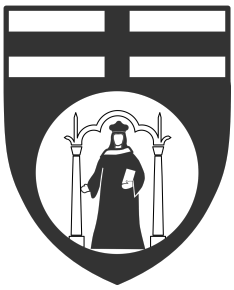

Università degli Studi di Genova

Computer Vision

Assignment 3: Edge detection

Giovanni Battista Borrè, Tommaso Gruppi, Federico Tomat

April 8, 2019

Contents

Introduction

1 Introduction

1.1 Importance of the study of the edges

Edge points contain a great amount of semantical information concerning the analized images. E.g. the boundaries of objects are usually delineated by contours, same for rapid change of orientation of the surface. Each of the isolated edge points can also be grouped into longer curves or contours, as well as straight line segments.[1]

1.2 Edge Detection

Qualitatively, edges occur at boundaries between regions of different color, intensity, or texture. The division of an image in coherent regions is somehow a difficult semantical task, so it's usually more reasonable to use a more operative definition for edge. So an edge is the location of a rapid intensity variation.
Mathematically the edge are the points on the surface associated to the image that have a steep slope. In order to define the slope for the surface of the image we use the gradient:

$$\mathbf{J}(\mathbf{x}) = \nabla \mathbf{I}(\mathbf{x}) = (\frac{\partial \mathbf{I}}{\partial \mathbf{x}} + \frac{\partial \mathbf{I}}{\partial \mathbf{y}})(\mathbf{x}) \tag{1}$$

The process of taking the gradient of the image accentuates the higher frequencies in the image and therefore amplify the noise. To avoid this effect we apply a low pass filter to the original image, usually a Gaussian low pass filter. Due to the linearity of the convolution we can just apply the gradient to the filter, this operation is convenient since the derivative is applied to the filter itself so we will not derive a signal affected by noise.

$$\mathbf{J}_\sigma(\mathbf{x}) = \nabla[\mathbf{G}_\sigma * \mathbf{I}(\mathbf{x})] = [\nabla \mathbf{G}_\sigma(\mathbf{x})] * \mathbf{I}(\mathbf{x}) \tag{2}$$

σ represents the standard deviation of the Gaussian.

For many purpose a thinner edge is required, so we look for the *maxima* in the edge strength. Finding this maximum corresponds to take a directional derivative of the strength field in the direction of the gradient and perpendicular to the edge itself, then looking for zero crossings. The desired directional derivative is equivalent to the dot product between a second gradient operator and the results of the first[1] convolution, so the Laplacian of the Gaussian filter will be computed:

$$\mathbf{S}_\sigma(\mathbf{x}) = \nabla^\mathbf{2}[\mathbf{G}_\sigma * \mathbf{I}(\mathbf{x})] = [\nabla^\mathbf{2} \mathbf{G}_\sigma(\mathbf{x})] * \mathbf{I}(\mathbf{x}) \tag{3}$$

The *Laplacian* of Gaussian operator is also known as *Mexican hat* due to his shape [2].

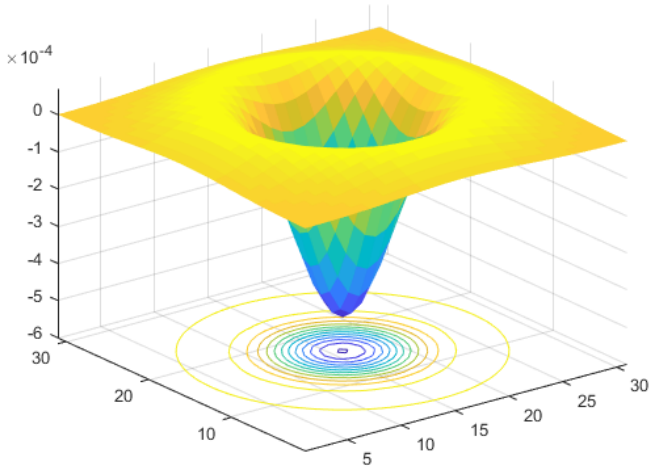

Figure 1: Laplacian of Gaussian function on x-y-z axes

2 Algorithms used

In order to obtain an edge detection we implemented in MATLAB® environment two of the most used algorithms of edge detection:*Marr and Hildreth* and *Canny*.

2.1 Marr and Hildreth edge detector through zero crossing

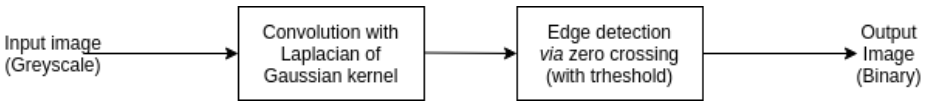

Figure 2: Block diagram of Marr Hildreth edge detection throught zero crossing

The Marr-Hildreth algorithm consists of three key steps:

- Noise reduction (usually by convolution with low pass Gaussian filter);
- Laplacian calculation;
- Edge Tracking (by zero crossing).

Due to the linearity of convolution operator steps 1 and 2 can be replaced by the convolution of the original image with a *LoG* (*Laplacian of Gaussian*) operator[2]. Therefore the algorithm maybe reduced to the two following steps:

- Noise reduction and Laplacian calculation (by convolution with LoG operator);
- Edge Tracking (by zero crossing).

2.2 Canny edge detector through hysteresis

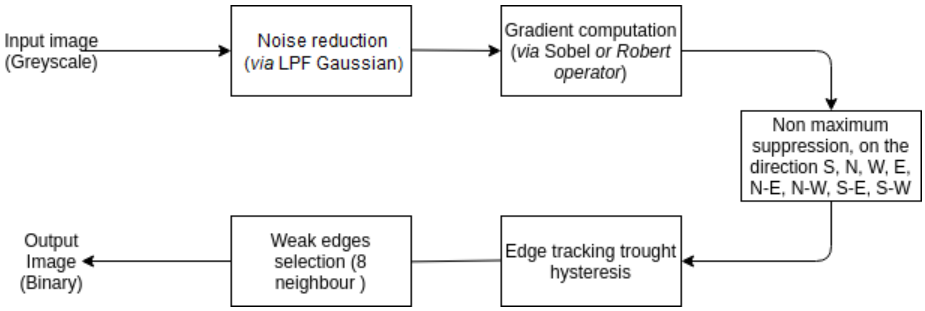

Figure 3: Block diagram of Canny edge detection

The Canny edge detection algorithm is composed by 4 steps:

- Noise reduction;
- Gradient calculation;
- Non-maximum suppression;
- Edge Tracking (by Hysteresis Double thresholding).

Noise Reduction

Since the mathematics involved is mainly based on derivative, edge detection results are highly affected by image noise. One way to get rid of the noise on the image, is by applying Gaussian blur to smooth it. To do that, image convolution technique is applied with a Gaussian Kernel. The kernel size depends on the expected blurring effect. Basically, the smaller the kernel is, the less visible the blur is.

Gradient calculation

The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators. Edges correspond to a change of pixels' intensity. To detect the edges, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal and vertical. When the image is smoothed, the derivatives are calculated by convolving the image with Sobel kernels K_x and K_y, respectively:

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{4}$$

Then, the magnitude **G** and the slope $\boldsymbol{\theta}$ of the gradient are calculated as follows:

$$\begin{cases} G = \sqrt{I_x^2 + I_y^2} \\ \theta = arctan(\frac{I_y}{I_x}) \end{cases} \tag{5}$$

Non-Maximum Suppression

Non-maximum suppression is an edge thinning technique. It is applied to find "the largest" edge. After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. Thus non-maximum suppression can help to suppress all the gradient intensity values (by setting them to 0) except for the pixels with the maximum value in the edge directions, which indicate the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

- Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
- If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

Let's take an easy example:

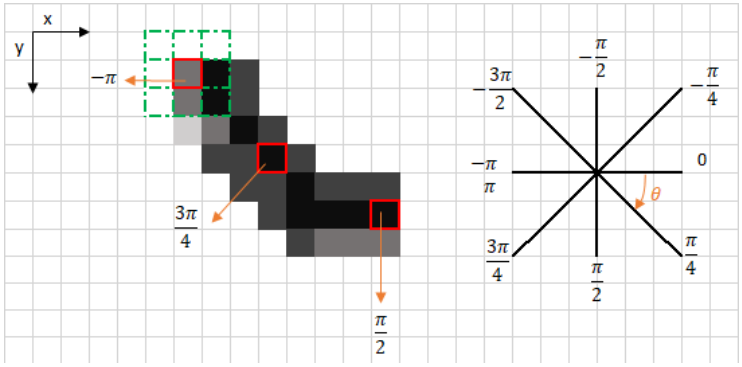

Figure 4: Focus on left corner red box pixel

The upper left corner red box present on the above image, represents an intensity pixel of the Gradient Intensity matrix being processed. The corresponding edge direction is represented by the orange arrow with an angle of $-\pi$ radians ($\pm 180°$).

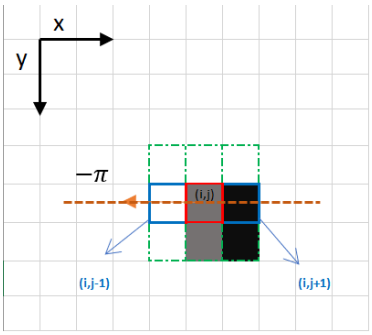

Figure 5: Focus on the upper left corner red box pixel

The edge direction is the orange dotted line (horizontal from right to left). The purpose of the algorithm is to check if the pixels on the same direction are more or less intense than the ones being processed. In the example above, the pixel (i, j) is being processed, and the pixels on the same direction are highlighted in blue $(i, j-1)$ and $(i, j+1)$. If one of those two pixels are more intense than the one being processed, then only the more intense one is kept. Pixel $(i, j-1)$ seems to be more intense, because it is white. Hence, the intensity value of the current pixel (i, j) is set to 0. If there are no pixels in the edge direction having more intense values, then the value of the current pixel is kept. For example:

- if the rounded gradient angle is 0° (i.e. the edge is in the north-south direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the east and west directions,
- if the rounded gradient angle is 90° (i.e. the edge is in the east-west direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north and south directions,
- if the rounded gradient angle is 135° (i.e. the edge is in the northeast-southwest direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north west and south-east directions,
- if the rounded gradient angle is 45° (i.e. the edge is in the north west–south east direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north east and south west directions.

In more accurate implementations, linear interpolation is used between the two neighbouring pixels that straddle the gradient direction. For example, if the gradient angle is between 89° and 180°, interpolation between gradients at the north and north east pixels will give one interpolated value, and interpolation between the south and south west pixels will give the other. The gradient magnitude at the central pixel must be greater than both of these for it to be marked as an edge.

Note that the sign of the direction is irrelevant, i.e. north–south is the same as south–north and so on.

Edge Tracking (by Hysteresis thresholding)

The double threshold technique is described in the following:

- High threshold is used to identify the strong pixels (intensity higher than the high threshold)
- Low threshold is used to identify the non-relevant pixels (intensity lower than the low threshold)
- All pixels having intensity between both thresholds are flagged as weak and the Hysteresis mechanism (next step) will help us identify the ones that could be considered as strong and the ones that are considered as non-relevant. Based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one and if the position of the pixel symmetrical to the strong one with respect to the one considered is not a weak one.

MATLAB Resolution

In this assignment we aim to:

- write a function that implements the Laplacian of Gaussian Operator and to plot it with different spatial support and standard deviation;
- convolve the test images with the Laplacian of Gaussian and display the results;
- detect zero crossings applying a threshold on the slope, first scanning for each row and then for each column;
- finally test the algorithm by varying the spatial support of the kernel and the threshold and compare the results.

For doing that we have implemented different functions on MATLAB®

1 Auto threshold

$$threshold = autothreshold(imgInput) \tag{6}$$

We use this function to calculate the best threshold for our edge detection algorithms. Here we have an adjustment constant used to determine the threshold. First we compute the mean value and the standard deviation of the of the image matrix $M = mean2(imgInput)$[7] and $s = std2(imgInput)$[6]. Then we by heuristics chose a parameter $C = 0.1$.
The two thresholds are so computed:

$$\begin{aligned} highThreshold &= min(1, C * (M - s)) * \frac{1}{255} \\ lowThreshold &= max(254, C * (M + s)) * \frac{1}{255} \\ threshold &= \begin{bmatrix} highThreshold & lowThreshold \end{bmatrix} \end{aligned} \tag{7}$$

2 Laplacian of Gaussian

For computing the Laplacian of Gauss we decided to implement the function

$$LaplacianMatrix = LaplacianOfGaussian(sigma) \tag{8}$$

Inside this function the spatial support is initialized to the value equal to the ceiling of three times the standard deviation sigma. After the application of

the MATLAB® function *meshgrid*[5] , with a domain ranging between - and + the value of the spatial support, we store and compute the Laplacian function inside the variable called LaplacianMatrix. The different standard deviations requested for the gaussian function in this assignment are initialized in the main function inside the section Parameters.

The next step consists of convolving the test images with the Laplacian of Gaussian and we decide, after a proper resize of the image through the function $matrixFramer(matrix, matrixSize)$ implemented before by us, to do that in a for cycle inside the main function.

3 Zero Crossing Edge Detection

After the convolution of the input image with the Laplacian of Gaussian, we have to detect the edges through the zero crossing method, for doing that we have implemented the function called

$$edgeMatrix = zeroCrossingEdgeDedector(threshold, convMatrix) \quad (9)$$

This function has as parameters the threshold that we want to use to evaluate if there is an edge or not and the matrix corresponding to the convolution between the Laplacian of Gaussian and the input image on which we want to apply the zero crossing. Inside the body of this function first of all we initialized the output image matrix with zeros and after that we scan all the function' s input matrix and we check the sign of each contiguous first for row then for column. When signs are different we evaluate the module of the difference of contiguous pixels, if it is lower of the selected threshold we switch on the pixel on the edgeMatrix coorrispondent to the index of the pixel in evaluation.

4 The Canny Edge Detector

For this method the following function is implemented by us:

$$imgCannyEdge = canny(imgInput, ThreshLow, ThreshHigh) \quad (10)$$

The Canny edge detector is an edge detection operator composed by some steps:

- Apply Gaussian filter to smooth the image in order to reduce the effects of noise on the edge detector.
- Find the intensity gradients of the image.
- Apply non-maximum suppression to get rid of spurious response to edge detection.
- Apply double threshold to determine potential edges.
- Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

We will use a 5×5 Gaussian kernel because is a good size for most cases, remember that the larger the size is, the lower the detector's sensitivity to noise. The Gaussian filter kernel is given by the MATLAB® functions *imfilter*[9], so

we convolute this matrix with the original image to remove noise. To find the intensity gradients of the image we used the *Sobel Filter* finding therefore the partial derivatives thanks to the matrices called KG_x and KG_y, so we have computed another convolution for each of them with the input image pre-filtered by the Gaussian filter. Then through the functon *atan2*[8] we have computed the orientation of the resultant gradient and then converted it in degree. Then we have made some approximations: adjusting negative directions, making all directions positive, simple adding 360° in case of negative direction and after approaching the angle to the nearest one belonging to the accepted cases. After this, we calculated the magnitude and used a non maximum suppression to get rid of spurious response to edge detection. Finally applying double threshold we determine potential edges and by hysteresis we removed the edges that are weak and not connected to strong edges.

Conclusion

1 Result

The following figures show how the LoG shape changes by varying σ and consequentially the spatial support that depends on it.

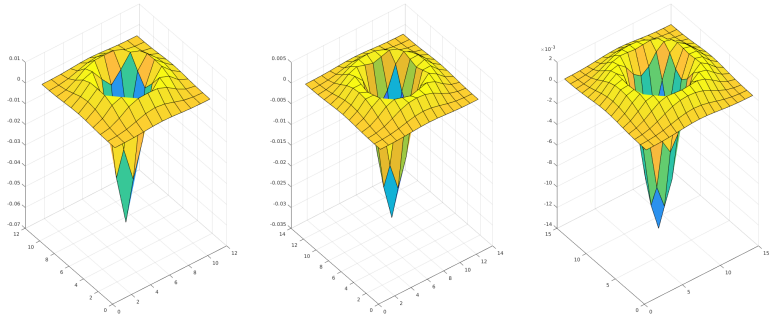

Figure 6: Comparison of LoG with $\sigma = (1.5 - 1.8 - 2.2)$

The LoG shown above have been convoluted with the Cameraman image. The result clearly highlights the *maxima* of the image intensity field. Then the edge detector algorithm is applied with different values of σ and threshold.

Figure 7: Comparison of convolution of original image wit LoG with $\sigma = (1.5 - 1.8 - 2.2)$

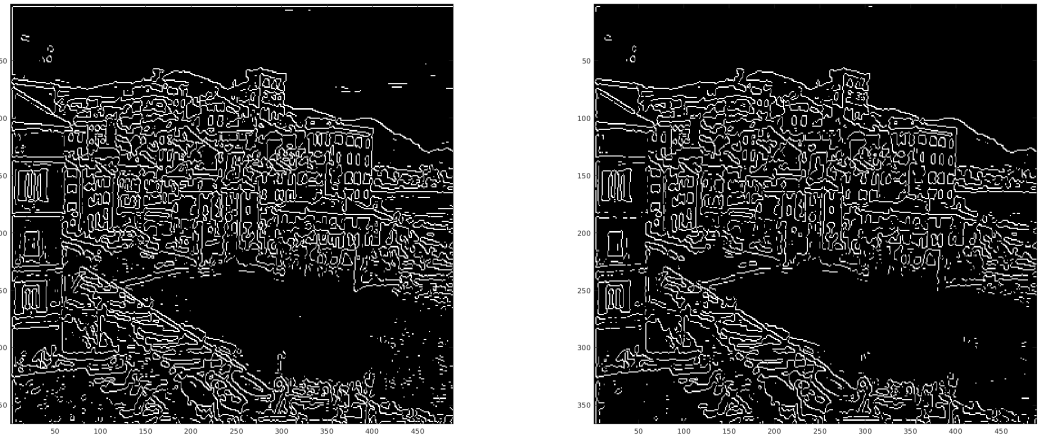

Figure 8: Boccadasse with $\sigma = (1.8 - 2)$ and threshold = 1.2)

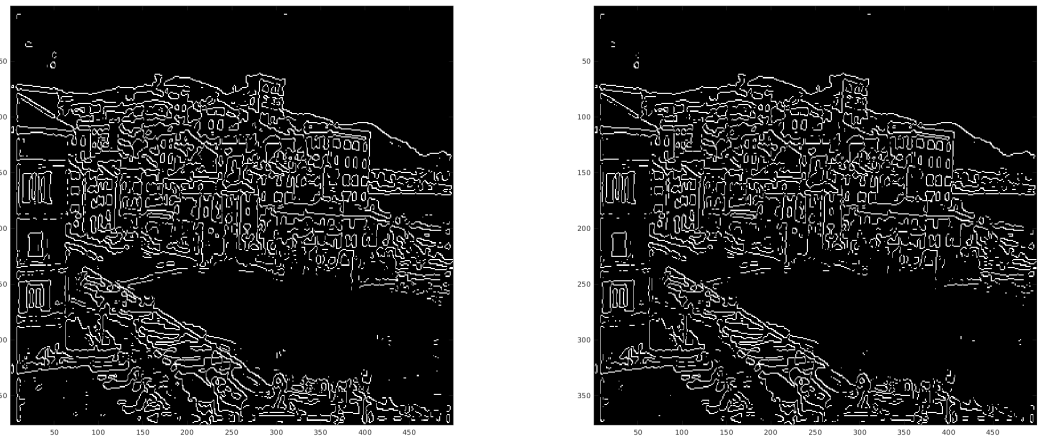

Figure 9: Boccadasse with $\sigma = 2$ and threshold = (1.2 - 1.6)

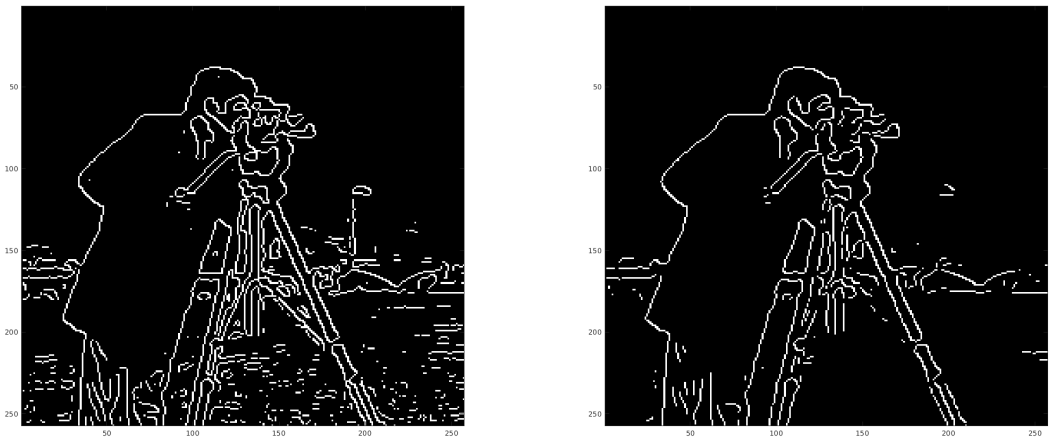

Figure 10: Cameraman with $\sigma = (2 - 2.5)$ and threshold = 1.2

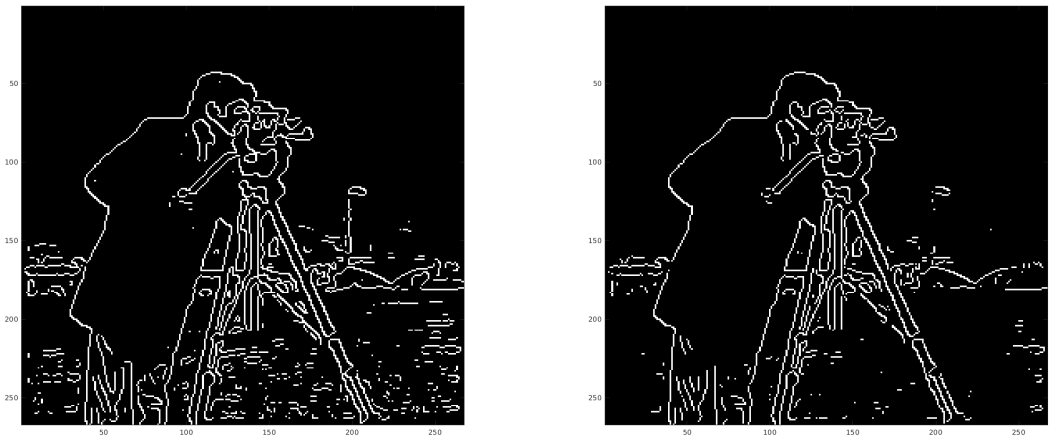

Figure 11: Cameraman with $\sigma = 2$ and threshold = (1.2 - 2)

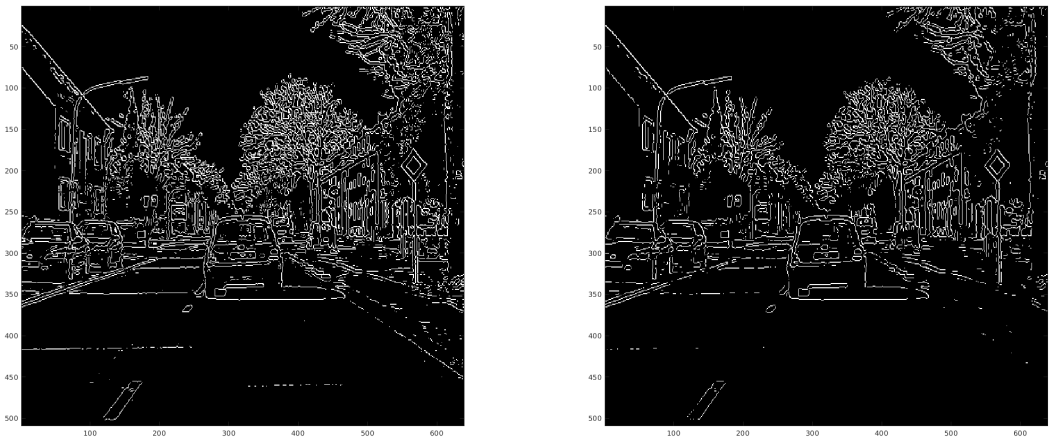

Figure 12: Car with $\sigma = (1.5 - 1.8)$ and threshold = 1.2

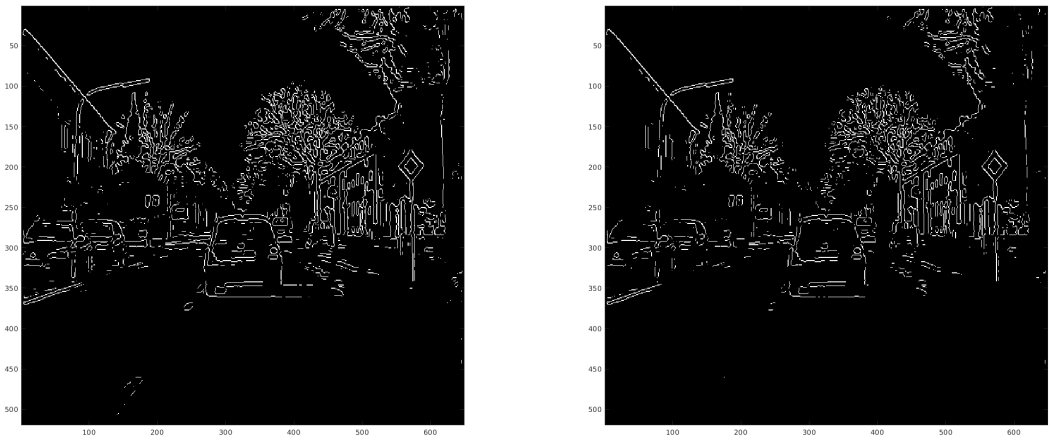

Figure 13: Car with $\sigma = 2$ and threshold = (1.2 - 2)

As the figure shows the Marr-Hildreth algorithm is very noise sensitive. To smooth the image the use of a strong LoG operator is required and this reduces the difference between edges and background.
If compared to the built in function (*edge*[4], with argument *'LoG'*), our implementation, with the same parameters for σ and threshold, acts fairly similarly, as shown below. In the comparison MATLAB® implementation is shown on the left side and our one on the right side. In the superposition, the coloured lines are from our image and white ones are from MATLAB®

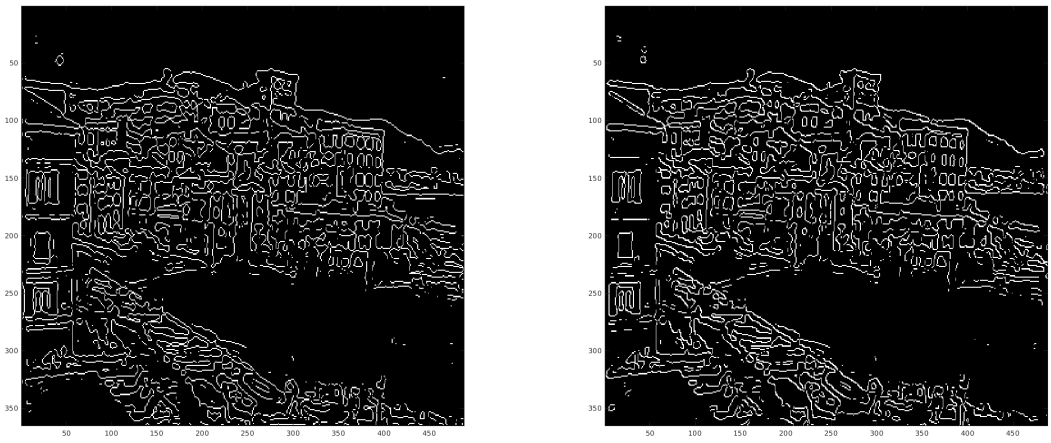

Figure 14: Boccadasse comparison with $\sigma = 2.2$ threshold $= 0.9$

Figure 15: Boccadasse superposition $\sigma = 2.2$ threshold $= 0.9$)

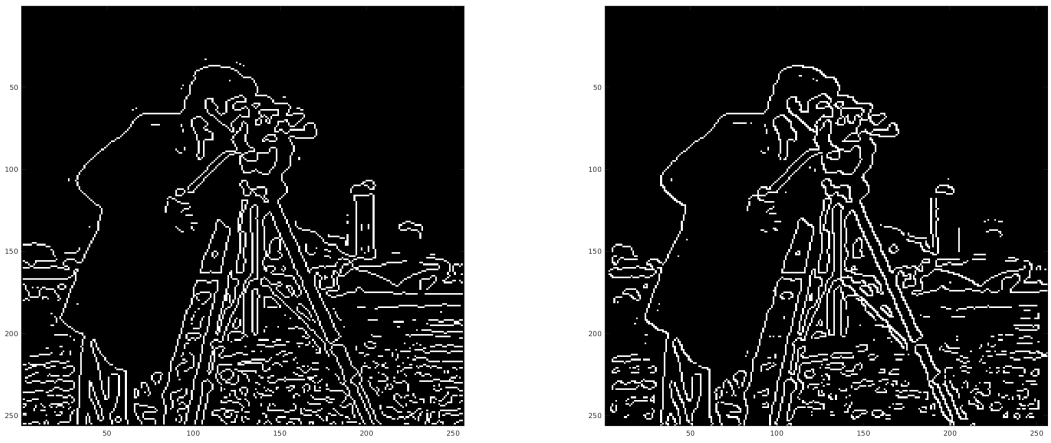

Figure 16: Cameraman comparison $\sigma = 2$ and threshold = (1.2 - 2)

Figure 17: Cameraman superposition $\sigma = 2$ and threshold = 0.8

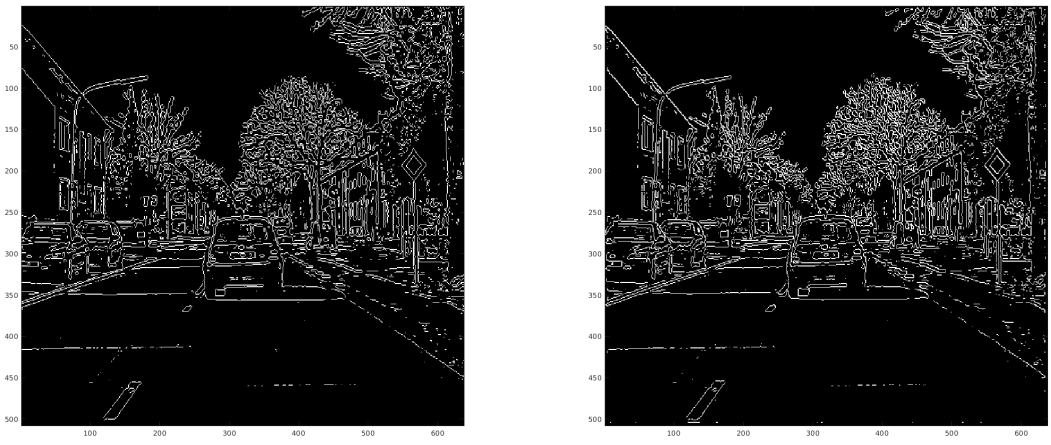

Figure 18: Car comparison with $\sigma = 1.5$ and threshold $= 1.2$

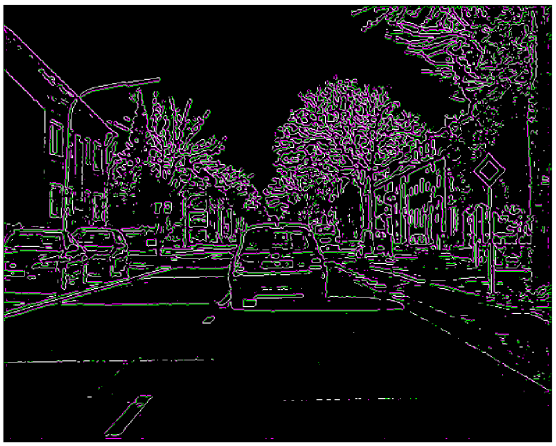

Figure 19: Car superimposition with $\sigma = 1.5$ and threshold $= 1.2$

We also implemented the Canny edge recognition algorithm with automatic thresholding based on statistical considerations. Comparing our results with those implementation (*edge*[4] function with *'Canny'* argument) our function performs fairly well. As before MATLAB®images are reported on the left side and our ones are on the right. In the superposition, the coloured lines are from our image and the white ones again from MATLAB®.

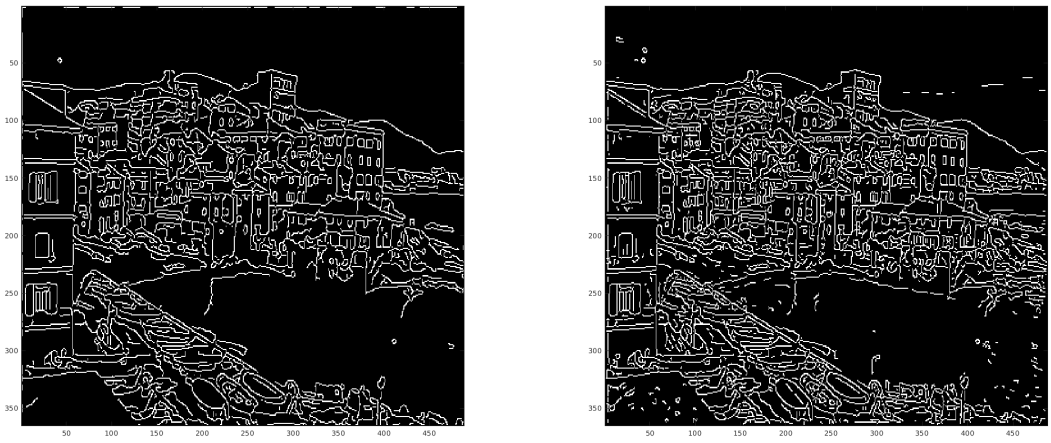

Figure 20: Boccadasse Canny comparison

Figure 21: Boccadasse Canny superposition

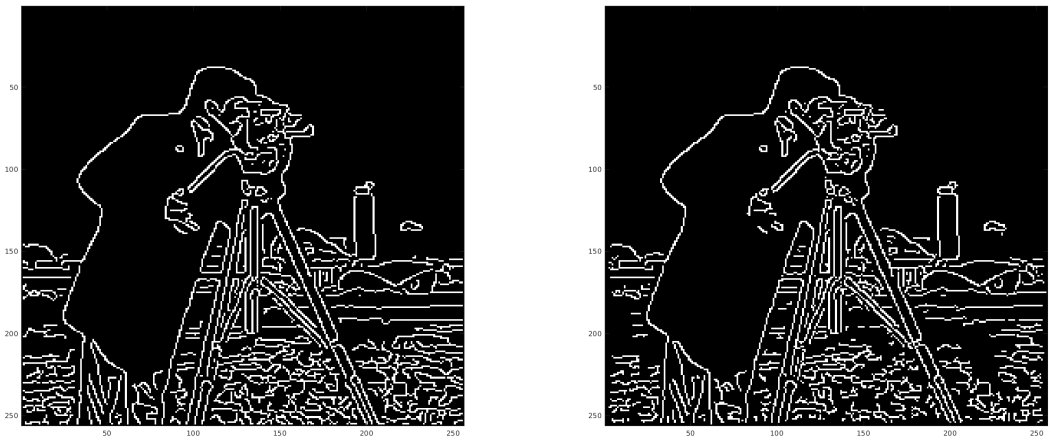

Figure 22: Cameraman Canny comparison

Figure 23: Cameraman Canny superposition

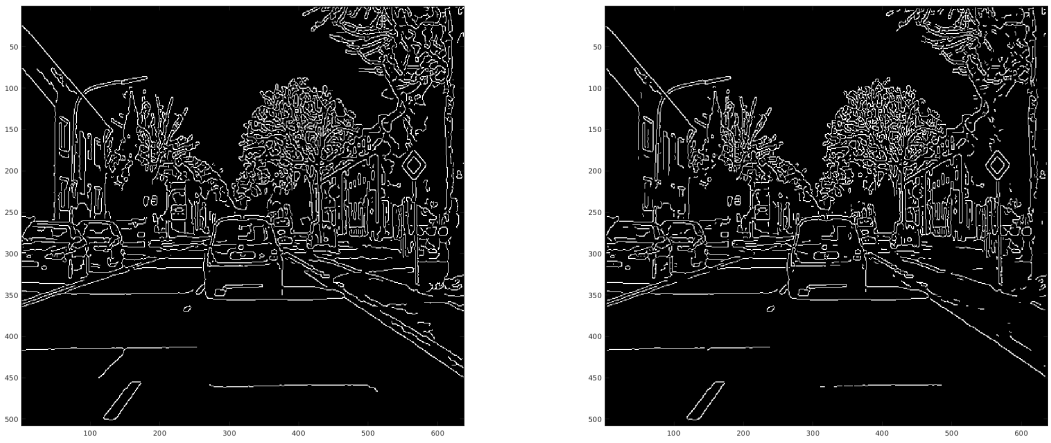

Figure 24: Car Canny comparison

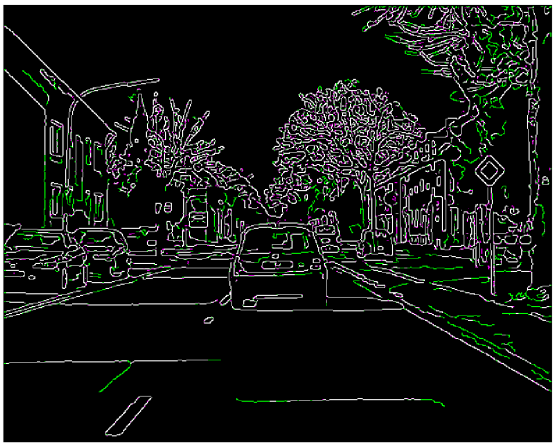

Figure 25: Car Canny superposition

2 Github Repository

https://github.com/federicotomat/Computer-Vision

Bibliography

[1] Richard Szeliski, *Computer Vision: Algorithms and Applications*, (University of Washington, 2010).

[2] Francesca Odone & Fabio Solari, *Computer Vision Course: Image Foundamental*, (University of Study of Genoa, 2019).

[3] Wikipedia page on Canny algorithm

[4] Mathworks documentation on *edge*

[5] Mathworks documentation on *meshgrid*

[6] Mathworks documentation on *std2*

[7] Mathworks documentation on *mean2*

[8] Mathworks documentation on *atan2*

[9] Mathworks documentation on *imfilter*