Computer Vision Assignment 2: Feature Extraction

Charlotte Moraldo

1. Feature Extraction

1.1 Harris Response

The challenge of this part is writing down an algorithm efficient enough to calculate quickly the Harris response for every pixel. The "traditionnal" way would be to go through every pixel in a double for-loop. In this assignment, the images have a size of 2448x3264, which requires approximately 5min of computation with a double for-loop. Even though it is still bearable, it isn't an efficient implementation and the computation time would get longer and longer as we use larger images.
Therefore, I decided to use the matlab function `movsum`, which returns an array of local k-point sums, where each sum is calculated over a sliding window of length k across the neighboring elements. After calculating the gradient matrices `Ix` and `Iy`, I computed the matrices `Ix_sq` (elementwise multiplication of `Ix` with itself), `Iy_sq` (elementwise multiplication of `Iy` with itself), and `Ixy` (elementwise multiplication of `Ix` with `Iy`). After defining the window size `W_size` to 3, I computed the Harris response as following:

```
H11 = movsum(movsum(Ix_sq,W_size,1), W_size,2)
H22 = movsum(movsum(Iy_sq,W_size,1), W_size,2)
 H12 = movsum(movsum(Ixy,W_size,1), W_size,2)
```

Where `H11`, `H22`, and `H12` are matrices such that the Harris matrix of a pixel (i, j) is given by:

$$H(i,j) = \begin{bmatrix} H_{11}(i,j) & H_{12}(i,j) \\ H_{12}(i,j) & H_{22}(i,j) \end{bmatrix}$$

Therefore, to compute the Harris response K for each pixel, I just needed to perform the elementwise determinant and trace of H:

```
K = (H11.*H22-H12.*H12)./(H11+H22)
```

Where the Harris response of a pixel (i, j) would be given by:

$$K(i,j) = \frac{H_{11}(i,j) \cdot H_{22}(i,j) - H_{12}(i,j) \cdot H_{12}(i,j)}{H_{11}(i,j) + H_{22}(i,j)} = \frac{det(H(i,j))}{trace(H(i,j))}$$

Having tried both this method and the double for-loop, I can confirm that this method is much more computationally efficient. Indeed, I went from approximately 5min to a few seconds of computation!

1.2 Non-Maximum-Suppression

In this part, I it would have also been possible to find a computationally efficient method. However, the efficiency of part 1.1 was the most important, and using a classic double for-loop here doesn't increase too much the computation time, as it remains equal to a few seconds.
In this part, I therefore run through each element of the Harris response matrix K and compute the 3x3 window matrix W if the element $K(i, j)$ is superior to a decided threshold. This threshold is a parameter that will need to be tuned later on, when matching the descriptors. I search for the index x of the maximum element in the window with `max(W(:))`, and then recover the corresponding index y using the matlab function `ind2sub`. If the index (x, y) of the maximum is equal to the Harris response of the pixel (i, j) we are currently looking at in our loops, then we add the coordinates of this pixel in the vector `corners`.

1.3 Illustration

The two images below illustrate the extracted Harris corner before and after non-maximum suppression (NMS).

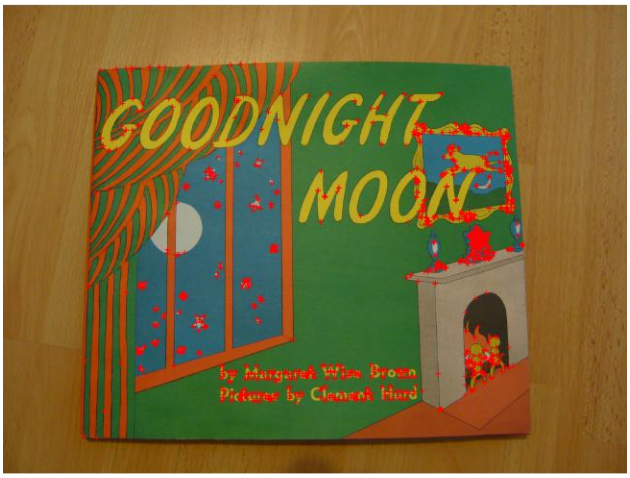

Figure 1: Before non-maximum suppression

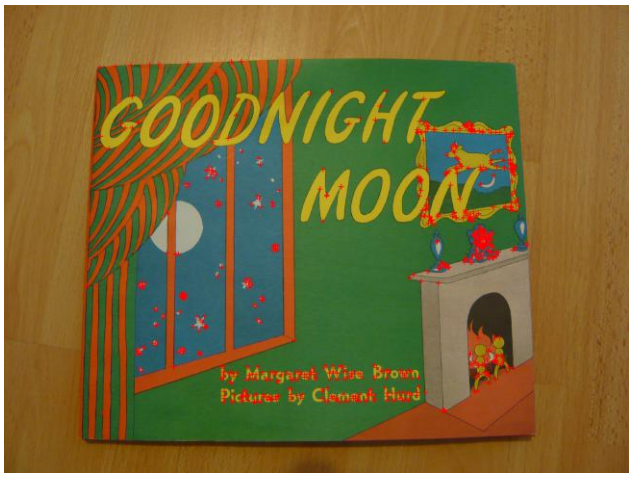

Figure 2: After non-maximum suppression

The image before NMS contains 4650 corners, while the image after performing NMS with a threshold of 0.005 contains 749 corners only, showing that the NMS works well at selecting the "relevant" corners. Choosing the threshold can be done by looking at the histogram of the Harris response K (figure 3):

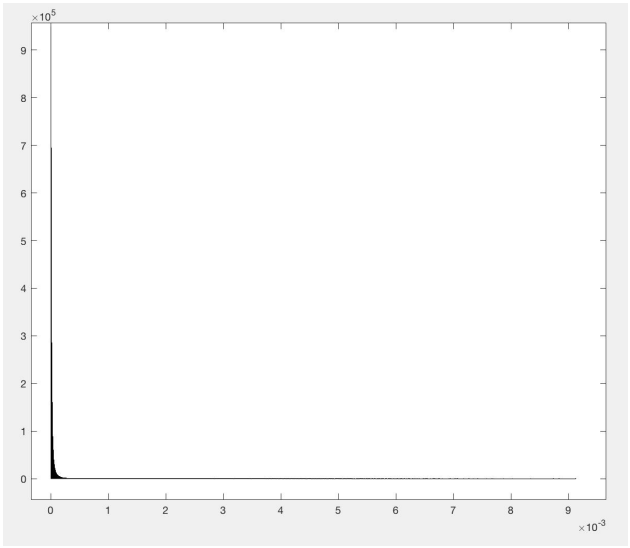

Figure 3: Harris Response Histogram

thresh = 0.001	thresh = 0.005	thresh = 0.008
1949 corners	749 corners	4 corners

Figure 4: Number of corners after different thresholds

In figure 4 above, we try different thresholds to show the impact of the non-maximum-suppression, showing that is necessary to select the threshold value well in order to optimize the descriptor-matching.

2. SSD Feature Matching

2.1 Simple Matching

After extracting a 9x9 descriptor matrix around each corner, we can perform the SSD feature matching. A small SSD between two descriptors denotes a high similarity between them. After some tuning, I decided to set a threshold at 0.08, meaning that only the points with a SSD under 0.08 are matched. I also decided to set the threshold for the non-maximum-suppression to 0.0035.
With these parameters, we obtain: 1301 corners for image 1, 2094 corners for image 2, and more importantly 711 matches, that can be seen in the following picture:

Figure 5: Matched descriptors

This result is already pretty good, as we can see that many features of the left image have been correctly matched to the right image.

2.2 Optimization

We can further optimize our results and reduce the number of bad matches by looking at the following idea:

- Let f_1 be a feature in image 1 we are trying to match
- Let f_2 be the best SSD match to f_1 in image 2
- Let f_2' be the second best SSD match to f_1 in image 2
- If the ratio $\frac{SSD(f_1,f_2)}{SSD(f_1,f_2')}$ is close to 1, then the match is ambiguous or bad

Therefore, in order to avoid any bad matches, in case the ratio of the two SSDs is close to 1, we can simply discard the matches of f_1 with both f_2 and f_2'. This is yet another parameter to tune, as we have to choose t such that we only keep the matches following the condition:

$$abs(\frac{SSD(f_1,f_2)}{SSD(f_1,f_2')} - 1) \geq t$$

The closer t is to 1, the more we filter the matches which have a close SSD. Below are a few examples of the results obtained for two different values of t:

t = 0	t = 0.5	t = 0.6
711 matches	171 matches	106 matches

Figure 6: Number of matches for different t

Figure 7: Matched descriptors with $t = 0.5$

Figure 8: Matched descriptors with $t = 0.6$

Unfortunately, this method reduces a lot the number of matched obtains, especially in the regions where the patterns are very similar (ex: letters of the text, stars in the sky, pattern of the curtain), which makes sense since the corners of image 1 can be easily matched to many different corners of image 2.
However, we also observe that the number of bad matches diminshes a lot as increase the threshold. Indeed, in Figure 7 ($t = 0.5$), by looking closely we observe around 20 bad matches for a total of 171 matches, corresponding to $\backsim$ 11.7%. However, for $t = 0.6$, we count approximately 10 bad matches for a total of 106, corresponding to $\backsim$ 9.4%. Even if some bad matches remain, there is still an amelioration, as now have +90% of good matches! Let's further note that the bad matches often occur in "confusing areas", where the patterns and the corners are very similar, for example in different areas of the curtain, between different stars, or between the text letters. Further optimization can always be performed to obtain better results.

3. SIFT Features

We must begin by putting the two color images in gray scale, then normalizing them in the [0,255] interval, and converting them in single precision images. Only then can we use the command `vl_sift` to compute the SIFT frames and descriptors. The SIFT detector is controlled by the peak threshold as well as the edge threshold. After different tries, the optimal matching performance I found is the one obtained with a peak threshold of 13. Below 13, wrong matches start to occur, and above, less good matches appear. Below is the illustration of this result: we can see that, the matches are absolutely perfect. In this case, the two images match at 100%:

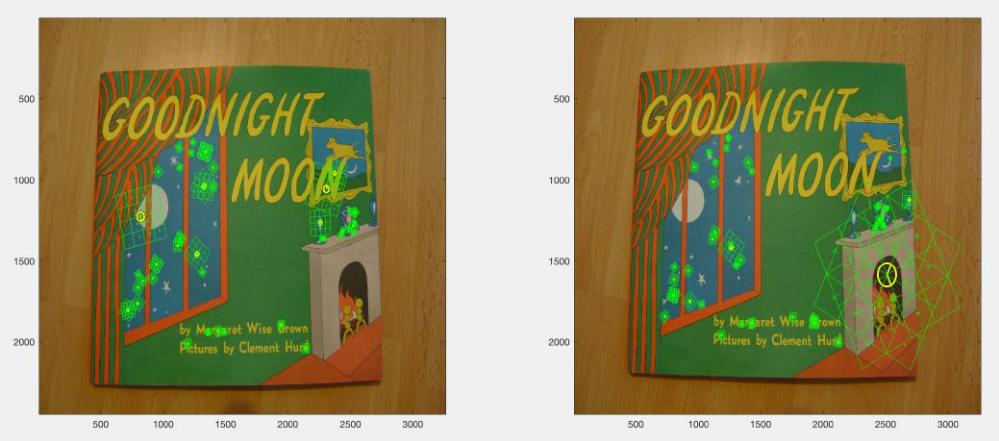

Figure 9: Features and descriptors with peak threshold 13

Figure 10: Matched descriptors with peak threshold 13

4. Conclusion

The results obtained with my implementation of the Harris Corner Detector are already very convincing. Indeed, after some simple optimization (explained in point 2.2), we obtain a good match percentage of +90%, which is already excellent. However, the results obtained with the SIFT detector are much better. Indeed, the algorithm used is much more complex and the tuning can be performed more finely, for example thanks to the peak and edge thresholds.