Homework 2

Thiago Henrique Pincinato
Introduction to Signal and Image Processing

April 7, 2019

1 Linear Filtering

Linear Filtering is an operation, in which a kernel is used to determine how the neighborhood pixel will influence the new value of the center pixel. Such a technique can be used to difference proposes as, for instance, removal of pepper and salts noise (e.g box filter) or enhance of edge (e.g Sobel)

1.1

The function boxfilter(n) can be summarized as a creation of a matrix nxn with all elements equal to 1 divide by nxn ($np.ones((n,n))/(nxn)$.)

In the development phase, the condition $boxfilter(n).sum() == 1$ was used to assure that the sum of the filter was equal to 1.

1.2

The function $myconv2$ was implement by acquiring the weight and height of filter and image ($np.shape$), applying padding in the image ($np.pad$), flipping the filter ($np.flit$), realizing a multiplication element by element and summing the result of the multiplications ($np.sum$ and $np.multiply$).

The result of such a function is a full convolution.

1.3

A boxfilter of size 11 was created and this filter was used in a given image ($cat.jpg$). The result of such a convolution can be seen in the figure 1.

As expected the filter blurred the image. This happens because the box filter is an average filter that, in our case, computes the average of the center pixel and its 120 neighboring pixels.

1.4

In order to write a function that returns a 1D Gaussian filter, an array x is created ($np.linspace$), then the Gaussian equation is applied ($np.exp$ and $np.power$) .

Figure 1: *Original and filtered image*

Furthermore, the argument filter_length is incremented of 1 if, and only if, is value is a even number.

As result, the function $gauss1d$ has no for loop, which improve efficiency and compactness.

1.5

The function $gauss2d$ was implemented based on the functions $gauss1d$ and $myconv2$. Firstly, the function $gauss1d$ is called, returning a 1D Gaussian array. After that, the 1D Gaussian array is copied and transposed. Finally, the function $myconv2$ is called with the 1D Gaussian array and 1D Gaussian array transposed, returning the final 2D Gaussian filter.

The resulting 2D Gaussian filter is illustrated in the figure 2.

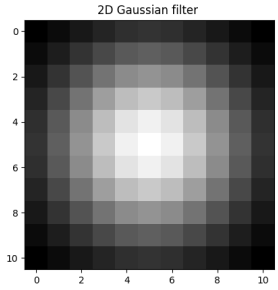

Figure 2: $2D$ *Gaussian filter*

1.6

The previous filter (exercise 1.5) was used in the image *cat.jpg* and the result is shown in the figure 3.

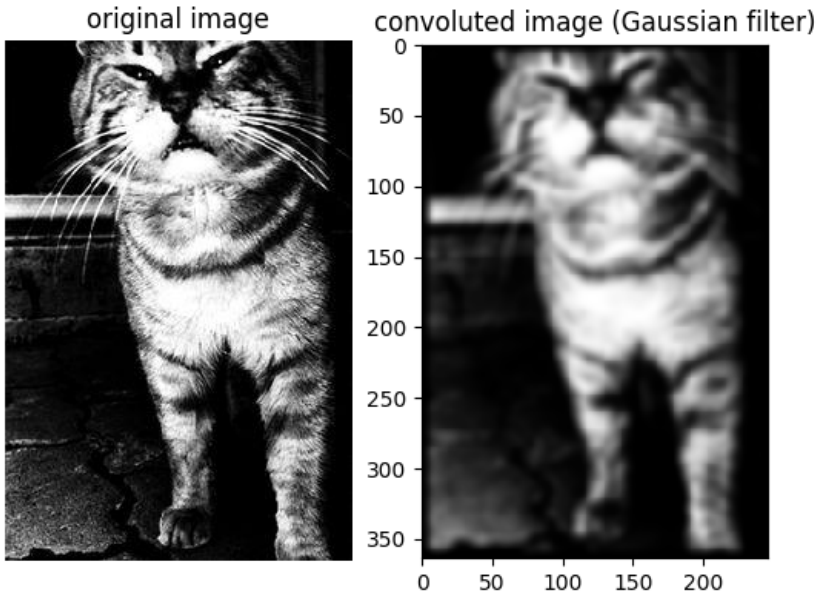

Figure 3: *Gaussian filter*

Different form the boxfilter, the Gaussian filter use different weight for the neighborhood, closer pixels are more weighted than farther ones. In addition, the value of sigma plays an important role in the Gaussian distribution. Lower values of sigma tend to preserve more the image, given that the neighborhood is low weighted, and thus, the central pixel is even more relevant. Higher sigma values, however, tend to blur more the image, due to the similar distribution of weight in the neighboring pixels. When sigma goes to infinity, we have a boxfilter.

Gaussian filters are frequently used because they can represent better the effect of optical lens when compared with boxfilters.

1.7

One could apply the convolution of the image with the 1D Gaussian filter and after an another convolution, with the result of the first convolution and the transpose of the 1D Gaussian filter. The result would be exactly the same as applying the convolution with 2D Gaussian filter.

The mathematical demonstration that the result would be the same can be seen below:

$$I * 2DG = F -> 2DG = 1DG * (1DG') -> (I * 1DG) * 1DG' = F$$

Where I is the image, $2DG$ is 2D Gaussian filter, $1DG$ is 1D Gaussian filter, $1DG'$ is the transpose of the 1D Gaussian filter and F is the filtered image.

Note that this operation is possible due to the fact that 2D Gaussian filter are symmetric and separable.

When doing the filtering process with 1D Gaussian filter and its transpose, we are realizing $m + m$ multiplication by pixel, instead of mxm (2D Gaussian filter). It means that filtering in two steps is of 1 first order $O(1)$ and 2D is a second order process $O(2)$.

Therefore, we can enhance our efficiency and velocity.

1.8

To show the concept developed in the previous exercise, a plot of filter size vs. computation time was done . The plot 4 elucidates the size-time relation when using 2D box filter of increasing size (3,100) and the method described in subsection 1.7.

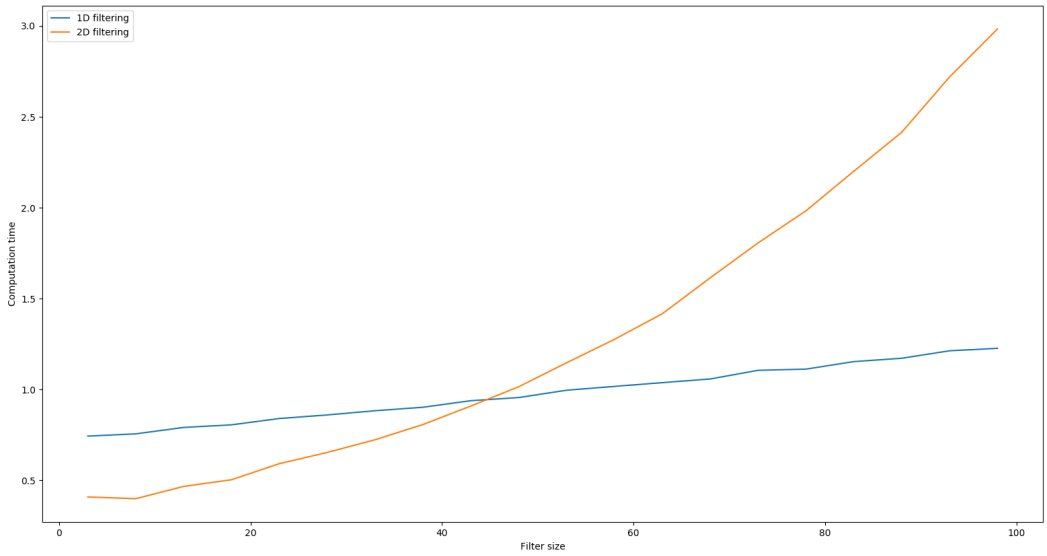

Figure 4: *Computation Time*

As we can see, the 1D filtering approach is more efficient than 2D approach.

At the beginning we can notice that the 2D approach is faster, which may happens due to function calls and how my code was implemented. Nevertheless, as the size of the filter increases, it is evident how much the 1D approach is better (0(1) against 0(2)).

2 Finding edges

Edges are extremely rich in information about the shape and details in images. Therefore, it detection is an important tool for image processing.

Edge are generated due to discontinuities in depth, illumination, reflectance and surface orientation.

2.1

By using the functions of the previous exercise a 1D derivative of Gaussian Y (Gy) and a 1D derivative Gaussian in X (Gx) were generates.

In my case, the chosen derivative operator were $dx = [-1, 0, 1]$ and $dy = [-1; 0; 1]$.

Gx and Gy can be seen in the figure 5

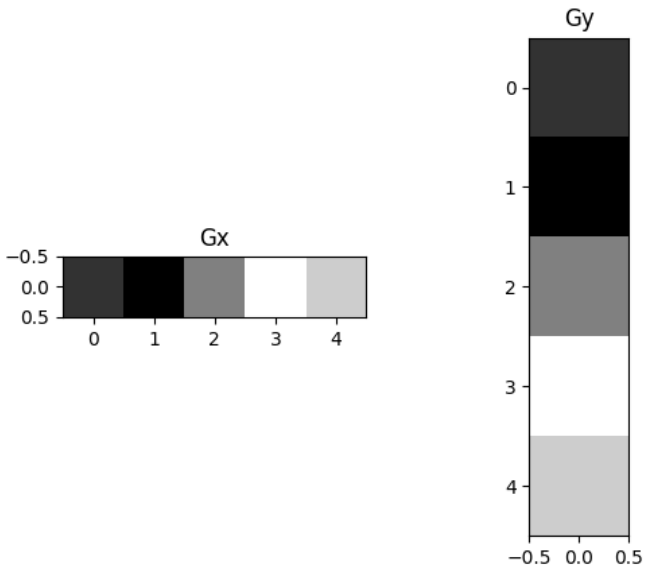

Figure 5: *Gx and Gy*

We can clearly notice the effect of dy and dx over the Gaussian filter.

2.2

An edge magnitude image was obtained by , basically, filtering the image with the filters Gx (to generate Ix) and Gy (to generate Iy). Then, the magnitude and the gradient orientation are acquired (function *np.sqrt* and *np.power* and *np.angle*). The result obtained is elucidated in the figure 6

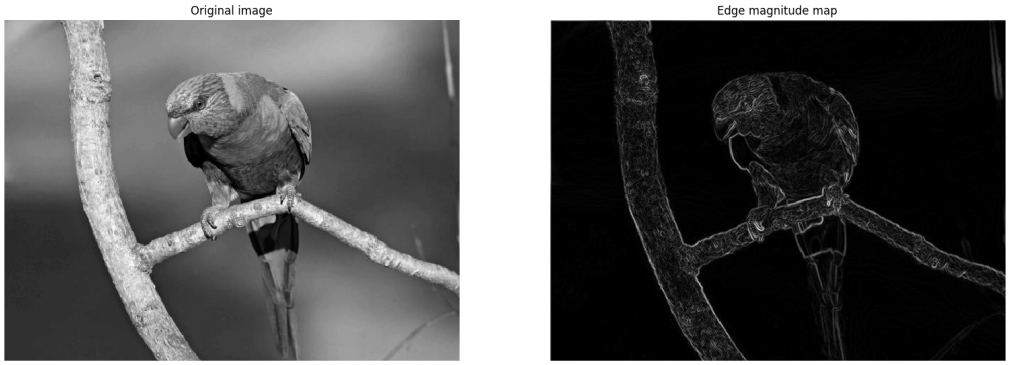

Figure 6: *edge magnitude map*

2.3

The edge images of particular directions were coded by using the function *np.where* and logical operators. The result of an image test (*circle.jpg*) can be seen in the figure 7.

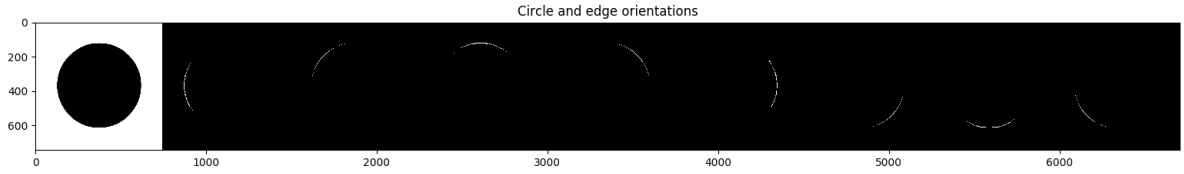

Figure 7: *edge images of particular directions in a circle*

The value chosen for r was 25. Others values were tested, but higher values of threshold leads to missing important edge, and lower values leads to an huge amount of edge that are not representatives. This difference between the threshold can be observed in the figure 8. The first subplot is result of a $r = 10$, the second $r = 25$ and the last (button) $r = 80$.

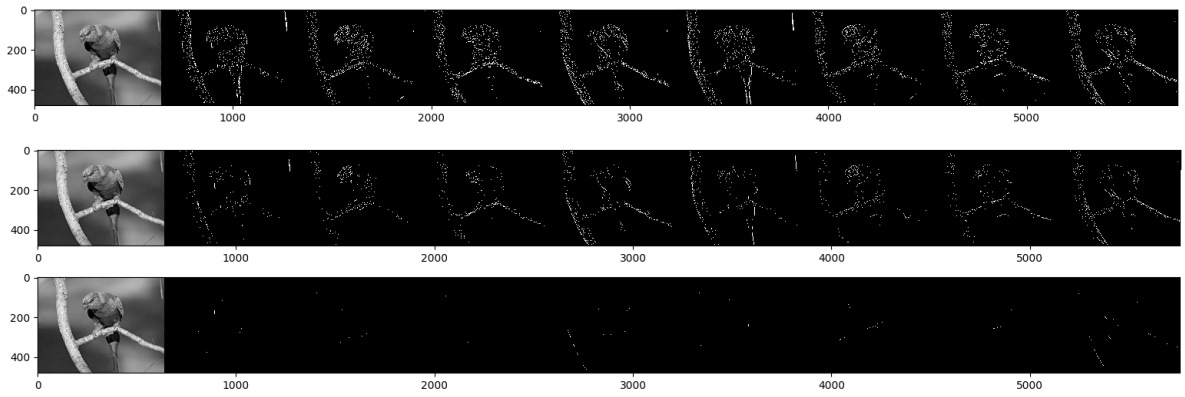

Figure 8: $Different\ r\ values$

2.4

Non-maximum suppression is an technique used to make edge thinner. In such a technique the pixel in an edge with higher value in a certain direction is maintained and the others are set to zero. The idea is to remove the blurred of the edge by finding the local maximum.

The results of the non-maximum suppression applied on the images *circle.jpg* and *bird.jpg* were shown in the figures 9 and 10.

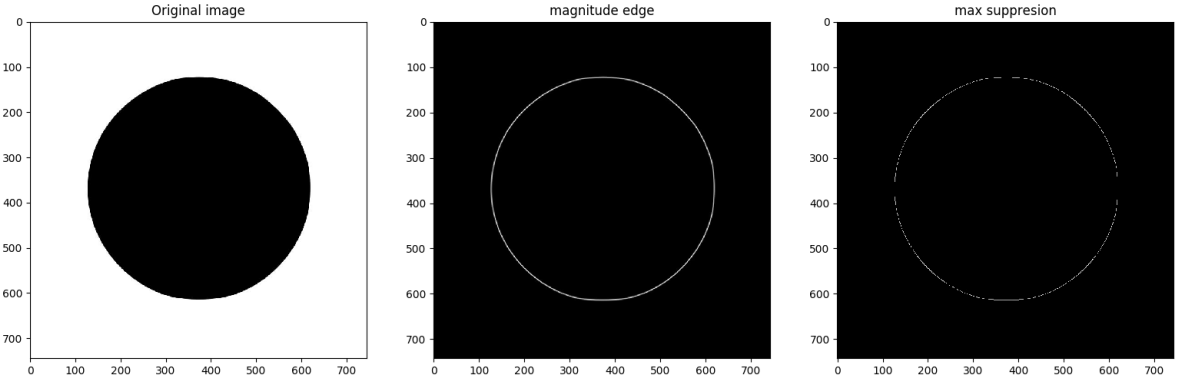

Figure 9: $Non-maximum\ suppression\ circle$

When looking at figure 9 , it seems that the final image (max suppression) has some holes. However, as shown in the figure 11, the circle is complete and the non-maximum suppression works perfectly fine.

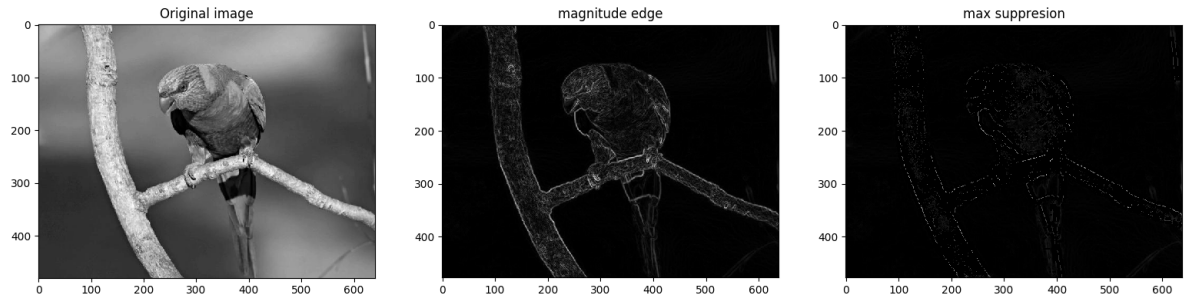

Figure 10: $Non - maximum\ suppression\ bird$

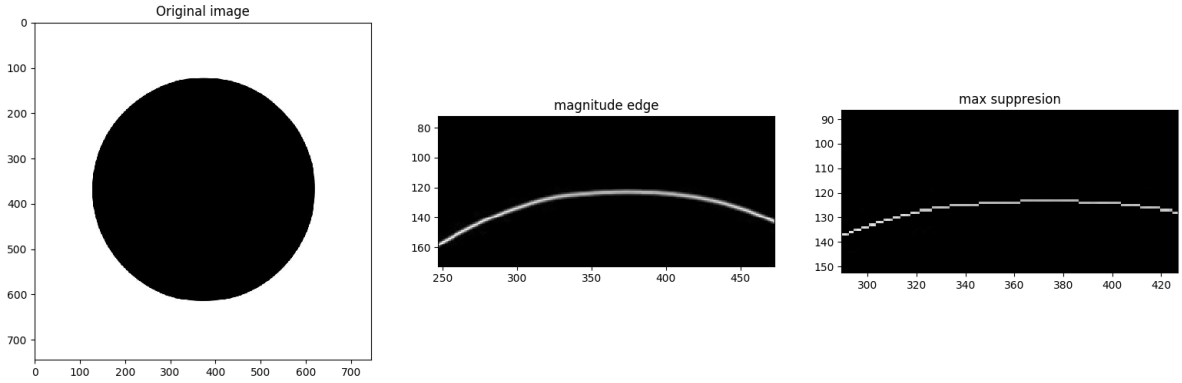

Figure 11: $Non - maximum\ suppression\ circle\ zoom$

3 Corner detection

Corner detection is technique used to obtain features of an image. It is frequently utilized in computer vision for different task, such as, for instance, object recognition and motion detection.

3.1

Harris corner detection is a method based on the placing of a square on the region of interest. The square is replaced (moved) in all direction and the deviation of the gradients are observed. This deviation is then used to determine if the region of analyze is a corner or not.

For the implementation of the Harris corner detection the functions *scipy.signal.convolve*, *np.multiply* , *np.linalg.det* and *np.trace* were used.

The values of sigma, k and window size are preserved, 0.2,0.1 and 5x5 respectively.

3.2

The algorithm implemented in the previous exercise (3.1) was used in the image *chessboard.jpeg*.

At the end, a threshold on R was applied ($R > 2$) to better visualize the results, which is shown in the figure 12.

The results shows the ability of the algorithm to detect corner. Furthermore, we can recognize the effect that the light source has in the corner detection. The reflections of light on the Chess pieces are detected as corner with a right value of R. Some corner were also missed due to the parameters sigma, k and window size.

3.3

The function *scipy.ndimage.rotate* was utilized to rotate the image (45 degrees). After that, the same function of the exercise 3.2 was called. The result is elucidated in the figure 13

We can notice that, besides the rotation, the detection of the corners is similar with the result of the previous exercise. That shows the rotational invariant property of the algorithm.

3.4

The function *scipy.misc.imresize* was utilized to downscale the image (50 percent). After that, the same function of the exercise 3.2 was called. The result can be seen in the figure 14

We can remark that the detection of the corners is different from the result of the exercise 3.2. The values of R did completely changed , and thus, the threshold must be adapted to the new range of R values . Therefore, we can point out that the Harris Corner Detection algorithm is size variant.

3.5

Looking at the results from (3.2), (3.3) and (3.4) we can say that Harris Corner Detection is invariant to rotation and variant to size.

Harris Corner Detection is based on the computation of R, that can be done by the following formula:

$$\text{R} = \det(\text{M}) - \text{k}(\text{trace}(\text{M}))^2 = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

By looking at this equation, we can say that R is dependent of the eigenvalues of the matrix M. Eigenvalues by definition are independent of any rotation in M. Therefore, a rotation in the image would generate a rotation in M, which would not change the values of R , and thus, the Harris Corner Detection is invariant to rotation.

Nevertheless, a change in size would leads to a change in the eigenvalues, making Harris algorithm variant to size.

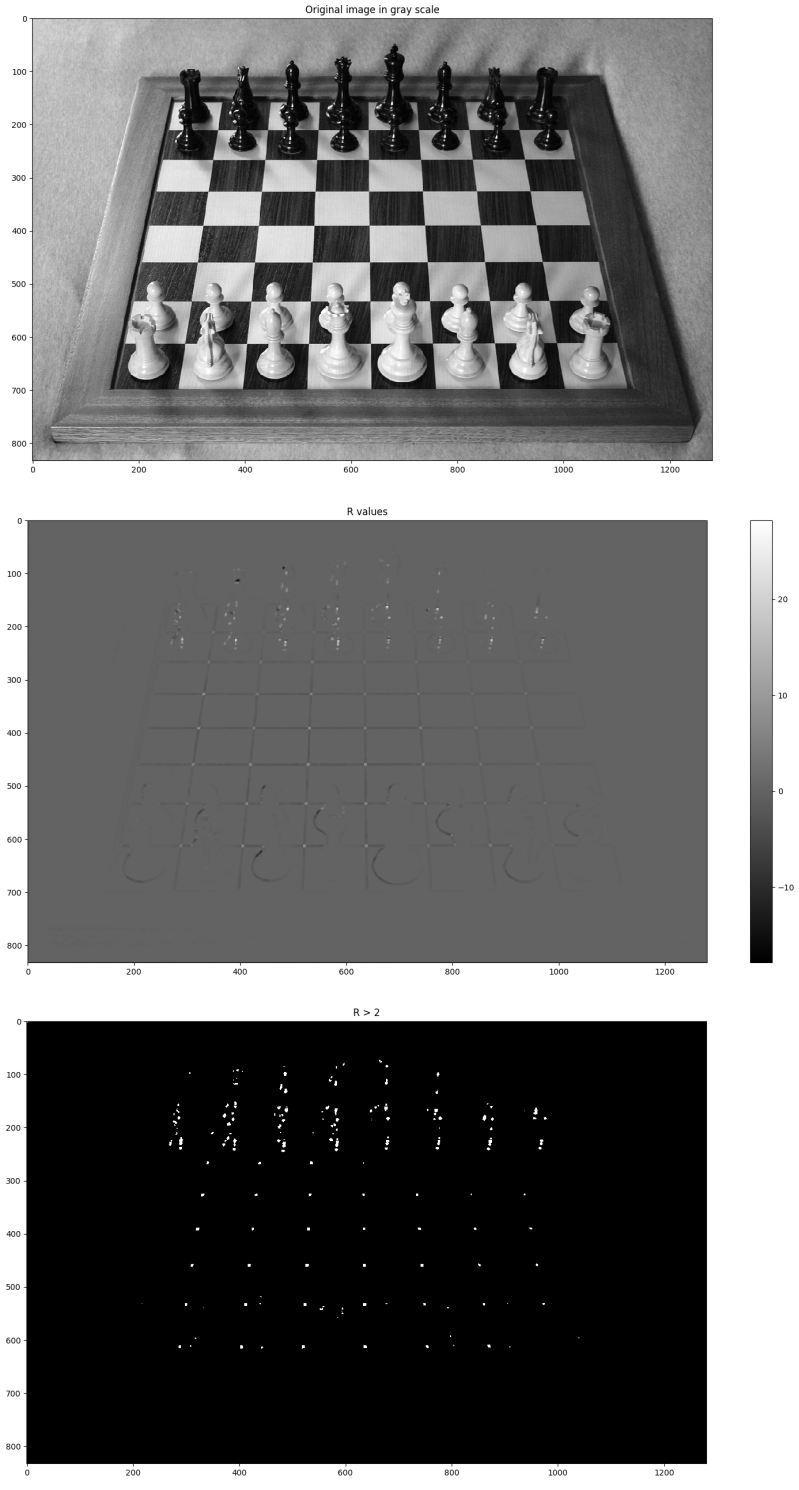

Figure 12: *Harris Corner Detection*

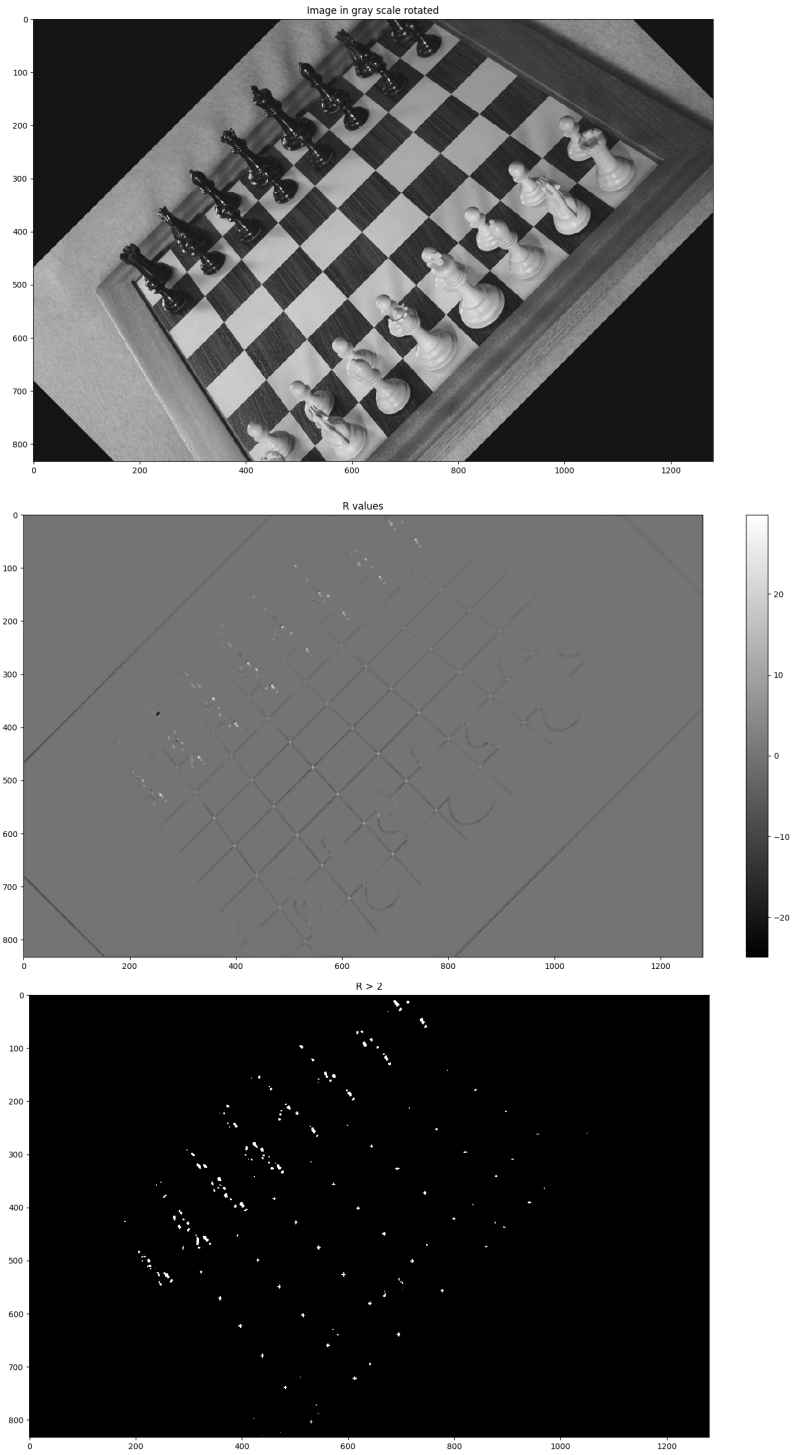

Figure 13: *Harris Corner Detection, rotated image*

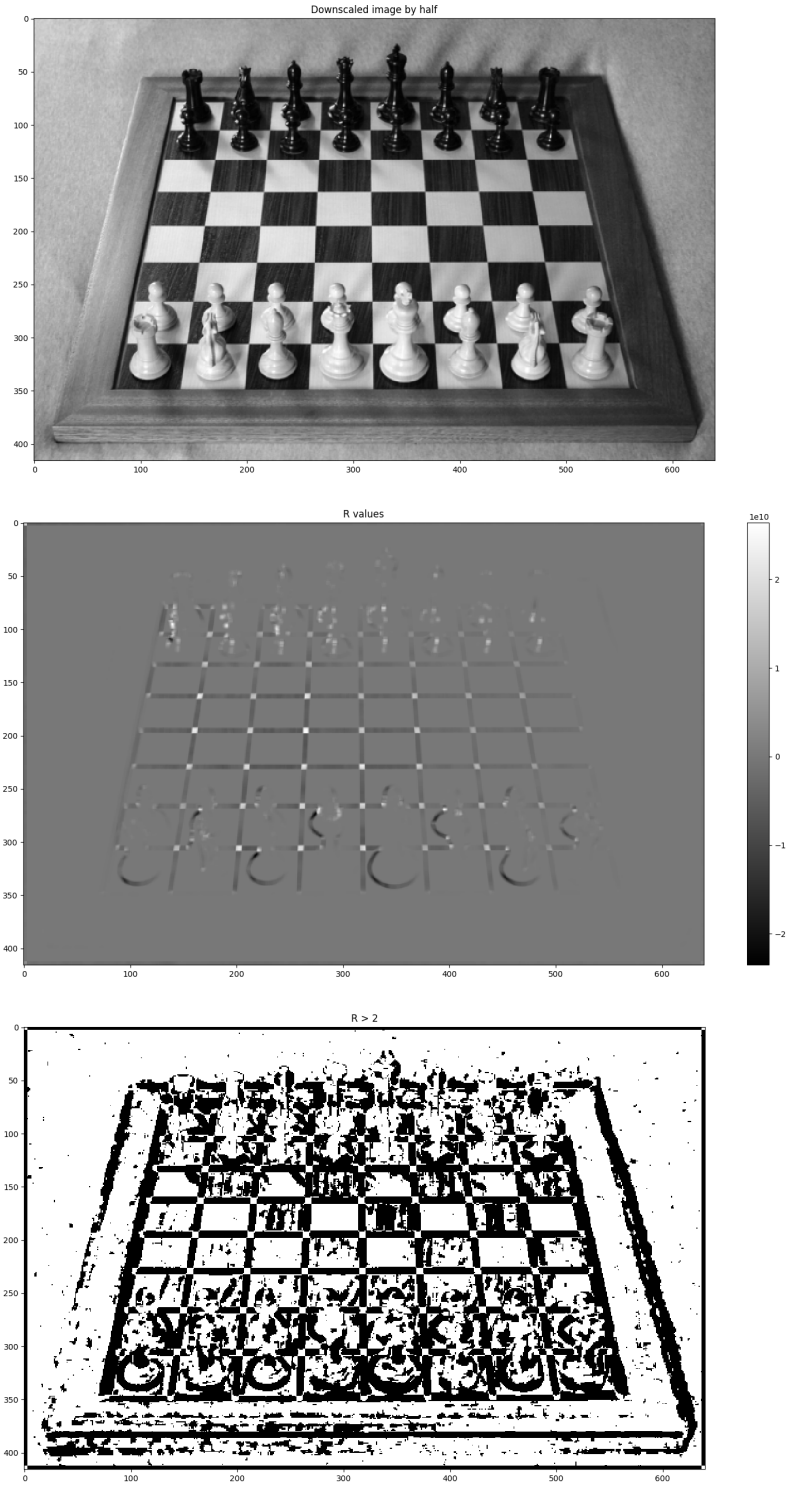

Figure 14: *Harris Corner Detection, down − scaled image*