5

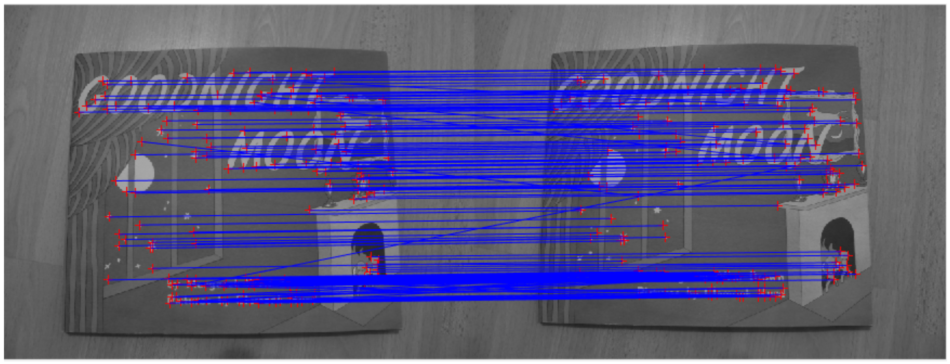

(c) Threshold = 0.8

Figure 10: Ratio test correspondences

From the pictures in Figure 10, it becomes clear how a different threshold affects the resulting correspondences, and as it was said previously, the selection of this value is arbitrary based on how strong the correspondences are meant to be.

With this, every part of the assignment has already been discussed, and finally the conclusions will be presented.

3 Conclusion

In this second assignment of the Computer Vision course, first the Harris corner detection algorithm has been implemented. This has been done step by step, explaining the meaning behind each operation and commenting the proposed implementation. In addition, a discussion about the effect of each hyperparameter in the algorithm has been done, with several figures that illustrate the mentioned effects.

Next, once the keypoints have been already computed, a simple descriptor based on the intensity of the surrounding pixels of each keypoint has been obtained. Then, different matching protocols have been implemented and compared, taking into account the differences between them and how each solution improves the standard one-way matching protocol.

All in all, it has been a complete assignment where, even though the techniques used are rather classical, and more complex descriptors haven't been used, it allows the students to become familiar with the general feature detection and matching pipeline, which is a basic competence for anyone working in the Computer Vision world.