uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Andres Solis Montero
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Towards Feature Detection Based on Morphology of Objects on Image
TITRE DE LA THÈSE / TITLE OF THESIS

Amiya Nayak
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Chung-Horng Lung

Ahmed Karmouch

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Towards Feature Detection based on Morphology of Objects on Image

by

Andres Solis Montero

A thesis submitted to
the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-65976-2
Our file Notre référence
ISBN: 978-0-494-65976-2

Acknowledgements

First, I would like to thank my supervisor Dr. Amiya Nayak for all of his help and support during my studies here at the University of Ottawa.

Thanks to Dr. Milos Stojmenovic who is a good friend and seems to be very interested in the Caribbean, mostly the Island of Puerto Rico.

Special thanks to Dr. Ivan Stojmenovic for his valuable comments and suggestions to my work.

Second, I would like to thank all of my old and new friends who cheered me on and helped me in many ways through this year of graduate school.

Finally, but most importantly, I would like to thank and dedicate this work to the two most important people in my life; my parents. Though, no longer with me, had been my inspiration and guidance.

Los extraño mucho! Sin ustedes nada hubiese sido possible.

Contents

List of Figures

List of Tables

Abstract

This thesis describes a new line segment detection and extraction algorithm for computer vision, image segmentation, and shape recognition applications. This algorithm uses a compilation of different image processing techniques such as normalization, Gaussian smooth, automatic threshold, and Laplace edge detection to extract edge contours from color input images. Contours of each connected component are divided into short segments, which are classified by their orientation into about ten discrete categories. Straight lines are recognized as the minimal number of such consecutive short segments with the same direction. This solution indeed gives us more precise line segments (including line endpoints) and requires a shorter time than the widely used Hough Transform algorithm for detecting line segments given any orientation and location inside an image. Its easy implementation, simplicity, parameter minimization, speed, ability to split an edge into straight line segments using the actual morphology of objects, accuracy and the use of OpenCV library are key features and advantages of the proposed approach. The algorithm was tested on several simple shape images as well as on real pictures, yielding a more accurate resemblance of straight lines in accordance with the human perception of line taxonomy. The line detection algorithm introduced here requires few parameters and is robust to standard image transformations such as rotation, scaling and translation. Furthermore, some of these parameters are selected by automatic unsupervised methods, thus improving the expected algorithm outcome in terms of the stated problem. Several experimental results are presented to support the validity of the algorithm.

Chapter 1: Introduction

The aim of this chapter is to introduce the topic of interest of this thesis, in particular background, problem statement, motivation, objective, and contributions. This thesis describes the problem of extracting digital line segments from any given input image. Following epigraph will present the requirements and goals to fulfill for our proposed algorithm which will be described in the next chapters.

1.1 Background

The problem of line detection and extraction in images is very important in the fields of computer vision and image processing. Images are usually a composition of basic shapes; all of them put together give us a sense of objects that can be recognized by humans. Combinations of basic shapes such as lines, squares, triangles, circles, ellipses and so on, create complex shapes that characterize real objects. The main problem in computer vision algorithms is to find a method that detects those compositions in a robust manner and subsequently use it to recognize objects in an input image.

A straight line is the simplest basic shape and the topic of discussion in this work. Digital straight lines are image representations of straight lines. More precision would even require adding 'segments' to their full name: digital straight line segments. In other words, they are digital straight line segments. For the sake of clarity, they will be called simply lines in this thesis. Lines are one of the most common elements in images: indeed many basic shapes such as triangles, trapezoids, parallelograms, etc are composed of lines. They can be found in every human-made structure and picture.

Lines are special cases of edges (or digital edges), which represent object boundaries and sharp changes in pixel colors or intensities in images. Edge detection is an important and well studied topic in literature. Canny 1, Sobel 2, and Laplace 3 are some of the most widely known algorithms for edge extraction in digital image processing.

1.2 Problem Statement

We consider here the problem of detecting digital straight line segments from any type of image. Line segments will be described as a portion of a straight line with endpoint information and containing all points between its endpoints. The list of points contained inside the segment must be connected and can be regarded as a straight line segment.

Existing algorithms retrieving line segments use isolated points to detect set of points that are highly linear. This brings about the problem of detecting ensemble of points that are not really connected and the need to know in advance what kind of line segments we are looking for in the images (e.g.: slope, gap between line segments) to reduce the computational complexity. Although highly linear, this could end up in a misleading representation of the actual segments present in the picture.

Our main goal is to create an algorithm which is able to detect significant line segments following the morphology of objects inside an image. Basically, it shows connected segments or portions of edges that can be considered as (straight) line segments, including their endpoints. This algorithm would be used as a preprocessing step to identify more complex shapes and/or other features of interest inside an image. Thus, we want it to be a fast approach needing fewer configuration parameters than peer algorithms in literature addressing the same problem.

1.3 Existing Solutions

There are a number of schemes for studying whether or not an edge is indeed a line; some sources in literature make reference to measurements that could be used to test linearity 11; so, if two or more segments are located on the same connected edge, these algorithms will not discover them.

The existing literature on line detection concentrates around Hough transform algorithms and its variants. They require a preprocessing stage in order to convert a given input color image into a binary image. For example, OpenCV library has only one algorithm for line detection, which is Hough-transform-based. Sources that discuss this line detection problem reference the widespread Hough Transform 10131518 model for detecting the imperfect instance of straight lines using a voting procedure in the accumulator parameter space explicitly constructed for computing the Hough transform. This method maps points (pixels) from their Cartesian image space *(x, y)* extracted by an edge-detecting operator to a curve described by polar coordinates *(r, θ)*. These polar coordinate curves create accumulations of points in this space, of which local maxima values correspond to existing lines in the image. The previous scheme uses complex trigonometric operations like sine and cosine calculation for each point and needs to extract the local maxima intensity values in the accumulator image generated by the polar coordinates. The other methods are mostly variations and extensions 5 of this classical one in order to improve their execution order but all of them rely upon the same idea of the parametric representation of lines using polar coordinates 51518 and the voting procedure from the accumulator space. Other implementation utilizes convolution matrix and eigenvalue analysis of pictures 6.

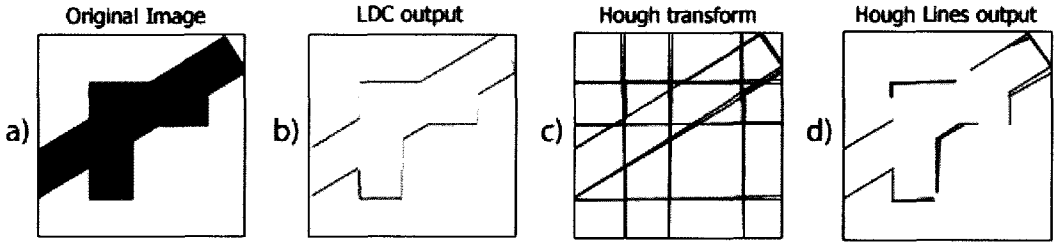

Fig. 1 a) Original Image; b) Output of our proposed LDC algorithm; c) the Hough transform line detection algorithm output; and d) its OpenCV implementation, Hough lines output, which detects line segment endpoints.

The aforementioned and other existing solutions do not enforce connectivity for discovered lines (that is, a declared line segment may consist of pixels from different parts of the image). In all these solutions, connectivity can in fact be enforced as a pre-processing step. For example, connected components of detected edges could be isolated, and Hough transforms (HT) or any other algorithm could be applied on each component separately. Our new algorithm, called LDC (Line Detection with Contours), finds connected components of detected edges and produces connected lines (see Fig. 1b). Hough-transform-based algorithms detect lines, not just line segments, as shown in Fig. 1c). An extra step is required to retrieve the line segment in the image instead of a whole line across the image. OpenCV extension of HT to produce connected line segments with endpoints is called Hough Lines HL, (see Fig. 1d). Hough Lines uses a progressive probabilistic Hough Transform PPHT 18 in order to considerably reduce the computation needed from the classical method and also give information on segment endpoints. Only a fraction of points is accumulated and a "maximum gap" parameter between line segments is applied to connect otherwise isolated pixels into straight line segments. Other parameters include the minimum gap between segments, the minimal length of line segments as well as those related to the accumulator space. We will elaborate on the algorithm in Chapter 2.

1.4 Motivations and Objective

There is no widely accepted criterion for judging whether or not an edge is a (straight) line. The judgment is either subjective or based on its usefulness for a particular application. Several schemes address a similar problem, yet none of them does it in terms of connectivity and morphology of discovered objects inside an image. These solutions detect isolated points that are highly linear all together. There is no correlation between the connectivity of such points. As such, there is no linkage between the contour of the object and the lines thereof detected.

Contours are a sequence of consecutive boundary pixels that describe outer edges and forms for any shape and object. No matter how complex an object might be, it is the contour that gives us the first inclination to recognize it. The contour of an object characterizes the object itself. This led us to believe that contours can provide a lot of useful information in order to detect line segments in images. It can be seen from the edges extracted from the image in Fig. 2a) that we can spot 13 (straight) line segments that compose such picture.

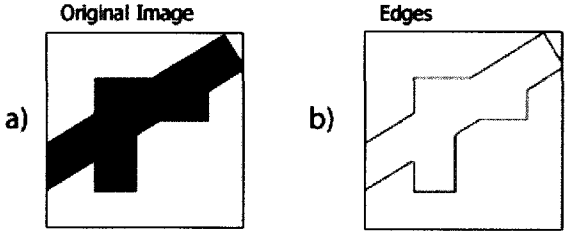

Fig. 2 Digital edge extraction example. The edges (b) are extracted from the original image (a) using an edge detection algorithm.

Our objective is to create a simple solution that is able to detect line segments (endpoint information) using the contours/edges of the objects inside the image. This solution would be exploited as a preprocessing step in the identification of more complex shapes and/or other features of interest inside an image. Thus, we want it to be a fast approach with fewer configuration parameters than those of existing algorithms targeting the same problem. Moreover, we strive to outfit our solution with improved accuracy of the connected line segments recognized with highly linear classification. Finally, our solution needs to be able to detect line segments from any input image and be robust to classical image transformations.

1.5 Contribution

We demonstrated (examples can be found in Appendix A) that our algorithm achieves superior accuracy among all peer approaches targeting our problem statement and is closer to a human classification than the Hough Lines algorithm for different input parameter configurations. It is also faster than HL and yields more accurate results in a shorter time. Although Hough transform is a straightforward algorithm to compute lines theoretically, it is not so simple in practice. Our algorithm does not need highly complex trigonometric operations like sine and cosine transforms or finding local maxima points in the accumulation space. The creation of the accumulator space is a time-consuming process since each pixel is converted into a digital line. Our proposal is a more intuitive solution for this problem, thus considerably simpler to implement and understand. Preprocessing steps applied to images in our algorithm are well-known image processing techniques. Furthermore, we define a specific procedure in order to retrieve line segments out of an actual color image, as opposed to HL which leans upon a binary image as input. There is no documented and widely accepted procedure in literature for extracting line segments from color images 4914. Obviously, the same preprocessing, applied here to the LDC algorithm, could be feasible to get a binary image from a colorful one as required

by the HL scheme. Thus, LDC is the first such algorithm with complete description of line segment extraction from color input images.

Our LDC solution accurately fits the idea of human perception classification of lines, is simple to understand and implement, fast and robust to any orientation and slope and also invariant to image transformations such as scaling, rotation and translation in order to detect lines. LDC also makes use of a smaller set of parameters. Compared to HL, LDC parameters are more predictable and can be applied almost independently on any input picture whereas HL requires fine tuning of its parameters for every picture so as to arrive at a satisfactory line extraction.

This work and extensions of it have been published in two IEEE International Conferences in the field of Computer Vision and Image Processing techniques (See Publications section for more details).

1.6 Thesis Structure

The literature review is in Chapter 2. Our new Line Detection Algorithm using Contours (called LDC algorithm) is explained in Chapter 3. It elaborates on each step involved in the process of retrieving line segments out of color images. Chapter 4 is dedicated to the analysis of our proposal and experimental results of the comparison between LDC and the built-in Hough Lines, HL 15 OpenCV implementation. This test among the algorithms is realized over a set of 30 random images with different resolution sizes. Appendixes depict all input images used in addition to the visual representation of the lines detected after the execution of each algorithm. The parameter configuration and execution time for each method are also outlined.

Chapter 2: Literature Review

In this chapter, we give a literature review on existing relevant algorithms for detecting and testing linearity in images. We also describe standard operations carried out over images. Finally, we introduce line and edge detection in OpenCV software package, a popular choice for image processing nowadays, and the way we exploited as per our problem statement, outlining a possible solution via ingredients available in OpenCV.

2.1 Measuring Linearity of Point Sets

A relevant problem is that of measuring how linear a given unordered set of points is. SNZ 11 studied linearity measures which are invariant to rotation, scaling, and translation operations. They are calculated very quickly and are resistant to protrusions in the data set. The most relevant and applicable shape measure to our work is the measurement of linearity of unordered data sets. Six linearity measures were proposed in SNZ [11]. The average orientation scheme first finds the orientation line of the set of points using moments. The method takes k pairs of points and finds the normalized unit with respect to the lines they form. The normalized unit describes all points in the same direction (along the normal to the orientation line). The average normal value (A, B) of all the k pairs is computed and the linearity value is calculated as $\sqrt{A^2 + B^2}$. Triangle heights take an average value of the relative heights of triangles formed by taking random triplets of points. Relative heights are those that are divided by the longest side of the triangle and then normalized to obtain a linearity value in the interval $[0, 1]$. Triangle perimeters take the normalized, average value of the area divided by the square of the perimeter of triplets of points as their linearity measure. Contour smoothness and eccentricity were adapted from measures of circularity. They are simple formulas involving moments and the resulting measurements were interpreted differently. Ellipse axis ratio is based on the minor/major axis ratio of the best ellipse that fits the set of points. These algorithms could

be adapted for extracting lines from images by, for example, applying a binary search over edges to find the longest straight line segments (consecutive points along the curve with high measure of linearity). This is essentially a polygonization of the curve. However, such algorithms are slow compared to one single pass along the curve proposed in this work.

2.2 Hough Transform

The Hough transform (HT) is a feature extraction technique used in image analysis, computer vision and digital image processing 14. The purpose of this technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure. Hough transform is a core component of Hough Lines (HL) routine in OpenCV for detecting lines in images.

HT method uses a binary image as input, where white pixels are interesting features to keep track of. It maps all possible lines passing through each point *(x, y)* from the image space into a parameter space *(r, θ)*. In this parameter space, *r* is the Hessian normal form representation of a line, $r = x * \cos(\theta) + y * \sin(\theta)$; it represents the perpendicular line from the Cartesian origin to the line and θ the angle between *r* and the X axis (see Fig. 3). Each point *(x, y)* is mapped to all pairs *(r, θ)* which represent all the lines that pass through *(x, y)*.

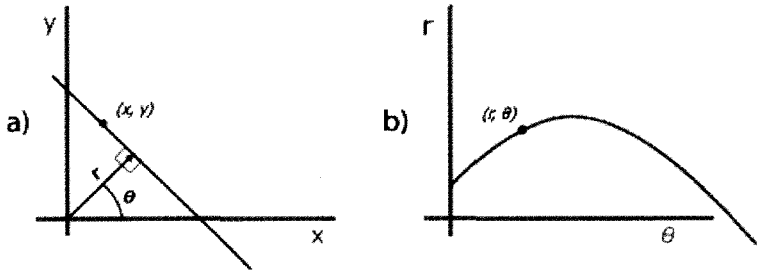

Fig. 3. For every *(x, y)* point in the Cartesian space a) we compute all the lines that pass across it in the parametric space b) using the Hessian normal form representation of a line as $\boldsymbol{r = x * \cos(\theta) + y * \sin(\theta)}$. A line in the Cartesian space is mapped to one point in the parametric space *(r, θ)*. A curve in the parametric space b) represents all the lines that pass through a point *(x, y)* in the Cartesian space a). A point *(r, θ)* in the parametric space b) represents a line in the Cartesian space a) that passes through the point *(x, y)*.

All pairs *(r, θ)* make up a digital curve in the parameter space. That is, a pixel in the original space corresponds to a digital curve in the parameter space. All the pixels belonging to a line in the original space will be mapped to corresponding digital curves in the parameter space such that they all pass through the same point. That point is exactly the reverse transform, mapping of pixel from the second space into a digital line in the original space. The Hough transform is then an accumulation of mappings in the parameter space. For every pixel in the contour, all accumulators in corresponding pixels (from the corresponding digital straight line) in parameter space are increased by one. At the end, these accumulated values are thresholded. A line of certain length corresponds to a pixel in the parameter space of certain cumulative value. The goal of this algorithm is to find points of local maxima intersection of the lines in the parameter space, which corresponds to existing straight lines in the image. The output of this approach is a list of entire line representations without any information of starting or ending points (see Fig. 4).

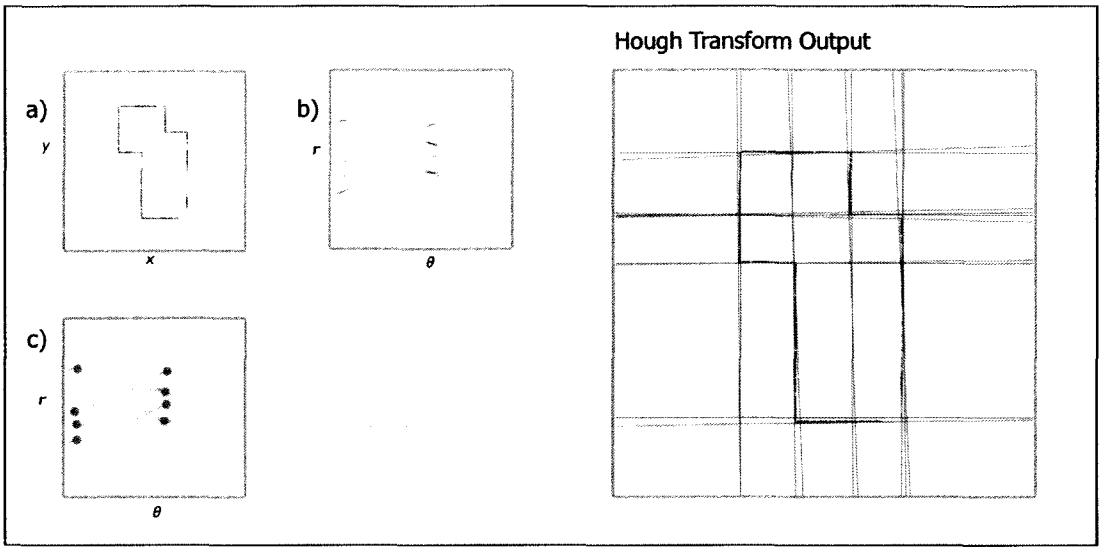

Fig. 4 Hough Transform algorithm. a) Binary image input, each pixel in the image space *(x, y)* is translated to b); accumulator image in polar coordinates. For each black point in the input image, it computes all the pairs *(r, θ)* to create the curves in the accumulator space c); points in the polar space with maxima local values describe lines in the image space output image.

2.3 Hough Lines (PPHT)

PPHT variant of HT, implemented in OpenCV, was designed to speed up the standard Hough transform algorithm. The basic modification is that not all input pixels are mapped to the accumulator 1518 yet still might lead to impractical line segment detection for some images compared to the classical implementation (see Fig. 1 or Appendix A for full examples). The advantage of PPHT lies in its ability to minimize the number of voting operations while almost retaining the quality of the HT output 18.

Probabilistic Hough Transform algorithm is a variant where standard Hough Transform is performed on a pre-selected fraction of input points. PPHT minimizes the

amount of computation needed to detect lines by exploiting the difference in the fraction of votes needed to reliably detect lines with different numbers of supporting points. The fraction of points used for voting need not be specified in advance or by using a priori knowledge, as in the Probabilistic Hough Transform. It is a function of the inherent complexity of data. The algorithm is ideally suited for real-time applications with a fixed amount of available processing time, since voting and line detection operations are interleaved. The most salient features are likely to be detected first. Experiments show PPHT has, in many circumstances, advantages over the standard Hough Transform HT 18.

PPHT variant also uses a binary input image. The main difference with the classical model is that this implementation gives us imperfect output line segment representation instead of entire lines using a voting procedure and some parameter configuration at hand, in order to decide which set of features may form a line segment. This input parameter configuration requires a high degree of a priori knowledge to behave satisfactorily, as they help build the accumulator space reducing the classical set of features to track in the HT model. Chapter 2.4 will shed light about this implementation in OpenCV.

2.4 OpenCV and EmguCV

OpenCV (Open Source Computer Vision) is a multiplatform library of programming functions mainly aimed at real time computer vision 12. This library was originally developed by Intel and used in several applications like Human-Computer Interaction; Object Identification, Segmentation and Recognition; Face Recognition; Gesture Recognition; Motion Tracking, Ego Motion, Motion Understanding; Structure from Motion; Stereo and Multi-Camera Calibration and Depth Computation and Mobile Robotics 13. Its outstanding features and performance implementation makes OpenCV one of the most widely accepted platforms in commercial and academic environments.

EmguCV is an Open Source multiplatform – Gnu Linux, Mac, and Windows – actually a .NET wrapper to the Intel OpenCV image-processing library. It allows OpenCV functions to be invoked from .NET compatible languages such as C#, VB, VC++, IronPython etc. For the sake of consistency, we are going to refer to OpenCV functions but we actually used this .NET wrapper and implemented our solution under C# language. All the image filtering and processing methods used in our solution are computed with this combination of OpenCV and EmguCV.

OpenCV image processing library implements cvHoughLines2 function. The OpenCV cvHoughLines2 function supports two different kinds of Hough-transform-based algorithms: Hough transform HT 4 to retrieve lines as explained in the above epigraph and the progressive probabilistic Hough transform PPHT 15, which is a variation of HT that, among other things, computes an extent for individual lines in addition to their orientation. It is probabilistic because, rather than accumulating every possible point in the accumulator space, it accumulates only a fraction of them. The idea is that if the peak is going to be high enough, then hitting it only a fraction of the time will be enough to find it. The outcome of this conjecture can be a substantial reduction in computational time. Both algorithms are accessible with the same OpenCV function 10. This variant of the Hough transform method returns a list of Segment2D objects, which are line segments with information of starting and ending points inside the image instead of entire lines. Both methods use a binary image (black and white) as input where the white pixels are features to be tracked. Those pixels are translated into the dual space and later on computed to detect straight lines.

The OpenCV cvHoughLines2 function definition computes a binary image in order to calculate lines in images. For both algorithms, the same built-in function is used. The 'method' input parameter stands for the algorithm to be used.

```
CvSeq* cvHoughLines2(
                CvArr*  image,
                void*   line_storage,
                int     method,
                double  rho,          (RR) Rho resolution
                double theta,         (TR) Theta resolution
                int threshold,        (HT) Hough threshold
                 double param1=0,     (GBL) Gap between lines
                double param2=0       (MLL) Minimum line length
            ).
```

The first input parameter for the cvHoughLines2 function is the binary image with the features to track for straight line detection. The second one refers to where the result will be stored. In the case of HT, it will be a set of entire lines whereas for PPHT, they will mean line segments with starting and ending point information. The "method" input parameter defines which method will be enforced for the computations. The following five parameters are labeled as follows. Rho resolution (RR) and Theta resolution (TR) are used to create the accumulator dual space. Hough Threshold (T) is the threshold used by the accumulator to find the points of local maxima. Gap between lines (GBL) is the maximum gap between line segments lying on the same line so as to treat them as a single line segment (i.e. to join them). Minimum line length (MLL) is the minimum line length to detect as a line segment. The two input parameters param1 (GBL) and param2 (MLL) are only used in the PPHT version.

In our document, we are going to refer to the OpenCV computation of the Hough transform method as HL (Hough Lines) in order to detect line segments using the PPHT method and we will denote by HT to the Hough transform algorithm for line recognition in images as presented in the previous epigraph. HL is the implementation used to compare with our proposed algorithm.

2.5 *Gray Scale Conversion*

Gray scale conversion transforms color image to a gray scale image so that colors get their corresponding brightness values in the black and white scale. In order to drop color information and keep a gray shade of the image with its correspondent pixel's bright, we need to map a three-depth color space to a one-depth color space. Any true color image can be represented as a combination of RGB values or channels (R = red value intensity, G = green value intensity, B = blue value intensity). This means that each pixel in the image is encoded as three bytes *–24* bits- color's combination, where each byte ranging from *0* to *255* defines the intensity of each pixel's constituent color. The final result is an image with a one-byte color depth, where only a value from 0 to 255 will define the gray color of that pixel.

The built-in OpenCV gray scale function used in our algorithm uses the perceptually weighted formula 10.

$$I'_{(x,y)} = (0.299)R_{(x,y)} + (0.587)G_{(x,y)} + (0.114)B_{(x,y)}.$$

This formula is applied to each pixel in the input image, resulting in a *29.9%* of the red channel of pixel's image value, a *58.7%* of the green one and finally an *11.4%* of the blue one.

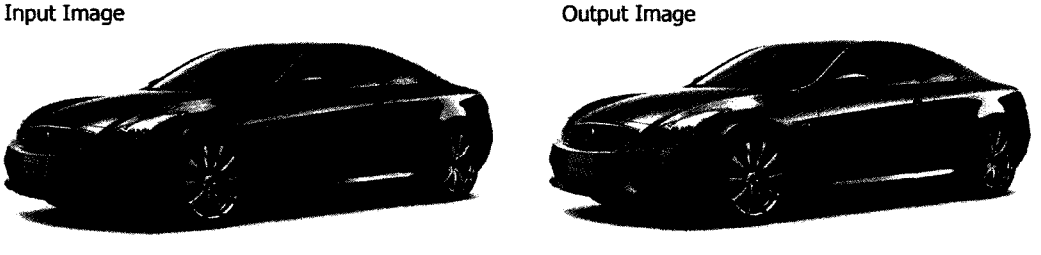

Fig. 5 Gray Scale conversion applied over a true color image – *24* bits RGB image- using OpenCV weighted formula.

2.6 Normalization

Normalization or Contrast Stretching is a process that changes the range of pixel intensity values in an image. Let *[a, b]* be the actual range of the image and we want to stretch it to the new interval *[c, d]*. The motivation is to achieve consistency in the contrast range for a set of pixel intensity values *[a, b]* of data to avoid saturation of the intensity values of the image. This process is linear when each intensity value $I_{(x, y)}$ of every pixel *(x, y)* in an image is scaled from the range of *[a, b]* to the final desired range *[c, d]*. Each pixel's intensity value $I_{(x, y)}$ will be translated into $I'_{(x, y)}$ by the following expression:

$$I'_{(x,y)} = \left(I_{(x,y)} - a\right)\left(\frac{d-c}{b-a}\right) + c.$$

This step in our algorithm is computed automatically. If after the gray scale conversion the histogram does not have values from *[0,255]*, we stretch the values accordingly.

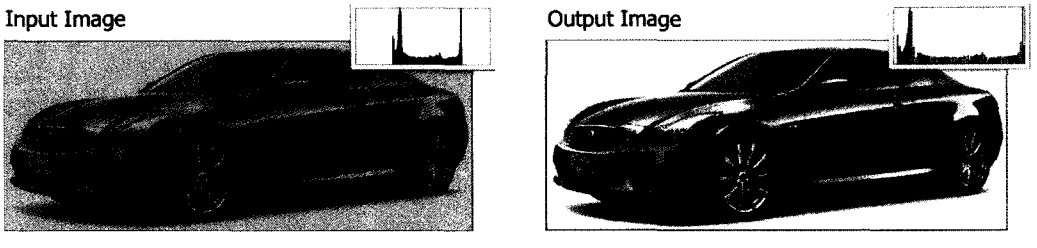

Fig. 6 Left image has an intensity range *[70, 201]*. Right image is the result of normalizing the left image to an intensity range *[0, 255]*. The top right square in both images represent their histogram, where the *x* axis stands for the intensity values from 0 to 255 and the *y* axis denotes the amount of points with the *x* intensity value.

2.7 *Image Filtering*

Image filtering is a technique used in computer graphics and image processing where a new image is created as a result of processing the pixels of the original image. This filter allows us to apply different effects to the original images like anti-aliasing, blurring, feature extraction and so on.

Each pixel in the output image is computed as a function of one or several pixels from the original image, usually located in the vicinity of the output pixel. Image filtering operations are often referred to as convolution operations in literature. The filter or kernel matrix is usually considerably smaller than the real image and with odd dimensions, for example *3x3*, *5x5*, *7x7*, because it has the targeted pixel as its central value. Then the output pixel value is the sum of the multiplications of the original pixel values times their corresponding positions inside the kernel matrix from the input pixel. This kernel matrix is applied to each pixel of the input image, thus returning a new matrix whose values define another image. Fig. 7 gives an example of a filtering operation, Laplace kernel, which is actually applied in edge detection process. There exist different types and sizes of filters used in image processing. This document applies it for blurring and edge detection.

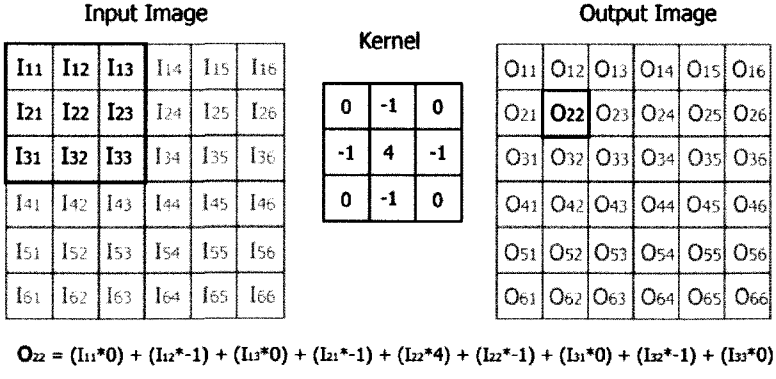

Fig. 7 Laplace Kernel filtering example applied over the targeted pixel I22, to compute O22.

2.8 Gaussian Smoothing

Gaussian smoothing is an image filter that describes the blurring of an image by using a Gaussian function for calculating the transformation to apply to each pixel in the image. This filter is usually employed to reduce noise in the image and as a pre-processing step in order to enhance image structures like edges. This technique produces a smooth blur effect over the original image which resembles an out-of-focus picture. Fig. 8 displays an example.

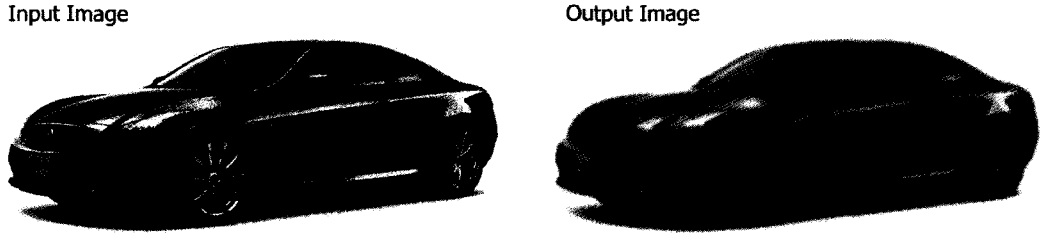

Fig. 8 The right picture is the result of convoluting the input image with a Gaussian kernel of size *3x3*.

In images, a Gaussian smoothing is generally computed by filtering (convolving) the image with a kernel of Gaussian values. The convolution matrix values are calculated by means of a Gaussian distribution. Each pixel in the input image is set to a weighted average of its neighbors, hence giving the highest weight to the original pixel and increasingly lighter weights to its neighboring pixels as their distance from the original pixel grows. The built-in OpenCV 10 function *GaussianSmooth* used in our algorithm entails a kernel size k as a configuration parameter, which defines the $(k \times k)$ Gaussian kernel to be applied on each pixel of the image. This convolution process preserves boundaries and edges in the input image and eliminates isolated pixels with elevated light contrast, which are not needed, for instance, in edge detection processes.

The kernel values are computed using σ, the standard deviation of the Gaussian distribution. The implementation of Gaussian smoothing provides a higher performance

optimization for common kernels, *3x3*, *5x5* and *7x7* with the "standard" $\sigma = 0.65$, and a zero mean 12.

Smoothing or blurring of an image with a Gaussian filter is done by using a *(k x k)* kernel applied to each pixel of the image, thus yielding a new intensity value to each pixel according to the weighted sum of its k^2 neighboring pixels. The resulting image can be regarded as a blurry version of the original gray image, keeping solely the most important edge features inside the image and eliminating some noise in the picture. For gray images (e.g. Fig. 5), the intensity value of the pixels goes from 0 to 255.

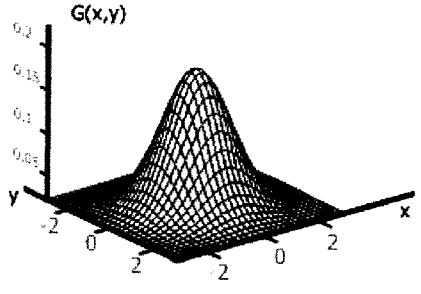

Fig. 9 2D Gaussian distribution example with mean *(0, 0)* and $\sigma = 1$, using function $G_{(x,y)} = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$.

Because of the discrete nature of the image space and the convolution process, we need to truncate the real values to integers in order to get a $k \times k$ filter for odd values of k. An exemplar of a discrete approximation to Gaussian functions for $k = 5$ and $\sigma = 1$ is:

$$\frac{1}{273}\begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}.$$

Then every pixel of the input image is computed with this kernel in order to create a smoothing effect in the image (see Fig. 8).

2.9 Edge Detection with Laplace Operator

In image processing, edge detection is a term used to refer to algorithms whose purpose is to identify points where image brightness changes or, more formally, has discontinuities. These discontinuities are likely corresponding to discontinuities in depth, material properties, surface orientations or brightness variations. Edges in images are areas with strong intensity contrasts. This may lead to a set of connected curves which describes the boundaries of objects in the image.

These methods play an important role in computer vision algorithms and are mainly divided into two categories: search-based and zero-crossing-based. The former is a search for local maxima of the gradient magnitude in a first-order derivative expression and the latter is a second-order derivative expression calculated in the image to find edges. Some of the most important edge algorithms in computer vision are Sobel, Canny and Laplace operators 12310. These algorithms use as input a gray image and output another same-sized image where the edges of the original object are brighter. These grayish outputs are usually thresholded in order to extract the brighter pixels and the edges. Sobel and Laplace outputs are gray images, thus after these convolution processes are computed, a binary threshold is needed. Canny's is a more elaborated edge detection scheme with combination of Gaussian smoothing, edge localization and some thresholding techniques. The output of Canny's algorithm is in fact a binary image rather than the ones mentioned before. Anyways, these three methods compute the same features of interest; i.e. edges.

Sobel's kernel returns first-order derivatives of grey levels in each possible direction. This image filter calculates the gradient of the image intensity at each point separately in

x and y directions; G_x and G_y. The resulting gradient is obtained after combining the two axis gradients using $G = \sqrt{G_x^2 + G_y^2}$ where the two kernels convolved with the original image are:

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

The Laplace operator is however based on a second-order derivative. This method searches for the zeroing values in the second derivative of the image to find edges. For the image intensity function $f_{(x, y)}$, if a given pixel (x_0, y_0) is on an edge segment then $\nabla^2 f_{(x,y)}$ has the zero-crossing properties around (x_0, y_0) in the following way: $\nabla^2 f_{(x,y)}$ would be positive on one side of the edge segment, negative on the other side, and zero at (x_0, y_0) or at some place(s) between (x_0, y_0) and its neighbors.

The built-in OpenCV10 Laplace operator which we exploit in our algorithm is an image filtering process that calculates the Laplacian operator of the image by summing the x and y derivatives using Sobel's discrete operator. Using an aperture size input parameter of *1*, the *3x3* matrix which computes a discrete approximation of the gradient is:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

This type of filter is also called "high pass" (convolution) filter 12. This operator takes a gray image as input. Then, each pixel of the image from the bi-dimensional collection of pixels in rectangular coordinates is convoluted (filtered) using the specified kernel, which is in turn applied to the image from left to right and from top to bottom. Each pixel in the image and its eight neighbors (needed for a *3x3* kernel) are multiplied by the kernel's corresponding value. This gives the final value of the pixel as the sum of all products. Fig. 7 is an illustration of the calculation of Laplace's operator with a *3x3*

kernel. This process creates a new gray image where the edges are the brightest area in the picture.

In order to reduce noise in the edge detection phase, a pre-processing smoothing step is nearly always applied. This trims some irrelevant information while preserving the important structure properties of the image. Since the convolution is associative, the Laplacian and Gaussian filters could be combined into one convolution filter ahead of image processing. Thus, the filter is the second derivative of the Gaussian function. Smoothing or convolving an image with Gaussian filter and then determining its Laplacian operator is the same as convolving the image with the Laplacian of Gaussian (LoG) operator in order to detect edges 9.

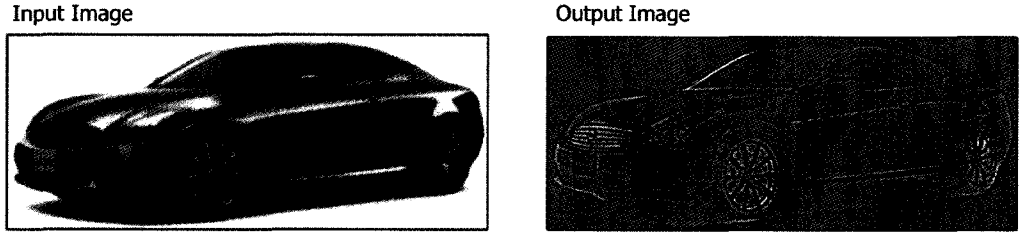

Fig. 10 Edge detection algorithm using Laplace's operator with kernel size = *3*. It uses a gray image as input whose edges are represented by brighter pixels in the output image (pixel intensities are inverted).

After the edge strength is computed using the Laplace's operator, our next step is to apply a threshold to decide whether the image points resemble edges or not.

2.10 Image Thresholding

Thresholding is a very simple method in image processing algorithms. There are several categories of thresholding methods, but we are only focused on the binary threshold. This function is typically used to get a binary image out of a grayscale image. After a convolution kernel is applied to an image, the gray image is transformed into a binary (black and white) image by means of a threshold. The white pixels/regions are of interest to us; they represent the brightest pixel area of the gray input image. In a binary thresholding setting, such pixels are turned into white ones and the remaining into black ones.

After we have these grayish regions in the input image, we scan every pixel's intensity value and compare it with the threshold value T. If this intensity value is greater than T, then we assign a white color to it, otherwise we assign a black color.

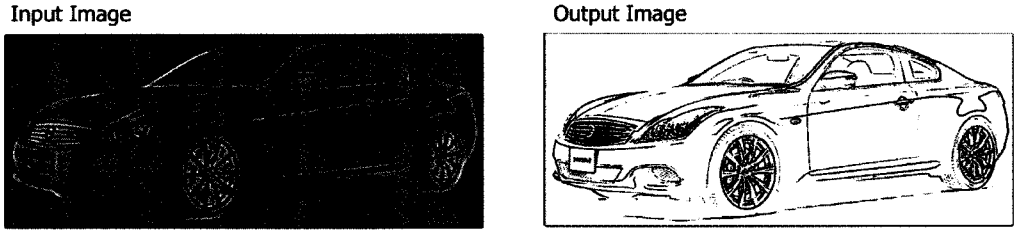

Fig. 11 The binary threshold takes the brighter pixels with $I_{(x, y)} \geq T$ in the gray image and assigns them a white color; the rest of the image is black. This output image example has been inverted for adequate visibility purposes. The original image is a white edge pixel's region over a black background.

2.11 Unsupervised Automatic Threshold

It is significant in image processing to select an adequate threshold of gray level for isolating objects from their background. This will lead us to retrieve better edges and, consequently, better line segments. A good nonparametric and unsupervised method of automatic thresholding is presented here 19 and used as part of our LDC algorithm.

A variety of techniques have been proposed in this regard to overcome the automatic threshold difficulty. The valley sharpening technique 20 restricts the histogram to the pixels with large absolute derivative values (e.g. Laplacian or gradient). The difference histogram method 21 selects the threshold at the gray level with the maximal amount of difference.

Choosing a threshold value for real pictures is very cumbersome because, first of all, there is no ''goodness" threshold evaluation in most methods so far proposed. In an ideal case, the histogram has a deep and sharp valley between two peaks representing objects and background, so the threshold can be chosen at the bottom of this valley. However, such valley and its peaks are not always evident. In most real-world images, it is very challenging to find the bottom valley due to, for instance, its flat and broad shape, or its noisy nature, or it having two peaks with uneven heights.

The technique we are going to introduce proved to be a simple solution with a short processing time and can be categorized as an unsupervised method with excellent results. The procedure utilizes the zeroth and the first-order cumulative moments of the gray-level histograms. In other words, it utilizes the histogram of gray images to select a threshold value in an unsupervised fashion. Such threshold must separate the background from the objects and finally yield a good edge's image result without any input parameter. The main idea of this procedure is to dichotomize the gray image histogram into two classes by a threshold value k, which is calculated by maximizing a function using the actual intensity values of the histogram.

The formulation of this procedure is given as follows. Let the pixels of a certain picture be represented in L gray level as *[1, 2... L]*. The number of pixels at level I is denoted by n_i and the total number of pixels by $N = n_1+n_2+...+n_L$. The gray-level histogram is normalized and regarded as a probability distribution:

$$p_i = \frac{n_i}{N}, \quad p_i \geq 0 \; and \; \sum_{i=1}^{L} p_i = 1. \tag{1}$$

Now, we want to dichotomize the pixels into two classes C_0 and C_1 (background and objects, or vice versa) by a threshold at level k; C_0 denotes pixels with levels *[1... k]*, and C_1 denotes pixels with levels *[k+1... L]*. Then the probabilities of class occurrence and the class mean levels, respectively, are given by:

$$\omega_0 = \Pr(C_0) = \sum_{i=1}^{k} p_i = \omega(k). \tag{2}$$

$$\omega_1 = \Pr(C_1) = \sum_{i=k+1}^{L} p_i = 1 - \omega(k). \tag{3}$$

$$\mu_0 = \sum_{i=1}^{k} i \Pr(i|C_0) = \sum_{i=1}^{k} ip_i/\omega_0 = \frac{\mu(k)}{\omega(k)}. \tag{4}$$

$$\mu_1 = \sum_{i=k+1}^{L} i \Pr(i|C_1) = \sum_{i=k+1}^{L} ip_i/\omega_1 = \frac{\mu_T - \mu(k)}{1 - \omega(k)}. \tag{5}$$

where

$$\omega(k) = \sum_{i=1}^{k} p_i. \tag{6}$$

and

$$\mu(k) = \sum_{i=1}^{k} p_i. \tag{7}$$

are the zeroth and the first-order cumulative moments of the histogram up to the k^{th}-level, respectively, and

$$\mu(L) = \sum_{i=1}^{L} ip_i. \qquad (8)$$

is the total mean level of the original picture. Then the between-class variance and the total variance of levels, respectively are:

$$\sigma_B^2 = \omega_0\omega_1(\mu_1 - \mu_0)^2. \qquad (9)$$

$$\sigma_T^2 = \sum_{i=1}^{L}(i - \mu_T)^2 pi. \qquad (10)$$

Finally, the optimal threshold k^* would be the value of k that maximizes σ_B^2.

$$\sigma_B^2(k^*) = \max_{1\le k\le L} \sigma_B^2(k). \qquad Refers\ to\ (9)\ and\ (2) - (5).$$

This value can be computed explicitly using (9) and (2)-(5) or by means of the simple cumulative quantities:

$$\sigma_B^2(k) = \frac{[\mu_T\omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]}. \qquad Refers\ to\ (6)\ and\ (7).$$

Then, we need to find the k^* value that maximizes σ_B^2 by evaluating k from *0* to *255*. Those are the values of the gray-level intensities in an image. From the respectively *256* values, the maximum will correspond to the optimal threshold value k and the one we are going to use in our algorithm to compute the binary threshold after the Laplace operator is applied.

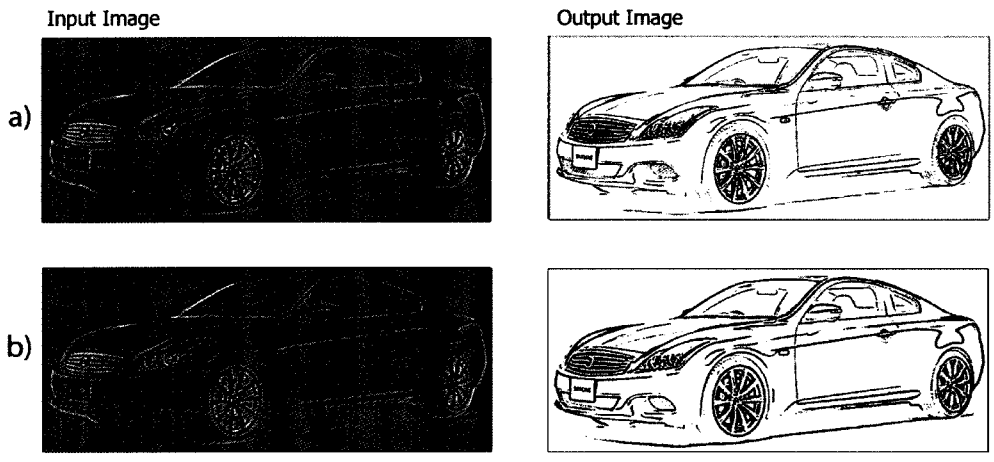

Fig. 12 Supervised and unsupervised method of binary threshold. Fig. a) is a sequel from Fig. 11, where the threshold parameter was selected in a human supervised environment. The value selected was *146*. Fig. b) is an example of the nonparametric and unsupervised method described above. All other parameters till the computation of the gray image are the same for this test. The computed value for this method was *138*.

2.12 Connected Components Labeling and Contours

Connected component labeling is a technique to process binary images in order to detect connected regions. This process goes over all the pixels in a binary image and its neighbors of the same color, connecting them with the same label. This labeling process is only applied to the white pixels, representing the edges in the image. The OpenCV library gave us a set of built-in functions to retrieve contours and connected components. An OpenCV contour is a list of sorted points that in a manner represent a curve in an image. These contours are represented by sequences in which every entry encodes information of the next point on the curve. The sequence of points then forms a cycle, which means that the starting point of the list and the ending point of the list will be the same pixel. The underlying data structure for implementation purposes is a list. A

contour is a list of finite ordered points which envelops connected components. In case of edges, the list normally traverses it twice (see Fig. 13).

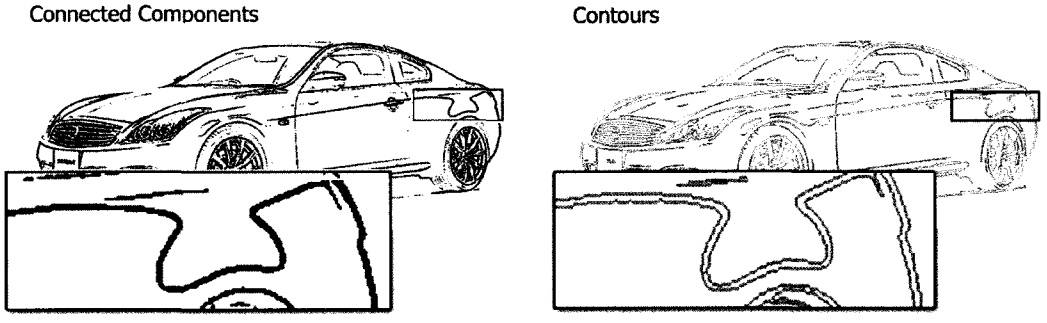

Fig. 13 Contours are bordering each connected component.

Due to the different and complex connected component situations, OpenCV treats these contours as a Hierarchical tree, keeping track of the exterior and interior boundaries. OpenCV's *findContour* built-in functionality gives us the opportunity to retrieve the contours of the connected components in different ways: EXTERNAL-only external contours-, CCOMP -connected components-, TREE – Hierarchical representation of all contours in the image-, and LIST – a list of all contours, external and internal, without the hierarchical representation of the tree-. In our algorithm we use LIST, because we do not need the hierarchical structure and we want to scan every external and internal contour of all connected components in order to find line segment patterns.

Using the binary images as the source, these contours and connected components are extracted using labeling scanner techniques, classifying white pixels that are next to each other with the same label (Fig. 13 shows contours over connected components). The list of points of a contour is a left-to-right, top-to-bottom list of ordered points (clockwise). The starting point and the points inside the contours might be different due to dissimilar parameters used to retrieve the initial binary image, but they all keep this pattern and create a cyclic list of points which model the contour boundaries of objects 1010.

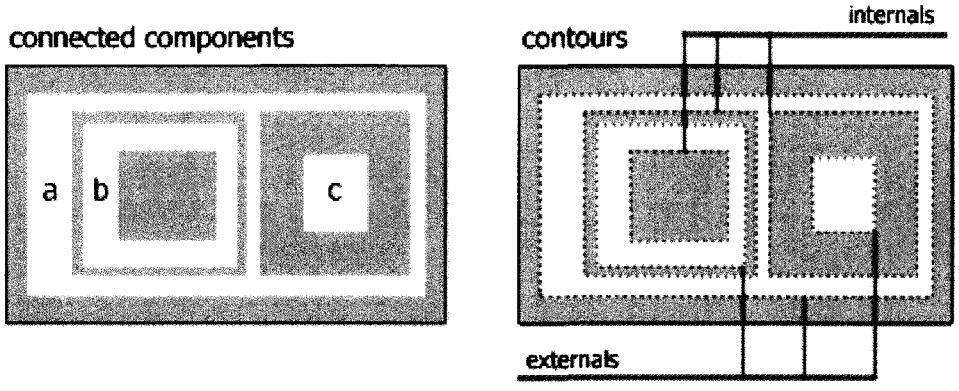

Fig. 14 Connected components *a*, *b*, *c* (white regions) are bordered by their external and internal contours.

Edges can be treated as connected components and regions themselves. The contours are one-pixel wide, where each pixel has information on the next one. The complete list of pixels inside a contour will form a cycle describing the external or internal surface of the connected component or the edge. If we consider the connected component *a* in Fig. 14 as a one-pixel-wide edge, then this edge will be represented by one external and two internal contours, as shown in the rightmost picture of the same figure. In the case of *c* (a bold, white area with no holes), the contour representing this area will be an external contour only.

The OpenCV's built-in *findContours* function is not aware of edges in images. This mean that for a contour sequence of points, an 'edge' is just a very thin 'white' area. As a consequence, for every exterior contour there will be an interior contour that almost exactly matches it. This hole is actually just inside the exterior boundary. It can be thought of as the white-to-black transition that marks the interior side of the edge 10.

An inconsistent terminology for connected component labeling exists, like region labeling, blob discovery or region extraction 1713. Connected component labeling is a widespread technique employed in computer vision to discover connected regions in

binary digital images. These connected areas are uncovered by using the intensity values of the white pixels and looking for connected neighboring pixels. To archive this goal, several iterative and recursive algorithms and implementations have been put forward.

Relatively simple to implement and understand, the two-pass iterative algorithm iterates throughout the two-dimensional representations of the digital image data 16. The algorithm traverses the matrix of pixels by labeling the vertices based on their connectivity and relative intensity values of their neighbors –scanning for white pixels having other white pixels among their eight neighbors. Basically, the algorithm performs two passes (but it could be devised with just one) over the image: one is for record connectivity and assignment of transient labels –it might be an integer number- and the second one for replacement of the temporary labels by those of their equivalence class and final output. A high-level description using two passes is described below in more detail.

High level description of the Labeling process:

The first pass:

1. Iterate through each element of the data by column, then by row
2. If the element is a white pixel
   a. Get the neighboring elements of the current element
   b. If there are not white neighbors, uniquely label the current element and continue.
   c. Otherwise, find the neighbor with the smallest label and assign it to the current element.
   d. Store the equivalence between neighboring labels.

The second pass:

1. Iterate through each element of the data by column, then by row
2. If the element is a white pixel
   a. Re-label the element with the lowest equivalent label.

Some of these steps in the above workflow can be merged for better efficiency, thus allowing a single sweep over the entire image. Multi-pass algorithms also exists, some of them run in linear time relative to the number of pixels in the image, such as the one used in the OpenCV library for computation of connected components 17.

Chapter 3: Line Segment Detection and Linearity

This chapter will cover our LDC algorithm in order to retrieve straight line segments from color images with linear execution time in number of pixels in the image. Here, we will explain each step of our algorithm solution using an image from our 30 image set (Appendix A) experimental data. Then, we can follow input and output for each step in this algorithm. The algorithm was implemented under OpenCV and EmguCV using the C# language on .Net platform. The analysis and experimental comparison results data between LDC and the OpenCV built-in Hough Line algorithm are in the following Chapter 4.

3.1 Overview of the LDC Algorithm

The input of our algorithm consists of: the true color image (of any resolution) and 3 configuration parameters (to be discussed in appropriate epigraph). They are labeled as follow:

- Gaussian kernel size (GKS),
- Laplace aperture size (LAS),
- Minimum length contour (MLC).

GKS and *LAS* are the sizes of the Gaussian's and Laplace's kernels, respectively, to convolute the image. *MLC* is the minimum amount of pixels that a contour must have to be accepted as a line in our final Segmenting Contour step. Therefore if a discovered contour has less than *MLC* pixels than it is not processed and no line is extracted from it. Otherwise it is passed to the next phase. This value was fixed to *MLC=50* in our tests, where the resolution are from *250x250* to *2048x1536* and always produced visually

satisfying results. The output of our LDC algorithm is a list of recognized connected lines in the image, each with its beginning and ending pixels, and all the intermediate pixels, with single pixel width.

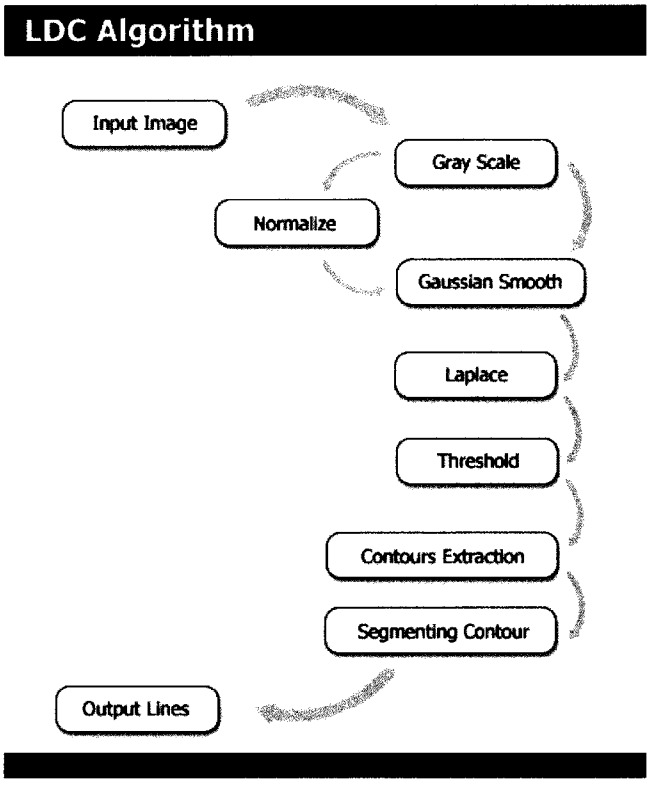

Fig. 15 LDC algorithm's steps used to compute straight lines segments from true color images.

Our LDC Algorithm is composed of 7 steps: Gray scale conversion, Automatic Normalization, Gaussian Smooth, Laplace operator, Automatic Threshold, Contours Extraction, and Segmenting Contours (see Fig. 15). The first 6 steps used are well known image processing algorithms and the last step, Segmenting Contour, is our implementation to extract straight line segments of the objects inside the image. It is the main novelty of this thesis.

We will first briefly describe the main idea. The color input image is passed through several image processing algorithms. The image is converted to gray scale; possibly

normalized, smoothed, edges are extracted and thresholded to produce a binary image input. Contours are then created for each candidate edge. After these steps, we scan the list of points of the contours with time complexity *O(n)* (where *n* is the total number of pixels in all of the contours), looking for segments, or sections of the list of pixels which gives the notion of linearity. The final result will be a list of line segments; each segment will be represented by a starting and ending pixel position in the image space.

The image in Fig. 15 is a color image, *24* bit depth with a resolution of *1024x420* and spatial resolution of *92* dots per inch. It will be used to illustrate all the steps in the LDC algorithm.

Fig. 16 Example input image to the LDC algorithm. This image has a resolution of *1024x420* and *24* bits color depth.

3.2 Gray Scale Conversion

Our first step is to convert input image color image to a gray scale image using OpenCV *CvGray* built-in function. This formula gives an intensity value to the output image from the 3 channels of the original image. The value of the gray pixel is a number between *0* and *255* and it is computed as the *29.9%* of the red component of the real image, *58.7%* of the green, and *11.4%* of blue channel of the same pixel in the input image. The result will be the same image but only with one channel, instead the

composite RGB. This function takes a color image and merges the *3* color channels in a gray one (see Fig. 17).

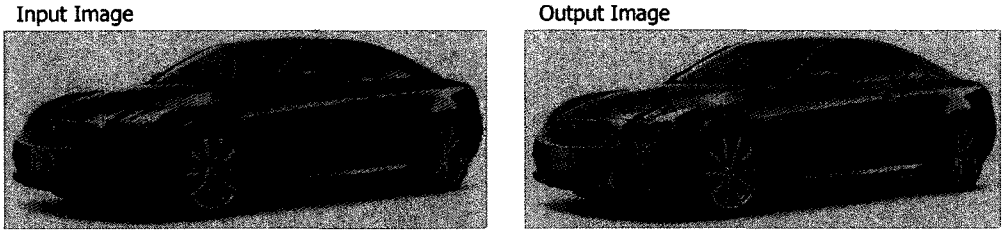

Fig. 17 Gray Image obtained after applying the OpenCV weighted function.

3.3 Normalization

The normalization procedure is generally optional, but we applied it automatically; not all the pictures need to be normalized. Automatic normalization is done by stretching the gray image histogram to its minimum and maximum values *[0, 255]*. If the minimum value of the gray image's histogram is greater than *0* or if the maximum is less than *255*, then the image is normalized to the range *[0,255]*. Edge detection may be difficult if image is affected by the illumination problems. To aid with this problem, we add some pre-processing steps in order to create higher intensity contrasts. Normalization changes the intensity of each pixel in the image. This process might assist further edge detection process. However sometimes it may be somewhat detrimental. Normalized images sometimes give more accuracy but in other cases may introduce noise in the image. Unfortunately we did not find any deterministic algorithm on deciding whether or not to normalize. As for the other input parameters of this and other algorithms, it is based on the human perception.

Our example gray input image has an intensity range *[70, 201]* then normalization will take place. This process will take the intensity value from each point of the gray

image and subtract it *70*; after this each pixel is multiplied by *255/201* and rounded. This will produce an image in final range *[0, 255]* in order to get more contrast in the image.

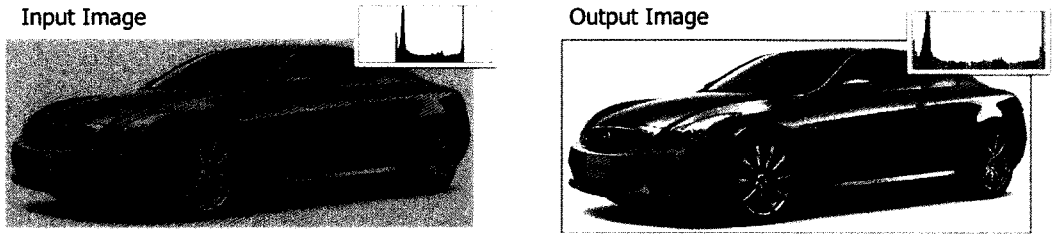

Fig. 18 Histogram and image after normalized from *[70, 201]* to a *[0, 255]* range (output image).

We can see in this case how the contours of the car are better shaped after the normalization step, since the background and the car object have more contrast. We can note sharper edges in the car contour and its components in the output image.

3.4 Gaussian Smooth

After normalization we can appreciate a better-contoured image, but there are still some aspects in the image that may introduce undesired result in the edge detection. The noise in the image can produce undesired edges to analyze.

The Gaussian smoothing step is used to remove noise in the image and leave only the main contours. This blur process has the particularity that it reduces useless information to process and leaving the more important boundaries of the object in the image. The smoothing is applied over the gray image from the normalization step output. If the image is not normalized then the output of color to gray scale conversion step process is taken as input of this step.

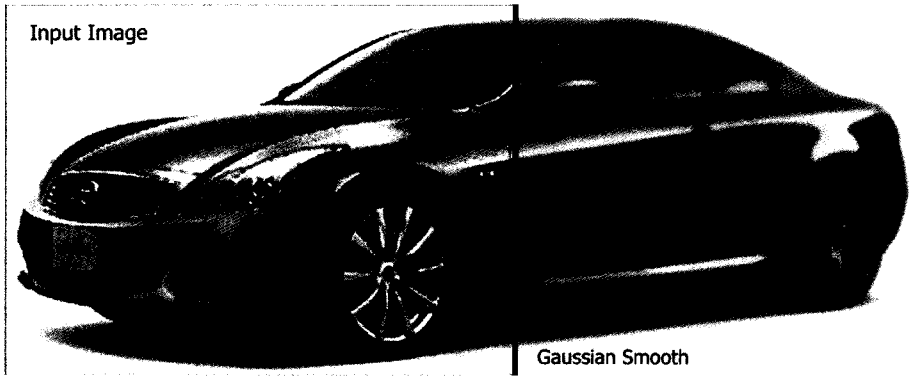

Fig. 19 Gaussian blur applied to the normalized image (right half of the image). The kernel size parameter used was 7.

The Gaussian kernel matrices to compute the Gaussian blurring/smoothing are computed inside the OpenCV image processing library. OpenCV give us only the opportunity to configure the size of the kernel to use over the original image. After this size is defined, like *3, 5, 7, ...19* (*GKS* input parameter; we have experimentally determined in our image testing set that *19* is a good upper bound) the image is convoluted with the predefined Gaussian kernels in the image library corresponding to the input parameter specified. Among all the tests done in our research, we concluded that a kernel size larger than *19* is not necessary and is not used in our experimental image set. This will lead us to use odd's numbers from *3* to *19* only; which reduce a lot the combination of input parameters for our algorithm.

Normalizing and Gaussian smoothing steps might yield different results capturing the full real edges in the image based on different input parameters and images. However, these methods are widely used with excellent results in computer vision algorithms.

3.5 *Laplace Operator*

After several tests on edge detection algorithms like Sobel, Canny, Predefined Convolution kernels, and Laplace operator, we found that the Laplace operator was the one that produced the best results for our goal. It had sharper edges describing the boundaries of objects for our purposes (see Fig. 20). Also, it only has one parameter to configure, the aperture size of the kernel, to apply to the image. This aperture size parameter must be *1, 3, 5* or *7*. In all cases except *1*, the aperture size will define the *k x k* kernel where *k* = *aperture size*. For the case of aperture size = *1* the kernel used is the *3x3* kernel (see Fig. 7). 12

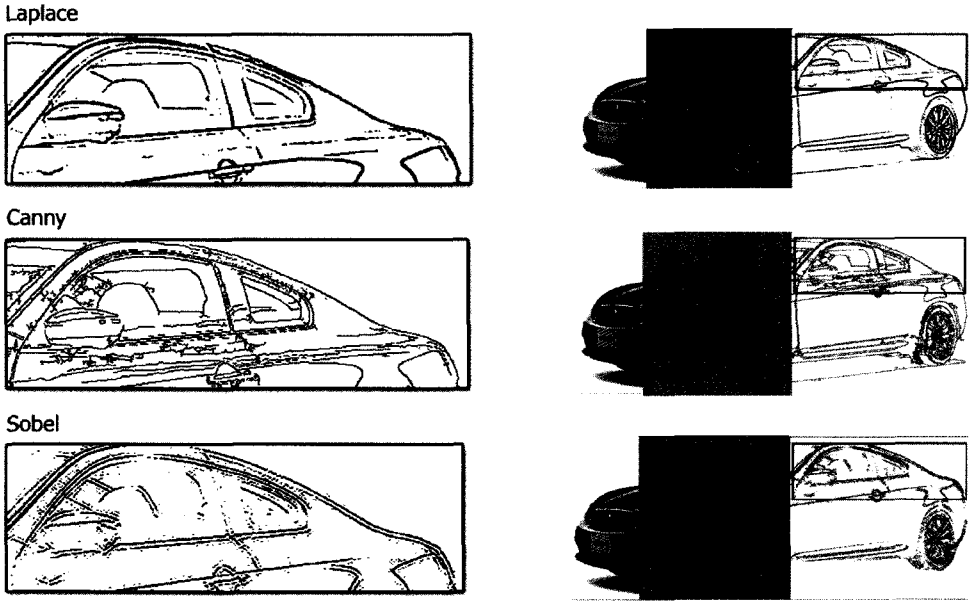

Fig. 20 Edge detection algorithm examples: Laplace, Canny and Sobel. We applied a binary threshold to each image for better view of the edges.

The input image for this step is the gray output image from the Gaussian smoothing step. This image as a matrix of point in image space will convolute with the appropriate

kernel defined by the input parameter Laplace aperture kernel size. This method will make brighter the image zones where the edges are founded.

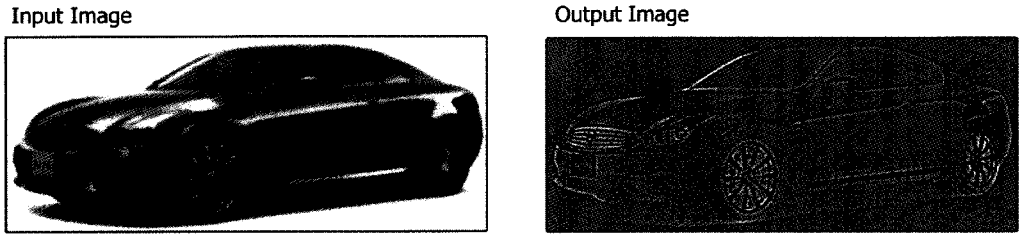

Fig. 21 Laplace operator applied over the output image from the Gaussian Smooth (input image). The output image is the result of applying a kernel aperture size of *5*. This is the result of our image after the previous steps.

3.6 Automatic Binary Threshold

After normalization, Gaussian Smooth and the Laplace operator steps, we obtain a gray scale image, which we apply an automatic binary threshold to create a binary image with two color's regions (white/black). The white color would describe the boundaries of objects and black will be the background. We can see the different connected components among the white regions. All the pixels with intensity value greater than or equal to the threshold parameter in our algorithm become white, representing brighter areas in the image and at the same time the edges. The other pixels become black. Fig. 22 shows an example of the image we traced from the beginning (again, output image has inverted black and white pixels for viewing convenience). The threshold value was computed minimizing the formula explained in Chapter 2.11.

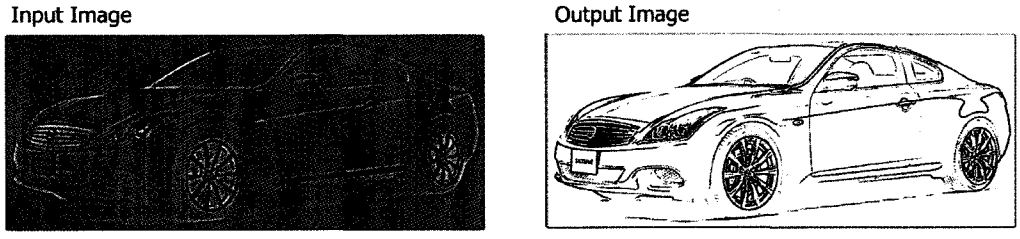

Fig. 22 After apply automatic threshold we have the edges of the original image. The output image is the result of applying an automatic threshold value of *138* over the input image (Output image from Laplace operator).

3.7 Extracting Contours

The binary image resulted after the threshold step will be the input image for this phase of our algorithm. We want to extract information about the edges (white areas of the image). We are going to retrieve the list of all contours, external and internal using the *findContours* method of the OpenCV library.

Contours are a list of sorted point's pixels inside the image where each pixel has information about the next neighboring point. These contours surround the different white connected components (displayed black, for display purposes) in the binary input image (see Fig. 23). Because of the complex object we might find inside an image, we are going to retrieve all the internal and external contours in order to analyze them. The points inside the contours are clockwise oriented. As part of our algorithm, we introduced a threshold parameter *MLC* (Minimum Length Contour). This parameter's function is to filter contours with at least a threshold value for the number of pixels. This means that any contour, with an amount of pixels less than this parameter's value will not be processed in order to extract lines. Because we are looking for representative's line segments, this parameter will help us to reduce line and contours computations which do not lead us to find an important line segment inside the image.

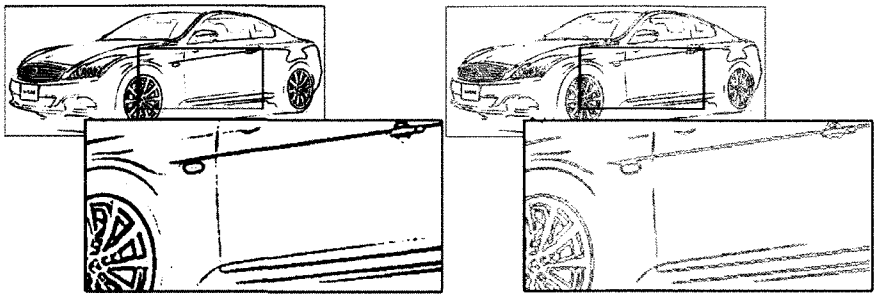

Fig. 23 Contours are represented by the green color that surrounds the black pixels from the connected components which are the edges. The minimum length contour (MLC) of *50* pixels is applied. (Image colors are inverted for better viewing, edges are white colored)

3.8 Repeated Segment Directions on Image Contours

3.8.1 Overview of the algorithm

Our next and final step is to analyze this list of points/contours in order to find straight lines under a discrete linearity criterion. These lines might not be perfect lines, but they are as close to being linear as possible, so that we can recognize them as important line segments. We are going to segment the ordered list of points, dissecting them by line segments that we are going to extract. We can see this as some kind of poligonalization with a little difference: we are going to drop the points that we don't recognize as lines or as parts of a line. In other words, not all the points inside the contour will belong to a line segment classification, only the ones that are accepted will be part of the output of the algorithm.

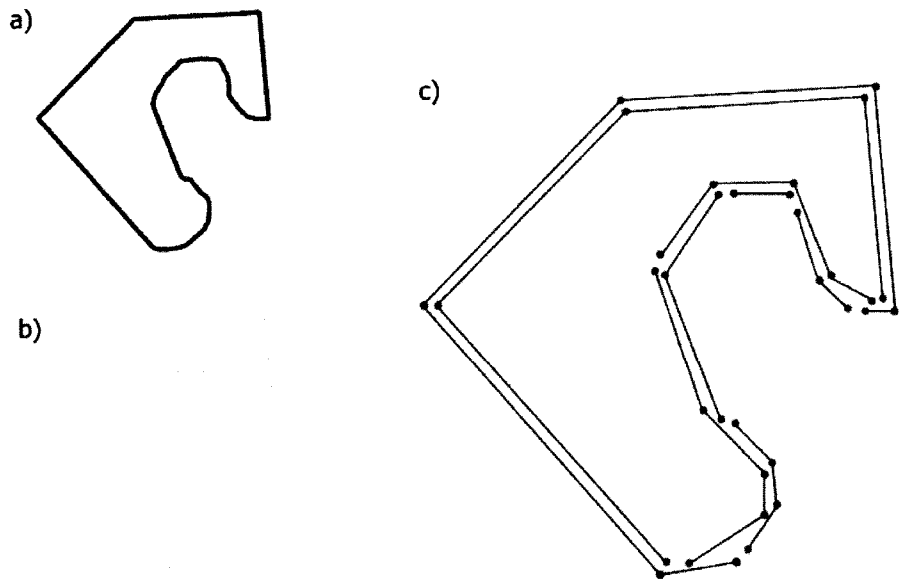

Fig. 24 Example a) gives us an edge extracted from an image. The b) image is the two contours extracted from the original picture a). The straight lines extracted from the contour with the endpoints are in picture c). We can see how the extracted lines resemble an image 'poligonalization' but we can dismiss some point from the contour.

Our algorithm, in this step identifies line segments with any orientation inside the contours in time complexity $O(n)$, where n is the number of points on the contours we are analyzing. Therefore this final step does not compromise the execution time for the whole line segment detection algorithm. The total time complexity is dominated by preprocessing steps that require processing of every pixel in the image, and has time complexity $O(N)$, where N is the number of pixels in the image.

We are processing the contour in the order given by the OpenCV built-in *findContours* algorithm. Each contour is divided into fragments. Each fragment contains $\Delta=5$ consecutive pixels. Neighboring fragments share endpoints. A line segment consists of several consecutive fragments. Some fragments will not be associated with any line segment, and each fragment may be part of only one line segment. In our implementation, we used a fragment size of $\Delta=5$ pixels. Because of sharing endpoints,

each fragment has $\Delta+1$ pixels, including endpoints. The rest of description is dependent on this choice; other choices require designing appropriate fragment classification scheme before detecting line segments.

Our algorithm will classify each fragment using its slope, and will then attempt to identify repeated patterns of the same slope to create a (straight) line segment. A segment will keep all the information about the contour pixels and two main points: the starting pixel (x_0, y_0) and the ending pixel (x_1, y_1). These endpoints are the first pixel of the first fragment and the last pixel of the final fragment in detected line segment.

The number of fragments inside a contour is equal to the truncated ratio of the number of pixels (length) on the contour and the fragment size Δ:

$$fragments = \left\lfloor \frac{|Contour|}{\Delta} \right\rfloor.$$

Each fragment will be classified according to its slope into nine possible categories; from 0 to 8. This produces an integer sequence of line slopes for further processing.

3.8.2 Slope Classification and Fragmentation

Fig. 25 shows basic idea for the classification of fragments into nine discrete slopes. The small black square in the center represents the initial point of the segment of the line we are analyzing and the matrix of image space points represent all the possible points a step of size 5 can end starting from the center. The four diagonal lines represent classifications of continues line slopes into 8 discrete values (regions between line slopes).

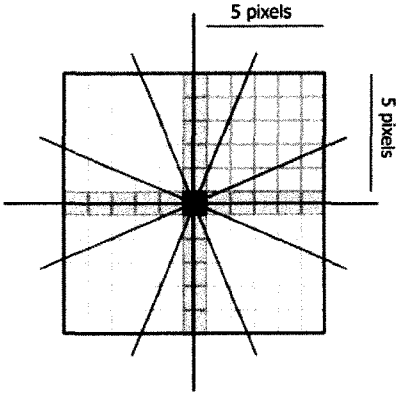

Fig. 25 Classification of fragments into discrete slopes.

Detailed classification is shown in Fig. 26. Diagonal lines are discretized to produce division of *11x11's* matrix into *9* areas, labeled *0, 1...8*. The central *3x3* region is labeled *0* and represents Class *0*. Classes *1-8* correspond to eight approximate directions respectively: east, south-east, south, south-west, west, north-west, north, and north-east. Each fragment is classified in the following way. The beginning of the fragment is placed in the center of the *11x11* matrix (used when $\Delta=5$). The fragment is classified according to the location of its endpoint in this matrix, and receives the label associated with it. Class *0* is a 'neutral' direction, or a lack of direction. It represents fragments that do not make sufficient progress in any direction, and can be treated as a kind of loops. This class was introduced due to the irregularity and noise on the edge detection step. Progress with distance of at least *2* from the center in one of directions suffices to classify slopes into one of *8* Classes.

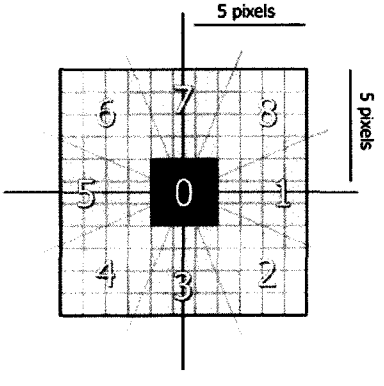

Fig. 26 Classification matrix with *9* Classes.

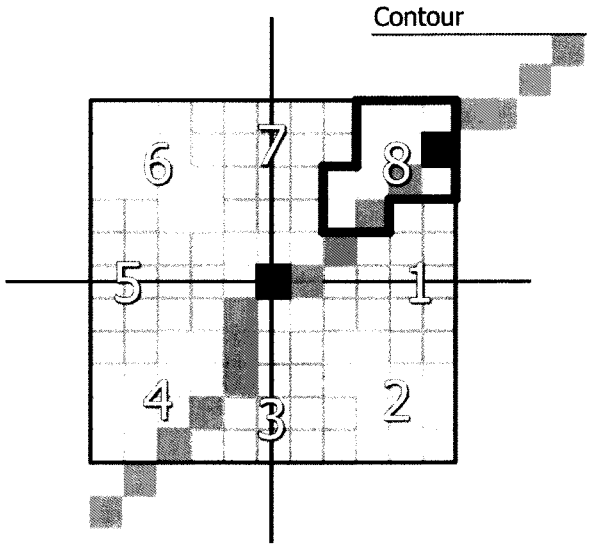

Fig. 27 The green line represents a section of a contour analyzed by the discrete classification matrix. Two black pixels inside the contours indicate the starting point and ending of a fragment; from left to right. The final classification of this fragment representing for those *2* points is Class *8*.

If a fragment begins with pixel located at *(x, y)* then its classification depends on the location of its endpoint, which could be anywhere within $[x \pm \Delta, y \pm \Delta]$ neighborhood. The endpoint is translated by *(-x,-y)* and located inside the *11x11* matrix to classify. The location of intermediate pixels does not impact the classification. For example, in Fig. 27, one of intermediate pixels belongs to a different region (Class*1*).

One reason to deal with this linearity classification instead of the normal representation of a line $(y = mx + n)$ which obtains the *m* value classification is that we wanted to have a faster algorithm instead of dealing with the high complexity of calculation of sine and cosine. Due to the discrete nature of the media, lines with a slope near $90°$ and $0°$ will cause a big noise in the calculation using the normal model of representing a line. Our basic classification idea may be fine tuned to use more discrete classes, e.g. for higher values of Δ.

3.8.3 Merging Fragments into Segments.

The sequence of fragment classification values will be further processed to identify line segments, corresponding roughly to a sufficient number of repeated fragment slopes. Our criteria to detect line segments from fragments are:

A). A segment has at least X fragments of the same slope classification.

B). Two segments with the same slope classification, separated by less than X fragments with different classification, will create a single segment with all the fragments from and between both segments.

C). The Class *0* slope classification is a neutral position for the algorithm. It will not interfere in the classification of any segments; it will be treated as being same class as the previous class in the sequence.

D). If the starting fragment and final fragment of a contour have the same classification then they join in a same segment with length equal to the sum of both.

Fig. 28 illustrates classification process for fragments. This representation will help us to explain some of the former criteria above.

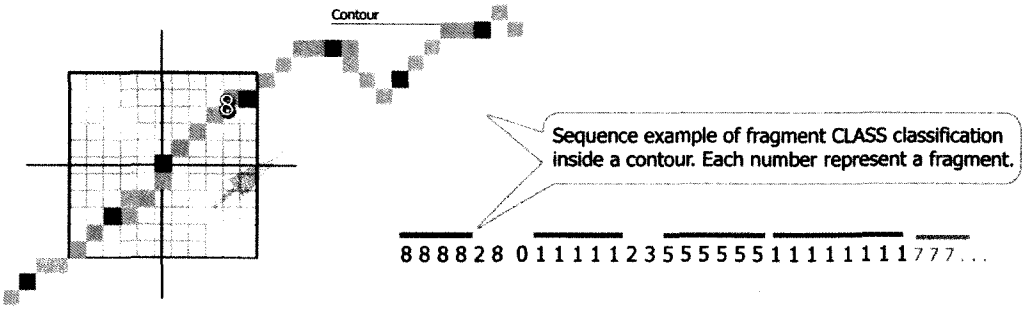

Fig. 28 Producing sequence of fragment classes. Green lines above the class number represent line segments recognized in the patterns.

Our algorithm uses an X value of 3; this means that for at least 3 consecutive Class value fragments are needed to declare it a segment. Fig. 29 shows some examples of extracting line segments from sequence of classified fragments. In Fig. 29; first row, 4 lines segments are detected, based on rule A), repeated patterns of 8 (line segment '888888'), 1 (line segment '11111'), 5 (segment '555555'), and 1 (segment '11111111'), respectively.

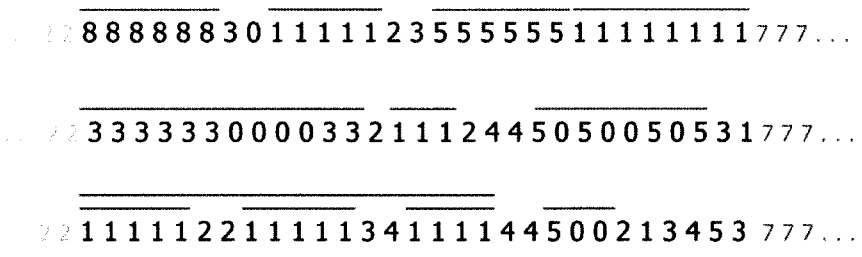

Fig. 29 Segment extraction from fragments, using rule A (the first row), rule C (the second row), and a mix of rules B and C (the third row).

In the second row in Fig. 29, the sequence '0000' is inserted into larger fragment '333333000033', using rule C. Similarly,'0', '00', and '0' are inserted into '5555' to produce line segment '50500505'.

In the third row, rule B is applied to produce a single segment '1111122111113411 11' out of initially produced segments '11111', '11111' and '1111', since inserted '22' and '34', used to merge them are shorter than minimal length $X=3$ needed to 'break' the sequence. In the same row, '500' is also declared a line segment (rule C). This corresponds to the current implementation and is subject to discussion, and can be changed later. We opted for this option to preserve simplicity and to allow segment extensions by single pixel distances per fragment. It can be also perceived as line segments with somewhat shorter length than the minimal specified which does not necessarily means erroneous classification.

We will now describe the segment construction from fragment repetitions more formally, via a flow chart and algorithm pseudo-code. The input for this *fragment_repetitions(f)* is an array f of m fragments with values in range *0-8*. The output will be two lists of ending points of p detected line segments, the beginning of line segments b_p, and the ending of the respective line segments e_p, where p is the number of detected line segments. Note that the whole algorithm has more elaborate output S_p of line segment detected, including list of pixels between endpoints, and their positions, which is not shown here for simplicity.

The algorithm will be first described as a flow chart, which shows how the number of repetitions Q of same fragment classifications is modified and used to create line segments. The value of Q can be treated as a state, in a kind of finite automata (note that the flow chart is not a finite automation in the strong mathematical sense). Q runs from *0* to *4*, where *0* is the initial state and the values of *1*, *2* and *3* represent the amount of consecutive fragments found with the same slope. The 'final state' value $Q=4$ indicates a new ending point of the line segment. This state is added to be more readable our 'fragments repetitions' solution; this means that a change of slope. After we compute a new line segment we jump immediately to $Q=1$ and continues parsing the input sequence.

Our goal is to find sequence of at least 3 fragments with the same Class classification slope. Then we are going to parse the initial list of fragments f_i, to give an output of line segment detected in this contour – list of fragments -.

Every time we read a new Class value for any f_i fragment we are going to keep the Q state variable up to date. This Q state will change as long we detect consecutive sequence of the same slope or not. This will keep us informed how many fragments consecutives have the same slope. A Q value of *1* will say that a beginning position of a fragment is founded. Each time we reach a Q value of *1* we actualize the beginning position of the line segment. The values of *2* and *3* for the Q variable help us to count the amount of fragments with the same slope. The particularity of a $Q = 3$ state is that if we are actually at that state and the following fragments keep being considering it as

same slope classification then we are not going to change this value. We are looking for at least *3* steps with the same classification. Finally we added a $Q = 4$ state. This state is when after a sequence of at least *3* fragments with the same slope, we found a change in the slope of the contour. Then this is an ending position for a line segment inside the *f* sequence. We will jump from *Q=0* to *Q=1* when we reach an input different of *0*. In our implementation we have a temporal variable start that will keep this position value. Once the first fragment different of *0* Class is reached we jump to *Q=1* and the algorithm is computed as described (see Fig. 30).

The final output of the algorithm is a list of line segment detected $S_p = [b_p, e_p]$ for each p. The variable p is the amount segments detected.

Let us go a litter dipper in the description of each step. Every time we reach *Q=1* then we actualize the actual starting point b_p to the m actual index value from the contour. This is when a line segment begins. The state $Q = 2$ is reached when we are in $Q = 1$ and a same Class fragment classification to the last one analyzed or *0* Class is reached. This means that at least two segments with the same slope Class are founded in the fragment sequence. We are going to use the variable *change* to know if there is a change in the Class slope classification or not. The variable *change* will be *0(false)* when no change is detected; this means that the following input Class analyzed is the same slope that the last one analyzed or it is a *0* Class. In the case a different slope is reached then *change* will be *1(true)*. Then, *Q=3* come from *Q=2* when the same situation is founded; *change=0*. We are going to keep in *Q=3* as long we keep having a *change*'s value equals to *0*. This state means that at least 3 fragments with the same classification were founded. Just when a change of slope is reached (*change=1*) we move to *Q=4*. This state will notify us when a new ending point is founded.

When we found a new ending point we actually have a line segment detected $S_p = [b_p,e_p]$. Once we have this new segment we need to check if the last segment detected $[b_{p-1}, e_{p-1}]$ is less equal *2* in fragment distance with the new one. If $e_{p-1} - b_p <= 2$, then the last segment is actualized as $[b_{p-1}, e_p]$; no new $[b_p,e_p]$ is added. In the other case $[b_p,$

$e_p]$ is considered a new line segment and p is incremented. After these checks we 'jump' to state $Q=1$ and continues parsing the fragment sequence. We jump to $Q=1$ without analyze other segment because we reached the $Q = 4$ state already when a new fragment slope was detected as we had been in $Q=3$. The state $Q=4$ is just for compute a new segment. Then we can see this like, every time we reach a *change=1* we jump to $Q = 1$ if we are in $Q = 1, 2,$ or 3. In the other case we will follow the sequence *1, 2*, and *3*; we will stay in $Q=3$ till a *change=1* value is founded then we just pass for the intermediate $Q = 4$ before automatically jump to $Q = 1$.

Basically when we are parsing the contour we have a change of slope Class or not. The variable *change* with possible values of *0* or *1* represents if they was not a change of slope or yes respectively. Finally *start* variable will be the step position of the first fragment different of *0* Class classifications in the *f* sequence. We use this variable to know how many *0* are in the beginning of contour sequence. This will help us to join the first segment with the last line segment computed from the sequence.

Let us follow the example sequence showed in Fig. 29c for a better understand of how the fragment detections method works. We will take the sequence f = '1111122111113411114450021345 3' as input of our *fragment_repetitions(f)* method. Let say that we began to process the sequence at the first '1'; the $m=0$ position; m is the length of the f sequence. Like the first Class input is not zero, from $Q=0$ we jump to Q=*1* and $b[0] = 0$. At position $m=0$ we compare this position Class with the next input in the sequence; when the second '1' arrives at $m+1$ there is no change in the sequence (*change = 0*) and then we jump to $Q=2$. With the next '1' we jump to $Q=3$ and we stay in this state till a '2' is reached from the sequence. Then *change = 1* at $m=5$, when the first '2' is parsed; at this point we jump to $Q=4$ and $e[0]=4$. Like there is no last step computed because $p=0$, then we just increment p (note that at this position we already have $[b_0, e_0] = [0, 4]$), jump to $Q=1$ and $b[p] =5$ (the value p is equal to 1 now). With the second '2' in the input we jump to $Q=2$. Then a '1' will make us jump to the $Q=1$ state because *change = 1*; a slope change had been reached and $b[p]$ will be reassigned to 7. As the first part we are going to pass Q from 1 to 4 till the position $m=12$ and a '3'

Class is reached. This time *p*; the amount of segments detected is not zero. Then we check if the distance between *e[p-1]* and *b[p]* is less equal than 2 fragments and have the same '1' Class. In this case this condition is true, then we make *e[p-1] = 11*, and *p* is not incremented because no new segment is founded, just joined with the last one. We jump again to *Q=1* and make *b[p] = 12*. The following input '4' will keep us in *Q=1*, because another *change = 1* was founded and *b[p]=13*. Again with a input of '1' we keep the *Q=1* state and *b[p] =14*. We will reach again the *Q=4* state at *m = 18* when a '4' is reached in '1111122111113411114...'. We check again the fragment distance between *e[p-1]* and *b[p]* and this is equal to 2 then we make the *e[p-1] = 17*. Again no increment is done to *p* because of the join segment. (*p* keeps the value of 1 at this point). When jump to *Q=1* and make *b[p] =18*. With the following '4' input we reach *Q=2* and return back to *Q=1* when the '5' is founded; *b[p] =20*. With the two '0's input we reach to *Q=3* and with the following '2' we jump to *Q=4*. This time the distance between the last segment founded and the actual '500' in segments is 2 but it is not the same CLASS classification. Then *e[p] =22* and we increment *p* because a new segment was founded. The reader can keep following the rest of the input '....213453' but as we can see no line segments will be detected. Finally only 2 line segments were detected in this example: '1111122111113411111' = *f[0,17]* and '500' = *f[20,22]*. We finish to parse the sequence because there is no more input and like we did not end in the state *Q=3*, we don't need to compare the last segment founded with the first one in order to join them. See the *fragment_repetitions* procedure for more implementation details.

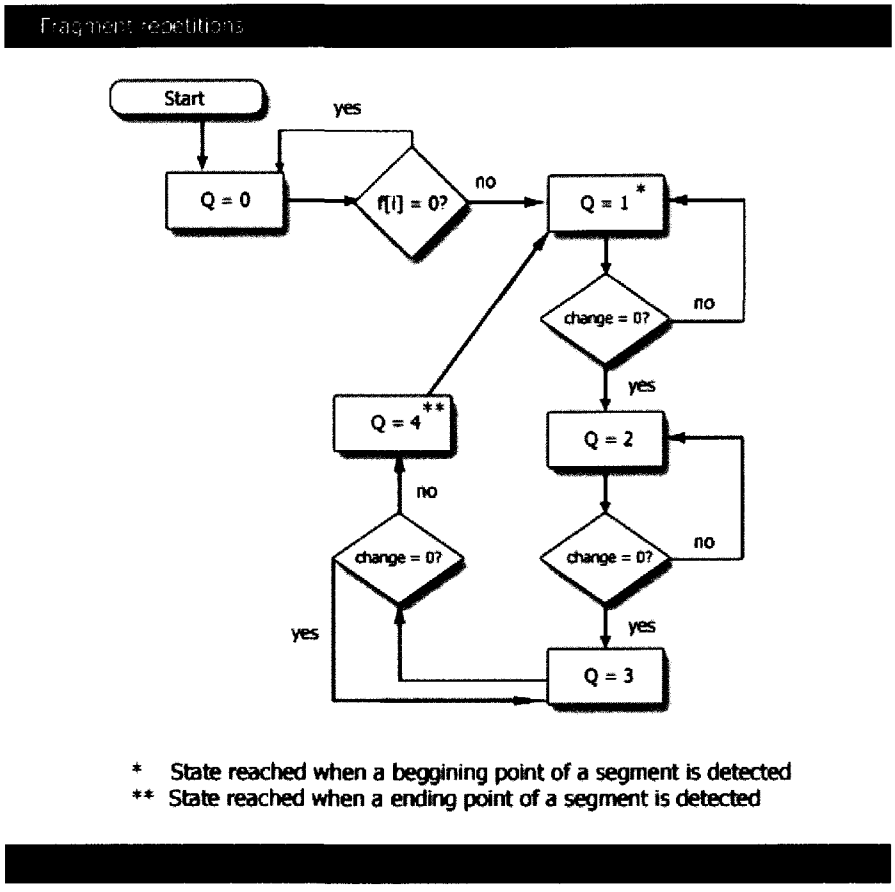

Fig. 30 Flow chart for fragments repetition step of the LDC algorithm. Every time the state *Q=4* is reached we compute a new line segment with the beginning information gathered from *Q=1*. The final output list of line segment of our algorithm can be computed using the $[b_p, e_p]$ computed in the *Q=1* and *Q=4* states.

After explaining what this step is about and how the flow execution is made, here we can follow with all details our *fragments repetitions* procedure.

3.8.4 Fragments Repetitions Procedure.

Procedure: *fragments_ repetitions (fi):*
Input: *fi* Fragments in contour where $0 <= i <= m$ and m total amount of fragments in contour.
Output: *bj* Beginning position of line segment j where $0 <= j <= p$; p amount of detected lines.
Output: *ej* Ending position of line segment j where $0 <= j <= p$; p amount of detected lines.

Q This variable holds the possible states of the algorithm. $0 <= Q <= 4$.
start Initial position of the fragment sequence with Class not *0*.

```
set i, j, Q = 0;
for i = 0 to m do
            if f[i] == 0 then start = i +1 else break
end
for i = start to m do
     if (f[i]==f[i+1] or f[i] == 0 ) then change = 0 else change = 1 end
     if ( change == 1 and Q == 3 ) then Q = 4 else
     if ( change == 1 and Q < 3 ) then Q = 1 else
     if ( change == 0 and Q == 3 ) then Q = 3 else
     if ( change == 0 and Q < 3 )  then Q = Q + 1

     if Q==1   then b[j] = i; end
     if Q==4   then
                  if j == 0 then e[j] = i ; j = j+1; end
                  else if ( b[j-1] == f[i] and b[j] - e[j-1] <=2 ) then e[j-1] = i;
                  else
                   e[j] = i ; j = j +1;
                  Q =1; b[j] = i+1;
                  end
end
if Q==3 then
            if j == 0 then e[j] = m ; j = j+1
           else if ( b[j-1] == f[i] and b[j] - e[j-1] <=2 ) then e[j-1] = m;
            else
                  e[j] = m ; j = j +1;
end
if  j > 1 and f[start] == f[e[j-1]] and (m==e[j-1] or m-e[j-1] + start < 3) then
            b[0] = b[j-1];  erase b[j-1];  erase e[j-1];  j = j-1;
end
```

Note the whole algorithm has more elaborated output: S_p (line segments including list of pixels on each). We can see $S_p = (b_p, e_p)$ for each value of p(number of line segments detected).

As we can see in the procedure above we need to check if the last state is $Q=3$ when the sequence is completed parsed and there is not more input. This case might be possible for any fragment sequence where the last Class' are the same till the last position of the sequence. In this scenario we will never reach the state $Q=4$ and consequently end in $Q=3$ once the sequence is over. An example sequence can be = '2323444444'. As we can see, the last part of the sequence is already a line segment (no more sequence to analyze), and there is no change of slope after that. Then, when parse the sequence completely we are going to check if the last state is $Q=3$. In this case, we check if this new segment to add can be joined to the last one detected; if not then we add the new last segment.

Let us see an example of this. In the sequence = '2323444444'. When we finish at $m=9$ and there is not more fragment's classification to analyze; our final status will be $Q=3$. We can see this because there is more than 3 consecutives '4's at the end of the chain. In this example, p will be zero and $b[0]=4$ when me reach $m=9$. Since there is no more fragment to parse, at the end of the sequence $Q=3$ state will act as $Q=4$. Like p is 0 in this example, we don't need to check if there is a join with the last detected segment, just $e[0]=9$ which represent a new line segment $S = [b_0, e_0] = [4,9] =$ *'444444'*.

The last check in our procedure is to see if the first and the last segment detected can be joining if there is more than *1* segment detected in the contour sequence. If the last segment ends in the m position, just at the end of the sequence and it has the same Class as the first segment detected. Then, these two segments are joining. No matter if start is not really the beginning of the sequence, if the last segment ends in the last position and both have the same slope classification the *0*'s from the beginning of the sequence till start position will included in the join too. Per example this sequence f = '00111123431111', will give us one final line segment '1111001111'. This is computed

after the algorithm detects both sequence of '1' and finally joined in the last check of our algorithm. If the last segment does not end in the last *m* position of the sequence, then in order to be joined this two segments(the last and the first one) must have the same Class and the fragment distance between both must be less equal than *2* fragments. For example, '011112331111' and '0111123311112' will be joined and yield us one line segment composed by '1's. We can see the solution like this '1111011111233'; the underlined '1's are the end of the sequence putted in front of the sequence, we can see how this line will be accepted as our algorithm. The same for '1111201111233'; there is only 2 fragments between the repeated sequence of '1's which is less than $X=3$.

3.9 Final output of the LDC algorithm

Once the *fragments_repetitions* procedure is applied to every contour detected by the previous extracting contour step. We already have our features of interest to display over the original image.

The accuracy of any line detection algorithm is evaluated by subjective judgment. Nevertheless, the appealing results are hard to deny. Fig. 31 shows the final output for our running example, where the line segments detected had been printed separately, and over the original picture.

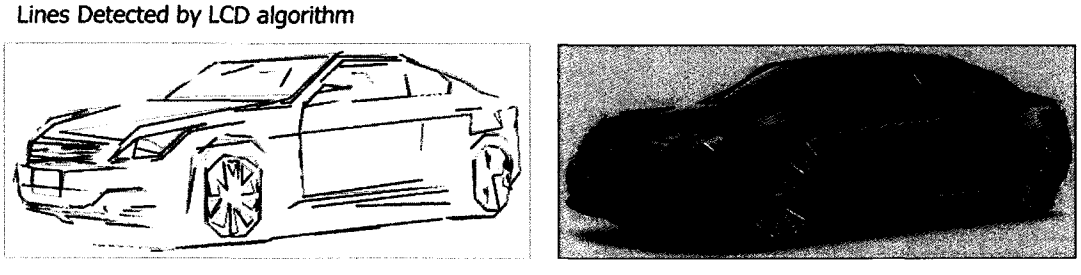

Fig. 31 Line Segments detected by LDC and printed over a white image and the original example image used to explain our algorithm.

3.10 Other Output Examples

Fig. 32 shows more examples, taken from our experimental data set (see Appendix A for complete image set). In images #14 and #17, all line segments were detected. Each of them is detected twice as per algorithm (duplications can be removed by post-processing). In example #3, it is not so easy for a human to compute the 'actual' number the lines the object has, or the number of lines needed so that the object can be accurately recognized, which is the main purpose of this overall discussion. It is however undeniable that the result obtained by LDC algorithm is very appealing in this case, and detected line segments together resemble the object quite closely.

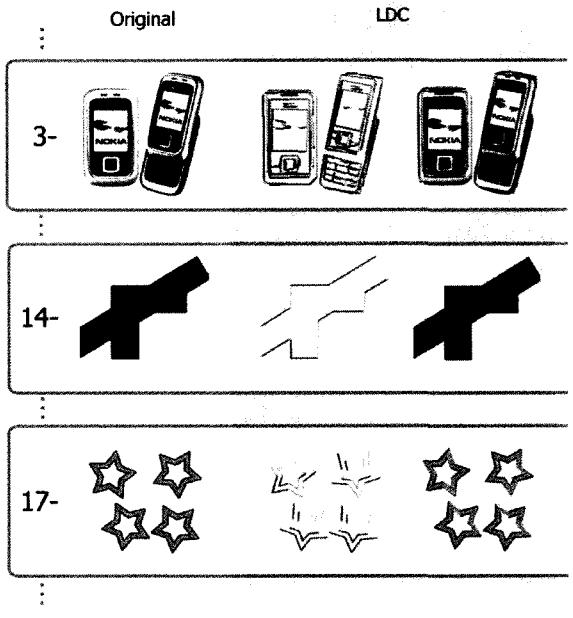

Fig. 32 Three examples taken from the 30 image set used in the experimental data. Following this order (GKS, LKS); the example #3 takes this configuration (1, 5). The configuration parameters for #14 and #17 were respectively (3, 5) and (1, 7).

Chapter 4: Analysis and Experimentation

This chapter is devoted to the analysis and empirical comparison between our LDC algorithm and the Hough Line algorithm. It will start with a description of the data set used in our experiments and the explanation of equivalent parts of both approaches for further comparison. The assessment will take into account the execution time, input parameters and accuracy of the line segments detected. The experiments will be conducted over the image set found in Appendix A. Following, an objective and contribution analysis in terms of the problem statement is included. For completeness of the evaluation of our proposed algorithm, some of its limitations will be highlighted. Last but not least, we will shed some light on the time and space complexity of our LDC proposal before stating the conclusions and future work perspectives in Chapter 5.

4.1. Experimental Setup

The *30*-image set analyzed in our experiments were randomly selected according to their shape content and spatial size resolution. All selected pictures consist of color-depth images (*24 bit-depth, 96 dpi*) containing planar shapes and real environment pictures. The spatial resolution of these images goes from *256x256* to *2048x1536*. Simulations were run under a *32*-bit Intel Mobile *2.0* GHz CPU architecture with *1* GB of RAM. Following subsections will discuss and investigate the average results for the benchmark image set. This chapter will focus on the experiments conducted on those images which are relevant for the analysis and subsequent discussion of our research objectives. Individual results for each image and applied method can be encountered in the appendices section.

We compared our newly proposed LDC algorithm with what appears to be a widely accepted fashion to extract line segments in literature by using the OpenCV library. Since Canny's edge detector was regularly associated with the Hough Lines 478, GCH

algorithm was chosen as the benchmark, which represents an ensemble of Gray Scale + Canny + HL (the OpenCV built-in PPHT Hough Lines) algorithm. In this implementation we relied upon the Open Sources OpenCV 1.0 library and the EmguCV 1.2.2.0 project, .NET wrapper to OpenCV and C# language. Our goal was to measure the processing time and the set of lines detected by both algorithms. Processing time was assessed individually for the two steps: from color image to edge extraction, and from edge to line segment extraction. Both LDC and GCH are then split into their corresponding building blocks, (see Fig. 33) LDC=GB+CS, where GB stands for the sequence of Gray scale conversion from color image, automatic normalization, Gaussian smoothing, Laplace operator and automatic threshold and CS is a sequence of contour extraction (which includes finding connected components) and segmenting contour. GCH=GC+HL consist of GC (Gray scale conversion followed by Canny edge detection) and HL (Hough Lines algorithm) which does not use connected components or contours to extract lines and instead maps some pixels to the dual accumulator space.

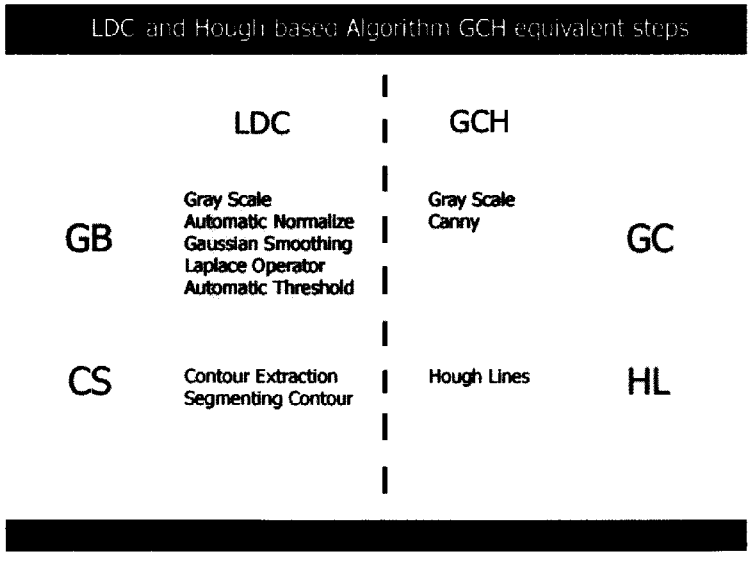

Fig. 33 Equivalent steps among LDC and Hough-transform-based GCH algorithms. GB and GC are edge extraction preprocessing steps that produce binary images from colorful ones. CS and HL compute line segments from binary images.

4.2. *Time Comparison*

The mean processing time of our LDC algorithm was 0.11 seconds with 0.11 ± 0.04 of confidence interval at the 95% confidence level. For GCH, results were 0.15 and 0.15 ± 0.05, respectively. Hence, there exists a small but statistically insignificant difference in favor of our LDC algorithm since confidence intervals overlap.

The next evaluation measured processing times starting from binary images. Our CS scheme is about twice faster than the competing HL algorithm, and the difference is statistically significant. The average processing time of CS was 0.03 seconds, with 95% confidence interval of 0.03 ± 0.015. The corresponding means and confidence intervals for HL were 0.07 seconds and 0.07 ± 0.028, respectively.

For the sake of completeness of this time measurement experiment, we computed these statistical indicators for the preprocessing steps in both algorithms. GB is part of our LDC algorithm and exhibits a confidence interval of 0.08 ± 0.029 while the corresponding interval for the competing GC was 0.08 ± 0.025. Thus, these times were found almost identical and are a major part of the overall computation time. The measured times are shown displayed in Fig. 34.

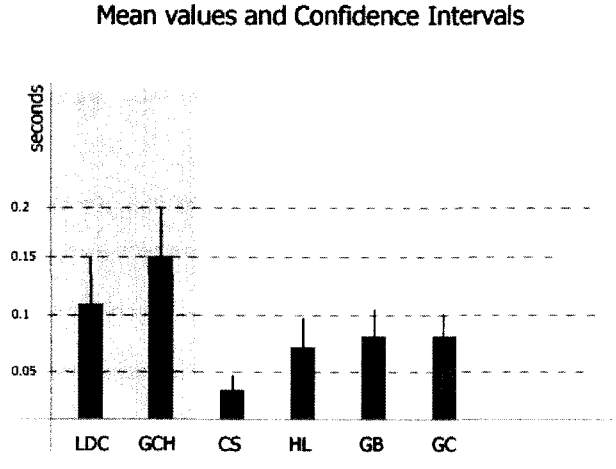

Fig. 34 Means values and Confidence Intervals for processing times.

4.3. Accuracy

The outcome of HL (and consequently GCH) was very sensitive, for all images, to the parameter choices. Even small changes in some parameters have led to quite different number of lines detected. Human intervention for obtaining satisfactory outcome in HL was essential and time-consuming. A larger number of parameters in GCH also contributed to this. The best set of parameter values is picture-dependent. For a fair comparison, we included two parameter choices in GCH for each of the 30 selected pictures and only one for LDC. One obvious observation from this exercise was that LDC proved to be a significantly more *robust* solution than GCH, due to substantially less sensitivity of the outcome to changes in parameter values.

Our next and most important experiment compares the accuracy of the line detection process. Fig. 35 shows 7 out of 30 images from our experimental test bed. In some examples, the actual number of lines is obvious. Images #10, #14, #17, and #27 have 12, 13, 80 and 27 lines respectively, and our LDC algorithm retrieves exactly twice as many lines because of easily removable duplication. The number of lines extracted by the GCH iterations is quite different, sensitive to parameter choices, and in disagreement with human observation, mostly because of missing many line segments. Human classification might yield about 20 line segments for image #9. LDC retrieves 46 (or 23 actual ones), while two parameter choices for GCH yielded only 13 and 10 lines respectively, despite edges being intuitively obvious. LDC algorithm also produces more appealing line segments for the car in image #19 and cellular phones in #9 compared to GCH Hough-transform-based algorithm. Overall, better results for our method were obtained for *every* image from our set.

The complete parameter configuration input can be viewed in Appendix F. Complete analysis of the 30 image set is shown in Appendix A of this thesis. For each image, you can see from left to right, the original picture, line segments detected by LDC, the LDC's segments over the original picture, GCH1's lines segments detected, and GCH2's line

segments detected (GCH1 and GCH2 denote two parameter choices for the same GCH algorithm).

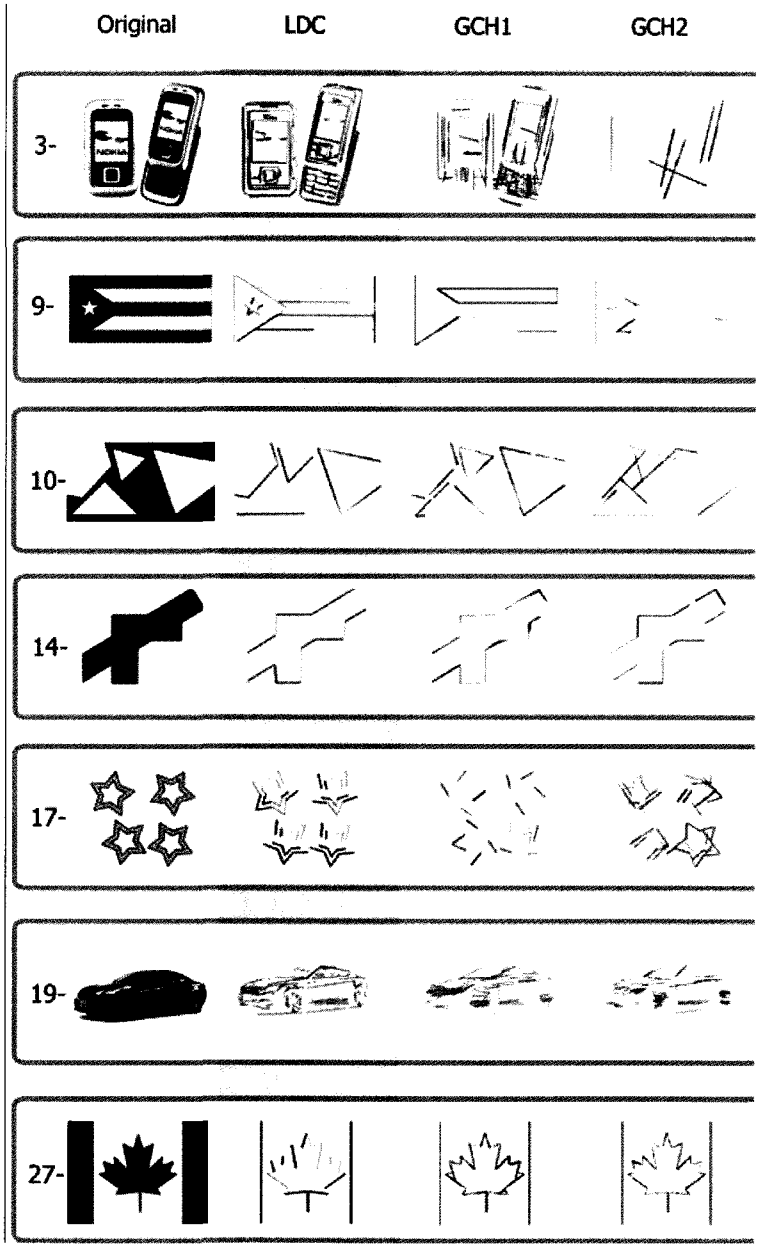

Fig. 35 Examples #9, #10, #14, #17 and #27 from the 30 image set in Appendix A. The amount of lines detected by LDC is 46, 22, 24, 26, 160, and 54, respectively. Two parameter choices HL1 and HL2 had 13, 34, 28, 40, 62, and respectively 10, 12, 13, 82, 65 lines.

Finally, we computed the linearity SNZ 1111 value for each of the line segments detected by LDC in the image set. The SNZ linearity values range from 0 to 1, where values nearest to 1 mean highly linear. The value is computed based on coordinates of each pixel from a given line segment. Because of discretization, perfect 1 scores are seldom likely. Results for individual images are given in Appendix A. The 95% confidence interval for the linearity values was 0.97 ± 0.006. This shows that detected line segments are indeed representative lines, with almost perfect linearity score and consequently small deviation.

4.4. Analysis of Input Parameters

Our combination/selection of input parameters is considerably lower than the HL traditional line detection method. Unlike HL, our solution does not need to have a prior knowledge of the line segment instances we are looking for in the image. The HL implementation needs prior knowledge of what we are looking for in the images: minimum gap between lines, theta angle definition, rho distance to the line (theta and rho are used to restrict the construction of the parametric space), and minimum length of the segment.

The Hough Transform HT method does not neither need to know anything in advance, but its computation of the complete parametric space is very expensive in time, and the sine and cosine translations make it a more complex and unmanageable algorithm. The lines detected by our algorithm are computed from the edge morphology rather than from isolated points that might mistakenly represent lines of not actually connected points in the image. We don't need to compute all the possible lines that can pass over a pixel in the image, as in the classic Hough transform method or Hough Line variant. Neither we require any knowledge on what length and orientation of line segments we are interested in, nor the gap space between different line segments over the same line. In addition to it,

LDC algorithm uses three input parameters versus seven in GCH (see Fig. 36). Out of them, we need to keep in mind that two of them are bounded: GKS is an odd number less than or equal to *19* and LKS can only take values in {*1, 3, 5* or *7*}.

If we compare only HL with our contour extraction and segment detection steps of the LDC algorithm (because both start off with a binary image as input), our combination steps resembling the HL procedure only employ one configuration parameter MLC, i.e. the Minimum Pixel length of the Contour to analyze versus five configuration parameters in HL. Also, we fixed the MLC value in our experimental data to be *50* pixels. This setting reported an increased performance over the complete experimental set.

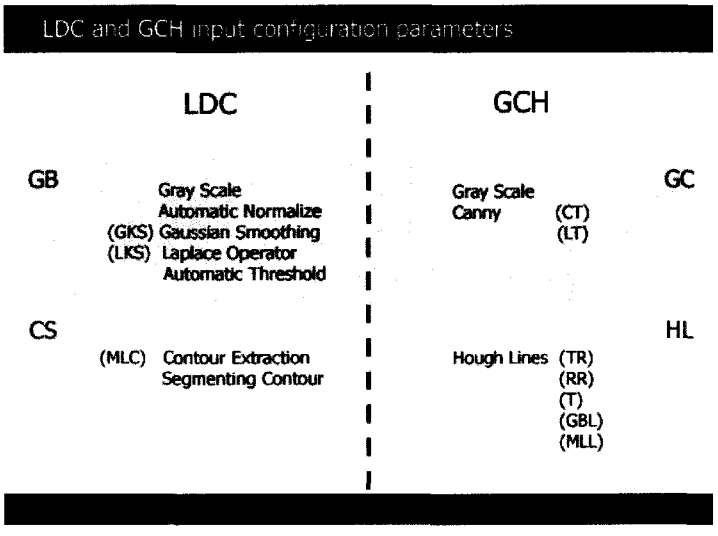

Fig. 36 Input configuration parameters for LDC and GCH (parameters are between parentheses). For our LDC algorithm in the preprocessing step GB (from Gray Scale to Binary Threshold), we have a total of two input parameters and one for CS (Contours extraction and segmentation). GKS, LKS, and MLC are in the same order Gaussian kernel Size, Laplace Kernel Size, and Minimum length Contour. For the GCH solution, we require two input parameters in the preprocessing phase versus five for HL. CT, LT, TR, RR, T, GBL, MLL are in the same order Canny Threshold, Linking Threshold, Theta Resolution, Rho Resolution, Threshold, Gap between lines, Minimum length line.

4.5. *Objective and Contribution Analysis*

Although there is no settled empirical criterion for judging the correctness of the lines extracted by any algorithm (other than the human evaluation), we can see how our approach attains better accuracy in terms of the stated problem and is more intuitive than peers with respect to the human classification for varying input parameter configurations. Also, for images where it is easily recognizable by a human expert the number of line segments present in the original image, LDC reported a more accurate number of lines and a more precise location in the image space. Furthermore, it proved to be robust against conventional images transformations such as rotation, scaling and translation and was able to uncover line segments across any orientation and slope over the entire image space with an acceptable execution time. LDC also required fewer configuration parameters than peer methods, as we can see in this chapter. The 95% confidence interval for the linearity values 0.97 ±0.006 SNZ [11] computed from the entire line segment detected by our algorithm demonstrates that the line segments reported are indeed representative lines with almost perfect linearity score and consequently small deviation.

Another advantage of LDC versus the Hough Line algorithm lies on its shorter execution time and more accurate results. Although Hough transform might be a theoretically simple algorithm to line detection, it is not that easy in practice. Our algorithm does not need high-complexity operations like sine and cosine transformations, or the finding of local maxima in the accumulation space. The creation of the accumulator space is a time-consuming process since each pixel is converted into a digital line. LDC exhibits a linear time complexity in the number of pixels of the input image. This time was measured with satisfactory results by testing spatial resolution images varying from a wide array of image resolutions *256x256* to *2048x1536*. Our discrete linearity criterion is a very intuitive, easy to implement solution that lacks round off and associated math calculation setbacks. In the experimental data, we can behold that the execution time of our line segmenting contour method is statistically different than that in the OpenCV Hough Lines implementation. This will in turn be translated into a faster overall solution achieved by the proposed algorithm. Preprocessing steps used

across the algorithm are well-known image processing operations and filters with very good linear time complexities in the order of the number of pixels comprising the input image.

Additionally, our model defines a specific procedure in order to retrieve line segments out of true color images. The Hough lines algorithm needs a binary image as input. Although some techniques are mentioned in computer vision literature as preprocessing steps in order to compute Hough-transform-based algorithm, there is no documented and widely accepted fashion in literature for extracting line segments from colorful images. Obviously, the same preprocessing, applied here to LDC, could be applied to generate a binary image from a colorful one as needed by HL. Our understanding is that LDC appears to be the first complete prototype of line segment extraction out of color input images. In existing literature, this preprocessing step compliant with HL is declared as just any image processing technique that yields a binary image as output after receiving a gray image as input 45810. In any case, the results may be different based on the selection of the input parameters and the different techniques required for arriving at the binary image. Our proposed solution proved to be very robust when it comes to computing any kind of image and yielding nice line segments compared to the human perception classification. Different from the current literature, our LDC algorithm can be envisioned as a sequence of steps in order to retrieve line segments from any kind of images.

4.6. Limitations

We can consider the following issues as limitations for our LDC approach, which might lead to obtaining diverse results.

The selection of input parameters for the edge detection step is subject to human recognition for better accuracy results. These can often be ascribed to the brightness contrast in images and the nature of the edge detection algorithms. This is in spite of LDC having only three configuration parameters to draw line segments from color images.

We can find duplicate lines describing the same real object's boundary because of the cyclic representation of the contours. A single line contour might produce two line segments with the same slope but different discrete Class classification and almost the same spatial location. In Fig. 35, sample images #14 and #17 detect nearly the same line twice.

The discrete linearity measure used; based in the *9* Class's classification matrix, the step size, and the *4* state might be subject of further discussion concerning their accurateness. For instance, we may assume we have a stairs with longer horizontal and shorter vertical steps. In our algorithm, following our linearity criteria it would be declared as a single line as long as any of the length of the horizontal or vertical steps have length less than *15* expressed in pixels, because the step size = *5* and X = *3* (*5 x 3 = 15*), a line in the image need to be at least *15*-pixel-long so as to be recognized (see Fig. 37). The problem lies whenever we want to reduce this *15*-pixel length, so we will need to lower either the step size or the X value. If we choose the former, e.g. *3*, then the original 9 Class classification matrix will be transformed into another Class classification set and the size of the matrix will be *7x7,* thus, the discrete classification of slopes will be bolder and not as complete as the *5 x 3* initial value, since in this configuration *5* and *3* don't detect stair steps with length lower than *15* pixels. On the other hand, if we lower

the X value, then the classification linearity is less 'strong' and more sequences of points could be labeled as a misleading line. After some step sizes and X values test, we found that *5 x 3* achieved a 'good enough' trade-off in order to disclose straight line segments in images with a good human perception accuracy that we can show in our test bed. The authors believe there is no perfect combination of values for this algorithm: a change in any of them will produce different outputs and entail different limitations. However, the measurement of the optimal values for our discrete classification of line segments in images is beyond the scope of this work and manuscript. Here, we are showing how 'accurate' this algorithm might be using the *9* Class classification matrix with a step size of *5* and X = *3* in order to detect digital straight line segments in images.

Another important topic to consider is the spatial resolution of the image and how it would affect a fragment size of *5*. It might sound reasonable that with very smaller images, the linearity criteria used with the discrete matrix and the fragment size *5* may lead to some noise in the segment detection phase. Maybe this step size used could be related to the image resolution size in some non-trivial way. But, as we can see in our experimental setting comprising images of quite different resolutions (from *256x256* which is regarded as a small image to *2048x1536*, which is way bigger than the highest resolution video cards can actually support in nowadays computers), LDC reports a great output line segment set. See Appendix A and Experimental Data chapter for complete information on image sizes. Anyways, this might be part of future research work about the correlation between step size and image resolution.

Another anomaly in our linearity classification may be founded in the example showed in Fig. 38, when we are trying to find lines and the image is a circle. This circle will be described by *8* line segments, each segment from the different *9* classification classes. This might be seen like an *8* segment poligonalization of the circle (see Fig. 38). Unfortunately, as we can see in the image, both figures may yield the same output.

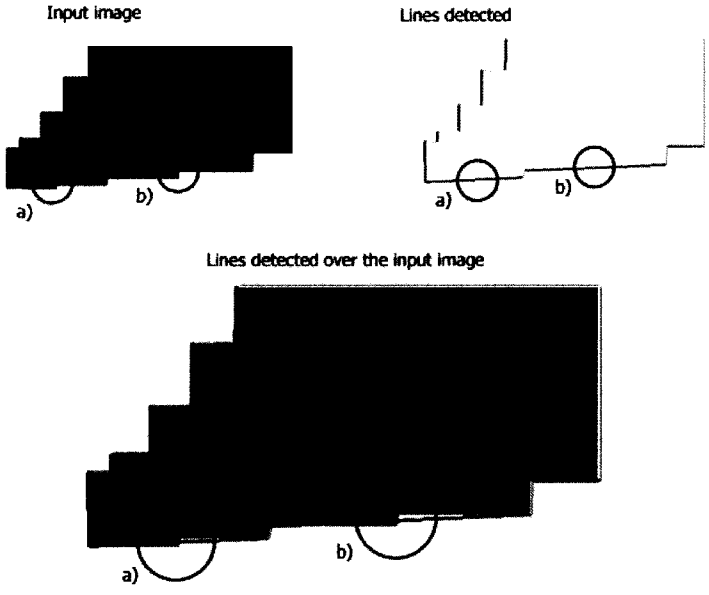

Fig. 37 In the above image a) and b) are line segments no detected by our linearity criteria – the verticals steps are not detected-. The 3 segment steps are recognized as a single line. The two big horizontal steps and one little vertical step are recognized as a little diagonal horizontal line segment. The red circles indicate the vertical steps not detected.

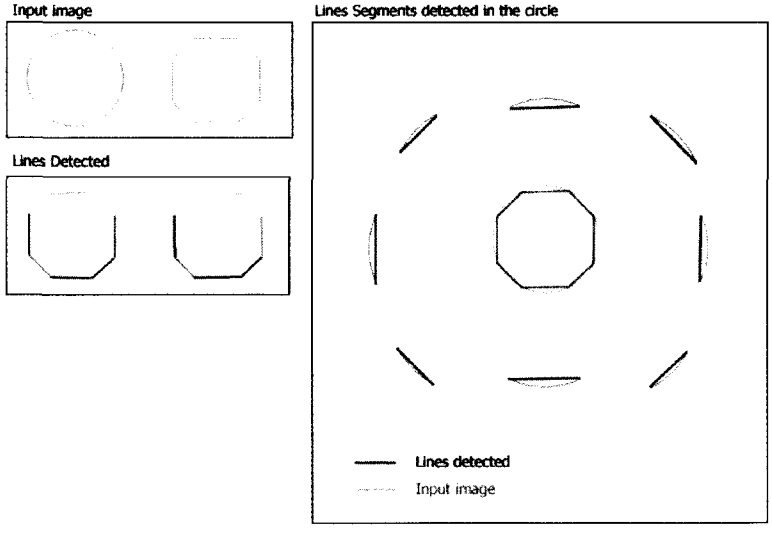

Fig. 38 The top left image is the input image tested, below the lines detected by the LDC algorithm. The right picture shows the *8* lines detected in the circle over the circle silhouette.

4.7. *Time and Space Complexity*

Time complexity of our proposed algorithm will be dominated by the preprocessing steps in GB. The *5* steps involved in GB are the Gray Scale, Automatic Normalization, Gaussian Smoothing, Laplace Operator and Automatic Thresholding. These steps will get a binary representation of the edges from any color image. The two last steps involved in our algorithm are Contour Extraction and Segmenting Contour: CS. As we portrayed in epigraph 4.2, GB takes 0.08±0.029 seconds; more than twice the time consumed by the two last steps in CS: 0.03±0.015 seconds. The time complexity involved in each of the five steps of GB are depending upon the number of pixels in the original image: N. All the five steps apply constant operations to each pixel of the original image; giving us an $O(N)$ complexity [2][3][12]. Contour Extracting will be also bounded by linear time $O(N)$ [10[13][16], because we need to traverse connected components found in the image. The second part of CS and final step of our solution: Segmenting Contour will be bounded by the total number of pixels of the contours; n. We can see how N is considerably bigger than n. Finally, our last step iterates only once over the list of contours adding a $O(n)$ to the overall complexity, which turns out to be $O(N + n)$; but taking into account that $N >> n$. The time complexity is therefore linear, i.e. $O(N)$.

In terms of space complexity, all the steps involved in GB take $O(1)$ because they can be carried out "on the go". No considerable extra memory needs to be allocated for the conversion of the color image to its edge binary representation. For CS, the space complexity will be compromised by the amount of pixels n detected as contours of connected components. This number n as seen before is considerably less than the number of pixels in the image. The final step, Segmenting Contours will not allocate more memory than *4* integers per line

detected. The *4* integers will be composed by the x and y coordinates of each ending point; two in each line segment. This number again will be a great deal less than n, the number of contour pixels. Finally, the space complexity of the LDC algorithm will be dominated by N; that will be the memory allocated by the input image. The space complexity is therefore also linear, i.e. $O(N)$.

Chapter 5: Conclusions and Future Works

In this thesis we described a new line detection algorithm called LDC. We compared our solution to the most widely used HL algorithm based on computation time, accuracy as well as number and meaning of the input parameters. Analysis of time complexity, space complexity and experimental data are supplied as part of the work to support our claims. Although some limitations are presented here, we concluded that our solution is better than current state-of-the-art approaches targeting the same problem. We created a simple solution capable of detecting line points (endpoint information) using the morphology of the objects inside the image. The LDC algorithm is the first method depicting the process of extracting line segments out of input color images. It is also more accurate in its output, yielding a better human perception of line classification. Furthermore, it is faster than Hough-based solutions, as demonstrated by its linear time and space complexity bounds. It detects line segments among any orientation or slope and is invariant to standard image transformations such as scaling, rotation and translation. LDC also makes use of fewer input parameters. Compared to HL, the set of parameters used in LDC is more predictable and can be applied almost independently on any input picture whereas HL requires fine tuning of its parameters for each picture in order to successfully extract a line. Because of its good time and space complexity, it can be used as a pre-processing step to identify more complex shapes and/or other features of interest within an image.

5.1. Extensions and Future Work

Our algorithm has several applications in the area of computer vision and can be used for measuring linearity in a linked, ordered set of points, segmentation of images using lines, corner detection, poligonalization of contours and basic shape detections. These are some of the areas of our future work. We believe our solution can be clearly adapted to other curve classifications and also be integrated with other computer vision algorithms. We are currently working on corner detection and basic shape detections using the line segments previously detected on images.

Publications

1. Solis Montero, Andres. Nayak, Amiya. Stojmenovic, Milos. Zaguia, Nejib. "Robust Line Extraction Based on Repeated Segment Directions on Image Contours", IEEE Symposium: Computational Intelligence for Security and Defense Applications (CISDA), July 8-10, 2009.

2. Solis Montero, Andres. Nayak, Amiya. Stojmenovic, Milos. "A simple, fast, nearly parameterless and accurate detection of straight lines from images", Journal IEEE Transactions on Image Processing (Signal Processing Society), submitted, 2009.

3. Solis Montero, Andres. Stojmenovic, Milos. Nayak, Amiya. "Robust Detection of Corners and Corner-line links in images", IEEE International Conference on Computer and Information Technology (CIT2010) Bradford, UK, 29 June - 1 July, 2010.

References

1. Canny, John. "A Computational Approach to Edge Detection", IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 8, pp. 679-714, 1986.
2. Sobel, Irwin. Feldman, Jerome A. "A 3x3 Isotropic Gradient Operator for Image Processing", unpublished, 1968, cited orig. in Duda, Richard O. and Hart, Peter E. "Pattern Classification and Scene Analysis". Wiley, 1973.
3. Gonzalez, Rafael C. Woods, Richard E. "Digital Image Processing" 3rd edition. Prentice Hall Inc. 2007.
4. Duda, Richard O. Hart, Peter E. "Use of the Hough Transformation to Detect Lines and curves in pictures", Comm. ACM, vol. 15, pp. 11-15, 1972.
5. Fernandes, Leandro A. Oliveira, Manuel M. "Real-time line detection through an improved Hough transform voting scheme", Pattern Recognition, vol. 41, issue 1, pp. 299–314, 2008.
6. Guru, D. S. Shekar, B.H. Nagabhushan, P. "A simple and robust line detection algorithm based on small eigenvalue analysis", Pattern Recognition Letters, vol. 25, issue 1, pp: 1 – 13. 2004.
7. Marr, David. Hildreth, Ellen.C. "Theory of Edge Detection", RoyalP (B-207), pp. 187-217, 1980.
8. Goshtasby, Arthr A. "2-D and 3-D Image Registration ", Wiley-IEEE, 2005.
9. Qiu, Peihua. "Image Processing and Jump Regression Analysis", John Wiley and Sons, 2005.
10. Bradski, Gary. Kaehler, Adrian. "Learning OpenCV: Computer Vision with the OpenCV Library", O'Reilly, 2008.
11. Stojmenovic, Milos. Nayak, Amiya. Zunic, Jovisa. "Measuring linearity of planar points sets" Pattern Recognition 41, pp. 2503-2511, 2008.
12. Emgu CV project API Reference and Documentation. [Online] Available: http://www.emgu.com/wiki/index.php/Main_Page, accessed on July 27th, 2009.
13. Shapiro, Linda. Stockman, George. "Computer Vision", 1st edition, Prentice-Hall Inc, 2001.

14. Martí, Joan. Benedi, Jose M. Mendoza, Ana M. Serrat, Joan. "Pattern Recognition and Image Analysis", 3rd Iberian Conference, IbPRIA, 2007.
15. Kiryati, Nahum. Eldar, Yuval. Bruckstein, Alfred.M. "A probabilistic Hough transform", Pattern Recognition, vol. 24, issue 4, pp. 303 – 316, 1991.
16. Suzuki, Kenji. Horiba, Isao. Sugie, Noboru. "Linear-time connected-component labeling based on sequential local operations", Computer Vision and Image Understanding, Image Underst, vol. 89, issue 1, pp. 1-23. 2003.
17. Chang, Fu. Chen, Chun-Jen. Lu, Chi-Jen. "A linear-time component-labeling algorithm using contour tracing technique", Computer Vision and Image Understanding, vol. 93 issue 2, pp. 206-220, 2004.
18. Galambos, C. Matas, J. Kittler, J. "Progressive Probabilistic Hough Transform for line detection", Computer Vision and Pattern Recognition, vol. 1, pp. 554-560, 1999.
19. Otsu, Nobuyuki. "A threshold Selection Method from Gray level Histograms", IEEE Transactions on systems, man, and cybernetics, vol. smc-9, No. 1, pp. 62-66, January 1979.
20. Weszka, J. S. Nagel, R. N. and Rosenfeld, A. "A Threshold selection technique", IEEE Trans. Comput, vol. C-23, pp. 1322-1336, 1974.
21. Watanabe, S. and CYBEST Group. "An automated apparatus for cancer prescreening: CYBEST", Comp. Graph, Image Process, vol. 3, pp. 388-410, 1972.

Appendices

Appendix A

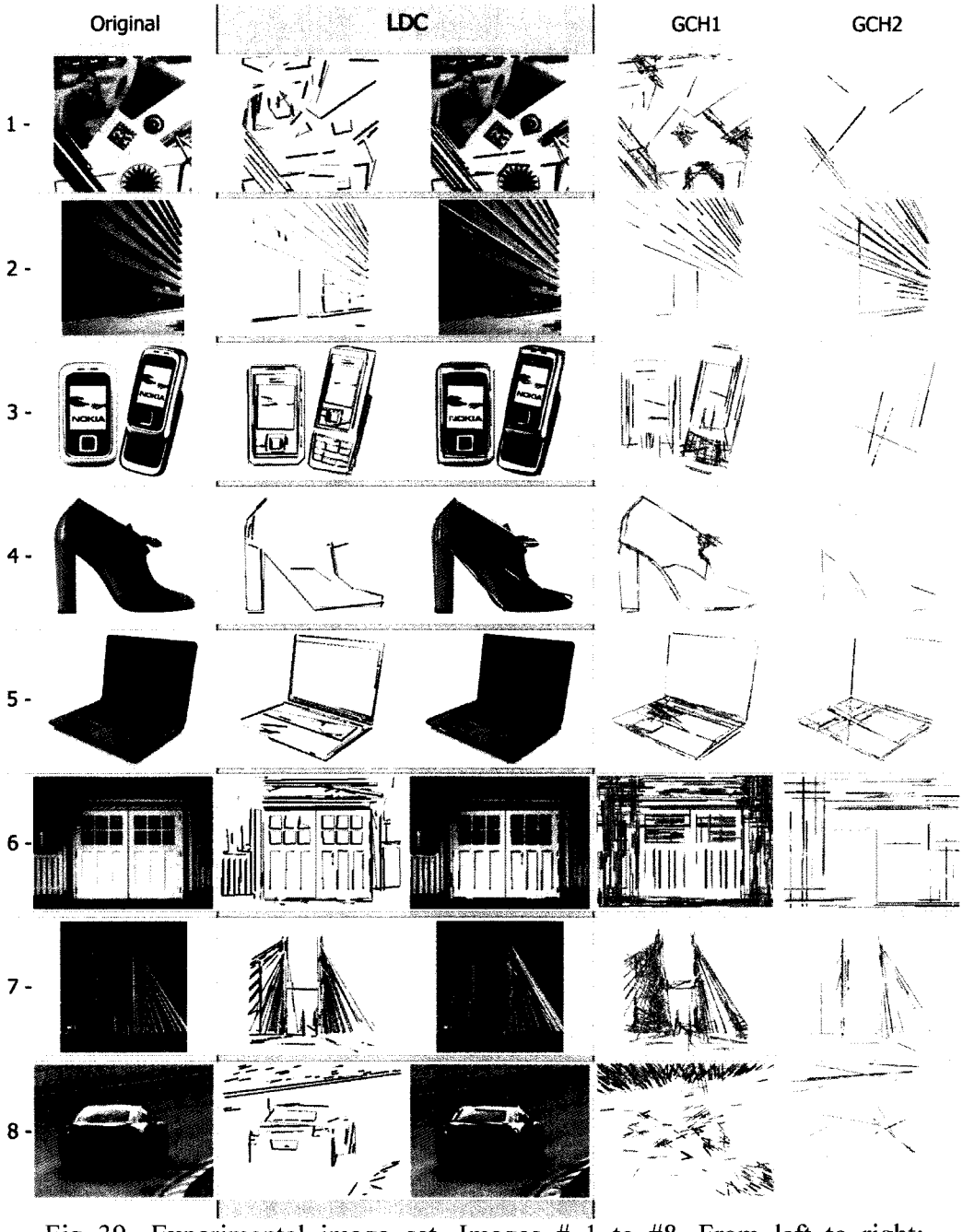

Fig. 39 Experimental image set. Images # 1 to #8. From left to right: Original Image, LDC output, LDC output over original image, GCH1 output, GCH2 output.

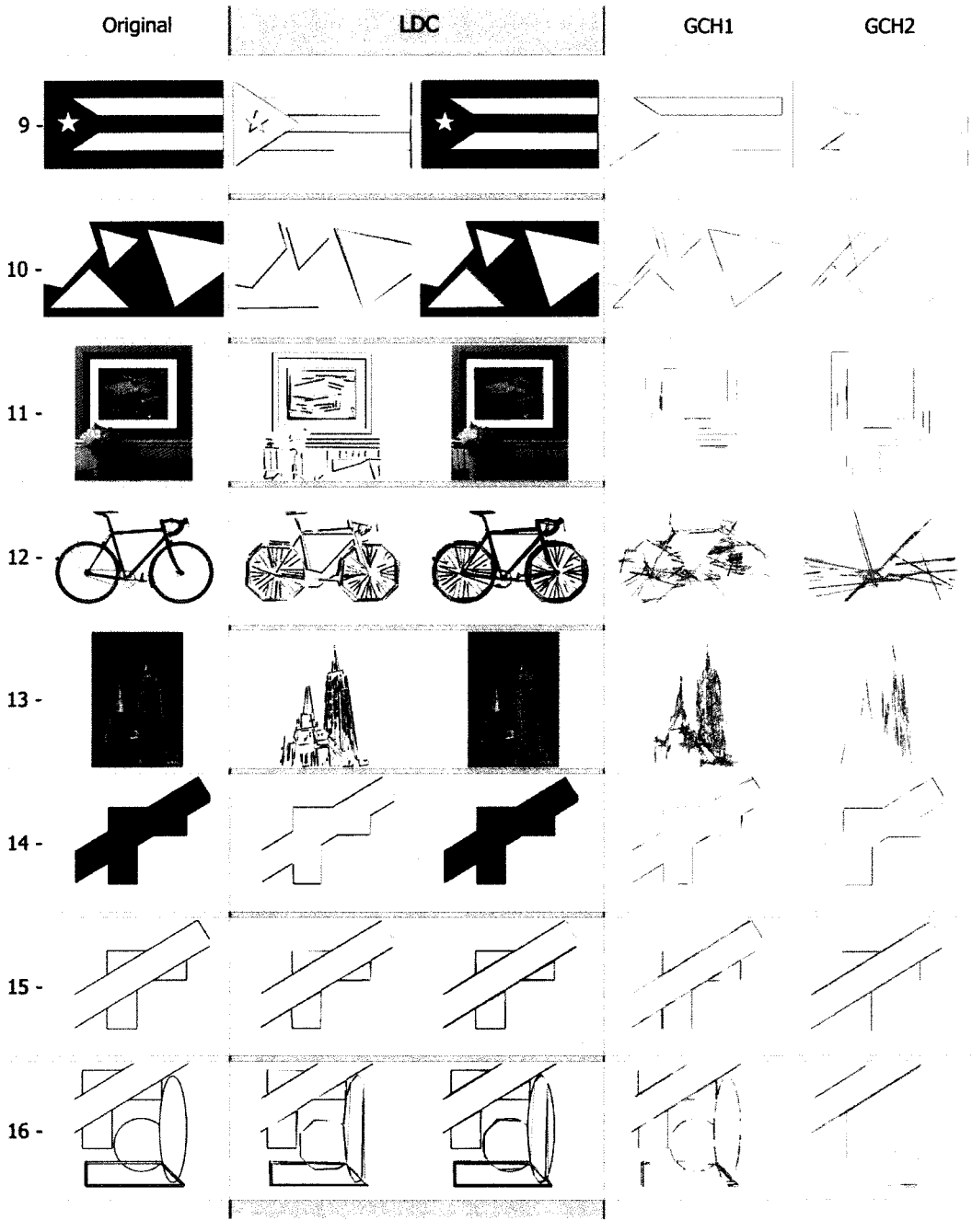

Fig. 40 Experimental image set. Images # 9 to #16. From left to right: Original Image, LDC output, LDC output over original image, GCH1 output, GCH2 output.

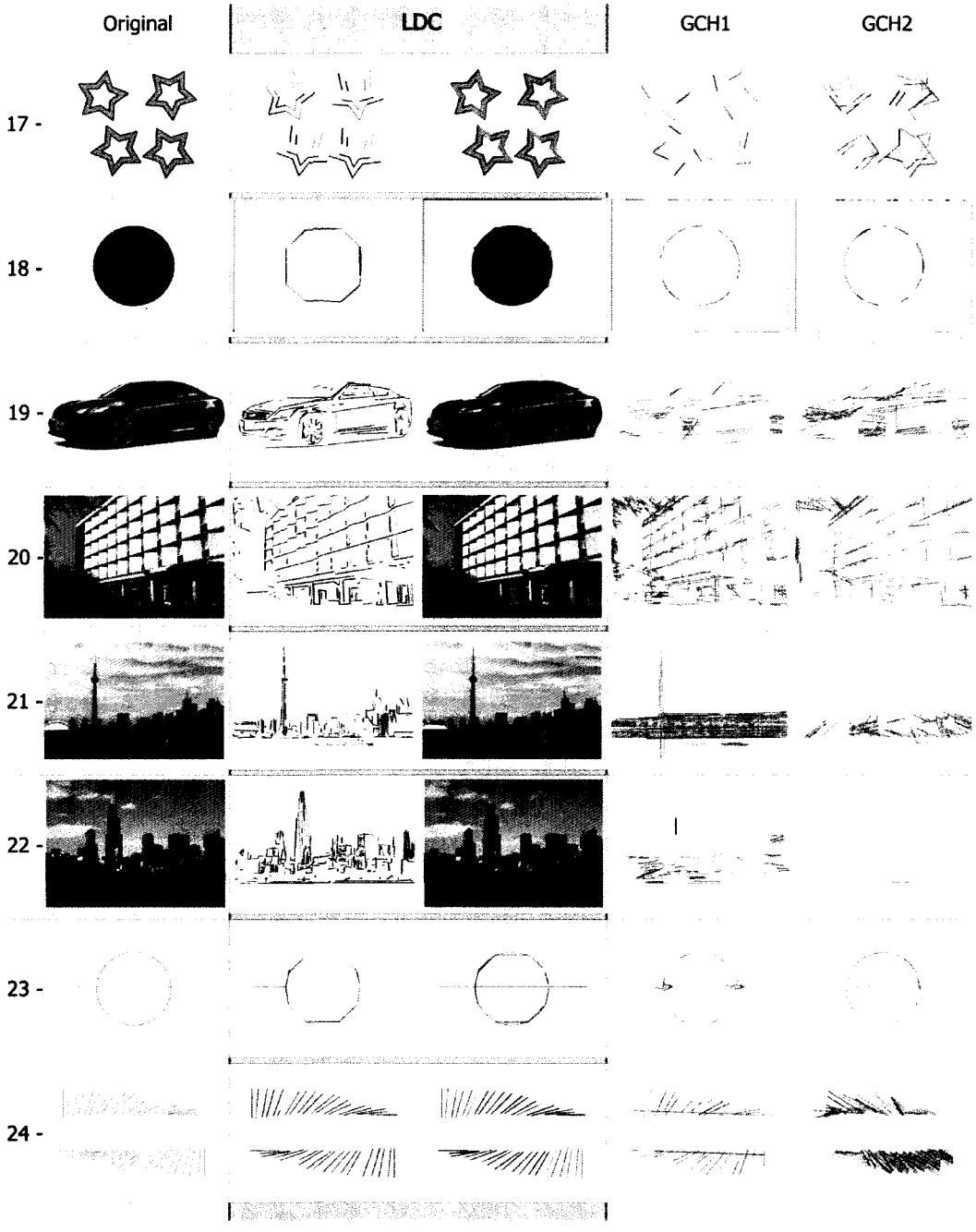

Fig. 41 Experimental image set. Images # 17 to #24. From left to right: Original Image, LDC output, LDC output over original image, GCH1 output, GCH2 output.

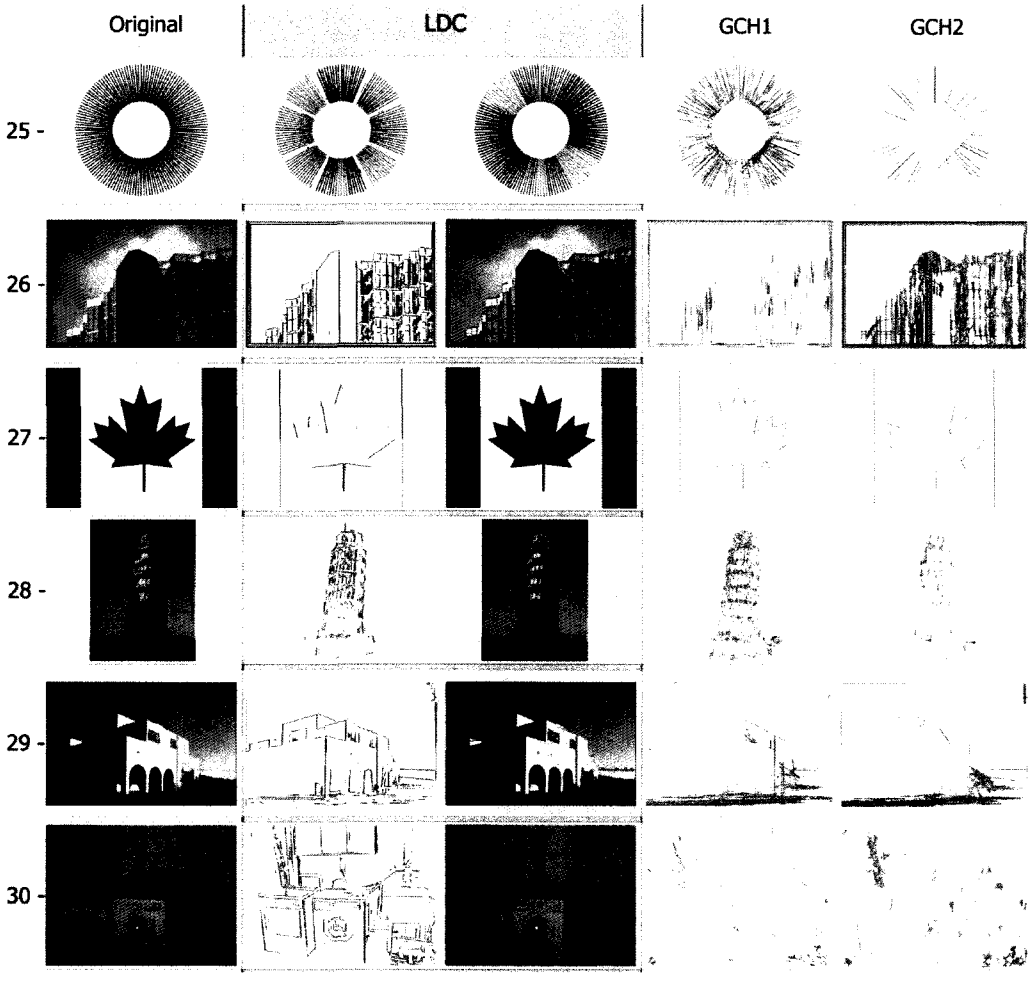

Fig. 42 Experimental image set. Images # 25 to #30. From left to right: Original Image, LDC output, LDC output over original image, GCH1 output, GCH2 output.

Appendix B

Algorithm LDC

Input:

```
image:        Image,
kernel_size:   integer,
aperture_size: integer,
mincount points: integer .
```

Output:

```
segments[]: Array of Segments.
// Segments = (P0(x0, y0), P1(x1, y1)).

image ← image.toGray();
image ← image.Normalize();

image ← image.GaussianSmooth(kernel_size);
image ← image.Laplace_EdgeDetection( aperture_size);
binary_image ← image.Threshold();
contours[]← image.FindContours();

foreach (contour in contours)
{
     if (contour.pixelCount >= minPixC)
     {

          fragments[]← contour.toFragment();
          S(b[],e[])← fragment_repetitions(fragments);
          Segments.add(S);

     }

}
return segments;
```

Appendix C

The 3 charts in this appendix show us the comparison elapsed time for the complete algorithms LDC, and GCH1, GCH2 iterations of OpenCV Hough lines based algorithm implementation. The elapsed time for the 30 image set is expressed in second. The order of the images follows the same as Appendix A, ascendant ordered per mega pixel size of the image. The time is measured in seconds.

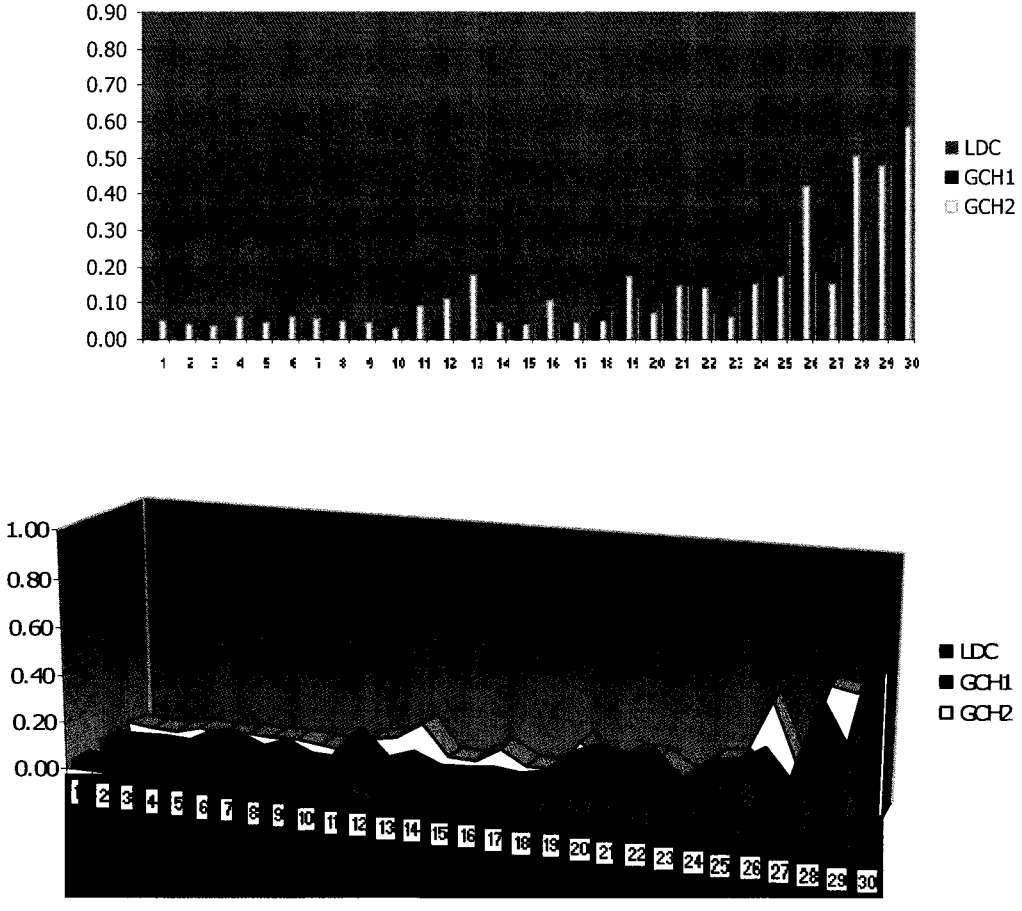

Fig. 43 Overall time Charts. Overall time for LDC and GCH (both tests).

Appendix D

This appendix shows us the elapsed partial time of the 3 iterations over the 30 image set; the time is measure in seconds. Both algorithms' time is measure without the edge detection process. At this point we are analyzing both algorithms taking a binary input image and computing the final result.

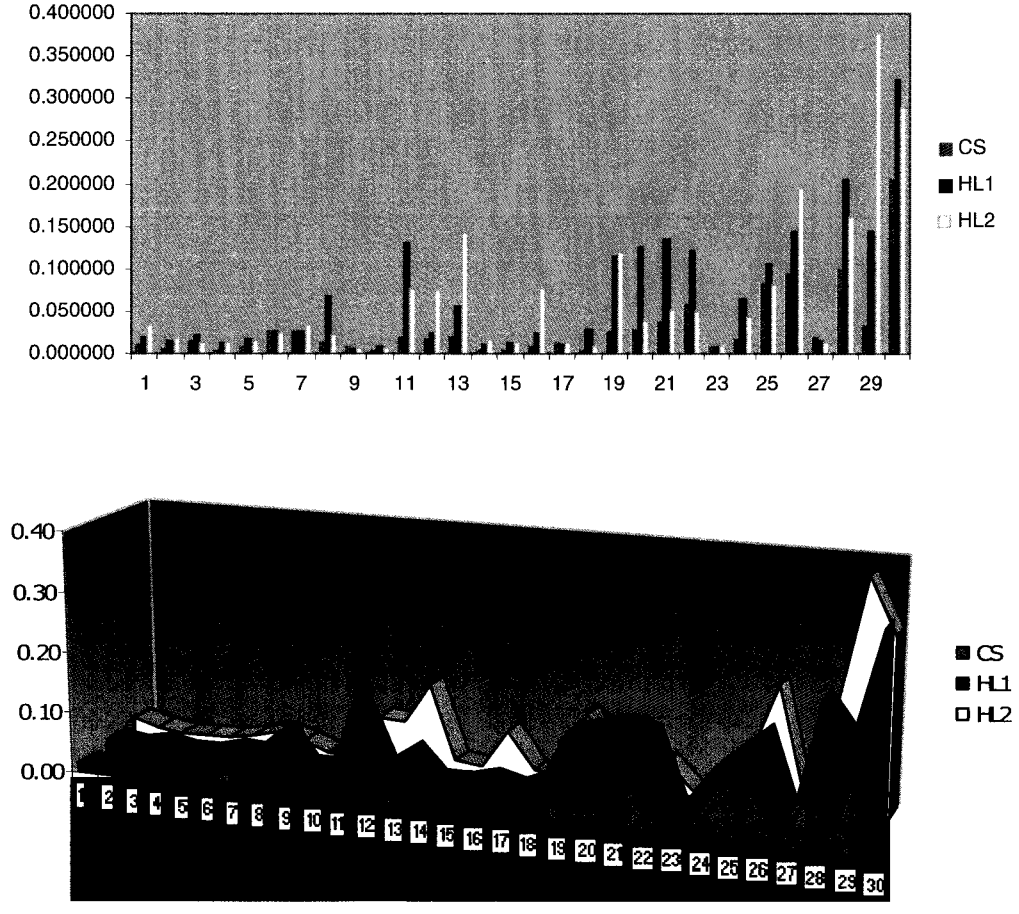

Fig. 44 Partial Time Charts. Partial times for CS and HL (both tests).

Appendix E

This appendix shows us the amount of line segments detected in images of the 3 test (LDC, GCH1, and GCH2) applied in the 30 image set.

Images with the result of these line segments detected are showed in Appendix A. and the elapsed complete and partial time charts in Appendix C and Appendix D. The complete sets of input parameter configuration used in each image are showed in Appendix F.

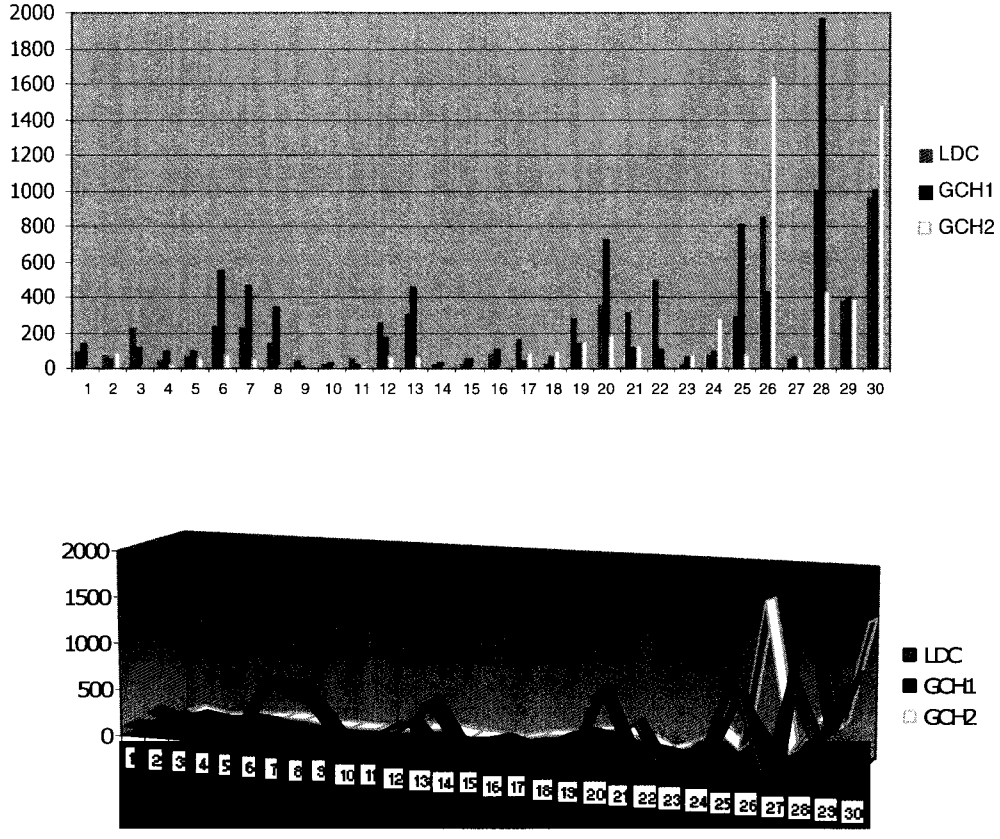

Fig. 45 Amount of segments detected charts. Amount of Segments detected by the two algorithms LDC and GCH (both tests).

Appendix F

Input parameters used in each image test. These parameters were selected by human classification over the lines detected images –Appendix A-.

Table 1 Parameter configurations.

	LDC		GCH1							GCH2						
	GKS								RR	CT						RR
1 -	1	7	117	141	10	30	20	45	1	111	25	10	53	89	60	1
2 -	1	5	714	190	1	30	20	45	1	714	190	120	30	20	45	1
3 -	1	5	135	549	10	38	20	29	1	287	641	16	156	132	27	16
4 -	1	5	147	10	10	19	20	45	1	147	202	4	31	31	38	1
5 -	1	7	30	440	10	30	20	45	1	30	440	56	104	20	45	1
6 -	1	7	40	10	10	30	20	8	1	40	10	10	30	141	10	1
7 -	3	3	74	10	10	30	20	45	1	74	10	10	30	120	38	1
8 -	1	7	56	10	4	22	132	209	38	68	10	16	153	71	25	31
9 -	1	5	141	178	10	56	1	32	1	196	312	53	56	147	23	16
10 -	1	7	208	10	10	34	20	45	1	208	10	71	92	19	34	1
11 -	9	7	56	488	1	34	1	163	4	56	488	1	34	241	163	4
12 -	5	7	86	464	10	38	50	27	7	50	1	16	7	114	32	1
13 -	3	5	178	10	10	31	19	51	1	178	10	7	68	34	170	1
14 -	3	5	74	10	77	30	20	89	4	74	10	77	30	86	126	4
15 -	1	5	135	10	10	30	20	45	1	135	10	10	126	50	40	1
16 -	1	5	65	165	10	30	20	45	1	65	165	50	83	147	121	1
17 -	1	7	38	105	10	44	1	27	1	31	13	16	44	7	25	1
18 -	3	5	105	11	34	1	13	113	1	38	10	47	13	1	27	1
19 -	7	5	86	68	7	47	95	75	1	86	117	34	47	95	75	1
20 -	15	7	208	190	16	30	20	62	1	208	190	16	62	20	62	1
21 -	3	7	385	10	345	114	95	113	1	829	232	25	86	147	251	348
22 -	5	7	92	44	5	68	28	62	4	354	44	6	123	83	104	4
23 -	1	5	360	232	77	56	4	19	92	360	232	77	56	4	38	1
24 -	5	7	184	10	114	77	62	45	1	421	1	47	101	101	25	141
25 -	1	3	141	10	7	34	44	40	1	141	10	7	141	44	40	1
26 -	17	7	80	26	3	56	20	22	1	66	26	26	56	20	22	1
27 -	3	7	184	287	107	30	20	45	1	184	287	16	1	20	45	1
28 -	5	5	30	105	7	30	20	45	1	30	105	7	56	126	45	1
29 -	15	7	105	10	41	22	62	69	1	98	31	38	132	22	24 6	1
30 -	7	5	190	10	10	30	20	45	1	190	10	25	47	1	13 9	1

The input parameters for the LDC are labeled: Normalize (N), Gaussian kernel size (SGK), Laplace aperture size (LAS), threshold (TH), and min length of contour (MLC).

The parameters for the GCH are labeled. Gap between lines (GBL); it is the maximum gap between line segments lying on the same line to treat them as the single line segment (i.e. to join them). Minimum line length (MLL); this is the minimum line length to detect as a line segment. Hough threshold (T); threshold used by the accumulator to find the points of local maxima and the lines. Theta resolution (TR), and Rho resolution (RR) are parameters to create the accumulator dual space. The other two parameters used for Hough transform based solution are the Canny Threshold (CT) and Linking Threshold (LT). Both belong to the Canny edge detection step.

Appendix G

Input parameters used in each image test. These parameters were selected by human classification over the lines detected images –Appendix A-.

Table 2 Image sizes and overall processing time.

		LDC	Gray Scale + Canny + HT		
	Image Size	LDC Time			
1 -	256x256	0.022396701	0.038463548	0.049976743	0.013853
2 -	300x337	0.018767190	0.040677516	0.039556703	0.012339
3 -	350x350	0.030943598	0.046170393	0.037006938	0.007666
4 -	429x300	0.025199292	0.039708957	0.061728236	0.018393
5 -	400x324	0.024821870	0.086807960	0.047070228	0.031402
6 -	448x336	0.046126812	0.094786628	0.059990027	0.025069
7 -	400x420	0.045539866	0.045004882	0.054615601	0.005401
8 -	480x360	0.035627992	0.084663554	0.053160946	0.024847
9 -	608x302	0.026335191	0.032142633	0.046262584	0.010249
10 -	587x316	0.027247876	0.033194722	0.031039699	0.003011
11 -	418x491	0.051857149	0.163165710	0.093260736	0.056259
12 -	580x357	0.048908451	0.058098166	0.113248472	0.034799
13 -	400x600	0.046337454	0.091276685	0.179591820	0.067794
14 -	500x500	0.027553781	0.040959954	0.047749085	0.010277
15 -	500x500	0.026896435	0.048923536	0.043119167	0.011417
16 -	500x500	0.029670251	0.058397087	0.108848471	0.040083
17 -	500x500	0.040062913	0.045439574	0.046929708	0.003612
18 -	640x480	0.034441249	0.068775221	0.048373746	0.017268
19 -	1024x420	0.070631603	0.161293684	0.174016251	0.056377
20 -	868x600	0.109475087	0.200524089	0.072196327	0.066015
21 -	900x599	0.097048927	0.174577216	0.145515142	0.039167
22 -	980x705	0.142885758	0.211807646	0.140879916	0.040384
23 -	1280x768	0.070896161	0.098777079	0.060113506	0.019952
24 -	1280x768	0.134137998	0.177371984	0.154949861	0.021622
25 -	1415x991	0.180014220	0.190818666	0.171008047	0.009919
26 -	1477x1007	0.320927101	0.253220451	0.420589768	0.084192
27 -	1459x1087	0.184850589	0.132874709	0.153281772	0.026187
28 -	1184x1598	0.253994293	0.456415550	0.503428508	0.132541
29 -	1700x1120	0.295997955	0.272502790	0.477448950	0.112160
30 -	2048x1536	0.480342055	0.805841525	0.583414931	0.166357

Table 3 Elapsed partial times.

		LDC	GCH		
	Image Size	GB			
1 -	256x256	0.010254	0.020125	0.033476	0.011654
2 -	300x337	0.007046	0.016854	0.019181	0.006440
3 -	350x350	0.016468	0.022589	0.014267	0.004312
4 -	429x300	0.004283	0.013594	0.013353	0.005308
5 -	400x324	0.008172	0.017030	0.014983	0.004638
6 -	448x336	0.026356	0.027797	0.025493	0.001164
7 -	400x420	0.026422	0.026506	0.033699	0.004177
8 -	480x360	0.013323	0.067904	0.023027	0.029118
9 -	608x302	0.007993	0.007134	0.006467	0.000765
10 -	587x316	0.005133	0.008232	0.006173	0.001577
11 -	418x491	0.020824	0.131845	0.077418	0.055514
12 -	580x357	0.017203	0.025224	0.074779	0.031185
13 -	400x600	0.021468	0.055375	0.141873	0.062087
14 -	500x500	0.003790	0.010996	0.016738	0.006488
15 -	500x500	0.004152	0.014688	0.012678	0.005594
16 -	500x500	0.009000	0.024214	0.076590	0.035457
17 -	500x500	0.012500	0.012214	0.014008	0.000964
18 -	640x480	0.004864	0.029751	0.009835	0.013170
19 -	1024x420	0.027175	0.116360	0.118814	0.052214
20 -	868x600	0.030141	0.126370	0.039507	0.053061
21 -	900x599	0.038338	0.135141	0.052027	0.052387
22 -	980x705	0.059032	0.122989	0.049697	0.039894
23 -	1280x768	0.009491	0.008399	0.008092	0.000736
24 -	1280x768	0.017833	0.066118	0.042283	0.024143
25 -	1415x991	0.084046	0.105631	0.080377	0.013645
26 -	1477x1007	0.093811	0.143948	0.194396	0.050292
27 -	1459x1087	0.019426	0.015715	0.013441	0.003021
28 -	1184x1598	0.098664	0.205871	0.161389	0.053862
29 -	1700x1120	0.033421	0.143691	0.378515	0.176253
30 -	2048x1536	0.206126	0.323206	0.288198	0.060096

The 3 tests time were measured just after the edge detection step and thresholded image was computed. The times show us how fast is the last two steps of our algorithms upon the HL OpenCV built-in algorithm.

Table 4 Lines detected by each algorithm's test.

		LDC	Gray + Canny + HL		
	Image Size	Lines Detected			
1 -	256x256	98	143	8	68.74
2 -	300x337	71	49	89	20.03
3 -	350x350	231	123	6	112.53
4 -	429x300	41	94	21	37.72
5 -	400x324	68	96	49	23.64
6 -	448x336	237	550	74	241.91
7 -	400x420	226	466	55	206.46
8 -	480x360	142	345	14	166.91
9 -	608x302	46	13	10	18.82
10 -	587x316	24	34	12	11.02
11 -	418x491	49	20	20	16.74
12 -	580x357	257	170	61	98.21
13 -	400x600	307	452	67	194.44
14 -	500x500	26	28	13	8.14
15 -	500x500	26	57	11	23.46
16 -	500x500	76	112	12	50.65
17 -	500x500	160	40	82	60.89
18 -	640x480	20	61	100	40.00
19 -	1024x420	277	144	154	74.07
20 -	868x600	358	726	188	275.01
21 -	900x599	310	116	133	107.44
22 -	980x705	496	103	5	259.85
23 -	1280x768	26	68	76	26.86
24 -	1280x768	72	94	286	117.72
25 -	1415x991	287	816	77	380.80
26 -	1477x1007	851	436	1639	611.06
27 -	1459x1087	54	62	65	5.69
28 -	1184x1598	1008	1967	435	774.06
29 -	1700x1120	380	395	390	7.64
30 -	2048x1536	966	1004	1483	288.15

Table 5 SNZ's linearity value of each line detected.

	LDC		
	Lines Detected	SNZ mean	
1 -	98	0.934554	0.125116
2 -	71	0.990746	0.038109
3 -	231	0.965833	0.072748
4 -	41	0.967099	0.082500
5 -	68	0.957474	0.080461
6 -	237	0.934675	0.145655
7 -	226	0.974441	0.070608
8 -	142	0.968754	0.060405
9 -	44	0.970346	0.073158
10 -	8	0.994375	0.011009
11 -	49	0.953591	0.120189
12 -	257	0.962210	0.076853
13 -	307	0.966489	0.076246
14 -	26	0.982007	0.066615
15 -	26	0.998001	0.002435
16 -	76	0.991494	0.026981
17 -	80	0.972731	0.043692
18 -	20	0.996181	0.004322
19 -	277	0.964996	0.073925
20 -	358	0.969676	0.056628
21 -	310	0.944627	0.135223
22 -	496	0.949342	0.106353
23 -	26	0.982485	0.059830
24 -	72	0.999279	0.000438
25 -	287	0.998429	0.006567
26 -	851	0.960725	0.080278
27 -	29	0.990878	0.016957
28 -	1008	0.951766	0.106373
29 -	380	0.964004	0.080964
30 -	966	0.952832	0.103912
	Mean	**0.970335**	**0.066818**
	Standard Deviation	**0.018806661**	**0.040520928**

For each line detected for our LDC algorithm we took all the list of points from the contour and compute the SNZ's linearity value. For each image we computed the average value of all SNZ linearity value from all the line segments detected and the standard deviation. We can see the amount of lines detected and the linearity SNZ mean and standard deviation for each of the 30 image set. The mean of the SNZ linearity value for all the lines detected in our 30 image set is 0.970775 and standard deviation 0.018806661. These values so close to the linearity criterion of a value 1 show us the mathematical accuracy of our solution.