IPOL IMAGE PROCESSING ON LINE

Published in Image Processing On Line on YYYY–MM–DD.
Submitted on YYYY–MM–DD, accepted on YYYY–MM–DD.
ISSN 2105–1232 © YYYY IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
http://dx.doi.org/10.5201/ipol

A Review of Classic Edge Detectors

Haldo Spontón[1], Juan Cardelino[2]

[1] IIE, UdelaR, Uruguay (haldos@fing.edu.uy)
[2] IIE, UdelaR, Uruguay (juanc@fing.edu.uy)

PREPRINT November 10, 2014

Abstract

In this paper some of the classic alternatives for edge detection in digital images are studied. The main idea of edge detection algorithms is to find where abrupt changes in the intensity of an image have occurred. The first family of algorithms reviewed in this work uses the first derivative to find the changes of intensity, such as Sobel, Prewitt and Roberts. In the second reviewed family, second derivatives are used, for example in algorithms like Marr-Hildreth and Haralick.
Results obtained from a qualitative point of view (perceptual) and from a quantitative point of view (number of operations, execution time) are compared, considering different ways to convolve an image with a kernel (step required in some of the algorithms).

Source Code

For all the algorithms reviewed, an open source C implementation is provided that can be downloaded from the IPOL publication of this article. An online demonstration[1][2] is also available, where you can test and reproduce our results.

1 Introduction

Edge detection is one of the oldest and most basic operations operations in image processing, which is often used as a basic building block for more complex algorithms. The basic idea of edge detection is to detect abrupt changes in image intensity. Detecting those changes can be accomplished using first or second order derivatives.

In the 70's, edge detection methods were based on using small operators (such as Sobel masks), attempting to compute an approximation of the first derivative of the image [6]. Section 2 describes such algorithms and serves as an introduction to a more sophisticated analysis of the edge detection process.

[1] http://dev.ipol.im/~juanc/ipol_demo/sc_edges/
[2] http://dev.ipol.im/~juanc/ipol_demo/sc_edges/

In 1980 Marr and Hildreth [?] argued that intensity changes are not independent of image scale, so edge detection requires the use of different size operators. They also argued that a sudden intensity change will be seen as a peak (or valley) in the first derivative or, equivalently, as a zero crossing in the second derivative. This algorithm is presented in Section 3. Haralick's algorithm [7] is an alternative approach based on the second derivative which is also reviewed in Section 3. This algorithm has the particularity of proposing a model to locally approximate the image around a point; then, using this model, an approximation to the second derivative of the image can be calculated analytically and finding edges is achieved by imposing a condition over the model parameters.

C1 The objective of this work is to provide reproducible implementations of the abovementioned algorithms, along with a comprehensive quantitative evaluation of the behaviour algorithms with respect to the parameters. Quantitative benchmarks are outside the scope of this work and will be addressed in future work. In addition, we will focus on low-level edge detection algorithms, which means that they will output a disconnected an unordered set of edgels (individual pixel facets where boundaries occur). Algorithms that perform edge integration into objects boundaries are of higher level of abstraction and not considered in this work.

Along with this paper, a detailed and well commented source code is presented, which implements the described algorithms. In section 4 some common mathematical developments are presented. Results of the implemented algorithms are presented in Section 5, along with examples to compare their performance. Conclusions are detailed in Section 6.

2 Algorithms based on the first derivative

Algorithms based on first derivatives studied in this paper share a common structure, with the only difference in the type of filtering used to compute those derivatives. Figure 1 shows a block diagram of that common structure.

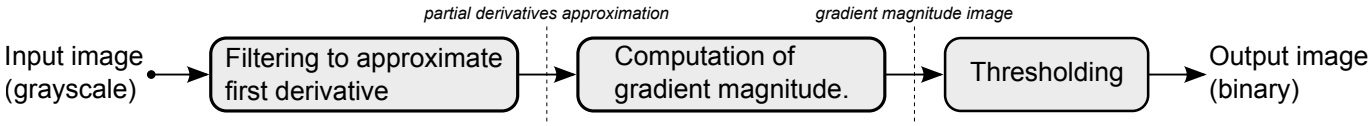


Figure 1: Block diagram of first derivative edge detection algorithms.

The usual tool to find the amplitude and direction of changes in intensity of an image f is the gradient operator (denoted as ∇), defined as the vector

$$\nabla f = \begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}. \tag{2.1}$$

The magnitude (M) and direction (α) of the gradient vector ∇f at location (x, y) are calculated as

$$\begin{cases} M(x,y) = \sqrt{f_x^2 + f_y^2} \\ \alpha(x,y) = \tan^{-1}\left(\dfrac{f_y}{f_x}\right). \end{cases} \tag{2.2}$$

The direction of an edge at an arbitrary location (x, y) of the image is orthogonal to the direction $\alpha(x, y)$ of the gradient vector.

To obtain the gradient, the partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$ need to be computed at every pixel in the image. When dealing with digital images, numerical approximations of the partial derivatives are required, calculated in a neighborhood of each point. In the following, the methods of Roberts, Prewitt and Sobel will be studied, whose main difference is how they perform this calculation.

2.1 The *Roberts* operators

C2 The Roberts edge detector was introduced in 1963 by L.G. Roberts in the context of 3D reconstrution [?], and it was based on computing the first derivative of the image. The most usual approach to approximate the first derivative is to use a the first order Taylor expansion with a small value of h. Thus $f'(x)$ is computed as C2

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}. \tag{2.3}$$

Digital images are commonly seen as a discretization of a continuous image, where a pixel is evenly sampled on a grid. If we denote the grid sizes by Δx and Δy, the continuous coordinates x and y can be written as $x = i\Delta x$ and $y = i\Delta y$. When no information about the spatial sampling is available, the size of the grid it is commonly assumed to be $\Delta x = \Delta y = 1$.
Thus, in the discrete space the first derivative of the image is computed as:

$$f_x = \frac{\partial f(x,y)}{\partial x} \cong f(i+1,j) - f(i,j) \tag{2.4}$$

and

$$f_y = \frac{\partial f(x,y)}{\partial y} \cong f(i,j+1) - f(i,j). \tag{2.5}$$

Equations 2.4 and 2.5 can be implemented for all values of x and y by filtering the image $f(x,y)$ with the 1-D masks shown in Figure 2. However, in order to be able to compute diagonal edges, 2-D masks are needed.
The *Roberts operators* are one of the earliest attempts to use 2-D masks for this purpose. These operators are based on computing the diagonal differences implemented by filtering an image with the masks in Figure 3. By convention in these figures, x-coordinates grow from left to right and y-coordinates grow top-down. It is also usual to scale the mask values, in order to have gain equal to 1.

Figure 2 shows the simplest way to compute derivatives, but Roberts [?] proposes to compute them as shown in Figure 3. In order to use the same convolution code for all algorithms, our implementation uses 3×3 matrices, adding a row and a column of zeros to the matrices in Figure 3, what produces the same result.

2.2 The *Prewitt* operators

Masks of size 2×2, although conceptually simple, are not symmetrical with respect to the central points. Having symmetrical edges is a desirable property and can only be achieved with oddly sized masks, the smallest of them being the 3x3. These masks provide more information to find the direction of the edges, because they take into account information on opposite sides of the central point.

The simplest digital approximation to the partial derivatives using masks of size 3×3 are obtained by taking the difference between the third and first rows (or columns) of the 3×3 region. The

difference between the third and first rows approximates the derivative in the x-direction, and the difference between the third and first columns approximates the derivative in the y-direction. These approximations can be implemented by filtering the image with the two masks in Figure 4. These masks are called *Prewitt operators* [**?**].

The left mask can be constructed (up to a scale factor) as the convolution of two filters: a vertical derivative $[-1\ 0\ 1]$ and an horizontal moving average $[1\ 1\ 1]^T$. The same reasoning could be applied to the right mask. Hence, these operators can be interpreted as the consecutive application of an horizontal low pass filter with a vertical derivative filter. Thus, they show smoothing properties in the direction orthogonal to the gradient.

2.3 The *Sobel* operators

A slight variation of the Prewitt operators which use more weight on the central coefficients of the difference. It can be shown that using more weight in the center location provides better image smoothing. This variation is implemented using masks shown in Figure 5. These operators are called *Sobel operators*.

Sobel masks can be seen (up to a scale factor) as the convolution of a horizontal derivation mask $[-1\ 0\ 1]$ with a vertical smoothing filter $[1\ 2\ 1]$ (closer to a Gaussian response than Prewitt), and vice versa. Hence, these operators have better smoothing properties, as mentioned in the preceding paragraph.

The computational cost of applying Prewitt and Sobel masks is exactly the same. Thus, it is preferable to use Sobel operators, because edges are better localized, and their filter are also less aliased, because of the weighted shape of $[1\ 2\ 1]$.

2.4 Computation of the edges

As mentioned before, edges can be seen as abrupt changes in intensity, i.e. higher values of gradient. An example of this behavior is shown in Figure 6, where gradient was computed using Sobel operators.

Then, once the respective operators are applied, an approximation of the gradient (stored in two matrices) is obtained, containing the approximation of the partial derivatives f_x and f_y. Gradient magnitude image M is calculated using the equation 2.2 (Figure 6(c)).

Finally, the edges image is obtained by thresholding the gradient magnitude image, i.e. for pixel a at position i, j in the image compute

$$\text{edges}[i,j] = \begin{cases} 1, & \text{if } M[i,j] \geq \text{th} \\ 0, & \text{if } M[i,j] < \text{th} \end{cases}$$

where th is the threshold. A black and white image is obtained, in which edge points are indicated in white (see Figure 7). It is noted that different thresholds lead to different results. Thicker or more edges for smaller thresholds, fewer edges for larger thresholds. This threshold, and the others mentioned below, are defined as a percentage of the maximum value of the gradient image. The advantage of this is to adapt the threshold to the dynamic range of the image. Therefore, the parameter *threshold* in the algorithms takes values between 0 and 1.

2.5 Pseudo-code

The pseudo-code of implemented algorithms is shown in Algorithm 1.

[!h]

-1
1

-1	1

Figure 2: One-dimensional masks used to implement equations 2.4 and 2.5.

[!h]

-1	0
0	1

0	-1
1	0

Figure 3: *Roberts cross-gradient* 2-D masks.

[!h]

$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$
0	0	0
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

$-\frac{1}{3}$	0	$\frac{1}{3}$
$-\frac{1}{3}$	0	$\frac{1}{3}$
$-\frac{1}{3}$	0	$\frac{1}{3}$

Figure 4: Normalized *Prewitt* 2-D masks of size 3×3.

[!h]

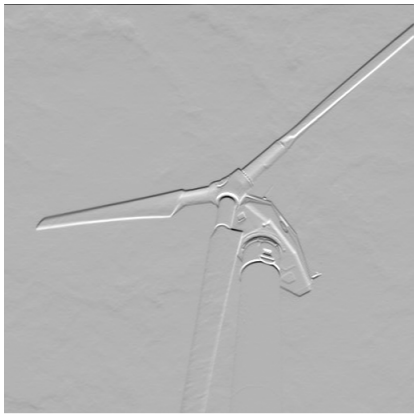
$-\frac{1}{4}$	$-\frac{1}{2}$	$-\frac{1}{4}$
0	0	0
$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$

$-\frac{1}{4}$	0	$\frac{1}{4}$
$-\frac{1}{2}$	0	$\frac{1}{2}$
$-\frac{1}{4}$	0	$\frac{1}{4}$

Figure 5: *Sobel* 2-D masks of size 3×3.

(a) Grayscale image.

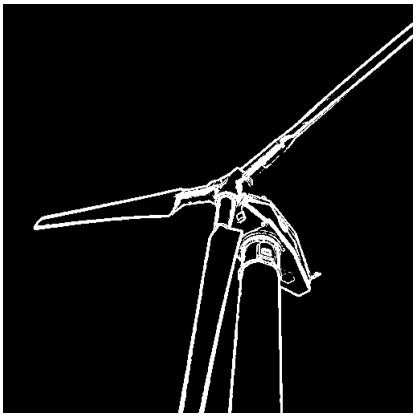
(b) Horizontal derivative f_x.



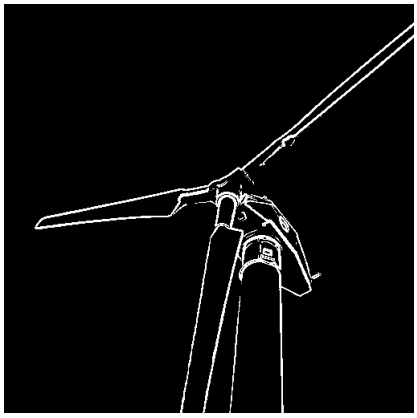
(c) Vertical derivative f_y.

(d) Gradient module image $M = \sqrt{f_x^2 + f_y^2}$.

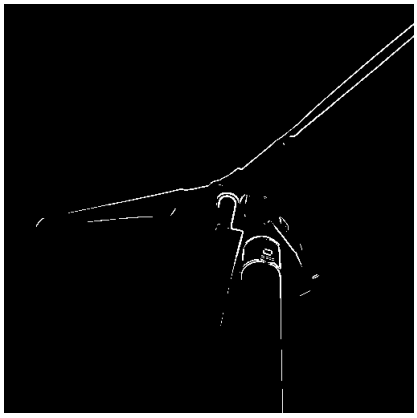
Figure 6: Example computation of the gradient. Original image and gradient image computed using Sobel operators (Operators in Figure 5).



(a) $th = 0.1 * \max(M)$.



(b) $th = 0.2 * \max(M)$.



(c) $th = 0.4 * \max(M)$.

Figure 7: Thresholded images. Different values of the threshold applied to the magnitude of the gradient.

Algorithm 1: First derivative edge detection algorithms.

Require: input image, threshold th.

$im \leftarrow$ input image
Define $operator_x$ and $operator_y$ {Roberts, Prewitt or Sobel.}
$g_x \leftarrow$ convolution(im,$operator_x$)
$g_y \leftarrow$ convolution(im,$operator_y$)
$max_M \leftarrow 0$
for all pixel i in image **do**
  $M[i] \leftarrow \sqrt{g_x^2 + g_y^2}$ {Gradient magnitude.}
  if $M[i] > max_M$ **then**
    $max_M \leftarrow M[i]$
  end if
end for
for all pixel i in image **do**
  if $M[i] \geq th \times max_M$ **then**
    $im_{OUT}[i] \leftarrow 255$
  else
    $im_{OUT}[i] \leftarrow 0$
  end if
end for
return output image $\leftarrow im_{OUT}$

3 Algorithms based on second derivative

The edge detection methods discussed in the previous section are simply based on filtering the image with different masks, without taking into account the characteristics of the edges or noise in the image.

C3 The two algorithms presented in this section (Marr-Hildreth [**?**] and Haralick [7]) are based on the second derivative of the image, and both take steps to reduce noise before detecting edges in the image. C3

3.1 The *Marr* and *Hildreth* algorithm

The Marr-Hildreth algorithm is a method of detecting edges in digital images. It is based on finding zero crossing points of the second derivative of the image. This can be done in several ways. Two different ways of doing this are implemented in this work (see block diagram in Figure 8): convolving the image with a Gaussian kernel and then approximating the second derivative (Laplacian) with a 3x3 kernel, or convolving the image with a kernel calculated as the Laplacian of a Gaussian function. There are more ways to do so, for example, using recursive Gaussian filters [4].

The algorithm is divided in two steps, each one described later:

1. Grayscale conversion of the input image (see section 4).
2. Convolution of the image with:
   - a Laplacian of Gaussian (LoG) kernel, (or)
   - a Gaussian kernel and then a Laplacian operator.
3. Search of zero crossing points in the filtered image.

Some auxiliary functions are needed such as Gaussian kernel and Laplacian of a Gaussian kernel generation, and 2-D convolution of an image with a given kernel, with different boundary conditions. These operations will be discussed in detail in section 4.

3.1.1 Gaussian and LoG kernels

The Marr-Hildreth algorithm consists on convolving the input image $f(x, y)$ with a LoG kernel;

$$g(x, y) = [\nabla^2 G(x, y)] \star f(x, y), \tag{3.1}$$

and then finding the zero crossings of $g(x, y)$ to determine the location of edges in $f(x, y)$. Because these are linear processes, equation 3.1 can be written also as

$$g(x, y) = \nabla^2 [G(x, y) \star f(x, y)] \tag{3.2}$$

indicating that the image can be smoothed with a Gaussian filter first, and then compute the Laplacian of the result[3] .

Then, generation of both Gaussian and LoG kernels is required (see section 4).

The Marr-Hildreth edge-detection algorithm may be summarized as follows:

1. Filter the input image with a $n \times n$ Gaussian lowpass filter obtained by sampling the Gaussian kernel (see equation 4.1).
2. Compute the Laplacian of the image resulting from step 1, using, for example, the 3×3 mask[4]:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3. Find the zero crossings of the image from step 2.

3.1.2 Zero crossing

A zero crossing at pixel p implies that the signs of at least two opposite neighboring pixels are different. There are four cases to test: left/right, up/down, and the two diagonals. In this case a threshold is used, so that not only the signs of the opposite pixels must differ, also their difference in absolute value must be greater than a certain threshold.

[3]The difference between these approaches lies in the compromise between the accuracy in the calculation and the computational cost (see Section 4.1.3).

[4]Steps 1 and 2 can be merged into one, using a $n \times n$ LoG lowpass filter obtained by sampling equation 4.7.

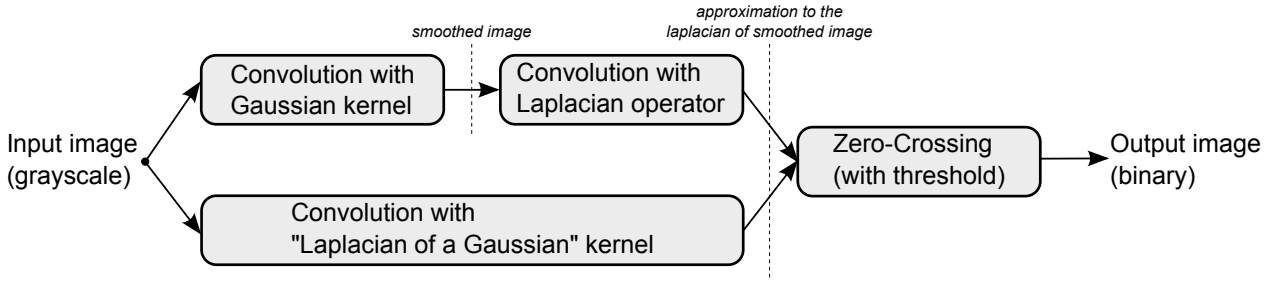


Figure 8: Block diagram of Marr-Hildreth algorithm.

The zero-crossing threshold (th_{ZC}) is given as a percentage of the maximum value max_L of the Laplacian image (both Gaussian and LoG kernels). Each pixel p has eight neighbors, named according to their position as follows.

$$\begin{array}{ccccc} p_{up,left} & & p_{up,middle} & & p_{up,right} \\ & \nwarrow & \uparrow & \nearrow & \\ p_{middle,left} & \leftarrow & p & \rightarrow & p_{middle,right} \\ & \swarrow & \downarrow & \searrow & \\ p_{down,left} & & p_{down,middle} & & p_{down,right}. \end{array}$$

Then a pixel p is considered as edge pixel if any of the following conditions is true (for simplicity the Laplacian image is denoted as $\mathcal{L}$):

- $(\text{sign}(\mathcal{L}[p_{up,left}]) \neq \text{sign}(\mathcal{L}[p_{down,right}])\ \&\ |\mathcal{L}[p_{up,left}] - \mathcal{L}[p_{down,right}]| > th_{ZC} * max_L$
- $(\text{sign}(\mathcal{L}[p_{up,middle}]) \neq \text{sign}(\mathcal{L}[p_{down,middle}])\ \&\ |\mathcal{L}[p_{up,middle}] - \mathcal{L}[p_{down,middle}]| > th_{ZC} * max_L$
- $(\text{sign}(\mathcal{L}[p_{down,left}]) \neq \text{sign}(\mathcal{L}[p_{up,right}])\ \&\ |\mathcal{L}[p_{down,left}] - \mathcal{L}[p_{up,right}]| > th_{ZC} * max_L$
- $(\text{sign}(\mathcal{L}[p_{middle,left}]) \neq \text{sign}(\mathcal{L}[p_{middle,right}])\ \&\ |\mathcal{L}[p_{middle,left}] - \mathcal{L}[p_{middle,right}]| > th_{ZC} * max_L.$

Zero crossing detection is the key feature of the Marr-Hildreth edge detection method. The technique presented in the previous paragraph is attractive for its simplicity of implementation and its low computational cost. In general it is a technique that yields good results, but if more precision in finding the zero crossings is needed, more advanced methods for finding zero crossings with subpixel accuracy could be used (e.g. marching squares [9]).

3.1.3 Pseudo-code

The pseudo-code of Marr-Hildreth's algorithm is shown in Algorithm 2.

3.2 The *Haralick* algorithm

In this section the original work of Haralick [7] on edge detection is presented in detail. The main idea behind this algorithm is identical to that of the previous method: find zeros in the second derivative of the image. In this method, however, the input image is smoothly approximated through local bi-cubic polynomial fitting. Then, when calculating the second derivative analytically, it is possible to find an equivalent expression to find the zeros of the second derivative of the polynomial as a function of its parameters.

3.2.1 Bi-cubic polynomial fitting

Here, the interpolation method used in the original work of Haralick is presented, although there are other options to do this [5].

C4 The surrounding neighborhood of a point (x_0, y_0) in the image f is approximated using the following bi-cubic polynomial C4

$$f(x,y) = k_1 + k_2(x-x_0) + k_3(y-y_0) + k_4(x-x_0)^2 + k_5(x-x_0)(y-y_0) + \tag{3.3}$$
$$k_6(y-y_0)^2 + k_7(x-x_0)^3 + k_8(x-x_0)^2(y-y_0) + k_9(x-x_0)(y-y_0)^2 + k_{10}(y-y_0)^3 \tag{3.4}$$
$$\tag{3.5}$$

Algorithm 2: Marr-Hildreth edge detection algorithm.

Require: input image, standard deviation σ, kernel size n and zero-crossing threshold t_{ZC}.

$im \leftarrow$ input image
if Gaussian kernel **then**
  Define Laplacian operator *laplacian*
  $im_{LAPL} \leftarrow$ convolution(im,*laplacian*)
else
  $kernel \leftarrow$ generate_kernel(n,σ) {Generated Gaussian or LoG kernel}.
  $im_{SMOOTHED} \leftarrow$ convolution(im,*kernel*)
  $im_{LAPL} \leftarrow im_{SMOOTHED}$
end if
$max_L \leftarrow 0$
for all pixel i in image im_{LAPL} **do**
  if $im_{LAPL}[i] > max_L$ **then**
    $max_L \leftarrow im_{LAPL}[i]$
  end if
end for
for all pixel i in image im_{LAPL}, except borders **do**
  for all pair (p_1, p_2) of opposite neighbors of p in im_{LAPL} **do**
    if $(\text{sign}(im_{LAPL}[p_1]) \neq \text{sign}(im_{LAPL}[p_2]))$ **and** $(|im_{LAPL}[p_1] - im_{LAPL}[p_2]| > th_{ZC})$ **then**
      $im_{OUT}[i] \leftarrow 255$
    else
      $im_{OUT}[i] \leftarrow 0$
    end if
  end for
end for
return output image $\leftarrow im_{OUT}$

To solve this fitting problem, it is necessary to take more neighbors than coefficients to be adjusted. As there are 10 coefficients to compute, the smallest neighborhood of odd size that accomplishes this has size 5×5. Having 10 coefficients and 25 data points leads to an overdetermined system, which can be solved using least squares or any other fitting technique.

In Haralick's work, this polynomial approximation is expressed in a local coordinate system centered on each pixel. In this way, the center of the neighboorhood (x_0, y_0) is always $(0, 0)$, and the coordinates (x, y) are offsets from this center point (e.g. in a neighborhood of size 5×5, x and y take values between -2 and +2). Thus, the expression of the polynomial is

$$f(x, y) = k_1 + k_2 x + k_3 y + k_4 x^2 + k_5 xy + k_6 y^2 + k_7 x^3 + k_8 x^2 y + k_9 xy^2 + k_{10} y^3. \tag{3.6}$$

Then the solution is computed using least squares, and the local expression of the polynomial ensures that the values of the k_i coefficients are computed in the same way for every pixel in the image. Thus, they can be computed by convolving the image with a set of fixed and precomputed masks, which results in a great speed-up of the algorithm. In the following we will state the approximation problem and show how to derive those masks from its solution.

Consider 25 points in a small neighborhood the point $(0, 0)$. This gives us an equal number of

equations to find 10 coefficients. As mentioned before this leads to an overdetermined system which can be solved by least squares. By substituting the 25 data points into the polynomial equation (3.6), the following system of equations is obtained,

$$
\begin{aligned}
f_1 &= f(x_1, y_1) = k_1 + k_2x_1 + k_3y_1 + k_4x_1^2 + k_5x_1y_1 + k_6y_1^2 + k_7x_1^3 + k_8x_1^2y_1 + k_9x_1y_1^2 + k_{10}y_1^3 \\
f_2 &= f(x_2, y_2) = k_1 + k_2x_2 + k_3y_2 + k_4x_2^2 + k_5x_2y_2 + k_6y_2^2 + k_7x_2^3 + k_8x_2^2y_2 + k_9x_2y_2^2 + k_{10}y_2^3 \\
&\vdots \\
f_{25} &= f(x_{25}, y_{25}) = k_1 + k_2x_{25} + k_3y_{25} + k_4x_{25}^2 + k_5x_{25}y_{25} + k_6y_{25}^2 + k_7x_{25}^3 + k_8x_{25}^2y_{25} + k_9x_{25}y_{25}^2 + k_{10}y_{25}^3.
\end{aligned}
$$

Using matrix notation, the system can be rewritten as

$$
\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{25} \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 & \dots & y_1^3 \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & y_2^2 & \dots & y_2^3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{25} & y_{25} & x_{25}^2 & x_{25}y_{25} & y_{25}^2 & \dots & y_{25}^3 \end{bmatrix} \times \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_{10} \end{bmatrix} \Rightarrow \mathbf{f} = \mathbf{A}\mathbf{k}.
$$

Then, the least squares problem can be solved by using the normal equations,

$$
(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{f} = \mathbf{k}
$$

Let us define a matrix $\mathbf{B} \in R^{10\times 25}$ such that $\mathbf{B} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$, thus

$$
\mathbf{k} = \mathbf{B}\mathbf{f}
$$

$$
\begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,25} \\ b_{2,1} & b_{2,2} & \dots & b_{2,25} \\ \vdots & \vdots & \ddots & \vdots \\ b_{10,1} & b_{10,2} & \dots & b_{10,25} \end{bmatrix} \times \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{25} \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_{10} \end{bmatrix}.
$$

For each coefficient (i from 1 to 10),

$$
k_i = b_{i,1}f_1 + b_{i,2}f_2 + b_{i,3}f_3 + \ldots + b_{i,25}f_{25} \quad \Rightarrow \quad \mathbf{k_i} = \mathbf{b_i}\mathbf{f}, \tag{3.7}
$$

where b_i is the vector containing the $i-th$ row of B, i.e.

$$
\mathbf{b_i} = \begin{bmatrix} b_{i,1} & b_{i,2} & \dots & b_{i,25} \end{bmatrix}. \tag{3.8}
$$

Let us rearrange b_i as a 5×5 matrix and denote it by $\mathbf{mask_i}$ as follows

$$
\mathbf{mask_i} = \begin{bmatrix} b_{i,1} & b_{i,2} & \dots & b_{i,5} \\ b_{i,6} & b_{i,7} & \dots & b_{i,10} \\ \vdots & \vdots & \ddots & \vdots \\ b_{i,21} & b_{i,22} & \dots & b_{i,25} \end{bmatrix}. \tag{3.9}
$$

We will compute one value of k_i for each pixel on the image. Thus we can store these coefficients in a matrix $\mathbf{K_i}$ of the same size as the original image. Then, using the mask shown in Equation 3.9 we can obtain $\mathbf{K_i}$ as the convolution of the original image with $\mathbf{mask_i}$.

$$
\mathbf{K_i} = f \star \mathbf{mask_i} \texttt{ for } i \in \{1 \dots 10\} \tag{3.10}
$$

As we expressed the approximation in a local coordinate system, the coefficients $b_{i,j}$ will be the same for every pixel, thus, this convolution is performed against a fixed and precomputed mask $\mathbf{mask_i}$. The numeric values of these 5x5 masks used in our experiments are shown in Table A.1.

3.2.2 Analytical calculation of the second derivative

The main idea of Haralick's algorithm, as stated in the original work, is to find "negative sloped zero crossings of the second derivative". C5 This means that it is actually looking for ascending step edges. According to Haralick's definition, a pixel will lie on such an edge if the following three conditions hold[5]:

1. the first derivative is non zero.
2. the second derivative is zero.
3. the third derivative is negative.

After the neighborhood of each point of the image is approximated using the bi-cubic polynomial expression in equation 3.6, the idea is to compute the derivatives of the image in the direction of the gradient using that approximation.

Let θ be the gradient angle, defined clockwise with respect to the y-axis[6]. We approximate it using the first order terms of the polynomial 3.6 obtaining the following expression:

$$\sin(\theta) = -\frac{k_2}{\sqrt{k_2^2 + k_3^2}} \tag{3.11}$$

$$\cos(\theta) = -\frac{k_3}{\sqrt{k_2^2 + k_3^2}}. \tag{3.12}$$

Let us consider the line r passing by $(0,0)$ with angle θ. The coordinates of each point (x, y) on this line can be described in terms of the angle θ and the distance ρ to $(0,0)$ as follows C6

$$x = \rho \sin\theta$$
$$y = \rho \cos\theta$$

If we substitute the above expressions in the bi-cubic polynomia of eq., we obtain

$$f_\theta(\rho) = C_0 + C_1\rho + C_2\rho^2 + C_3\rho^3, \tag{3.13}$$

where

$$\begin{aligned} C_0 &= k_1 \\ C_1 &= k_2 \sin(\theta) + k_3 \cos(\theta) \\ C_2 &= k_4 \sin^2(\theta) + k_5 \sin(\theta)\cos(\theta) + k_6 \cos^2(\theta) \\ C_3 &= k_7 \sin^3(\theta) + k_8 \sin^2(\theta)\cos(\theta) + k_9 \sin(\theta)\cos^2(\theta) + k_{10} \cos^3(\theta). \end{aligned} \tag{3.14}$$

[5]Please note that there is a typo in Haralick's original paper. The statement of the idea is presented at the beggining of Section III, in the first column of page 42. The three conditions are presented at the end of the same section, in the second column. There it reads $f''_\alpha(\rho) < 0$, $f''_\alpha(\rho) = 0$ and $f'_\alpha(\rho) \neq 0$. Clearly, the first and second conditions can't be true at the same time, thus the first one should be $f'''_\alpha(\rho) < 0$.

[6]Although it seems a non standard choice, we followed exactly from the original work

The derivatives are obtained as follows.

$$\begin{aligned} f'_\theta(\rho) &= C_1 + 2C_2\rho + 3C_3\rho^2 \\ f''_\theta(\rho) &= 2C_2 + 6C_3\rho \\ f'''_\theta(\rho) &= 6C_3. \end{aligned} \tag{3.15}$$

The simplest way to impose the above-mentioned conditions is starting with the third one, i.e. that the third derivative has to be negative. Thus,

$$f'''_\theta(\rho) = 6C_3 < 0 \quad \Rightarrow \quad C_3 < 0. \tag{3.16}$$

By construction θ is the direction of the gradient, which means that the edge will always be an ascending step. Thus by definition the function $f_\theta(\rho)$ is non-decreasing in a neighborhood of $\rho = 0$, which means that $f'_\theta(0) \geq 0$. This in turn implies that $C_1 \geq 0$.

The second condition that the second derivative is equal to zero becomes

$$f''_\theta(\rho) = 2C_2 + 6C_3\rho = 0 \tag{3.17}$$

Ideally, if the edge point falls exactly on a grid location, the polynomial function should be zero at $\rho = 0$. But due to the finite nature of the image grid, the edge could be located at non integer locations. Thus, the condition is relaxed asking that the polynomial becomes zero in a neighborhood of radius $\rho_0 > 0$ centered at $\rho = 0$. With this relaxation, the condition 3.17 becomes

$$\exists \rho \in [0, \rho_0) \ \texttt{such that} \ 2C_2 + 6C_3\rho = 0 \tag{3.18}$$

Thus, the solution is $\rho_* = -\frac{C_2}{3C_3}$ and imposing $|\rho_*| < \rho_0$ we obtain the second condition

$$\left| \frac{C_2}{3C_3} \right| < \rho_0, \tag{3.19}$$

Finally, the first condition is that the first derivative is non zero,

$$f'_\theta(\rho) = C_1 + 2C_2\rho + 3C_3\rho^2 \neq 0 \tag{3.20}$$

C7 If we evaluate f'_θ on the solution ρ_* using the above formula, we obtain C7

$$f'_\theta(\rho_*) = C_1 - \frac{C_2^2}{3C_3} \tag{3.21}$$

as $C_1 >= 0$ and $C_3 < 0$, $f'_\theta(\rho_*)$ will always be positive, thus the first condition always hold by construction and does not need to be verified.

In Haralick's work ρ_0 is a fixed threshold which acts as a parameter of the algorithm. Its value must be greater than zero and and less than the size of the grid (i.e. $\rho_0 < 1$). In this work we obtained the best results with values between 0.4 and 0.6.

Algorithm 3: Haralick edge detection algorithm.

Require: input image (width w, height h), edge point condition threshold ρ_0.

$im \leftarrow$ input image
Define the ten 5×5 masks ($mask_1 \dots mask_{10}$) used to determine coefficients k_1 to k_{10} {See Table A.1.}
for all pixel i in image im **do**
  $neighbors \leftarrow 5 \times 5$ neighborhood of pixel i in image im
  for $j = 1$ **to** 10 **do**
    $k_j \leftarrow$ convolution($neighbors$,$mask[j]$)
  end for
  $C_2 \leftarrow \frac{k_2^2 k_4 + k_2 k_3 k_5 + k_3^2 k_6}{k_2^2 + k_3^2}$
  $C_3 \leftarrow \frac{k_2^3 k_7 + k_2^2 k_3 k_8 + k_2 k_3^2 k_9 + k_3^3 k_{10}}{(\sqrt{k_2^2 + k_3^2})^3}$
  if $|C_2/3C_3| \leq \rho_0$ **and** $C_3 < 0$ **then**
    $im_{OUT}[i] \leftarrow 255$
  else
    $im_{OUT}[i] \leftarrow 0$
  end if
end for
return output image $\leftarrow im_{OUT}$

3.2.3 Algorithm

The Haralick edge detection algorithm is summarized in the following 4 steps:

1. For each pixel in the image, find the coefficients $k_1 \dots k_{10}$, as shown in section 3.2.1.
2. Compute $\sin(\theta)$ and $\cos(\theta)$ (equations 3.11).
3. Compute C_2 and C_3 (equations 3.14).
4. If $\left|\frac{C_2}{3C_3}\right| < \rho_0$ and $C_3 < 0$, then that point is an edge point.

3.2.4 Pseudo-code

The pseudo-code of Haralick's algorithm is shown in Algorithm 3.

4 Common Mathematical Operations

All implemented algorithms use some common mathematical operations that are independent of the algorithms themselves. These operations, although basic, have a great impact on the algorithm outcome, and thus they need to be implemented with care. In this section those operations are reviewed.

4.1 Kernel generation

Some of the algorithms presented above require the use of a Gaussian kernel or a LoG kernel. These kernels are generated by sampling the corresponding analytical function, which in each case depends on the standard deviation σ of the Gaussian function. The result is an array of size n by n.

4.1.1 Gaussian kernel

The Gaussian convolution mask is generated by sampling the 2-D Gaussian function (centered at $(0, 0)$)

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.1}$$

where σ is the standard deviation (sometimes σ is called the *scale space constant*).

The size of the kernel n and the standard deviation of the exponential function σ are both input parameters, but these are not strictly independent of each other. The value of n needs to be large enough to ensure that no information is lost when creating the kernel G. To ensure this, n is taken equal to the first odd integer greater than 6σ. Larger values of n do not add more significant samples of G, and increase the number of operations in the convolution.

Figure 9 shows a Gaussian kernel, generated with $\sigma = 4$ and $n = 25$ (first odd integer greater than $6\sigma = 24$). C8 C8

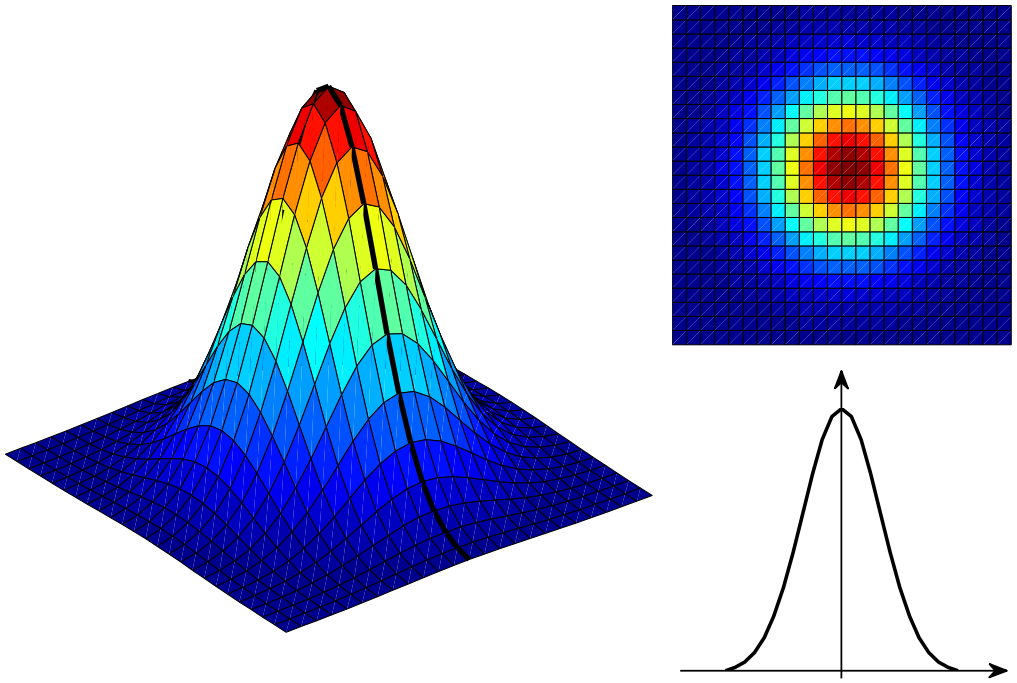


Figure 9: Gaussian kernel, $\sigma = 4$, $n = 25$. It is easy to see that the selected value of n is large enough to have a good approximation of the Gaussian function in the kernel.

4.1.2 LoG kernel

The Laplacian of Gaussian $\nabla^2 G(x, y)$ can be obtained analyically first and then a discrete mask can be computed by sampling the analytical kernel. Using this kernel for edge detection involves only one convolution with the input image (unlike the Gaussian kernel case, in which two convolutions have to be performed, one with the kernel and another with the Laplacian operator), but the kernel support must be greater in order to obtain the same precision.

$$LoG \triangleq \nabla^2 G(x, y) = \frac{\partial^2}{\partial^2 x} G(x, y) + \frac{\partial^2}{\partial^2 y} G(x, y) \tag{4.2}$$

We first compute

$$\frac{\partial}{\partial x}G(x,y) = -\frac{1}{\sqrt{2\pi\sigma^2}}\frac{x}{\sigma^2}e^{-(x^2+y^2)/2\sigma^2}, \tag{4.3}$$

$$\frac{\partial}{\partial y}G(x,y) = -\frac{1}{\sqrt{2\pi\sigma^2}}\frac{y}{\sigma^2}e^{-(x^2+y^2)/2\sigma^2}, \tag{4.4}$$

$$\frac{\partial^2}{\partial^2 x}G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}}\frac{x^2-\sigma^2}{\sigma^4}e^{-(x^2+y^2)/2\sigma^2}, \tag{4.5}$$

$$\frac{\partial^2}{\partial^2 y}G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}}\frac{y^2-\sigma^2}{\sigma^4}e^{-(x^2+y^2)/2\sigma^2}. \tag{4.6}$$

Therefore we obtain

$$LoG(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}}\frac{x^2+y^2-2\sigma^2}{\sigma^4}e^{-(x^2+y^2)/2\sigma^2}. \tag{4.7}$$

Now the LoG kernel is generated by sampling the function defined in equation 4.7. Figure 10 shows a LoG kernel, generated using the values $\sigma = 4$ and $n = 31$. C9

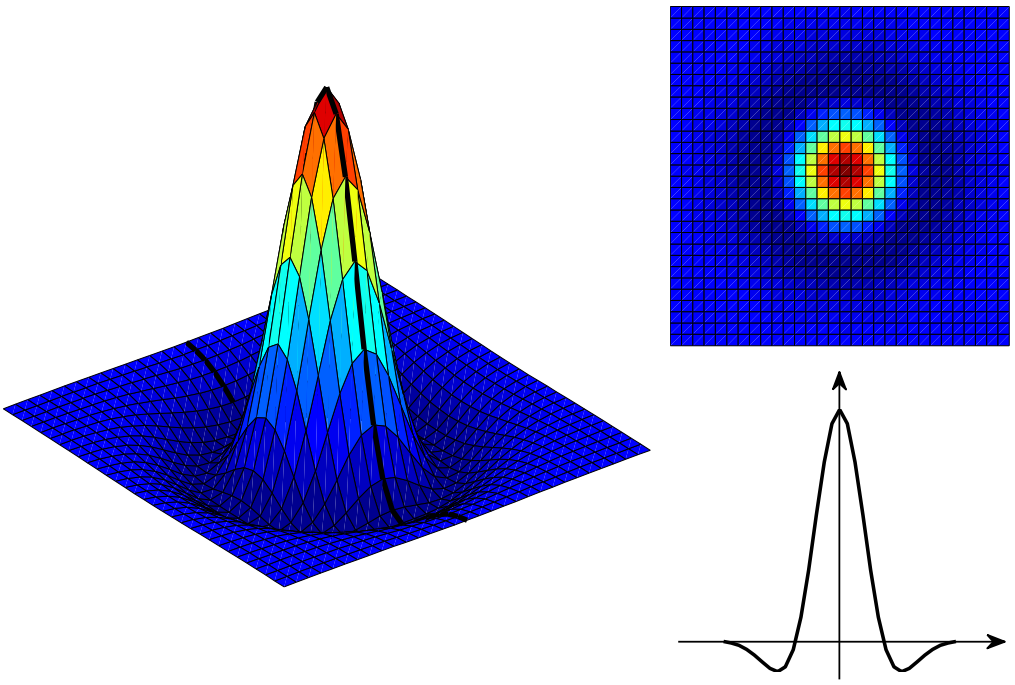


Figure 10: Laplacian of a Gaussian kernel, $\sigma = 4$, $n = 31$. The selected value of n is sufficient to have a good approximation of the LoG function in the kernel, but it is greater than in the case of Gaussian kernel.

4.1.3 Gaussian and LoG functions comparison

As mentioned before, the size n of the kernel and the standard deviation σ of the exponential function are not independent. For the same σ value , the function LoG has wider support than the Gaussian

function (i.e. LoG function has a slower decay), so a greater value of n is needed to correcly sample the LoG kernel.

Figure 11 shows both functions generated with the same value of σ. It is clear that we must use a larger kernel size in the LoG function case (approximately 18% larger). For example, using $\sigma = 4$, the optimum value for n in the case of the Gaussian function is the first odd integer greater than 6σ, which is 25, and for the case of LoG function, would be 29. C10 C10

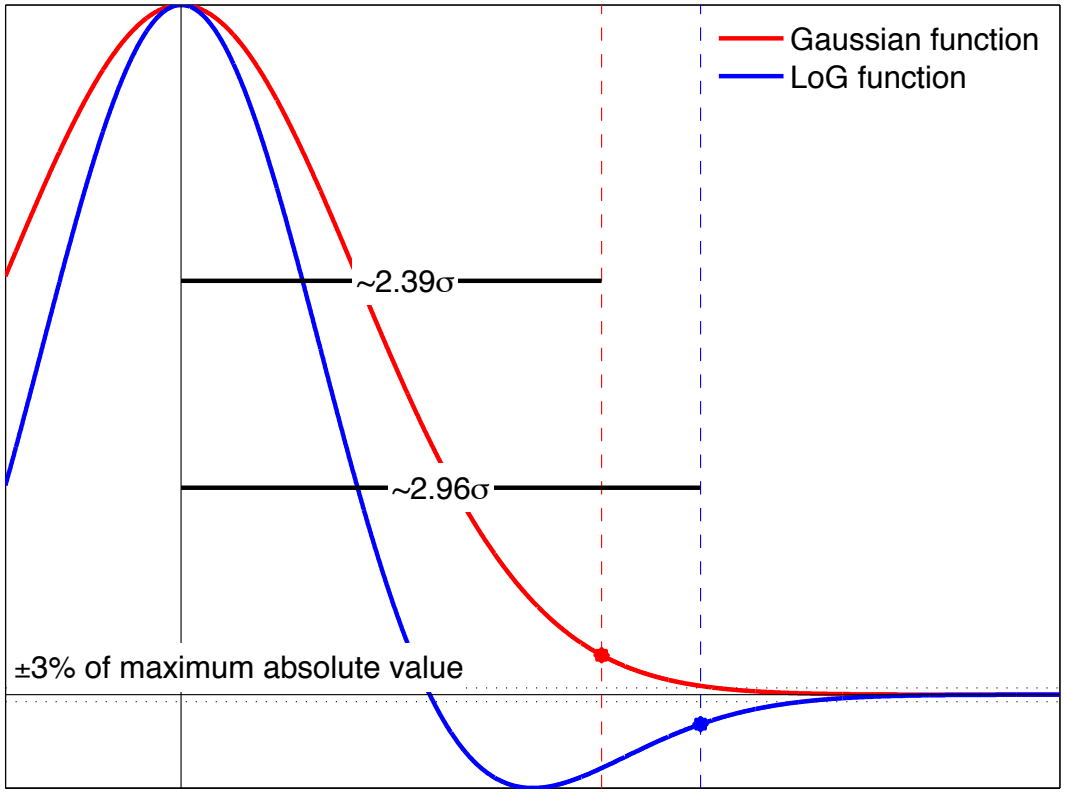


Figure 11: Comparison of the Gaussian and LoG functions.

4.2 Convolution

When making a convolution, it is necessary to define the boundary conditions used to compute it near the edges of the image, and to get a valid output the same size as input image. In this paper, two methods were implemented: zero-padding and boundary reflection. Zero-padding completes the borders of the image with valued pixels. Reflection completes those pixels with the corresponding symmetric pixel value relative to the edge of the image. Direct convolution is not the only way of filtering an image with a kernel. There are other methods such as FFT convolution or recursive filtering.

5 Results

The results of each algorithm are first shown for a simple image composed of a white square on a black background, see Figure 12.

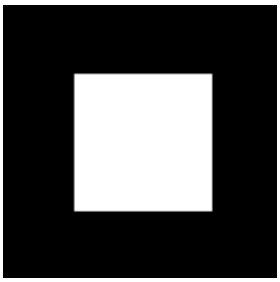
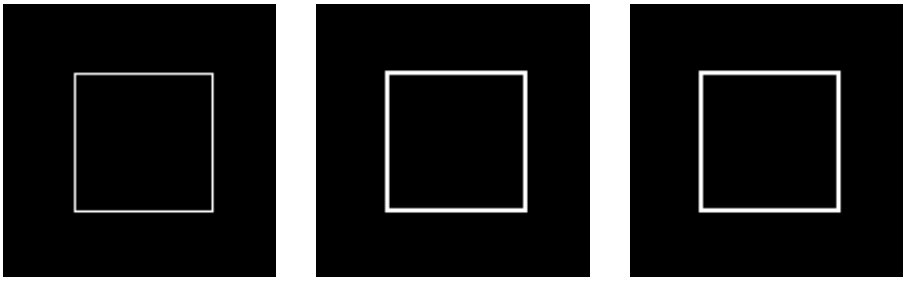


Figure 12: Test image 1: white square on black background (127×127 pixels).

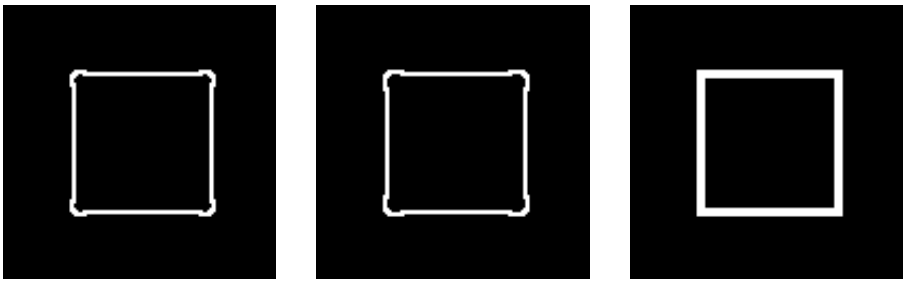
Figures 13(a) to 13(f) show the output of each algorithm (including execution times[7]).



(a) Roberts. $th = 0.1$. Exec-time : $0.02s$.

(b) Prewitt. $th = 0.1$. Exec-time : $0.02s$.

(c) Sobel. $th = 0.1$. Exec-time : $0.02s$.



(d) Marr-Hildreth (Gaussian). $\sigma = 1.5$, $n = 13$, $th_{ZC} = 0.1$. Exec-time : $0.03s$.

(e) Marr-Hildreth (LoG). $\sigma = 1.5$, $n = 17$, $th_{ZC} = 0.1$. Exec-time : $0.05s$.

(f) Haralick. $\rho = 0.4$. Exec-time : $0.04s$.

Figure 13: Results of the algorithms using the Figure 12 as input image. The first three ((a), (b) and (c)) correspond to the first derivative methods (Roberts, Prewitt and Sobel), then the following two ((d) and (e)) are from the Marr-Hildreth algorithm using Gaussian and LoG kernels, and the last one ((f)) corresponds to the Haralick algorithm. Note the rounded corners on the Marr-Hildreth results, due to Gaussian blur.

In this simple case, the first derivative edge detection algorithms run better and faster. All algorithms show consistent results. Thicker edges appear in the results of Haralick, due to smooth image model that is imposed in this algorithm.

[7]Running on Intel Core i3 CPU (2.53GHz), 3 Gb RAM, standard laptop. Note that the execution time of the

Figure 14: Test image 2: windmill (1000×563 pixels).

Algorithm	**Execution time (s)**
Roberts, Prewitt and Sobel	0.550 s
Marr-Hildreth (Gaussian)	1.050 s
Marr-Hildreth (LoG)	1.440 s
Haralick	0.930 s

Table 5.1: Execution time of the algorithms (including I/O) using Figure 14 as input image (1000×563 pixels, 3 channels).

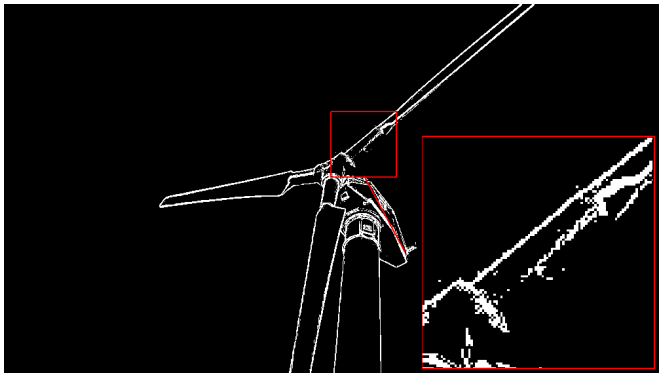
Now the performance of each algorithm is analyzed using a natural image (*Windmill*) shown in Figure 14. The results of the first derivative edge detection algorithms are shown Figures 15(b) to 15(d). Figures 15(e) and 15(f) show the results obtained using the Marr-Hildreth algorithm (both Gaussian and LoG kernels), and Figure 15(g) shows the results obtained using the Haralick algorithm. Table 5.1 summarizes the execution times for each of the algorithms.

In this example the difference in performance between the first derivative and the second derivative algorithms is more meaningful. For each image in Figure 15, we show an area of interest, enlarging it to have a better view of the details. Note that none of the first derivative methods (even with a loose threshold) detect the lower edge of the blade (which is shaded). Neither the Haralick algorithm is able to detect it. However, the Marr-Hildreth algorithm detects it, using either a Gaussian or a LoG kernel.

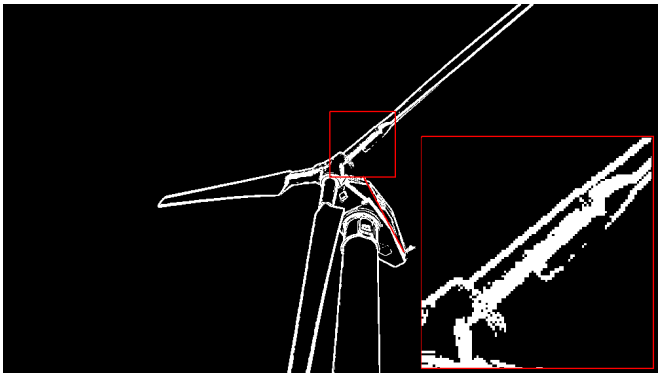
Another observation is that edges detected by Haralick's algorithm appear to be thicker than those detected with other ones. This may be due the regularity that Haralick assumes.

algorithms Roberts, Prewitt and Sobel are the same, because they run simultaneously.

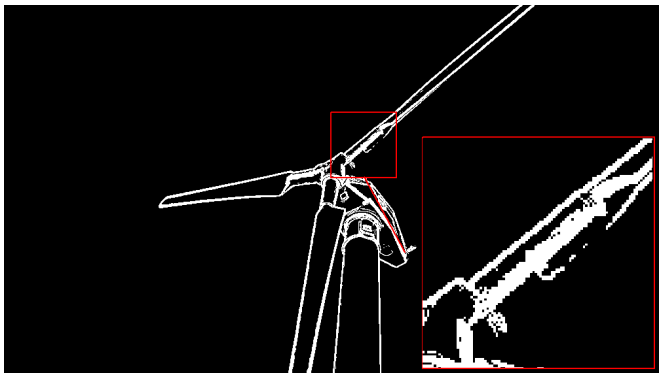
(a) Original.



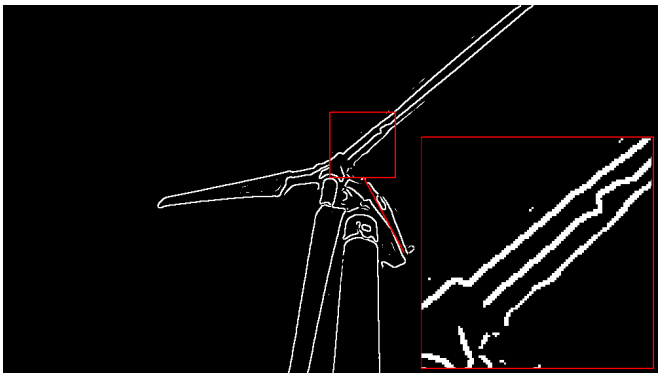
(b) Roberts. $th = 0.1$. Exec-time : $0.55s$.



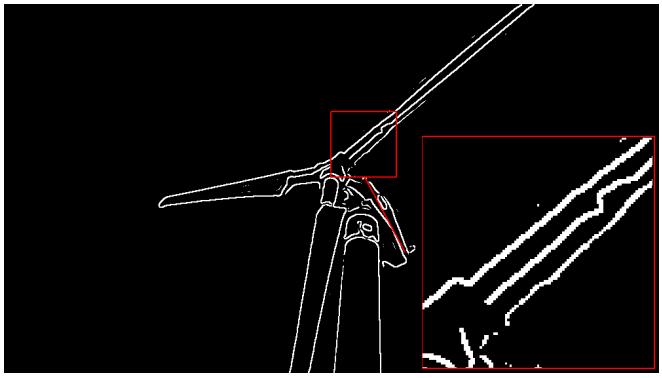
(c) Prewitt. $th = 0.1$. Exec-time : $0.55s$.



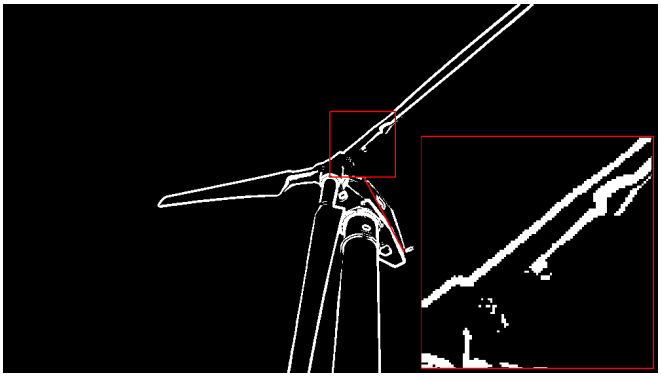
(d) Sobel. $th = 0.1$. Exec-time : $0.55s$.



(e) Marr-Hildreth (Gaussian). $\sigma = 3$, $n = 25$, $th_{ZC} = 0.07$. Exec-time : $1.05s$.



(f) Marr-Hildreth (LoG). $\sigma = 3$, $n = 29$, $th_{ZC} = 0.13$. Exec-time : $1.44s$.



(g) Haralick. $\rho = 0.4$. Exec-time : $0.93s$.

Figure 15: Results of the first derivative, Marr-Hildreth and Haralick's algorithms on the *Windmill* image.

5.1 Further examples

Further examples obtained with the implemented algorithms are shown in Figures 16 and 17. More examples can be found in the online demo[8].

(a) Original.

(b) Roberts. $th = 0.1$.

(c) Prewitt. $th = 0.1$.

(d) Sobel. $th = 0.1$.

(e) Marr-Hildreth (Gaussian). $\sigma = 2$, $n = 21$, $th_{ZC} = 0.15$.

(f) Marr-Hildreth (LoG). $\sigma = 2$, $n = 25$, $th_{ZC} = 0.15$.

(g) Haralick. $\rho = 0.35$.

Figure 16: Example: Lena (512×512 pixels).

[8] http://dev.ipol.im/~haldos/ipol_demo/xxx_edges/

Some observations:

- The first derivative algorithms, although they work properly and run fast, have some problems like discontinuity of the edges. They are also very sensitive to noise, which make them detect spurious edges (More sophisticated methods help to avoid and improve this, e.g. Canny edge detector [2]).

- The Marr-Hildreth algorithm (in its two versions) achieved interesting results in the edge detail. This algorithm provides a first attempt to obtain regular edges, because of the filtering with a Gaussian kernel. This behavior can be seen in Lena's hair (Figure 16), or inside the oranges (example 17).

- As mentioned earlier, Haralick's algorithm is the first to assume greater regularity of the image in the neighborhood of a pixel (third order). This causes thicker edges, as shown in both examples. But, note that some edges are only detected with this algorithm, e.g. the lower edges of the oranges, which are in the shadow, in Figure 17. These edges are quite smooth, so that the Haralick model is well suited to them, and they do not represent an abrupt change in the intensity to be detected by other methods.

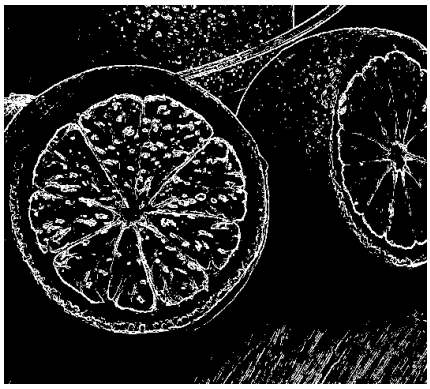
5.2 Effect of the scale parameter

C11 In this section we will discuss the effect of the parameters for each of the reviewed algorithms. As a general rule, they all take one threshold parameter which acts as a scale parameter. As Figure 18 shows for the *Roberts* algorithm, the bigger that threshold is, the coarser is the edge map obtained.

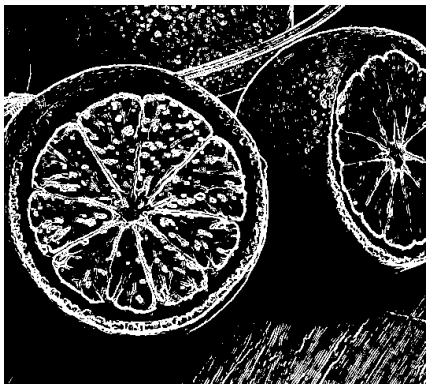
In Figure 19 we show a summary of the threshold's behavior for this algorithm, in the whole range of possible values $[0, 1]$. For each image, this information is summarized in a saliency map which serves to asses the quality of the hierarchical set of edges obtained. The saliency map is constructed by sampling the scale range (using steps of 0.05) and computing the edge map corresponding for each scale. Then each edge is labeled with its scale of disappearance α^-, which is the maximum scale at which an edge is present in the output edge map. This scale is drawn with a *hot* colormap (black, red, yellow, white). Intuitively, this means that 'hotter' edges last longer across scales and thus are more meaningful. These results show two important properties. First, the obtained stack of edges are causal, i.e. as the scale parameter increases the edge map gets coarser. This can be verified by observing the values of α^- assigned to each contour. If a contour has a high α^- value it means that it will survive for a long time in the merging process. On the other hand, small values mean that it will be removed at early stages. As the figure shows, the highest values of α^- are assigned to the most coarser segmentation. And second, the stack of edges is geometrically accurate, i.e. the borders present in each scale are a strict subset of the previous (finer) scale. If they were not included they will be seen as new contours in the saliency map. These results empirically show that the obtained hierarchy holds the strong causality property introduced by Morel and Solimini [11].

In Figures 20 and 21 we compare the three first derivative algorithms by means of their saliency maps. As the results show, they all present a similiar behavior, however the are small differences in the results of Roberts vs the other two algorithms. For example, edges surrounding the segments of the orange are better defined, because they have more relative weight than other spurious borders. This is explained by the smoothing introduced by Prewitt and Sobel masks explained in Sections 2.3 and 2.3, which removes theses spurious edges and thus improves the meaningfulness of the stable ones.

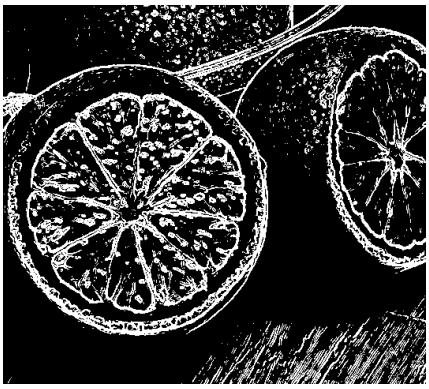
(a) Original.



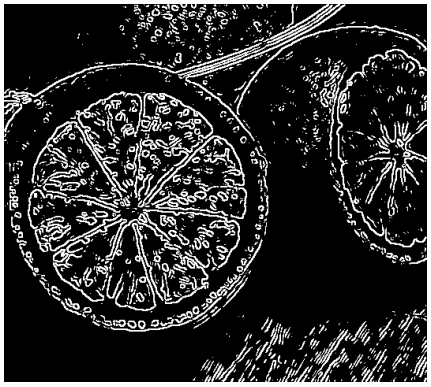
(b) Roberts. $th = 0.1$



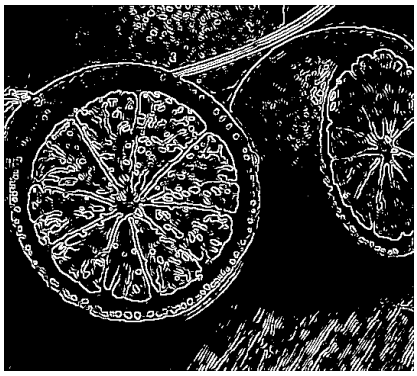
(c) Prewitt. $th = 0.1$



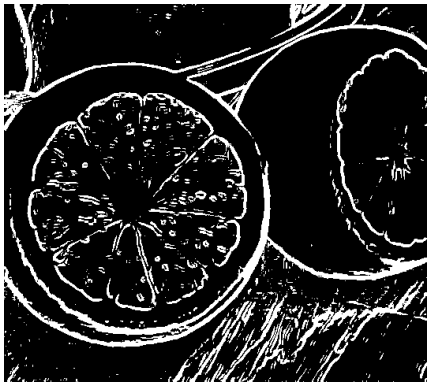
(d) Sobel. $th = 0.1$



(e) Marr-Hildreth (Gaussian). $\sigma = 2$, $n = 21$, $th_{ZC} = 0.15$.



(f) Marr-Hildreth (LoG). $\sigma = 2$, $n = 25$, $th_{ZC} = 0.15$.



(g) Haralick. $\rho = 0.4$.

Figure 17: Example: Oranges (536×480 pixels).

In Figures 22 and 23 we compare the second derivative algorithms by means of their saliency maps. In the first place, we obtain very similar hierarchies for both Marr-Hildreth implementations, which validates the affirmation that the difference is merely about computational effort if both are correctly implemented. However, the LoG implementation seems to give slightly better results, this is probably because it provides a more accurate discretization of the differential operators than the Gaussian approach. See for instance the clothing sticks in Figure 23, where the boundaries in the LoG case seem better defined.

5.3 Effect of the smoothing parameter

C12 As we stated in previous section, all the reviewed algorithms share a scale parameter (threshold), C12

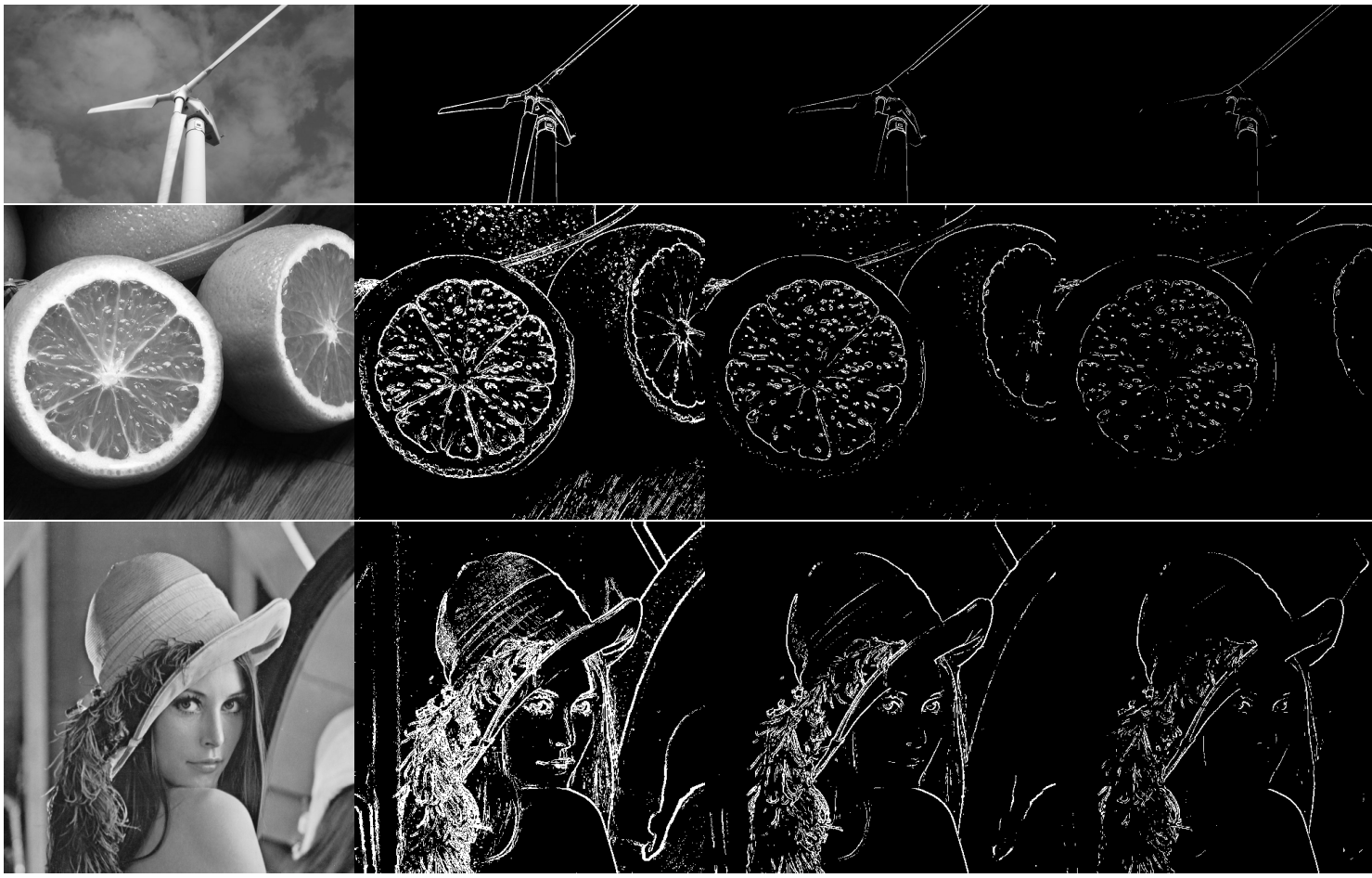


Figure 18: Results of the *Roberts* algorithm our sample images. For each example we show the original image on the left and the results corresponding to parameter values of 0.1, 0.2 and 0.3 respectively.

however, the Marr-Hildreth Algorithm also has a smoothing step which adds an additional parameter σ, which controls the amount of regularization. In Figure 24 we show the results using different σ values. As sigma increases more spurious edges are removed and the boundaries of the objects look smoother. However, if we keep increasing it we start removing meaningful edges (see the cloth band on the hat of Lena).

5.4 Discussion

C13 In this section we will discuss the obtained results and contrast them with the claims made by the original authors. In the case of the first derivative algorithms, they are very simple and the claims made by the original authors are straight forward to verify. The Roberts algorithm was presented in his Phd Thesis ([**?**], Chapter IV) as a step of a more complex 3D reconstruction system. For this reason, only qualitative results are shown on few images. The effects of the parameter or the discretization is not validated. The only observation made in that work is the presence of spurious edges due to noise (pg. 31), which are removed by a line fitting postprocessing. With respect to Prewitt and Sobel Operators,

In Haralick's work, there is an interesting quantitative validation over a synthetic grid. On that evaluation, sensitivity to noise as well as comparative performance with Prewitt operators. As in this work we didn't benchmark quantitaive

The obtained results validate the original claims made by Marr [**?**], were he made the following observations:

- "A major difficulty with natural images is that changes can and do occur over a wide range of scales (Marr 1976a, b). No single filter can be optimal simultaneously at all scales, so it follows that one should seek a way of dealing separately with the changes occurring at different scales. ".
- "The reason is that strict bandlimiting gives rise to sidelobes in the spatial spatial filter, and the consequence of these is that, in the zero-crossing image, strong intensity changes give rise to echoes as well as to the directly corresponding zero-crossings..."
- "Physical edges will produce roughly coincident zero-crossings in channels of nearby sizes. The spatial coincidence assumption asserts that the converse of this is true, that is the coincidence of zero crossings is sufficient evidence for the existence of a real physical edge. If the zero-crossings in one channel are not consistent with those in the others, they are probably caused by different physical phenomena, so descriptions need to be formed from both sources and kept somewhat separate."

The first claim is validated in our experiments varying the σ parameter, where we observed that no single value will allow us to obtain all the meaningful edges. The same reasoning applies to the zero crossing threshold t_{ZC} .

The second observation is evidenced by the halos seen in the blades of the windmill, see inside left blade and left of the top blade in Figure 15(e) . [9]

The third claim is evidenced by the structure of the saliency maps of the threshold parameter (Figures 22 and 23) and the smoothing parameter (Figure 25). In both maps we observe that meaningful edges are coincident across scales and the spurious ones are not.

[9] Please look at these issues in the electronic version or the online demo, as they are not noticeable in the printed version.

Figure 19: Results of the *Roberts* algorithm on our sample images. For each example we show the original image on the left and a saliency map on the right. The value of each contour corresponds to the scale of disappearance α^- of each region.

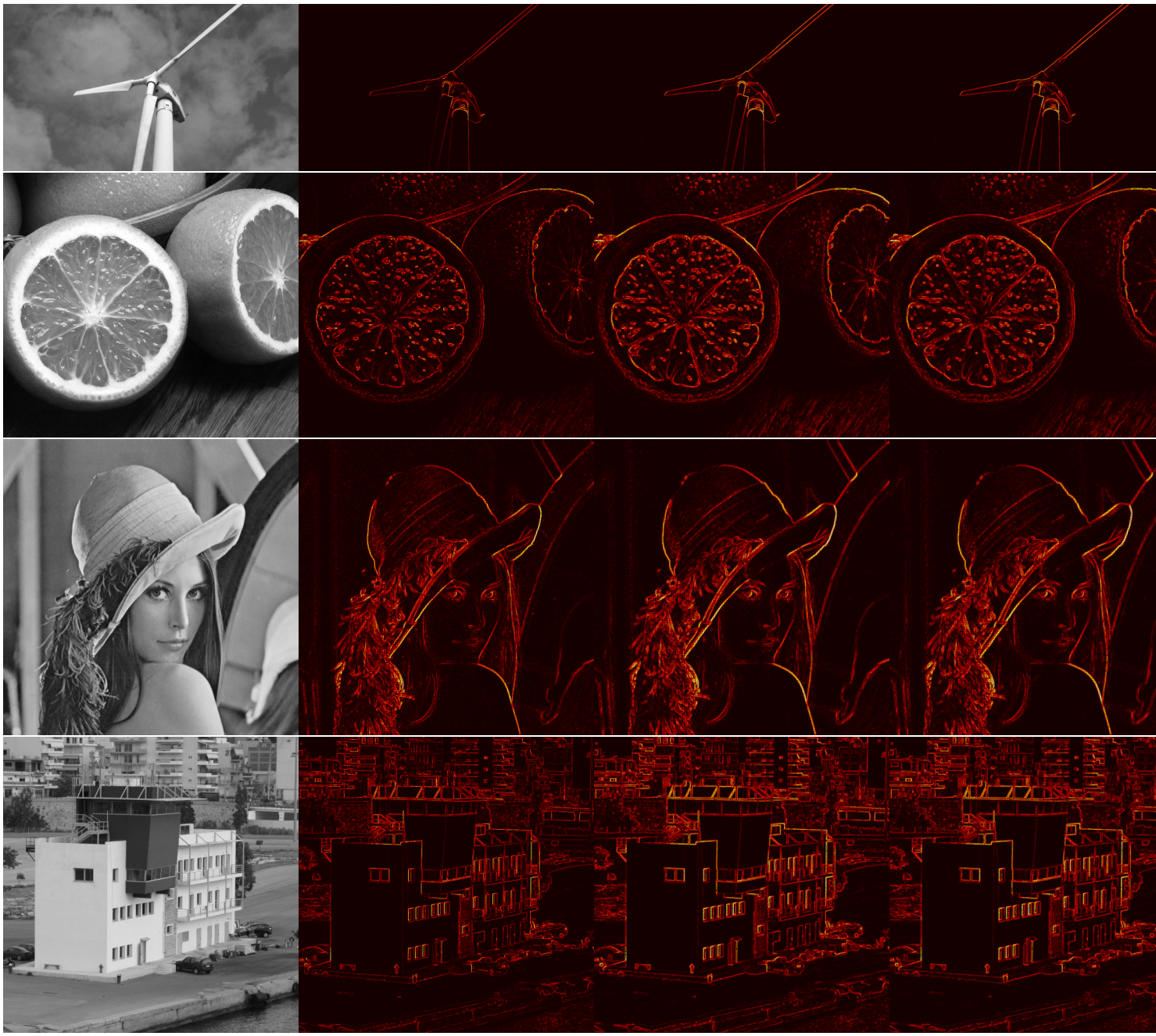


Figure 20: Results of the first derivative algorithms our sample images. For each example we show the original image on the left and the saliency map of each algorithm (Roberts, Sobel, Prewitt) respectively. The value of each contour corresponds to the scale of disappearance α^- of each region.

Figure 21: Results of the first derivative algorithms our sample images. For each example we show the original image on the left and a saliency map of each algorithm (Roberts, Sobel, Prewitt) respectively. The value of each contour corresponds to the scale of disappearance α^- of each region.

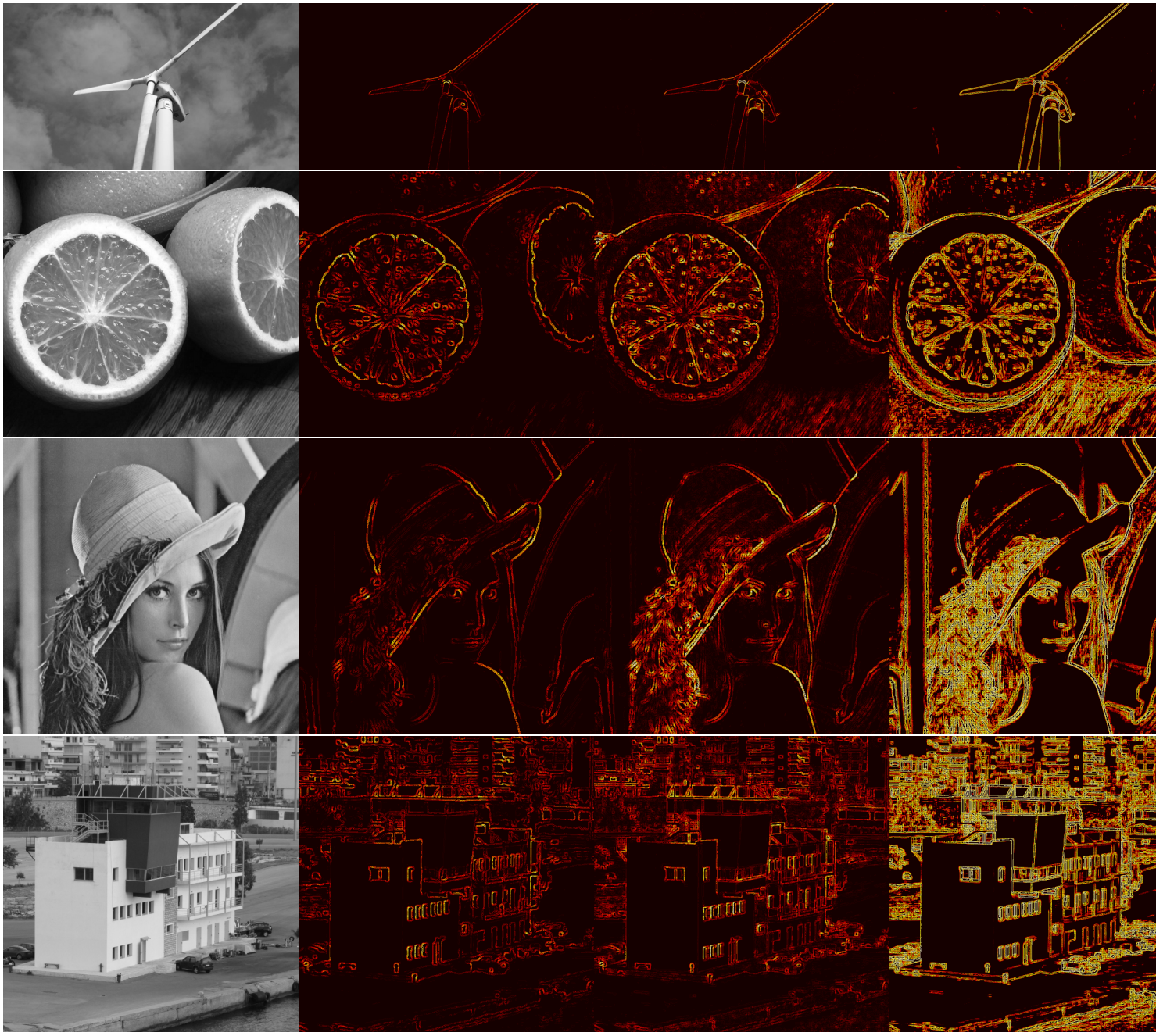


Figure 22: Results of the second derivative algorithms our sample images. For each example we show the original image on the left and a saliency map of each algorithm (Marr-Hildreth-Gaussian, Marr-Hildreth-Log, Haralick) respectively. The value of each contour corresponds to the scale of disappearance α^- of each region.

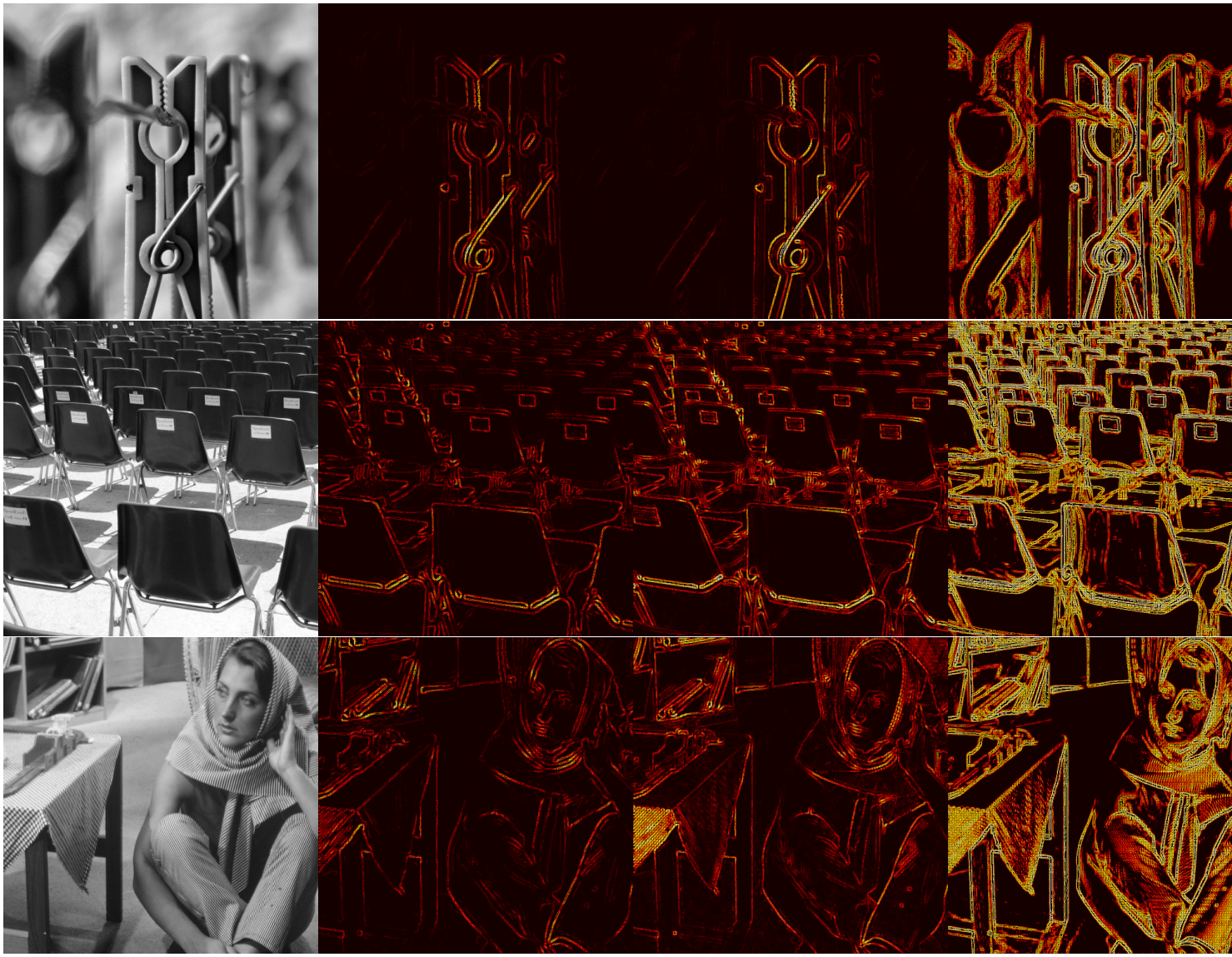


Figure 23: Results of the second derivative algorithms our sample images. For each example we show the original image on the left and the saliency map of each algorithm (Marr-Hildreth-Gaussian, Marr-Hildreth-Log, Haralick) respectively. The value of each contour corresponds to the scale of disappearance α^- of each region.

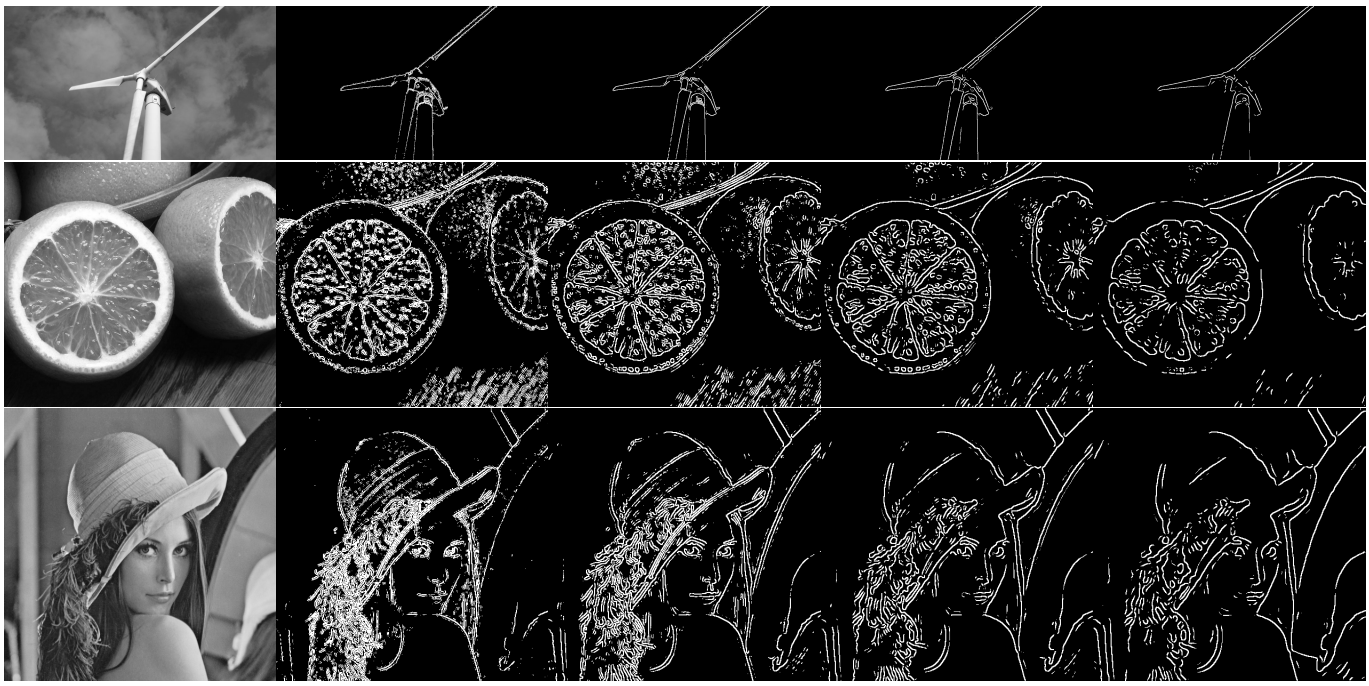


Figure 24: Effect of the σ parameter for the *Marr-Hildreth* (Gaussian) algorithm on our sample images. For each example we show the original image on the left and various consecutive results with different smoothing values ($\sigma = 1, 2, 3, 4$) on the right.

Figure 25: Effect of the σ parameter for the *Marr-Hildreth* (Gaussian) algorithm on our sample images. For each example we show the original image on the left and the generated saliency map on the right. σ is sampled regularly in the range $[0.5, 5]$ using a step of 0.5.

6 Conclusions

Some of the most traditional methods of edge detection in digital images were discussed and carefully implemented in this work. The implemented algorithms were tested with synthetic and real images, obtaining generally the expected results, taking into account the limitations of these methods.

The first derivative algorithms (Roberts, Prewitt and Sobel) have the advantage of having a very simple implementation. They also run extremely fast, because their main operation is a convolution with a very small kernel (2×2 or 3×3 pixels). The results obtained with these methods are quite good, considering their simplicity, but they have problems such as noise and discontinuity of the edges. Gaussian filtering would reduce noise, alleviate (but not solve) the discontinuity of the edges, and would make the comparison fairer with the second derivative methods.

The second derivative algoritms (Marr-Hildreth & Haralick) involve several more operations, since convolutions with larger kernels are performed. In real images, they have better behavior than first derivative algorithms.

Comparing the two versions of the Marr-Hildreth algorithm, the version with Gaussian kernel runs significantly faster that the LoG one. The latter, while slower (since it needs a larger kernel to achieve similar results) is more accurate, because it makes no approximation to calculate the Laplacian (it is calculated analytically before creating the kernel). It is also possible to manage the size of the kernel, which represents a scale parameter of the algorithm, being able to obtain a highly detailed edge image using small kernels, or just more noticeable edges using larger kernels.

Haralick's algorithm, although it shares the Marr-Hildreth idea of finding zero crossings of the second derivative, has some quite different ideas. It is the only one of these algorithms which works with an approximation of the intensity of the image in the neighborhood of a pixel, using a bicubic polynomial function. This supposes a certain regularity in the image, which is not always true, and sometimes leads to detect edges where none of them exist due to ringing effects, and get some thicker edges too.

All algorithms have their advantages and disadvantages. Choosing one or the other may depend on requirements of the application. Furthermore, we must remark that we focused on low level edge detectors, and thus none of the implemented methods ensure edge connectivity. This algorithms only serve as the input for a further edge integration step. There are many proposals in the literature aiming to integrate edgels into meaningful object boundaries. Examples of these approaches are the Canny [2] edge detector or the meanignful level lines algorithm [3].

With respect to the parameters, these methods require the manual determination of the parameters, and it is very hard to find a single threshold value suitable for a broad category of examples. In this sense, *A contrario* methods provide a better approach for setting parameters. For this reason, most modern approaches use a multiscale approach, were edges are detected at many different scales and the optimum scale is chosen a posteriori, as proposed in [12] or [1]. This idea can be even extended to pick edges at different scales depending on the region of the image (see [**?**] for a similar idea applied to regions).

6.1 Future work

The goal of this work was to provide reference implementation for some of the most classical edge detectors. Some of them have been published for many years and are quite simple, however, it is surprisingly hard to find trustable implementations. We focused on low-level edge detection, which means that we do not consider post processing steps to integrate the detected edgels into meaningul lines. Extensive qualitative validation has been performed, showing the effects of the parameters and the scale-space properties of the algorithms in order to faithfully compare them. We

consider this work as the first step towards a complete benchmark of edge detectors, which has to be completed with a systemic quantitative analysis. A suitable framework for this quantitative analysis is discussed and applied in [**?**], but this is clearly outside the scope of this work. In particular, quantitative challenging the claims of the reviewed works will be a great contribution for the area, which is very weak with respect to experimental validation. From the original works reviewed, only Haralick presents a quantitative analysis but only over one synthetic images. Even at the moment of writing (2014), there are only a dozen edge detection and segmentation works presenting a serious quantitative analysis over a standard database of images.

A Haralick convolution masks

(a) **mask$_1$**.

425	275	225	275	425
275	125	75	125	275
225	75	25	75	225
275	125	75	125	275
425	275	225	275	425

(b) **mask$_2$**.

-2260	-620	0	620	2260
-1660	-320	0	320	1660
-1460	-220	0	220	1460
-1660	-320	0	320	1660
-2260	-620	0	620	2260

(c) **mask$_3$**.

2260	1660	1460	1660	2260
620	320	220	320	620
0	0	0	0	0
-620	-320	-220	-320	-620
-2260	-1660	-1460	-1660	-2260

(d) **mask$_4$**.

1130	620	450	620	1130
830	320	150	320	830
730	220	50	220	730
830	320	150	320	830
1130	620	450	620	1130

(e) **mask$_5$**.

-400	-200	0	200	400
-200	-100	0	100	200
0	0	0	0	0
200	100	0	-100	-200
400	200	0	-200	-400

(f) **mask$_6$**.

1130	830	730	830	1130
620	320	220	320	620
450	150	50	150	450
620	320	220	320	620
1130	830	730	830	1130

(g) **mask$_7$**.

-8260	-2180	0	2180	8260
-6220	-1160	0	1160	6220
-5540	-820	0	820	5540
-6220	-1160	0	1160	6220
-8260	-2180	0	2180	8260

(h) **mask$_8$**.

5640	3600	2920	3600	5640
1800	780	440	780	1800
0	0	0	0	0
-1800	-780	-440	-780	-1800
-5640	-3600	-2920	-3600	-5640

(i) **mask$_9$**.

-5640	-1800	0	1800	5640
-3600	-780	0	780	3600
-2920	-440	0	440	2920
-3600	-780	0	780	3600
-5640	-1800	0	1800	5640

(j) **mask$_{10}$**.

8260	6220	5540	6220	8260
2180	1160	820	1160	2180
0	0	0	0	0
-2180	-1160	-820	-1160	-2180
-8260	-6220	-5540	-6220	-8260

Table A.1: 5x5 masks used to compute the coefficients of the bicubic fit.

B Implementation

In this section a detailed explanation of the source code of the different implemented algorithms is presented.

To the best of our knowledge, the only freely available codes implementing this algorithms are the Matlab implementation of edge detection algorithms using 2D masks (`edge` command[10]) and a Haralick algorithm implementation in Matlab published in Matlab Central[11]. There is also an implementation similar to the Marr-Hildreth algorithm, as well available in Matlab Central[12]. All these links were last checked in July 2012.

This code documentation was generated using a perl script to extract some parts of the code and comments into latex files. This code documentation tool is available as a GIT project[13].

Also a Doxygen generated documentation of the functions used is available here[14].

Required external libraries:

- `iio`[15].
- `libpng`[16].

B.1 Roberts, Prewitt and Sobel

The source code for this section can be found in the file `test_fded.c`.
First derivative edge detectors (Roberts, Prewitt and Sobel), main C file.

Parameters:

- `input_image` - Input image.
- `threshold` - Threshold for gradient image ($0 \leq th \leq 1$).
- `padding_method` - Padding method flag (in convolution): 0 means zero-padding, 1 means image boundary reflection.

Note: Output images are saved with the filenames "`roberts.png`", "`prewitt.png`" and "`sobel.png`".

Includes:

```
#include "iio.c"
#include "2dconvolution.c"
#include <time.h>
```

Macros:

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define THRESHOLD(x, th) (((x) > (th)) ? (255) : (0))
```

[10] http://www.mathworks.com/help/toolbox/images/ref/edge.html
[11] http://www.mathworks.com/matlabcentral/fileexchange/35329-simple-edge-detection-using-classical-harali
[12] http://www.mathworks.com/matlabcentral/fileexchange/11572-sdgd-edge-detection-filter
[13] https://github.com/juan-cardelino/source_comment_extractor
[14] http://iie.fing.edu.uy/~haldos/ipol/red_v0.1
[15] http://dev.ipol.im/git/coco/iio.git
[16] http://www.libpng.org/

Main function

```
int main(int argc, char *argv[]) {
```

Load input image (using *iio*):

```
        int w, h, pixeldim;
        float *im_orig = iio_read_image_float_vec(argv[1], &w, &h, &pixeldim);
```

Grayscale conversion (if necessary): explained in **??**.

Define the normalized Roberts, Prewitt and Sobel operators:
(We use 3×3 operators)

- Roberts:

$$R_1 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$R_2 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- Prewitt:

$$P_1 = \begin{bmatrix} \frac{-1}{6} & \frac{-1}{6} & \frac{-1}{6} \\ 0 & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \end{bmatrix}$$

$$P_2 = \begin{bmatrix} \frac{-1}{6} & 0 & \frac{1}{6} \\ \frac{-1}{6} & 0 & \frac{1}{6} \\ \frac{-1}{6} & 0 & \frac{1}{6} \end{bmatrix}$$

- Sobel:

$$S_1 = \begin{bmatrix} \frac{-1}{8} & \frac{-1}{4} & \frac{-1}{8} \\ 0 & 0 & 0 \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \end{bmatrix}$$

$$S_2 = \begin{bmatrix} \frac{-1}{8} & 0 & \frac{1}{8} \\ \frac{-1}{4} & 0 & \frac{1}{4} \\ \frac{-1}{8} & 0 & \frac{1}{8} \end{bmatrix}$$

```
        double roberts_1[9] = {-1, 0, 0, 0, 1, 0, 0, 0, 0};      // ROBERTS
        double roberts_2[9] = { 0,-1, 0, 1, 0, 0, 0, 0, 0};      // OPERATORS
        //---------------------------------------------------------------------
        double prewitt_1[9] = {-1,-1,-1, 0, 0, 0, 1, 1, 1};      // PREWITT
        double prewitt_2[9] = {-1, 0, 1,-1, 0, 1,-1, 0, 1};      // OPERATORS
        //---------------------------------------------------------------------
        double sobel_1[9] = {-1,-2,-1, 0, 0, 0, 1, 2, 1};        // SOBEL
        double sobel_2[9] = {-1, 0, 1,-2, 0, 2,-1, 0, 1};        // OPERATORS
        //---------------------------------------------------------------------
        for (z=0;z<9;z++) {                                      // NORMALIZATION
            roberts_1[z] /= sqrt(2);
            roberts_2[z] /= sqrt(2);
            prewitt_1[z] /= sqrt(6);
            prewitt_2[z] /= sqrt(6);
            sobel_1[z] /= sqrt(12);
            sobel_2[z] /= sqrt(12);
        }
```

The input image is convolved with the defined operatos, using the `conv2d` function in `2dconvolution.c`:

```
int padding_method = atoi(argv[3]);
double *im_r1 = conv2d(im, w, h, roberts_1, 3, padding_method);
double *im_r2 = conv2d(im, w, h, roberts_2, 3, padding_method);
double *im_p1 = conv2d(im, w, h, prewitt_1, 3, padding_method);
double *im_p2 = conv2d(im, w, h, prewitt_2, 3, padding_method);
double *im_s1 = conv2d(im, w, h, sobel_1, 3, padding_method);
double *im_s2 = conv2d(im, w, h, sobel_2, 3, padding_method);
```

Allocate memory for final images:

```
float *im_roberts = malloc(w*h*sizeof(float));
float *im_prewitt = malloc(w*h*sizeof(float));
float *im_sobel = malloc(w*h*sizeof(float));
```

For each method, two images are obtained (one for each operator). Then the gradient magnitude image is constructed using $M = \sqrt{g_x^2 + g_y^2}$.
Also the absolute maximum value of the constructed images is computed, for each method.

```
int i,j, fila, col;
double max_r = 0;
double max_p = 0;
double max_s = 0;
int imax = w*h;
for (i=0;i<imax;i++){
    fila = (int)(i/w);
    col = i - w*fila + 1;
    fila += 1;
    j = col + (w+2)*fila;
    // Max in each case
    im_roberts[i] = sqrt(im_r1[j]*im_r1[j] + im_r2[j]*im_r2[j]);
    im_prewitt[i] = sqrt(im_p1[j]*im_p1[j] + im_p2[j]*im_p2[j]);
    im_sobel[i] = sqrt(im_s1[j]*im_s1[j] + im_s2[j]*im_s2[j]);
    // Absolute max
    max_r = MAX(max_r,im_roberts[i]);
    max_p = MAX(max_p,im_prewitt[i]);
    max_s = MAX(max_s,im_sobel[i]);
}
```

Thresholded images of each method are created, using the THRESHOLD macro:

```
float th = atof(argv[2]);
for (i=0;i<imax;i++){
    im_roberts[i] = THRESHOLD(im_roberts[i],th*max_r);
    im_prewitt[i] = THRESHOLD(im_prewitt[i],th*max_p);
    im_sobel[i] = THRESHOLD(im_sobel[i],th*max_s);
}
```

Save output image (using *iio*):

```
iio_save_image_float_vec("roberts.png", im_roberts, w, h, 1);
iio_save_image_float_vec("prewitt.png", im_prewitt, w, h, 1);
iio_save_image_float_vec("sobel.png", im_sobel, w, h, 1);
```

B.2 Marr-Hildreth

The source code for this section can be found in the file `test_mh.c`.
Marr-Hildreth edge detector, main C file.

Parameters:

- `input_image` - Input image.

- `sigma` - Standard deviation σ of the Gaussian kernel.
- `n` - Size n of the Gaussian kernel ($n \times n$).
- `tzc` - Threshold in the Zero-Crossing calculation ($0 \leq t_{zc} \leq 1$).
- `padding_method` - Padding method flag (in convolution): 0 means zero-padding, 1 means image boundary reflection.
- `output_image` - Output image (edges).

Includes:

```
#include "iio.c"
#include "gaussian_kernel.c"
#include "2dconvolution.c"
#include <time.h>
```

Main function

```
int main(int argc, char *argv[]) {
```

Load input image (using *iio*):

```
        int w, h, pixeldim;
        float *im_orig = iio_read_image_float_vec(argv[1], &w, &h, &pixeldim);
```

Grayscale conversion (if necessary):
First is allocated the memory for the grayscale image `im`, with the corresponding correct allocation check. Then the number of channels of the image is checked: if `pixeldim`=3, RGB image is assumed and conversion is needed, else, single channel image (grayscale) is assumed and no conversion is required.
The computation of the gray intensity from RGB levels is:

$$I = \frac{6968 \times (\text{float})R + 23434 \times (\text{float})G + 2366 \times (\text{float})B}{32768}$$

To perform this operation, a cast is made to float on the values R, G and B of the original image. This can be slow, but ensures the correct image conversion.
These coefficients also ensure there is no saturation in the calculation, since $\frac{6968 \times 255 + 23434 \times 255 + 2366 \times 255}{32768} = \frac{8355840}{32768} = 255$.

```
        double *im = malloc(w*h*sizeof(double));
        if (im == NULL){
            fprintf(stderr, "Out of memory...\n");
            exit(EXIT_FAILURE);
        }
        int z;
        int zmax = w*h;
        if (pixeldim==3){
            for(z=0;z<zmax;z++){
                im[z] =  (double)(6968*im_orig[3*z] + 23434*im_orig[3*z + 1]
                                                    + 2366*im_orig[3*z + 2])/32768;
            }
            fprintf(stderr, "images converted to grayscale\n");
        } else {
            for(z=0;z<zmax;z++){
                im[z] = (double)im_orig[z];
            }
            fprintf(stderr, "images are already in grayscale\n");
        }
```

Generate Gaussian kernel using the `gaussian_kernel` function in `gaussian_kernel.c`:

```
double *kernel = gaussian_kernel(n,sigma);
```

Smooth input image with the Gaussian kernel previously generated, using the `conv2d` function in `2dconvolution.c`:

```
double *im_smoothed = conv2d(im, w, h, kernel, n, padding_method);
```

Computation of the Laplacian of the smoothed image:
A 3×3 approximation of the laplacian operator is used:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Using the `conv2d` function, the `laplacian` image is obtained.

```
double operator[9] = {1, 1, 1, 1, -8, 1, 1, 1, 1};
double *laplacian = conv2d(im_smoothed, w+n-1, h+n-1, operator, 3, padding_method);
```

Now the maximum absolute value of the `laplacian` image is calculated. This value is required for thresholding in zero-crossing calculation.

```
double max_l = 0;
int p;
int pmax = (w+n+1)*(h+n+1);
for (p=0;p<pmax;p++){
    if (abs(laplacian[p])>max_l){
        max_l = abs(laplacian[p]);
    }
}
```

Zero-crossing:
The image is explored, looking in every pixel a change of sign between neighboring opposite pixels. In every pixel p the funcion `get_neighborhood` (in `2dconvolution.c`) is used to get the 9 pixels of its neighborhood:

$$\begin{bmatrix} p_{up,left} & p_{up,middle} & p_{up,right} \\ p_{middle,left} & p & p_{middle,right} \\ p_{down,left} & p_{down,middle} & p_{down,right} \end{bmatrix}$$

Then the pixel p is marked as edge pixel if it occurs that:

- $sign(p_{up,left}) \neq sign(p_{down,right})$, or
- $sign(p_{up,middle}) \neq sign(p_{down,middle})$, or
- $sign(p_{up,right}) \neq sign(p_{down,left})$, or
- $sign(p_{middle,left}) \neq sign(p_{middle,right})$.

```
float *zero_cross = calloc(w*h,sizeof(float));
                // this image will only content values 0 and 255
                // but float type is used for saving using iio.
if (zero_cross == NULL){
    fprintf(stderr, "Out of memory...\n");
    exit(EXIT_FAILURE);
```

```
        }
        int ind_en_lapl, fila, col;
        int *offsets = get_neighbors_offset(w+n+1, 3);
        pmax = w*h;
        int dif_fila_col = (n+1)/2;
        for (p=0;p<pmax;p++){
            fila = ((int)(p/w));
            col = p-(w*fila) + dif_fila_col;
            fila += dif_fila_col;
            ind_en_lapl = col + (w+n+1)*fila;
            double *n3 = get_neighborhood(laplacian, ind_en_lapl, 3, offsets);
            if ((n3[3]*n3[5]<0)&&(abs(n3[3]-n3[5])>(tzc*max_l))) {
                // horizontal sign change
                zero_cross[p] = 255;
            } else if ((n3[1]*n3[7]<0)&&(abs(n3[1]-n3[7])>(tzc*max_l))) {
                    // vertical sign change
                    zero_cross[p] = 255;
                } else if ((n3[2]*n3[6]<0)&&(abs(n3[2]-n3[6])>(tzc*max_l))) {
                        // +45deg sign change
                        zero_cross[p] = 255;
                    } else if ((n3[0]*n3[8]<0)&&(abs(n3[0]-n3[8])>(tzc*max_l))) {
                            // -45deg sign change
                            zero_cross[p] = 255;
                        }
            free_neighborhood(n3);
        }
        free_neighbors_offsets(offsets);
```

Save output image (using *iio*):

```
        iio_save_image_float_vec(argv[6], zero_cross, w, h, 1);
```

Note: the main function in ***test_mh_log.c*** *is essentially the same. The only difference is that a LoG kernel is generated (instead of a Gaussian kernel) using the* ***LoG_kernel*** *function in* ***gaussian_kernel.c****. Therefore there is no need to use the laplacian operator, so only one convolution is made.*

B.3 Haralick

The source code for this section can be found in the file `test_haralick.c`.
Haralick edge detectors, main C file.

Parameters:

- `input_image` - Input image.
- `rhozero` - Threshold for the Haralick condition $|\frac{C_2}{2C_3}| \leq \rho_0$.
- `padding_method` - Padding method flag (in convolution): 0 means zero-padding, 1 means image boundary reflection.
- `output_image` - Output image (edges).

Includes:

```
#include "iio.c"
#include "2dconvolution.c"
#include <time.h>
```

Main function

```
int main(int argc, char *argv[]) {
```

Load input image (using *iio*):

```
        int w, h, pixeldim;
        float *im_orig = iio_read_image_float_vec(argv[1], &w, &h, &pixeldim);
```

Grayscale conversion (if necessary): explained in B.2.

Masks calculated by 2-d fitting (using LS) with the function:

$$f(x,y) = k_1 + k_2x + k_3y + k_4x^2 + k_5xy + k_6 * y^2 + k_7x^3 + k_8x^2y + k_9xy^2 + k_{10}y^3$$

```
        double masks[10][25] = { {   425,   275,  225,  275,  425,
                                     275,   125,   75,  125,  275,
                                     225,    75,   25,   75,  225,
                                     275,   125,   75,  125,  275,
                                     425,   275,  225,  275,  425},
                               { -2260,  -620,    0,  620,  2260,
                                 -1660,  -320,    0,  320,  1660,
                                 -1460,  -220,    0,  220,  1460,
                                 -1660,  -320,    0,  320,  1660,
                                 -2260,  -620,    0,  620, 2260},
                              // matrix continues, too large to display in documentation.
```

Initialise edge image using `calloc`:

```
        float *edges = calloc(w*h,sizeof(float));
```

Padding: a larger auxiliary image `aux` is required to compute the coefficients k_1 to k_{10} in every pixel of the original image.
Two different methods are implemented: zero-padding (`padding_method`= 0) and reflection of original image (`padding_method`= 1).

```
        int wx = (w+8);
        int hx = (h+8);
        double *aux = calloc(wx*hx,sizeof(double));
        int fila,col;
        int imax = wx*hx;
        if (padding_method == 0) {
            for(i=0;i<imax;i++){
                fila = (int)(i/wx);
                col = i-(wx*fila);
                if ( (fila>=4)&&(col>=4)&&(fila<h+4)&&(col<w+4) ) {
                    aux[i] = im[(col-4)+(w*(fila-4))];
                }
            }
        }
```

```
        if (padding_method == 1) {
            int fila_refl, col_refl;
            for(i=0;i<imax;i++){
                fila = (int)(i/wx);
                col = i-(wx*fila);
                if (fila<4) {
                    fila_refl = 7 - fila;
                    if (col<4) { //zone1
                        col_refl = 7 - col;
                    } else if (col<w+4) {   //zone2
                        col_refl = col;
                    } else { //zone3
                        col_refl = 2*w + 7 - col;
                    }
                } else if (fila<h+4) {
                    fila_refl = fila;
                    if (col<4) { //zone4
                        col_refl = 7 - col;
```

```
                } else if (col<w+4) { //image
                    col_refl = col;
                } else { //zone5
                    col_refl =  2*w + 7 - col;
                }
            } else {
                fila_refl = 2*h + 7 - fila;
                if (col<4) { //zone6
                    col_refl =  7 - col;
                } else if (col<w+4) {   //zone7
                    col_refl = col;
                } else { //zone8
                    col_refl =  2*w + 7 - col;
                }
            }
            aux[i] = im[(col_refl-4)+(w*(fila_refl-4))];
        } //for
    }
```

Haralick algorithm: coefficients k_1 to k_{10} are computed in every pixel of the original image (using the function `get_neighbors_offset` to get the index offsets of the neighbor pixels and the function `get_neighborhood` to get the neighborhood of a pixel using those index offsets). Once the coefficients are calculated, are computed

$$C_2 = \frac{k_2^2 k_4 + k_2 k_3 k_5 + k_3^2 k_6}{k_2^2 + k_3^2}$$

and

$$C_3 = \frac{k_2^3 k_7 + k_2^2 k_3 k_8 + k_2 k_3^2 k_9 + k_3^3 k_{10}}{(\sqrt{k_2^2 + k_3^2})^3},$$

and then the edge condition is evaluated in every pixel; $|\frac{C_2}{2C_3}| \leq \rho_0$.

```
    int i_zp, u, v, num_edges;
    num_edges = 0;
    double k[10];
    int *offsets = get_neighbors_offset(wx, 5);
    double acum;
    double C2, C3, denom, sintheta, costheta;
    for(fila=0;fila<h;fila++){
        for(col=0;col<w;col++){
            i = col + w*fila;                   // original image & edges image index
            i_zp = (col+4) + wx*(fila+4);   // padded image index
            double *neighborhood = get_neighborhood(aux, i_zp, 5, offsets);
            // k1 to k10 (note: k1 (u=0) is not necessary)
            for(u=0;u<10;u++){
                acum = 0;
                for(v=0;v<25;v++){
                    acum += neighborhood[v]*masks[u][v];
                }
                k[u] = acum;
            }
            // compute C2 and C3
            denom = sqrt( k[1]*k[1] + k[2]*k[2] );
            sintheta = - k[1] / denom;
            costheta = - k[2] / denom;
            C2 = k[3]*sintheta*sintheta + k[4]*sintheta*costheta + k[5]*costheta*costheta;
            C3 = k[6]*sintheta*sintheta*sintheta + k[7]*sintheta*sintheta*costheta +
                 k[8]*sintheta*costheta*costheta + k[9]*costheta*costheta*costheta;
            //if ((fabs(C2 / (3*C3))<=rhozero)&&(C3<=0)) {
            if ((fabs(C2 / (3*C3))<=rhozero)) {
                edges[i] = 255;
                num_edges += 1;
            }
            // free neighborhood
            free_neighborhood(neighborhood);
        }
    }
```

Save output image (using *iio*):

```
        iio_save_image_float_vec(argv[4], edges, w, h, 1);
```

B.4 Gaussian kernel generation

The source code for this section can be found in the file `gaussian_kernel.c`.
This file implements the necessary functions for generating Gaussian and LoG (Laplacian of a Gaussian) kernels. It is also done here the alloc and free of the memory needed.

Includes:

```
#include <math.h> // exp
#include <stdlib.h> // malloc, calloc, free
#include <stdio.h> // fprintf
```

Function `gaussian_kernel`

This function generates a Gaussian kernel of size $n \times n$ and standard deviation σ.

```
double *gaussian_kernel(int n, float sigma){
```

Memory allocation for the kernel:

```
    double *kernel = malloc(n*n*sizeof(double));
```

A normalized Gaussian kernel is generated using the expression $e^{-\frac{x^2+y^2}{2\sigma^2}}$.

```
    int i;
    int fila, col, x, y;
    double suma = 0;
    int imax = n*n;
    for(i=0;i<imax;i++){
        fila = (int)(i/n);
        col = i-(n*fila);
        y = ((int)(n/2))-fila;
        x = col-((int)(n/2));
        kernel[i] = exp(-(x*x + y*y)/(2*sigma*sigma));
        suma += kernel[i];
    }
```

Kernel normalization, using the sum of its components.

```
    for(i=0;i<n*n;i++){
        kernel[i] = kernel[i]/suma;
    }
```

Return:

```
    return kernel;
```

Function `free_gaussian_kernel`

This function frees the memory allocated in the function `gaussian_kernel`. It receives as parameter the pointer to the array to be freed.

```
void free_gaussian_kernel(double* kernel){
```

Free memory:

```
    free(kernel);
```

Function `LoG_kernel`

This function generates a Laplacian of a Gaussian kernel (LoG kernel) of size $n \times n$ and standard deviation σ.

```
double *LoG_kernel(int n, float sigma){
```

Memory allocation for the kernel:

```
    double *kernel = malloc(n*n*sizeof(double));
```

We generate a Laplacian of a Gaussian kernel using the expression $\frac{x^2+y^2-2\sigma^2}{\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$.

```
    int i;
    int fila, col, x, y;
    double suma = 0;
    int imax = n*n;
    for(i=0;i<imax;i++){
        fila = (int)(i/n);
        col = i-(n*fila);
        y = ((int)(n/2))-fila;
        x = col-((int)(n/2));
        kernel[i] = ( (x*x + y*y - 2*sigma*sigma)/(sigma*sigma*sigma*sigma) )
                    *exp(-(x*x + y*y)/(2*sigma*sigma));
    }
```

Return:

```
    return kernel;
```

B.5 2D convolution

The source code for this section can be found in the file `2dconvolution.c`.
This file contains the necessary functions to compute the convolution between an input array and a kernel (usually the input array is larger than the kernel).

Includes:

```
    #include <stdlib.h> // malloc, calloc, free
    #include <stdio.h> // fprintf
```

Function `free_gaussian_kernel`

This function frees the memory allocated in the function `get_neighborhood`. It receives as parameter the pointer to the array to be freed.

```
void free_neighborhood(double* neighborhood){
```

Free memory:

```
    free(neighborhood);
```

Function `free_gaussian_kernel`

This function frees the memory allocated in the function `get_neighbors_offset`. It receives as parameter the pointer to the array to be freed.

```
void free_neighbors_offsets(int* offsets){
```

Free memory:

```
    free(offsets);
```

Function `get_neighbors_offset`

This function computes the linear array index offsets required to access the neighbor pixels in a neighborhood of size $n \times n$. This computation requires to know the width w of the image.

```
int *get_neighbors_offset(int w, int n){
```

Memory allocation for the index offsets array:

```
    int *aux = malloc(n*n*sizeof(int));
```

Computation of the index offsets array:

```
    int imax = n*n;
    int dif_fila_col = (n-1)/2;
    for (i=0;i<imax;i++){
        delta_fila = (int)(i/n)-dif_fila_col;
        delta_col = i-n*((int)(i/n))-dif_fila_col;
        aux[i] = delta_fila*w+delta_col;
    }
```

Return:

```
    return aux;
```

Function `get_neighborhood`

This function returns an array of size $n \times n$ containing the neighbors of the pixel at the position `pos` of the image `im`. For this calculation, the index offsets array calculated with the function `get_neighbors_offset` is needed.

```
double *get_neighborhood(double *im, int pos, int n, int* offsets){
```

Memory allocation for the neighborhood array:

```
    double *aux = malloc(n*n*sizeof(double));
```

Assigning values of the neighborhood:

```
    int i;
    int imax = n*n;
    for (i=0;i<imax;i++){
        aux[i] = im[pos+offsets[i]];
    }
```

Return:

```
    return aux;
```

Function `conv2d`

This function calculates the convolution of image `input` (size $w \times h$) with the kernel `kernel` (size $n \times n$). Returns an image of size $(w + n - 1) \times (h + n - 1)$ (due to padding). The integer `padding_method` manages the padding method: 0 means zero-padding, 1 means image boundary reflection.

```
double *conv2d(double *input, int w, int h, double *kernel, int n, int padding_method){
```

Zero padding: reserve memory for an array of size $(w + 2(n-1)) \times (h + 2(n-1))$.

```
    int wx = (w+2*(n-1));
    int hx = (h+2*(n-1));
    double *aux = calloc(wx*hx,sizeof(double));
```

Fill in the values of the image `aux`, centering the original image `input` on it (zero-padding).

```
    int i,j,fila,col;
    int imax = wx*hx;
    if (padding_method == 0) {
        for(i=0;i<imax;i++){
            fila = (int)(i/wx);
            col = i-(wx*fila);
            if ( (fila>=n-1)&&(col>=n-1)&&(fila<h+n-1)&&(col<w+n-1) ) {
                aux[i] = input[(col-n+1)+(w*(fila-n+1))];
            }
        }
    }
```

Other padding method: reflection of the image (if `padding_method` equal to 1).

```
    if (padding_method == 1) {
        int fila_refl, col_refl;
        for(i=0;i<imax;i++){
            fila = (int)(i/wx);
```

```
            col = i-(wx*fila);
            if (fila<n-1) {
                fila_refl = 2*n - 3 - fila;
                if (col<n-1) { //zone1
                    col_refl = 2*n - 3 - col;
                } else if (col<w+n-1) { //zone2
                    col_refl = col;
                } else { //zone3
                    col_refl = 2*w + 2*n - 3 - col;
                }
            } else if (fila<h+n-1) {
                fila_refl = fila;
                if (col<n-1) { //zone4
                    col_refl = 2*n - 3 - col;
                } else if (col<w+n-1) { //image
                    col_refl = col;
                } else { //zone5
                    col_refl =  2*w + 2*n - 3 - col;
                }
            } else {
                fila_refl = 2*h + 2*n - 3 - fila;
                if (col<n-1) { //zone6
                    col_refl =  2*n - 3 - col;
                } else if (col<w+n-1) { //zone7
                    col_refl = col;
                } else { //zone8
                    col_refl =  2*w + 2*n - 3 - col;
                }
            }
            aux[i] = input[(col_refl-n+1)+(w*(fila_refl-n+1))];
        } //for
    } //if
```

Reserve memory for the output array of size $(w + n - 1) \times (h + n - 1)$.

```
    double *out = malloc((w+n-1)*(h+n-1)*sizeof(double));
```

Compute the convolution. Most of the operations are intended to calculate the relative positions between the images `aux` and `out`.

```
    double acum = 0;
    int pos;
    // Convolution
    imax = (w+n-1)*(h+n-1);
    int jmax = n*n;
    int *offsets = get_neighbors_offset(wx, n);
    int dif_fila_col = (n-1)/2;
    for(i=0;i<imax;i++){
        fila = (int)(i/(w+n-1));
        col = i-((w+n-1)*fila);
        fila += dif_fila_col;
        col += dif_fila_col;
        pos = wx*fila + col;
        // compute convolution:
        acum = 0;
        for (j=0;j<jmax;j++){
            acum += aux[pos+offsets[j]]*kernel[j];
        }
        out[i] = acum;
    }
```

Free and return:

```
    free(aux);
    free_neighbors_offsets(offsets);
    return out;
    //return aux; //TEST
```

References

[1] P. Arbelaez, *Boundary extraction in natural images using ultrametric contour maps*, (2006).

[2] J. Canny, *A Computational Approach to Edge Detection*, Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-8 (1986), pp. 679–698.

[3] F. Cao, P. Musé, and F. Sur, *Extracting meaningful curves from images*, (2005).

[4] R. Deriche, *Recursively implementing the Gaussian and its derivatives*, Tech. Report 1893, INRIA, May 1993.

[5] P. Getreuer, *Linear Methods for Image Interpolation*. http://www.ipol.im/pub/algo/g_linear_methods_for_image_interpolation/, 2011. ISSN:2105-1232, DOI:.[17]

[6] Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing (3rd Edition)*, Prentice Hall, 3 ed., Aug. 2007.

[7] R. M. Haralick, *Digital step edges from zero-crossings of second directional derivatives*, IEEE Trans. Pattern Analysis and Machine Intelligence, 6 (1984), pp. 58–68.

[8] *Image Processing On Line*. http://www.ipol.im/. ISSN:2105-1232, DOI:10.5201/ipol[18].

[9] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, SIGGRAPH Comput. Graph., 21 (1987), pp. 163–169.

[10] D. Marr and E. Hildreth, *Theory of edge detection*, Tech. Report AIM-518, MIT Artificial Intelligence Laboratory, Apr. 6 1979.

[11] J. M. Morel and S. Solimini, *Variational Methods in Image Segmentation*, Birkhäuser, 1995.

[12] L. Najman and M. Schmitt, *Geodesic saliency of watershed contours and hierarchical segmentation*, 18 (1996), pp. 1163 –1173.

[17] http://dx.doi.org/10.5201/ipol.2011.g_lmii
[18] http://dx.doi.org/10.5201/ipol