DIPLOMA THESIS

Hardware Acceleration for Line Segment Detection

Submitted at the Faculty of Electrical Engineering and Information Technology,
TU Wien
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Prof. Axel Jantsch
Dr. Nima Taherinejad

Institut für Computertechnik (E384)
TU Wien

by

Christoph Ossimitz
Matr.Nr. 01126614

March 3, 2021

Abstract

Line segment detection is an important preprocessing step in Computer Vision applications. However, most of the existing algorithms are designed to be run on general purpose processors with high power consumption and will not achieve high framerates on low-cost embedded hardware. Furthermore, these algorithms are usually not suited for hard real-time applications. In this thesis, two novel methods are proposed: The first is an optimization for the Line Segment Detector (LSD), a widely used line segment detection algorithm, using a lookup table (LUT). We show that this optimization decreases the processing time by 13.08% on average, up to a maximum of 28.8%, while keeping 99.84% of the output identical. Secondly, a novel line segment detection algorithm is proposed, designed for implementation on field-programmable gate arrays (FPGAs). An implementation of this algorithm on a Xilinx XC7Z015 FPGA runs at 100 MHz while using less than 10% of the available on-chip resources. Images in 640×480 resolution can be processed at 325.5 frames per second and a latency of 32.27 μs, while for the maximum resolution of 1080p (1920×1080) 48.23 frames per second can be achieved with an latency of 96.27 μs. Additionally, the latency and throughput of the algorithm are only depending on the image resolution, but not on the image contents, making it compatible with hard real-time environments. The algorithm beats existing FPGA-based line segment detectors in terms of latency, while using less on-chip resources and achieving a comparable detection quality.

Kurzfassung

Linien-Segment-Erkennung ist ein Verarbeitungsschritt, der in vielen Computer-Vision Anwendungen Verwendung findet. Die meisten dieser Algorithmen wurden allerdings für Desktop-CPUs mit einer hohen Stromaufnahme entwickelt, und skalieren schlecht auf jener kostengünstigen Hardware, die im Embedded-Bereich häufig Anwendung findet. Desweiteren sind diese Algorithmen nur teilweise für harte Echtzeit-Anwendungen geeignet. In dieser Diplomarbeit werden zwei neue Ansätze präsentiert: Zum einen wird eine Verbesserung für den Line Segment Detector (LSD), einen häufig verwendeten Linien-Segment-Erkennungs-Algorithmus, vorgeschlagen. Durch diese Verbesserung reduziert sich die Verarbeitungszeit pro Bild im Durchschnitt um 13,08% und in einem Testfall um 28.8%, während die Ausgabe des Algorithmus zu 99,84% identisch blieb. Desweiteren wird ein neuer Algorithmus zur Linien-Segment-Erkennung gezeigt. Dieser wurde speziell zur Implementierung auf Field Programmable Gate Arrays (FPGAs) entwickelt. Eine Implementierung auf einem Xilinx XC7Z015 FPGA erreicht eine maximale Taktfrequenz von 100 MHz und benutzt weniger als 10% der verfügbaren Resourcen des Chips. Bilder in VGA-Qualität (640×480) können damit mit einer Bildrate von 325,5 Bildern pro Sekunde und einer Latenz von 32,27 μs berechnet werden. In der höchsten unterstützten Bildauflösung, 1920×1080, können 48,23 Bilder pro Sekunde berechnet werden, bei einer Latenz von 96,27 μs. Da Latenz und Durchsatz nur von der Auflösung und nicht vom Bildinhalt abhängig sind, lässt sich der Algorithmus auch in harten Echtzeit-Umgebungen einsetzen. Der Algorithmus hat im Vergleich zu bestehenden FPGA-basierten Lösungen zur Linien-Segment-Erkennung eine niedrigere Latenz, benötigt dabei weniger Chip-Resourcen und hat eine vergleichbare Zuverlässigkeit bei der Erkennung.

Acknowledgements

First and foremost, I'd like to thank Mission Embedded GmbH, in particular Michael Kreilmeier, MSc, MBA, for providing me with necessary resources and making this project possible.
I'd like to thank my supervisor, Prof. Axel Jantsch, for his time and the feedback he provided me with throughout the project.
I'd like to thank Dr. Nima Taherinejad for the invaluable knowledge and editing work he contributed throughout the project, as well as Clemens Reisner, MSc, from Mission Embedded for his inputs.
I'd like to thank my colleague and dear friend, Tobias Watzinger, BSc, who helped me with proofreading this document.
Finally, I'd like to thank Isabella and Vincent for giving me emotional support and encouragement throughout the project, as well as my father for his continued support throughout the years.

Preface

A version of Chapter 3 will be published as *"A Fast Line Segment Detector Using Approximate Computing"* at the 2021 IEEE International Symposium on Circuits & Systems (ISCAS), pp.1-5, 2021 [OT21].

A version of Chapter 4 was submitted for publication as *"A Novel Hardware Accelerator for Line Segment Detection"* to the IEEE Transactions on Circuits and Systems for Video Technology journal in March 2021.

Table of Contents

Glossary

ASIC application-specific integrated circuit. 38

BRAM block RAM. 38, 62, 66

CORDIC Coordinate Rotation Digital Computer. 22

CPU central processing unit. 1, 19, 26, 62

DSP digital signal processor. 66

ED Edge Drawing. 14

FIFO first in, first out buffer. 44, 50, 53, 57, 58, 59, 60, 62, 63, 66

FPGA field-programmable gate array. 1, 2, 19, 23, 38, 62, 63, 64, 65, 75

GPU graphics processing unit. 19, 26

LSD Line Segment Detector. 1, 2, 7, 9, 10, 13, 14, 15, 18, 24, 26, 27, 28, 29, 35, 66, 68, 69, 71, 75

LSF Least Squares Fitting algorithm. 16, 18

LUT lookup table. 2, 22, 27, 29, 30, 31, 51, 75

NFA number of false alarms. 9, 13, 14, 18, 27, 28, 30, 31, 32, 31, 36, 69, 71, 73

PLL phase-locked loop. 63

RAM random-access memory. 1, 2, 44, 53

RTL register-transfer level. 53

SAR Synthetic Aperture Radar. 26

VHDL Very High Speed Integrated Circuit Hardware Description Language. 53

WRM weighting resistor matrix. 26

1 Introduction

In recent years, research interest in autonomous vehicle driving has risen continuously. From the years 1970 to 2019, a total of 18,153 publications on this subject were recorded on the Scopus[1] publication database - more than half of them (9,387) were published between 2017 and 2019, with more than a quarter of them (4,841) in 2019 alone [MWP20]. For this reason, there is an increasing demand for fast and real-time computer vision algorithms. For a certain class of applications, like the detection of road lanes [LLZZ18], it is necessary to detect primitive geometric structures as a preprocessing step, with *lines* and *line segments* among the most basic of those structures. Applications in other domains include stereo matching [LYXL16], target tracking [LGL16], and detection of barcodes [CA16], power lines [SMA+17], sea-sky lines [MJ16] and airports [BAS18] [LCC+18].

A line in a Cartesian plane is a set of points that can be defined by one or more linear equations that the coordinates of all points in that line must satisfy. In an infinite plane, a line stretches infinitely in two directions. In contrast, a line segment is a subset of a line, limited to all points lying between two endpoints. Line segment detection is employed when small local features are relevant for the application, whereas line detection is used when image-wide characteristics are more relevant. Different algorithms exist for both tasks. While we will give an introduction to the existing work on line-detection in Section 2.4, the major focus of this work lies on the detection of line segments. In the context of computer-vision, a line segment is a locally straight contour on an image [GJMR12]. Contours are zones of a sharp change in a characteristic of the image [GJMR12]. This usually refers to the intensity or brightness of the image. For this reason, the algorithms and methods presented in this work use grey-scale images as input, which only contain the intensity value for each pixel.

Different methods for line segment detection have been proposed. Among these, the Line Segment Detector (LSD) algorithm by von Gioi et al. [GJMR12] is very commonly used and has been part of the OpenCV[2] library. However, there are problems in employing LSD for embedded computer applications. For one, the algorithm runs at a moderate speed, taking several hundred miliseconds for a 1200×1200 pixel image on our reference machine (an i5-3450 desktop central processing unit (CPU) with 6 MB Level 3 Cache, and 8 GB DDR3 random-access memory (RAM)) Additionally, the algorithm is not compatible with hard real-time environments, as run-time is highly dependent on the contents of the image. As part of our experiments, the fastest and slowest processing times of the algorithm on our test image set, containing 40 images of the same resolution, differed by a factor of 4. For more details on the experiment, refer to Chapter 3.

[1] `https://www.scopus.com/`
[2] `https://opencv.org` (Accessed: November 20, 2020)

Mission Embedded[3] is using LSD as part of an algorithm for locality determination in a rail-based vehicle. The detection results of LSD are used to find the course of the rail tracks on the image, which in turn is used to find the position of the vehicle on the current route. This solution currently runs on an Industrial PC. The scope of this work is to evaluate whether it is feasible to outsource the line segment detection to a hardware accelerator - in this case, implemented on an field-programmable gate array (FPGA). However, porting the LSD algorithm to hardware poses a few challenges:

1. LSD requires the usage of multiple frame buffers, including random access to them. With on-chip RAM being a scarce resource, the connection to the off-chip RAM can quickly turn into a bottle neck. While image compression could aid the issue with high on-chip RAM usage, Hagara et al. [HSB+20] have shown that this has a negative impact on the detection quality of commonly used edge detectors (Section 2.1).
2. LSD is largely incompatible to a pipelined design due to internally used data structures (lists) with unbound size, unbound numbers of iterations on certain processing steps and data dependency between iterations.
3. LSD uses floating-point arithmetic and highly demanding mathematical operations for some calculations.

While we were able to solve the third problem by eliminating the most involved calculations with the use of a lookup table (LUT) (Chapter 3), we came to the conclusion that eliminating the need for a frame buffer in particular would require a major redesign of the algorithm.

For this reason, a new algorithm was implemented from scratch, titled the *step-length algorithm*. During our work on the step-length algorithm, a similar work called the *Modified LSD* algorithm was published [ZCW18]. We could show that the step-length algorithm improves upon the Modified LSD in FPGA resource usage and latency, while evaluations of the detection quality have shown very comparable results.

In Chapter 2, different existing algorithms for line and line segment detection are discussed. In Chapter 3, we present a LUT based optimization for the LSD algorithm and analyze performance improvements and impacts on the detection results. In Chapter 4, the step-length algorithm is presented and compared to the Modified LSD algorithm. Finally, we draw our conclusions in Chapter 5.

[3] `https://mission-embedded.com/en/` (Accessed: November 29, 2020)

2 Related Work

In this chapter, existing line segment detection algorithms designed for software and hardware implementations are reviewed. In Section 2.1, edge detection is explained, as it is a common element of most line segment detection algorithms. The succeeding Sections cover state-of-the-art line segment detection algorithms that are explained in detail.

2.1 Edge detection

Several algorithms for detection of lines and line segments use edge detection methods, either by partially employing some of their techniques or fully incorporating an entire algorithm for this purpose. A very commonly used algorithm for edge detection is Canny's algorithm [Can86]. While Canny's algorithm was primarily designed for software implementations, Canny's algorithm can be implemented in hardware if slight modifications are applied.

Canny's algorithm can be decomposed into several separate processing steps, which are covered in the succeeding subsections.

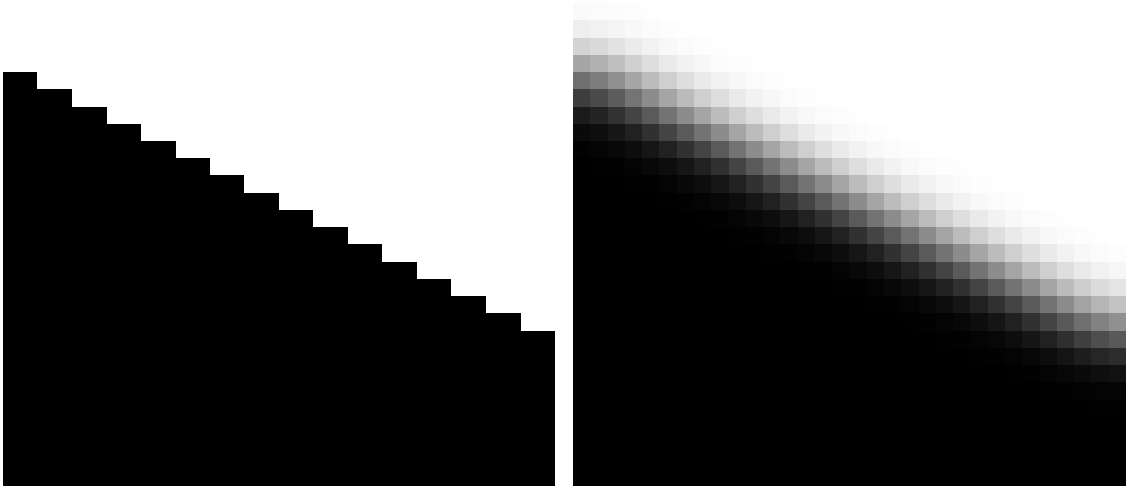
2.1.1 Gaussian blurring

Image noise refers to random variations of pixel intensity and is usually introduced during the capture of an image at the sensor level. Noise can be hard to avoid, depending on the available equipment. The effect is very noticeable in flat areas of similar intensity, such as blue skies. Image noise has a negative impact on the gradient, which is our main detection mechanism. Therefore, countermeasures to image noise are desirable.

Gaussian blurring (also known as Gaussian filtering) is a commonly used technique to reduce image noise, performed before any other processing step. For each pixel, a weighted average of intensities within a neighborhood of the pixel is calculated. Gaussian blurring can be implemented as convolution with a kernel - a 5×5 kernel for Gaussian blurring is shown in Equation 2.1.

$$K_G = \begin{bmatrix} 0.00377 & 0.01502 & 0.02379 & 0.01502 & 0.00377 \\ 0.01502 & 0.05991 & 0.09491 & 0.05991 & 0.01502 \\ 0.02379 & 0.09491 & 0.15034 & 0.09491 & 0.02379 \\ 0.01502 & 0.05991 & 0.09491 & 0.05991 & 0.01502 \\ 0.00377 & 0.01502 & 0.02379 & 0.01502 & 0.00377 \end{bmatrix} \tag{2.1}$$

Gaussian blurring also reduces negative side-effects of quantization artifacts like the staircase effect, which can result in a straight diagonal contour being detected as several disconnected horizontal line segments on each step, or not being detected at all (Figure 2.1).



(a) Original **(b)** Blurred

Figure 2.1: Effect of Gaussian blurring on a staircase.

The filtering can be decomposed into two passes. Instead of a convolution using a two-dimensional kernel, one can apply blurring the image twice using an one-dimensional kernel - once performing convolution on neighbor values in the same row and once performing convolution on neighbor values in the same column. This approach is often faster in a software implementation, and can also be used in hardware implementations to save resources and achieve a higher clock frequency since this approach requires less multipliers and smaller adder chains [ZCW18].

2.1.2 Gradient calculation

Detection of contour pixels is often based on the *image gradient*. Interpreting the intensity of pixels in a row or column of the image as a discrete function, the image gradient can be defined as the derivative of that function. Since derivatives only exist for continuous and differentiable functions, the gradient has to be approximated. Usually the gradients of the intensity in horizontal and vertical direction will be calculated, thus resulting in two values G_x (horizontal gradient) and G_y (vertical gradient) for each pixel. Commonly, the calculation is implemented as a convolution of the image with a kernel. The Prewitt (Equations 2.2a and 2.2b) and Sobel (Equations 2.3a and 2.3b) kernels are commonly used in the Canny algorithm. Implementations from von Gioi et al. [GJMR12] and Zhou et al. [ZCW18] use a 2×2 kernel instead (Equations 2.4a and 2.4b - the

matrix calculates the gradients for the top left pixel).

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.2a}$$

$$K_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.2b}$$

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.3a}$$

$$K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.3b}$$

$$K_x = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \tag{2.4a}$$

$$K_y = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \tag{2.4b}$$

From the horizontal and vertical gradient values G_x and G_y, the gradient magnitude G and gradient angle ϕ of each pixel can be calculated. In Canny's algorithm, equations 2.5 and 2.6 are used for this purpose. While the magnitude gives us an information about the strength of the contour, the angle is equal to the contour's orientation (within a tolerance).

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.5}$$

$$\phi = \arctan(\frac{G_x}{-G_y}) \tag{2.6}$$

Due to the complexity of the square root and arctangent operators, they were simplified in the Modified LSD algorithm [ZCW18]: Instead of the Pythagorean sum, the sum of absolutes is used for the gradient magnitude (Equation 2.7). Using multiplications and comparisons, the gradient angle is quantized into one of 8 direction bins and stored in a 3-bit angle value, which the authors deemed as sufficiently accurate.

$$G = |G_x| + |G_y| \tag{2.7}$$

2.1.3 Edge thinning

For some algorithms, sharp edges that are one pixel wide are required. Since the output of the gradient map is often diffuse, an *edge-thinning* step is performed. In Canny's algorithm, this step is called *non-maximum suppression*.

A pixel is only considered an edge pixel if its gradient magnitude is higher than the gradient magnitude of the two neighbors that are orthogonal to the edge direction (based on gradient angle, Figure 2.2). If one of the neighbors has a higher gradient magnitude, the pixel's gradient magnitude is set to zero.

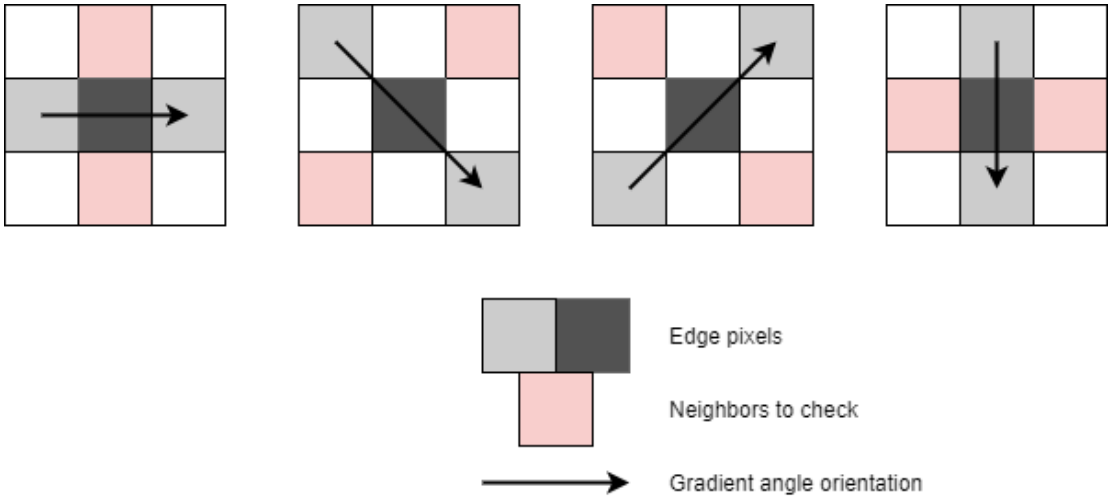


Figure 2.2: Non-maximum suppression. The gradient magnitude of the dark grey pixels is checked against the gradient magnitude of the pink pixels.

2.1.4 Double thresholding

Gaussian blurring alone is often insufficient to deal with the image noise. Noise is especially relevant in flat areas of similar intensity, which might cause edge pixels to appear within them. These false-positive edges are often characterized by a weak gradient magnitude, small extent and isolation from other edges. Canny's algorithm uses *double thresholding* to reduce the effects of noise on the edge map.

Canny's algorithm deals with this problem by introducing two threshold values $T_l, T_h : T_l \leq T_h$. These thresholds are used to categorize a pixel (x, y) with regards to its gradient magnitude $G(x, y)$:

- A pixel is a non-edge if $G(x, y) < T_l$
- A pixel is a strong edge if $G(x, y) > T_h$
- A pixel is a weak edge if $T_l \leq G(x, y) \leq T_h$

A pixel is only considered an edge if it is either categorized as a strong edge or if it is a weak edge and adjacent to another edge pixel. This allows noise-induced edges in isolated, flat areas

to be removed from the edge map. To fully update the status of each weak edge pixel, we need to iterate over all pixels twice: Once in the default scanline order (row-wise left to right, top to bottom), and once in the reverse order (row-wise right to left, bottom to top). While this is less of a problem in software implementations, we cannot reverse the pixel order for the second iteration on hardware without buffering the whole image and delaying operation until the first pass has been completed. Therefore, the double thresholding step is either omitted completely, or pixels are only updated once (in the default order), which is the approach used by Zhou et al. [ZCW18].

2.1.5 Sliding windows for hardware implementations

Software algorithms in computer vision applications are often designed under the assumption that the global image data is available at any calculation step, thus making the usage of frame buffers mandatory. For high-performance hardware implementations, frame buffers are undesirable due to limited on-chip memory and a relatively high delay for accessing off-chip memory. To limit access to the global memory, the maximum locality needs to be limited for any calculation step carried out in hardware. The locality needs to be sufficiently large to allow the algorithm to produce qualitatively good results while keeping it as small as possible to reduce on-chip memory usage and the delay required to fill the buffers.

Many operations in the domain of computer vision can be formally described as a discrete convolution of the image with a matrix of weights, called the *kernel*. The locality of the convolution operation is defined by the size of the kernel. We assume that one pixel is being processed per clock cycle, starting with the top-left corner pixel, proceeding row-wise from left to right, top to bottom. Therefore, for a 3×3 kernel, we need buffers to delay and store two lines, since we require pixels from a total of three different lines (Figure 2.3). In Zhou's work [ZCW18], these rectangular localities are referred to as *sliding windows*. A generic hardware module for a sliding window-based operation is shown in figure 2.4. All of the processing steps for the Canny algorithm can be performed using sliding windows.

A special case is the sliding window for the double thresholding. The output data is used for the calculation of the succeeding pixels, therefore the top and left part of the sliding window consist of pixels that were already updated. This is referred to as a *heterogeneous sliding window* due to its input coming from different (heterogeneous) sources (Figure 2.5).

2.2 Line Segment Detector

The LSD algorithm was published by von Gioi et al. in 2010 [GJMR12]. The detection is based on grouping adjacent pixels with similar gradient angle. It uses the *a contrario* method by Deolneux et al. [DMM00] [DMM08] to validate each line segment candidate. LSD implements a basic form of edge detection, limited only to downscaling (which has the same purpose as blurring) and gradient calculation steps. LSD was designed for software and due to its iterative design, the weak locality in the separate processing steps and the complex arithmetic operations used, it is not well suited for a hardware implementation.

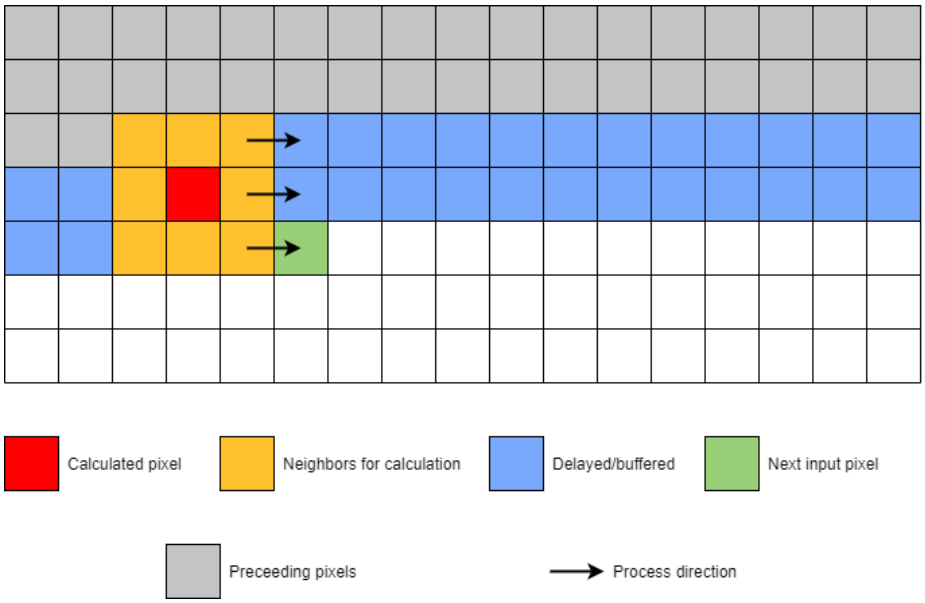


Figure 2.3: A 3×3 kernel operating on an image. The sliding window consists of the orange and red pixels [ZCW18].

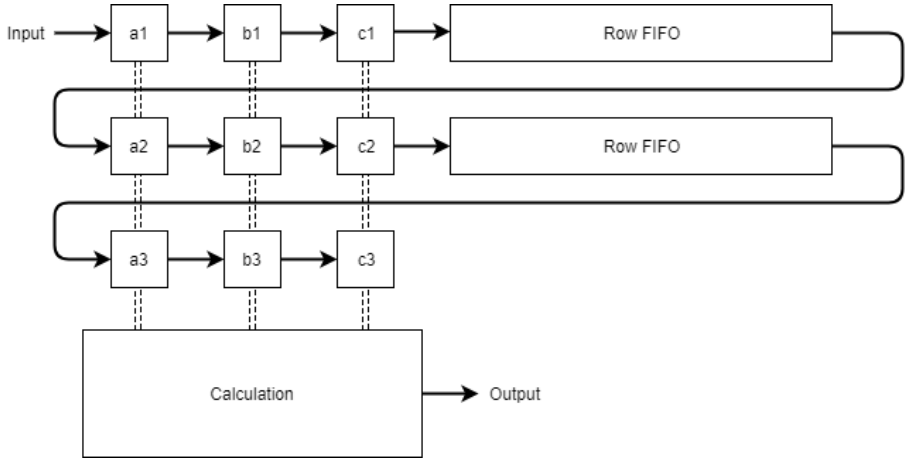


Figure 2.4: Hardware module for a 3×3 sliding window. The registers of pixels in the window a1 - c3 are connected to an arbitrary output calculation module

2.2.1 Algorithm overview

1. **Scaling:** Scale the image down.
2. **Gradient:** Calculate the image gradient.
3. **Ordering:** Pseudo-Order pixels by gradient magnitude, in descending order.
4. **Threshold:** Pixels with magnitudes below a certain threshold are marked as `USED`, others as `UNUSED`.
5. For each `UNUSED` pixel P from the ordered list, perform the following steps:

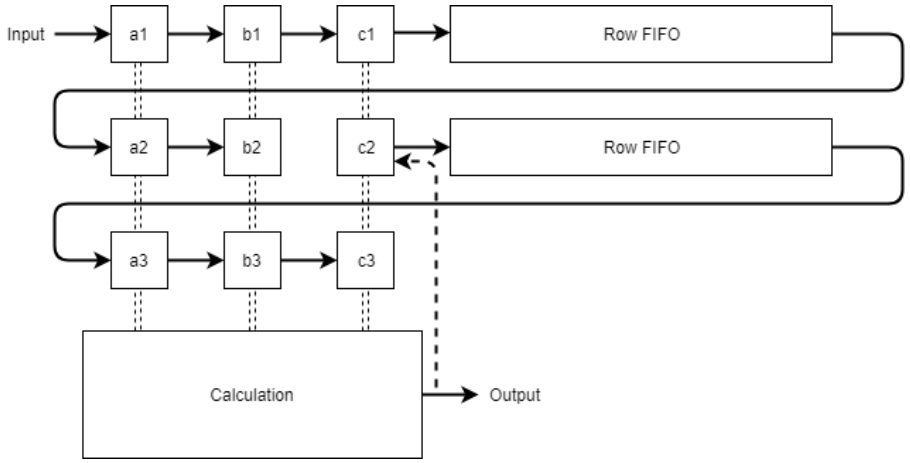


Figure 2.5: Hardware module for a 3×3 heterogeneous sliding window

(a) **Region Growth:** Starting from a seed pixel (initially P), `UNUSED` neighboring pixels with a similar gradient angle (within a tolerance angle τ) are added to the region. Added pixels are marked as `USED`.

(b) **Rectangle Approx.:** A rectangle that covers the region is calculated.

(c) **Region Cut:** In case the density of angle aligned pixels within a rectangle is low, the algorithm tries to cut the region, resulting in smaller regions (enclosed by tighter rectangles). Pixels removed from the region are marked as `UNUSED`.

(d) **Rectangle improvement and Validation:** Various improvements on the current rectangle are performed and each time, the number of false alarms (NFA) value of the improved rectangle according to the *a contrario* model is calculated. In case the NFA of an improved rectangle is below a threshold, the rectangle is regarded as a detected line segment.

2.2.2 Scaling

LSD has trouble operating on images where quantization artifacts like the staircase effect are present, since these edges might not be detected reliably. To cope with this problem, the image is scaled down, using Gaussian sub-sampling. This approach is favored over simply using a blurring filter on the image since the latter would interfere with the *a contrario* validation, as blurring changes statistics of a white-noise image. Using this scaling approach, the properties of a white-noise image are preserved. By default, the image is scaled by a factor of 0.8. This value was chosen because it is the smallest reduction in image size that reasonably solves the staircase issue, according to von Gioi [GJMR12].

In [Gio14], von Gioi proposed an alternative method: Instead of downscaling, it is feasible to use a blurred version of the image to find line segment candidates, and then validate them using the unblurred, original image. The disadvantage of this approach is that the magnitudes have to be calculated twice - once for the blurred and once for the unblurred image.

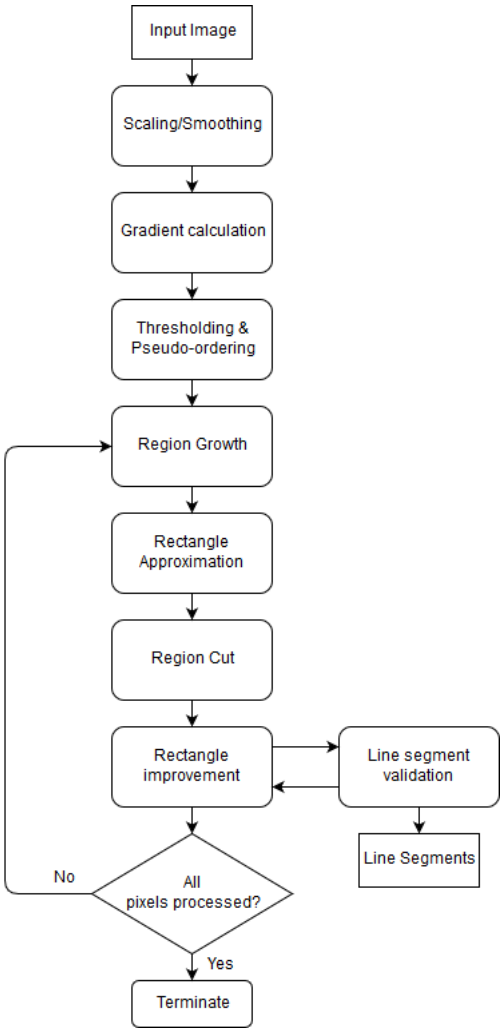


Figure 2.6: Overview of the LSD algorithm.

2.2.3 Gradient

Gradient calculation uses a 2×2 kernel, described in Equation 2.4a and Equation 2.4b. A simple gradient calculation scheme is used to minimize the dependency of the gradient values, which according to the author is important for the *a contrario* method. The pythagorean sum (Equation 2.5) and the arctan function (Equation 2.6) are used to calculate gradient magnitude and angle, respectively.

2.2.4 Ordering

LSD is a greedy algorithm, trying to build regions starting from promising pixels with a high gradient magnitude. Therefore, pixels need to be ordered. However, since sorting algorithms are expensive with regards to complexity, and since a rough ordering is sufficient for our purposes, a linear pseudo-ordering is used: Similar to histogram construction, each pixel is put into one of n bins (default value $n = 1024$). The bin with index i (indexes ranging from 1 to n) contains mag-

nitude values in the range $[\frac{i-1}{n}G_{max}, \frac{i}{n}G_{max})$, where G_{max} is the maximum gradient magnitude among all pixels.

2.2.5 Threshold

Pixels with small gradient magnitudes are not relevant for line detection. Therefore, a threshold ρ is defined, and all pixels below this threshold are marked as `USED` and can therefore no longer be added to regions later in the algorithm. ρ is calculated depending on another internal parameter, the angle tolerance τ since the angle error introduced by quantization depends on the gradient magnitude of the pixel. ρ is calculated by Equation 2.8.

$$\rho = \frac{q}{\sin \tau} \tag{2.8}$$

q is the bound of the gradient magnitude error due to quantization. The algorithm uses $q = 2$ per default for a more conservative bound, and experimentation by von Gioi showed that $\tau = 22.5°$ is a good value [GJMR12].

2.2.6 Region Growth

Each region R has a single pixel as its starting point (seed) and `UNUSED` pixels are used as seeds in descending order. The region's angle θ_r is initialized to the gradient angle of the starting pixel and the starting pixel is marked as `USED`. Next, the algorithm iterates over all pixels in the neighborhood of the region R. If a neighboring pixel (x, y) with gradient angle $\phi(x, y)$ is `UNUSED` and $|\theta_r - \phi(x, y)| \leq \tau$ holds, (x, y) is marked as `USED`, added to the data structure of region R and θ_r is updated according to Equation 2.9. This process is repeated until there are no pixels in the neighborhood of R eligible for region growth.

$$\theta_r = \arctan \frac{\sum_{(x,y)\in R} \sin \phi(x, y)}{\sum_{(x,y)\in R} \cos \phi(x, y)} \tag{2.9}$$

2.2.7 Rectangle Approximation

Rectangle approximation is done in three steps:

1. The center of the rectangle is set to be the "center of mass" of R, using the gradient magnitudes as weight values. The center point (c_x, c_y) of the region R is calculated as described in Equations 2.10 and 2.11

2. The angle of the rectangle is determined. This is defined by the angle of the smallest eigenvector of the matrix (Equations 2.12 - 2.15).

3. The width and length of the rectangle are chosen so that all pixels in the region fit inside the rectangle.

$$c_x = \frac{\sum_{(x,y)\in R} G(x,y) \cdot x}{\sum_{(x,y)\in R} G(x,y)} \tag{2.10}$$

$$c_y = \frac{\sum_{(x,y)\in R} G(x,y) \cdot y}{\sum_{(x,y)\in R} G(x,y)} \tag{2.11}$$

$$M = \begin{pmatrix} m^{xx} & m^{xy} \\ m^{xy} & m^{yy} \end{pmatrix} \tag{2.12}$$

$$m^{xx} = \frac{\sum_{(x,y)\in R} G(x,y) \cdot (x - c_x)^2}{\sum_{(x,y)\in R} G(x,y)} \tag{2.13}$$

$$m^{yy} = \frac{\sum_{(x,y)\in R} G(x,y) \cdot (y - c_y)^2}{\sum_{(x,y)\in R} G(x,y)} \tag{2.14}$$

$$m^{xy} = \frac{\sum_{(x,y)\in R} G(x,y) \cdot (x - c_x) \cdot (y - c_y)}{\sum_{(x,y)\in R} G(x,y)} \tag{2.15}$$

2.2.8 Region Cut

Sometimes, two lines with an angle difference below τ are caught in a single region. This will cause the resulting rectangle to be large, but sparsely populated with region pixels and will thus likely fail the validation step. Therefore, in case the density of correctly aligned pixels in the rectangle is below a certain threshold D ($D = 0.7$ per default), two different strategies are used to decrease the size and increase the density of a region: Reduction of the angle tolerance, and reduction of the region radius.

2.2.8.1 Reduce Angle Tolerance

The first approach is to repeat the region growth and rectangle approximation process with a smaller value for angle tolerance. The new angle tolerance is selected by examining pixels that are no further distanced from the seed pixel than the width of the rectangle. The new angle tolerance is set to twice the standard deviation of all examined pixels' gradient angles. In case the new rectangle satisfies the density requirement, the algorithm proceeds to rectangle improvement. Otherwise, another method is used to cut regions.

2.2.8.2 Reduce Region Radius

This method is repetitively applied to the region resulting from the previous step. The *radius* of the region is defined as the maximum distance from the seed pixel to any pixel in the region. Each iteration, the maximum radius is reduced by 25% and all pixels outside the new maximum radius are removed from the region. This process is repeated until a region/rectangle with sufficient density is found, or until the region is too small to be meaningful (default: 2 pixels per region is the minimum).

2.2.9 A contrario validation

Detection algorithms often require threshold parameters that have to be tuned. In case the threshold is set to high, some important structures might not be detected. In case the threshold is set too low, accidental structures are detected as well. The *a contrario* approach implicitly selects the parameter using the following idea: The parameter is set so that a fixed, small number of structures with a certain characteristics is detected on a white-noise image. This way, the threshold is set high enough that most accidental structures are rejected, while allowing the best possible number of real detections.

The *a contrario* approach uses the NFA as a measure of how likely a detection is in a random image. In LSD, the NFA is calculated as follows:

$$\mathrm{NFA}(n,k,p) = (N \cdot M)^{5/2} \cdot \gamma \cdot \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} \tag{2.16}$$

N and M are the height and width of the (scaled) image, γ is the maximum number of different precision values tested (default: $\gamma = 11$), p is the precision (initially set to $p = \tau/\pi$, later to be reduced during rectangle refinement), k is the number of p-aligned pixels, that is, pixels with gradient angle equal to the rectangle's angle up to a tolerance of $\tau = \pi \cdot p$, and n is the number of pixels in the rectangle. If the NFA is above a certain threshold ε, the line segment is regarded as "non-meaningful" and rejected (default value: $\varepsilon = 1$). Although other values have been tested as well, this change has shown little impact on the results according to von Gioi [GJMR12].

In the reference implementation $-log_{10}(NFA)$ is calculated instead. The binomial is calculated using the logarithmic gamma function, which itself is approximated using Windschitl and Lanczos methods.

2.2.10 Rectangle Refinement

After the NFA check of the region/rectangle with $p = \tau/\pi$ has failed due to the NFA value being above the threshold ε, various improvements on each rectangle are performed. There is a set of five improvement steps, and in each step either the precision or the width are adjusted up to five times.

Every time one of these two parameters is updated, the NFA value is recalculated. In case a meaningful rectangle ($NFA \leq \varepsilon$) is found, the rectangle improvement routine is stopped and the detection of the line segment is reported. If no meaningful rectangle was found after the fifth step has been completed, the line segment candidate is rejected. The following subsections describe the five improvement steps.

2.2.10.1 Try finer precision values

The NFA calculation is redone with finer precision values: Starting from the initial precision p, the values $p/2$, $p/4$,$p/8$, $p/16$, $p/32$ are tested. In case the pixels in our region are within a narrow range of gradient angles, k will be about the same, while a smaller p causes the NFA to be smaller. The p value with the smallest NFA is kept.

2.2.10.2 Reduce Width from both sides

The width of the rectangle is reduced, from both sides. Assuming W is the initial width, the following values are tested: W, $W - 0.5$, $W - 1$, $W - 1.5$, $W - 2$, $W - 2.5$. The best value with the best NFA is kept.

2.2.10.3 Reduce Width from one side (2x)

The width is reduced, but this time only pixels from one side of the rectangle are removed, moving the center accordingly in the process. The same width reductions as in the previous step are tested. Afterwards, the value with the best NFA is kept and the whole process is repeated for the other side of the rectangle. The width reduction steps are supposed to help in case the rectangle width was increased for only a small number of pixels with the correct angle alignment.

2.2.10.4 Try finer precision values

This repeats the first step. Starting from the precision value p kept previously, the precision is halved with every iteration, up to five times. In total, up to 11 precision values are tested, which explains the selected value $\gamma = 11$.

2.3 EDLines

EDLines was published in 2011 by Akinlar and Topal [AT11]. Like LSD, EDLines is based on gradients and *a contrario* validation, but uses a different approach. The algorithm first runs Edge Drawing (ED) [TAG10], an edge detection algorithm by the same authors, and then tries to fit lines in the detected edge segments (which Akinlar and Toal refer to as *pixel chains*). The line segment candidates are validated using the *a contrario* method. The algorithm takes a gray scale image as an input, and returns a list of detected line segments.

2.3.1 Algorithm overview

1. Perform edge drawing (ED):
   (a) **Noise suppression:** Smooth the image with a filter.
   (b) **Gradient:** Calculate the image gradient.
   (c) **Anchors:** Find anchors (pixels with gradient magnitude peaks).
   (d) **Linking:** Draw edges by connecting anchors.
2. For every detected edge segment (pixel chain):
   (a) **Line segment Fitting:** Map lines segments onto each pixel chain.
   (b) **Line segment Validation:** Validate the line segment by calculating the NFA and checking it against a threshold ε.

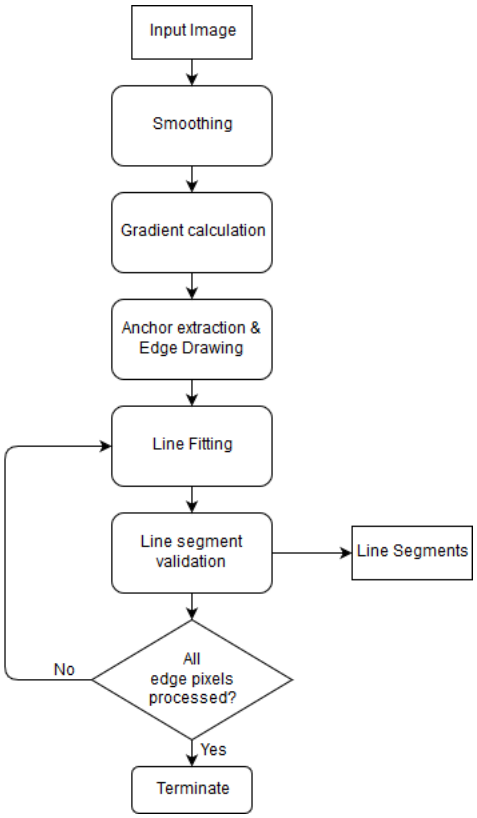


Figure 2.7: Overview of the EDLines algorithm.

2.3.2 Noise suppression

To reduce the amount of noise in the picture, a smoothing filter is applied. Akinlar and Topal used a 5x5 Gaussian kernel with $\sigma = 1$ for this purpose [AT11]. Note that the author does not mention the effect of image blurring on the white noise statistics of the image (refer to Section 2.2.2 for details).

2.3.3 Gradient calculation

Like LSD, EDLines is based on image gradients. EDLines uses the same kernels as LSD (Equations 2.4a and 2.4b) to compute the horizontal and vertical derivatives. The gradient magnitude is calculated using the Pythagorean sum (Equation 2.5), and the angle is stored as a binary value to indicate whether the gradient is oriented stronger towards the horizontal or vertical direction (Equation 2.17).

$$Dir(x,y) = \begin{cases} horizontal & |g_x(x,y)| < |g_y(x,y)| \\ vertical & |g_x(x,y)| \geq |g_y(x,y)| \end{cases} \tag{2.17}$$

Additionally, the magnitude of each pixel is checked against a threshold value. A default value of 36 is used for the threshold, assuming the input image uses intensity values in the range $[0, 255]$ [TAG10] [AT18]. Pixels with magnitudes below this threshold are marked as *non-edge*

area pixels and not considered for edge drawing. This is similar to the thresholding done in LSD (Section 2.2.5).

2.3.4 Anchors

The anchors are peak pixels in the gradient magnitude map. There are two internal parameters in the algorithm related to anchors:

The *scan interval* (also called *detail ratio*) controls how many anchors are found - a *scan interval* of k means that only every k-th row and column is scanned for anchors. For EDLines, Akinlar and Topal recommend to set the scan interval to 1, so that every pixel is checked [AT11].

The *anchor threshold* is the minimum difference of gradient magnitude between an anchor and its horizontal/vertical neighbors (assuming the gradient direction is horizontal/vertical). In the default implementation, this parameter is set to 3.

To find the anchors, pixels in rows and columns as defined by the scan interval are checked. If the gradient direction of a pixel is horizontal, the pixel's gradient magnitude is compared to the gradient magnitudes of its horizontal neighbors. The same is done for vertical pixels and their vertical neighbors. In case the difference to both neighbors is at least the *anchor threshold*, the pixel is marked as an anchor.

2.3.5 Linking

The linking step is repeated for all anchors. The algorithm performs a walk on the image, following pixels with high gradient magnitudes and marking them as edge pixels.

Starting from an anchor, the routing is done in the direction of the anchor's gradient angle. Every step, three neighbors are considered and the one with the highest gradient magnitude is picked (for example, in case the routing is done towards the top, the top-left, top and top-right neighbors are considered). The direction changes with the gradient direction of the selected pixel. The routing ends once a pixel that is either marked as an edge pixel or non-edge pixel, with the gradient magnitude below the threshold, is selected.

The result from the linking step is a list of pixel chains/edge segments that are used as the basis for line segment fitting.

2.3.6 Line segment fitting

For each edge segment, the algorithm first tries to fit a line segment with a certain minimum length, and then extends the line segment until the error exceeds a certain threshold. An overview on the line fitting procedure is given in Figure 2.8.

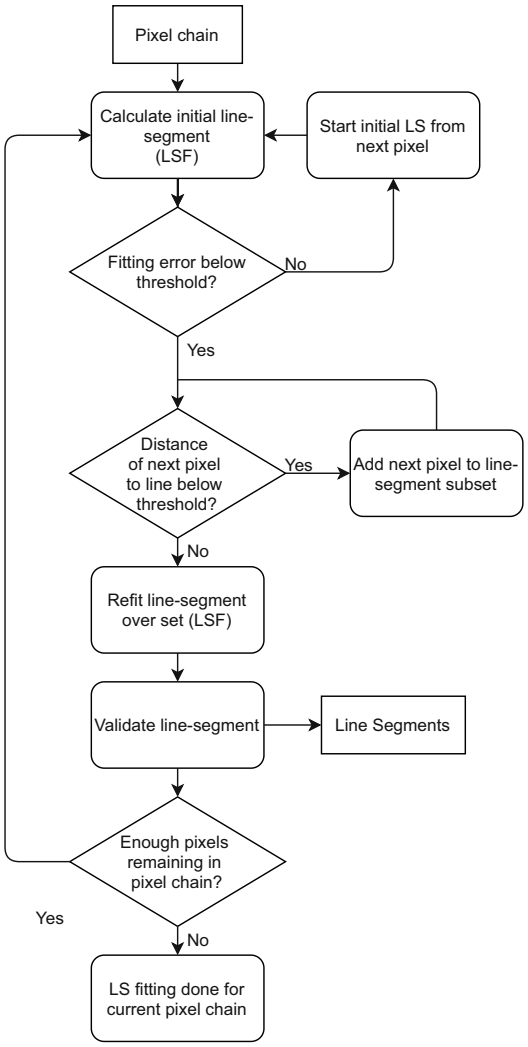


Figure 2.8: Overview of the EDLines line fitting algorithm

2.3.6.1 Least Squares Fitting algorithm

The Least Squares Fitting algorithm (LSF) algorithm tries to find a line $y = a_1 \cdot x + a_0$ (for *horizontal lines* with $45° < |\theta| < 135°$) or $x = a_1 \cdot y + a_0$ (for *vertical lines* with angle $|\theta| \leq 45°$ or $|\theta| \geq 135°$) that covers a consecutive subset of an edge segment so that the total fitting error (Equation 2.18 for horizontal lines and Equation 2.19 for vertical lines) is minimized. Two different forms of line equations are used since a line at 90° angle would result in $a_1 = \infty$ using the first form (the same is true for 0° angle lines using the second form).

$$err := |y - (a_1 x + a_0)| \tag{2.18}$$

$$err := |x - (a_1 y + a_0)| \tag{2.19}$$

Whether a line is *horizontal* or *vertical* is decided by comparing the sum of deviations from the mean (Equations 2.20 and 2.21), where P is the set of pixels to be covered by the line and $\bar{x}$ and $\bar{y}$ are the mean values of x and y in P. If $dev_y \leq dev_x$, equation $y = a_1 \cdot x + a_0$ is used to describe the line, and $x = a_1 \cdot y + a_0$ otherwise.

$$dev_x := \sum_{(x_i,y_i)\in P} |x_i - \bar{x}| \tag{2.20}$$

$$dev_y := \sum_{(x_i,y_i)\in P} |y_i - \bar{y}| \tag{2.21}$$

Equations 2.22 and 2.23 are used to calculate a_1 and a_0. The equations are valid for horizontal lines in the $y = a_1 \cdot x + a_0$ form. To calculate the parameters for the other form, the occurrences of x and y have to be swapped.

$$a_1 = \frac{\sum_{(x_i,y_i)\in P}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{(x_i,y_i)\in P}(x_i - \bar{x})^2} \tag{2.22}$$

$$a_0 = \bar{y} - a_1\bar{x} \tag{2.23}$$

2.3.6.2 Initial line segment

Beginning from the starting point of the edge segment, the algorithm tries to fit a line over the first n_{min} pixels (with n_{min} being the smallest number of pixels required for a line segment to be validatable), using LSF algorithm. If the fitting error is below a certain threshold (default value: 1.0), the algorithm proceeds to the next step, extending the line segment. Otherwise, the starting point is set to the next pixel in the chain. If the algorithm is unable to fit any minimum-length line segment into the current edge segment, the process is repeated for the next edge segment.

2.3.6.3 Extend line segment

In this step, the algorithm tries to extend the set of pixels to be covered by the line segment. To do so, the distance from the next pixel in the edge segment to the line is calculated. If the distance is below a certain threshold (default value: 1.0), the pixel is added to the set. Once a pixel with distance above the threshold is encountered, least squares fitting is used again to map a line on the pixels in the set as good as possible, and then the line segment will be checked in line segment validation. The line segment fitting algorithm is repeated for the remaining pixels in the edge segment.

2.3.7 Line segment validation

In this step, the line segments are validated according to the *a contrario* method. The validation uses a similar notion of p-aligned pixels as LSD (see Section 2.2.9), that is, pixels with gradient angle equal to the line segment angle, within a tolerance of πp. Therefore, for a set of pixels P covered by a line segment with angle θ_l in an $N \times M$ image, NFA is calculated as follows:

$$k := |\{(x,y) \in P \mid |arctan(\tfrac{g_x(x,y)}{-g_y(x,y)}) - \theta_l| \leq \pi p\}|$$

$$n := |P|$$

$$\mathrm{NFA}(n,k) = (NM)^2 \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} \tag{2.24}$$

Like in LSD, the NFA is checked against a threshold ε (default value: 1.0). In case the NFA is smaller, the line segment is considered meaningful and reported. The default value for p is 0.125 - this results in an angle tolerance of 22.5°, the same value LSD uses.

The aforementioned n_{min} variable is smallest n for which a k exists so that $\mathrm{NFA}(n_{min}, k) < \varepsilon$. Values of n_{min} for common resolutions can be taken from Table 2.1.

Table 2.1: n_{min} values for various resolutions, assuming $p = 0.125$ and $\varepsilon = 1.0$.

Resolution	n_{min}
512×512	12
640×480	13
800×600	13
1024×768	14
1280×720	14
1920×1080	14

2.4 Hough transformation

The concept of the Hough Transformation was first described in a 1962 patent by Paul Hough [Hou62]. The use of Hough Transformation to detect lines and other geometric shapes was proposed by Richard Duda and Peter Hart in 1972 [DH72].

The Hough Transformation operates on an image's edge map, generated by an edge detection algorithm like the Canny algorithm (Section 2.1. Variants of Hough Transformation are able to detect various shapes, including circles, although in this Section the focus lies on line detection.

While implementations of the Hough Transformation exist for CPU-, graphics processing unit (GPU)- and FPGA-based architectures, this Section will, following a brief introduction and overview on the Hough Transformation technique, focus on major innovations of hardware implementations.

2.4.1 Introduction

In the Hesse normal form, each line in the (x, y) Cartesian space can be described by two parameters (ρ, θ) with line equation:

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{2.25}$$

ρ is the length of the normal between the line and the origin and θ describes the angle between the x-axis and the normal (see Figure 2.9). For image processing, the origin is either defined to be the center of the image or one of the corners. The pair (ρ, θ) is used to describe a point in the *Hough space* (or *parameter space*), where each point (ρ, θ) in Hough space corresponds to a line in the 2-dimensional Cartesian space with Equation 2.25.

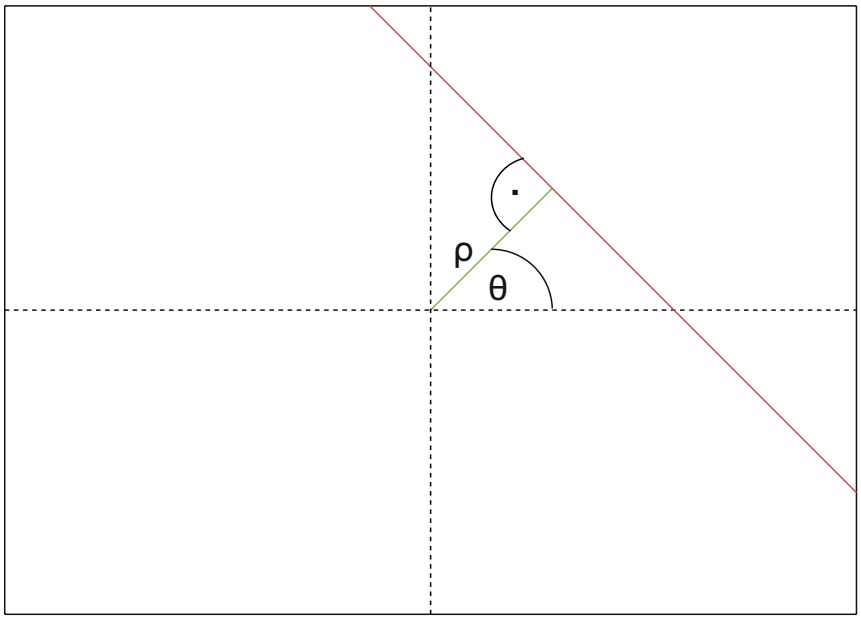


Figure 2.9: Parametrization of a line (red). The parameter ρ is the length of the normal (green) between line and the origin (in this case the center of the image), while θ is its angle to the x-axis.

A certain range of values for the parameters is necessary to express all possible lines in an $X \times Y$ image. For now, we assume the origin point to be set in the image center. One possibility is to limit angles to the range $0° \leq \theta \leq 180°$, which results in ρ taking a negative value in case the perpendicular point of the normal is in the 3rd or 4th quadrant of the image. The required range for ρ is the length of the image's diagonal.

$$\rho \in [-\frac{\sqrt{X^2+Y^2}}{2}, \frac{\sqrt{X^2+Y^2}}{2}] \tag{2.26a}$$

$$\theta \in [0°, 180°] \tag{2.26b}$$

Alternatively, the need for negative ρ values can be eliminated if we extend the range of θ to the whole 360°, since the lines $(-\rho, \theta)$ and $(\rho, \theta + 180°)$ are identical.

$$\rho \in [0, \frac{\sqrt{X^2+Y^2}}{2}] \tag{2.27a}$$

$$\theta \in [0°, 360°] \tag{2.27b}$$

The Hough Transformation uses a two-dimensional memory array in the parameter space, so the array can store one memory value for every possible line in the image. The size of this array is defined by the ranges of the parameters (equations 2.26 or 2.27) as well as the desired resolutions $\Delta\rho$ and $\Delta\theta$ of the parameters. A smaller resolution value results in a better detection accuracy, but at a higher computation cost and with higher memory requirements. For a 640×480 image with $\Delta\rho = 1$ pixel and $\Delta\theta = 2°$, this results in an array with $\frac{\sqrt{640^2+480^2}}{2\times 1} \times \frac{360}{2} = 72000$ entries.

The whole memory array, which is also called *voting memory* or *accumulator memory*, is initialized with zeros. Then, for every edge pixel P[x,y], the counter value of all possible lines going through that point is incremented - this process is called *voting*. In case a line is present on the image, all edge pixels lying on this line will vote for it, thus resulting in a peak in the Hough space for that line's parameters. A line is detected if the counter value reaches a chosen threshold, or in other words, if a certain minimum number of edge pixels is present on the space covered by that line. However, since no informations regarding the positions of the edge pixels voting for a line were stored, the endpoints of the lines cannot be found. As a consequence, Hough Transformation in its basic implementation cannot be used for line *segment* detection.

The main design challenges of Hough Transformation hardware acceleration lie in the architecture of the voting memory as well as the computational method to find the (ρ, θ) pairs of all lines going through an edge pixel.

2.4.2 Classified Hough Transform

A different set of parameters for the Hough space was proposed by Wang et al. [WTZZ02]. Instead of using the Hesse normal form (Equation 2.25), a line can also be represented by Equation 2.28.

$$c = y - mx \tag{2.28}$$

This form uses the slope m and the y-intercept c as the parameters of the line. Using this representation has the advantage of not requiring sine and cosine functions. However, the problem with this approach is its numerical instability for very steep slopes, as well as the required infinite range of the parameters. Since the numerical mostly occur on slopes above 45° angle, one solution is to use a different parametrization for these lines, based on the inverse slope $k = \frac{1}{m}$ and the x-intercept l (Equation 2.29).

$$l = x - ky \tag{2.29}$$

Therefore, Wang et al. [WTZZ02] proposed to separate the Hough space into a (c, m) and a (l, k) space.

Since there is a correlation between the angle of the line and the gradient angle, the number of votes required can be reduced by comparing the directional gradients G_x and G_y. If $|G_x| \leq |G_y|$ holds, the edge pixel is part of a flat slope line and therefore only votes in (c, m) space are made. If $|G_x| > |G_y|$ holds, the edge pixel is part of a steep slope line and only votes in (l, k) space are made.

To find the voting parameters of the lines, the slope parameter m (resp. k) is initialized to -1 and the intercept parameter is set to $x + y$ (where x and y are the coordinates of the pixel). Next, the slope is iteratively incremented by a resolution value ε, and the intercept parameter c (resp. l) is

updated accordingly (Equation 2.30 and Equation 2.31). By choosing $\varepsilon = 0.5^N$ (with N being a natural number), the multiplication can be replaced by a shift operator.

$$m_0 = -1 \tag{2.30a}$$
$$c_0 = x + y \tag{2.30b}$$
$$m_n = m_{n-1} + \varepsilon \tag{2.30c}$$
$$c_n = c_{n-1} - \varepsilon x \tag{2.30d}$$

$$k_0 = -1 \tag{2.31a}$$
$$l_0 = x + y \tag{2.31b}$$
$$k_n = k_{n-1} + \varepsilon \tag{2.31c}$$
$$l_n = l_{n-1} - \varepsilon y \tag{2.31d}$$

2.4.3 Parallel voting

For every edge pixel, a total number of $\frac{180°}{\Delta\theta}$ votes are required. This results in a high number of read and write accesses to the voting memory, which limits the performance of the design. To increase the throughput, one solution is to split the voting memory, so multiple votes are possible simultaneously. Research by Guan et al. [GAZ+17] describes an architecture to parallelize the voting procedure. The Hough space uses the (ρ, θ) parametrization, using the ranges defined by Equation 2.27, and is divided into n regions: The i-th region contains lines with θ values in the range $\theta \in [i\frac{360°}{n}, (i+1)\frac{360°}{n})$.

In hardware, this separation is mirrored by using n computation modules, where each module contains two LUTs for efficiently solving $sin(\theta)$ and $cos(\theta)$ in Equation 2.25, voting memory for the region of the Hough space, as well as the required logic for arithmetic operations and initialization.

- The LUTs contains sine and cosine values for all relevant θ values of that Hough space region, which results in $\frac{360}{n\Delta\theta}$ entries per LUT. The advantage of this approach is the high accuracy, without the need of slow solutions like the Coordinate Rotation Digital Computer (CORDIC) [Vol59] algorithm to solve the trigonometric functions.
- The memory layout of a module's voting memory is shown in Table 2.2. $\theta_{i,min}$ and $\theta_{i,max}$ denote the maximum angles θ in the Hough space of module i, and ρ_{max} is the maximum value of the ρ parameter (half the diagonal length of the image, rounded up).
- When an edge pixel is reported to the module, θ is initialized to θ_{min}. The ρ value is calculated by solving Equation 2.25 - the values $sin(\theta)$ and $cos(\theta)$ are read from the LUTs and multiplied with the x- and y-coordinates of the pixel. In case the calculated ρ value is non-negative, the content on voting memory indexed by ρ is incremented by 1. In the next

cycle, θ is increased by $\Delta\theta$, and ρ is updated accordingly. Since the θ value changed, an offset of $\frac{\rho_{max}}{\Delta\rho}$ has to be added to the address to account for the change of the θ parameter. This process is repeated until the condition $\theta = \theta_{i,max}$ is met. At this point, the module is ready to process the next edge pixel.

The delay $p\frac{360}{n\Delta\theta} + \alpha$ is dependent on the number of edge pixels p, the parallelization degree n, the parameter resolution $\Delta\theta$ and a pipeline delay α.

Table 2.2: Memory layout of the voting memory in module i.

Address	θ	ρ
0	$\theta_{i,min}$	0
1	$\theta_{i,min}$	$\Delta\rho$
2	$\theta_{i,min}$	$2\Delta\rho$
⋮	⋮	⋮
$\frac{\rho_{max}}{\Delta\rho} - 2$	$\theta_{i,min}$	$\rho_{max} - \Delta\rho$
$\frac{\rho_{max}}{\Delta\rho} - 1$	$\theta_{i,min}$	ρ_{max}
$\frac{\rho_{max}}{\Delta\rho}$	$\theta_{i,min} + \Delta\theta$	0
$\frac{\rho_{max}}{\Delta\rho} + 1$	$\theta_{i,min} + \Delta\theta$	$\Delta\rho$
⋮	⋮	⋮
$(\frac{360}{n\Delta\theta} - 1)\frac{\rho_{max}}{\Delta\rho} - 2$	$\theta_{i,max}$	$\rho_{max} - \Delta\rho$
$(\frac{360}{n\Delta\theta} - 1)\frac{\rho_{max}}{\Delta\rho} - 1$	$\theta_{i,max}$	ρ_{max}

2.4.4 Parametrization for lane detection

El Hajjouji et al. [EMAE20] proposed a parametrization for Hough Transformation that takes advantage of the intended application of road lane detection. When the camera positioned at the center of the car's windshield, road lanes have a number of special properties: The first is that road lanes are never detected as a horizontal line, therefore $\theta \neq 0°$ and $\theta \neq 180°$. From this property, a new parametrization (b, θ) can be derived, using line equation:

$$b = x\cot(\theta) + y \tag{2.32}$$

In addition, the range of θ is limited to two regions of interest - a range $[\theta_{Lmin}, \theta_{Lmax}]$ for the left lane, and a range $[\theta_{Rmin}, \theta_{Rmax}]$ for the right lane. Pixels with a gradient angle outside these ranges are discarded, as are pixels with gradient magnitude below the threshold. If this is not the case, pairs of (b, θ) will be calculated for voting using Equation 2.32 in one of the two ranges, at a precision of $\Delta\theta = 0.1$ rad. An implementation of this algorithm on an Xilinx XC5VLX50T FPGA achieves clock frequencies of up to 200 MHz and is able to process 640×480 images within 1.47 ms.

2.4.5 Detecting line segments

As previously mentioned, using Hough Transform, only the number of edge pixels on a line is counted. The locations, lengths, and count of line segments on this line can not be extracted

easily. Kim et al. [KJT+08] proposed a method to extract line segments from the line detection result of Hough Transformation.

In the implementation, a (ρ, θ) parametrization was used for the Hough Transformation. Using the parameters of the detected lines, the coordinates of all pixels on this line are calculated in the *Inverse Line Equation Calculator*. This works by solving Equation 2.33 for lines with $45° \leq \theta \leq 135°$ by iteratively inserting values for $x = 1...X$ (with X being the image width). The same is done analogously for lines with $\theta < 45°$ or $\theta > 135°$, by inserting values for $y = 1...Y$ (with Y being the image height) in Equation 2.34.

$$y = \frac{\rho - x\cos(\theta)}{\sin(\theta)} \tag{2.33}$$

$$x = \frac{\rho - y\sin(\theta)}{\cos(\theta)} \tag{2.34}$$

The edge map is read out at these coordinates and stored in the connection list of the *Line Identifier unit*. The connection list is an array of 15 memory cells. Each memory cell is one bit wide and determines whether a pixel of the line is an edge or not. By searching for patterns of the form "0000000X1111111", the start points of line segments can be found. A pattern "1111111X0000000" indicates the end of the line segment. The endpoints of consecutive line segments are stored, together with the length of the segment.

2.5 Modified LSD

A hardware-based line segment detection algorithm was proposed by Zhou et al. in 2017 [ZCW18]. The idea of the algorithm is based on the LSD algorithm by von Gioi et al. [GJMR12].

The algorithm contains a fully pipelined implementation of the Canny edge detector (Section 2.1). With every clock cycle, a pixel value is input (horizontal-scan, left-to-right, top-to-bottom). After a delay, a 1-bit edge map value as well as the gradient angle of that pixel is output (in case the pixel is not an edge, the gradient angle is not meaningful and set to a default value). The output of the Canny edge detector is used by two functionally identical *region growing modules*. One of the instances uses a horizontally mirrored copy of the image as the input. The mirroring is achieved by row buffers operated in first in, last out mode.

A *region* is a connected set of edge pixels with a similar gradient angle (within a tolerance). The regions are tracked by a data structure and can grow from top left to bottom right - therefore, a second instance of the module operating on the mirrored image is required to detect diagonal line segments oriented from bottom left to top right.

Each region growth module uses a 3×3 heterogeneous sliding window, shown in Figure 2.10. The *precedent regions* (PRs) are records to track the growth of regions - the record R[i] stores the following information about the region currently ending in column i:

- Gradient angle of the region
- Start point

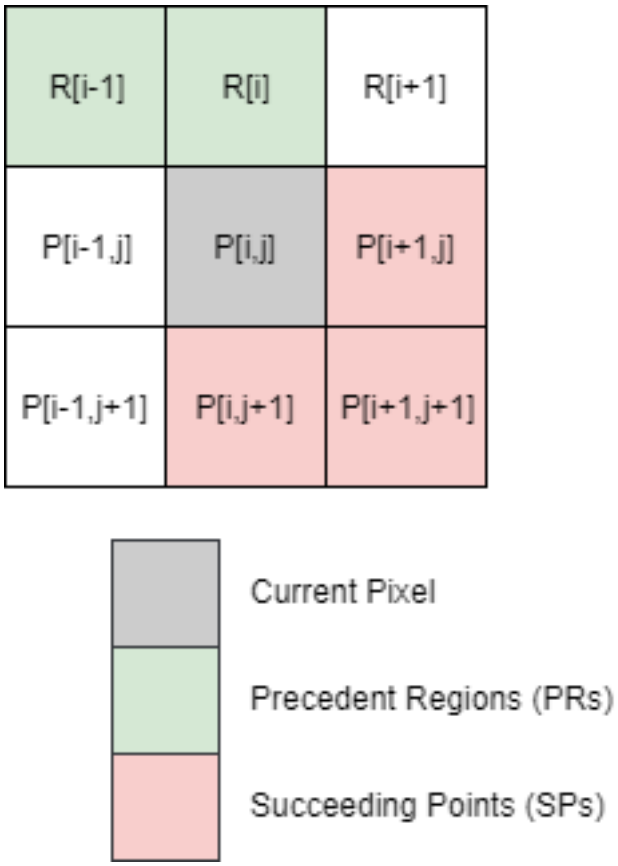


Figure 2.10: Sliding window of the region growth module [ZCW18].

- Current endpoint
- Length (number of pixels in region)

Pixel P[j,i] is a candidate to extend the precedent regions R[i-1] or R[i]. A pixel is considered *aligned* with a region if they share a similar gradient angle. We can distinguish four different cases:

1. P[j,i] is not an edge pixel. In this case, no changes to data structures are made.
2. P[j,i] is an edge and aligned with R[i-1]. In this case, R[i] will be merged with R[i-1] and inherits all information. Then R[i] is updated with regards to the addition of P[j,i] to the region.
3. P[j,i] is an edge and aligned with R[i]. R[i] is updated with regards to the addition of P[j,i] to the region.
4. P[j,i] is an edge but aligned with neither of the two PRs, or the PRs are empty/uninitialized. In this case, a new region will be initialized in R[i].

In case P[j,i] was added to or initialized a PR, the *succeeding points* (SPs) are checked. If none of them is an edge and aligned with R[i], then R[i] has no more candidate pixels for growing

and can be sent to the output unit. In the case of the region growth module operating on the mirrored image, the start- and endpoint coordinates of the reported line segments are flipped back to match the coordinates in the original, unmirrored image.

2.6 Other techniques

More algorithms using the *a contrario* validation as used in LSD and EDLines have been proposed. Liu et al. [LAGT20] proposed a new *a contrario*-based algorithm optimized for Synthetic Aperture Radar (SAR) applications. Zhang et al. [ZWL21] proposed a new group and validation strategy, further reducing the processing time while increasing the detection quality. Using an Intel Core i7–7700HQ CPU, the algorithm requires an average of 21.12 ms to calculate a 640×480 image.

Abdallah et al. [ACA14] proposed a line segment detector based on the weighting resistor matrix (WRM). This technique was originally development for high energy physics research and employs a resistor network to detect an 8-pixel long line segment on an 8×8 section of an edge map. The same authors also proposed an FPGA implementation for this technique [AFAC17]: Edge detection and the WRM algorithm are performed on the FPGA, and a software post-processing software algorithm reconstructs line segments from set of shorter 8-pixel long segments. In total, the algorithm requires about 20 ms to process an 640×480 image.

Almazàn et al. [ATQE17] proposed an algorithm that combines Hough Transformation with dynamic programming to detect line segments. The problem of finding a mapping between lines detected by Hough Transformation and line segments is modeled by a Markov Chain, and dynamic programming is used to find the maximum probable solution. The algorithm is implemented in MATLAB and requires about 5.2 seconds to process a 640×480 image on an Intel Core i7 CPU with 3.4 GHz and 8GB RAM.

In recent years, deep learning techniques became more and more popular in various domains, including computer vision. Huang et al. [HWZ+18] proposed two convolutional neural networks that are able to extract lines and junctions from images. While the method showed promising detection quality on their test image set, the throughput of their method was limited to 2 frames per second on their NVIDIA Titan X GPU.
Xu et al. [XXCT21] proposed a method for line segment detection based on transformers. A convolutional neural network extracts coarse features from the input image, which is then processed by a series of encoders and decoders. Notably, this method is not based on any edge detection method as an intermediate step.
Nguyen et al. [NJR21] proposed a line segment detector for power line detection, using a feature extractor, a classifier and a regressor. Using a GeForce GTX 1080 Ti GPU, 20.4 frames per second could be achieved on 512×512 images. When processing images taken by a fisheye or spherical camera, the distortion causes straight features in the image can appear curved, which causes conventional line segment detection algorithms to perform poorly in such cases. Li et al. [LYY+20] proposed a deep-learning based algorithm that is able to perform line segment detection on undistorted images taken by pinhole cameras as well as distorted images. Using an NVIDIA GTX 2080 Ti GPU, 40.6 frames per seconds could be achieved on 512×512 images.

3 Improved Line Segment Detector

In this chapter, a method to improve the performance of the the LSD algorithm by Gioi et al. is proposed. With the use of a LUT, we were able to optimize the a contrario validation (Section 2.2.9). On our test image set, this speeds the algorithm up by 13.08% on average and by 28.8% on one instance.

3.1 Background

In the LSD algorithm, every line segment candidate is validated using the *a contrario* approach. This is done by calculating the NFA and checking it against a certain threshold ε. If the NFA value is smaller than ε, the line segment is verified. The NFA is calculated according to the equation

$$\mathrm{NFA}(n,k,p) = (N \cdot M)^{5/2} \cdot \gamma \cdot \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} \tag{3.1}$$

where n denotes the number of pixels in the rectangle covered by the line segment and k is the number of pixels with a gradient angle aligned to the angle of the line segment, within a precision value p. $(N \cdot M)$ denotes the (scaled) image resolution and γ denotes the number of possible values for p (in the default configuration: $\gamma = 11$).

The proposed optimization replaces the need to calculate the NFA value using the formula above. Instead, a LUT based approach is used to approximate the equation. In addition, instead of using the NFA value as is, its decadic logarithm is used instead, since it allows for easier approximation as well as better numeric stability. Do note that the original implementation of LSD uses the decadic logarithm as well in its calculation of the NFA value [GJMR12].

This approach needs to limit the amount of parameters in the equation, to keep the size of the LUT reasonable. In particular, the resolution $(N \cdot M)$ is assumed to be defined by the application while $\gamma = 11$ and $\varepsilon = 1.0$ are hereby assumed to be fixed values that are defined by the implementation of the LSD algorithm. p is limited to γ different values by design, and a reasonable limit for n can be selected, denoted n_{max} (in case $n > n_{max}$, the NFA will be calculated in the regular fashion, thus allowing the algorithm to still operate in such situations). Therefore, the LUT has $\gamma \cdot n_{max}$ entries.

For every pair of (n, p), the equation can be interpreted as a function in k:

$$\mathrm{NFA}_{n,p}(k) = t \cdot \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j} \tag{3.2}$$

where $t := (N \cdot M)^{5/2} \cdot \gamma$ is a constant. Analyzing this equation, the following observations can be made:

- $\log_{10}(\mathrm{NFA}_{n,p}(k))$ is monotonically falling with rising k
- $\log_{10}(\mathrm{NFA}_{n,p}(k))$ converges towards $\log_{10}(t)$ for $k \to 0$

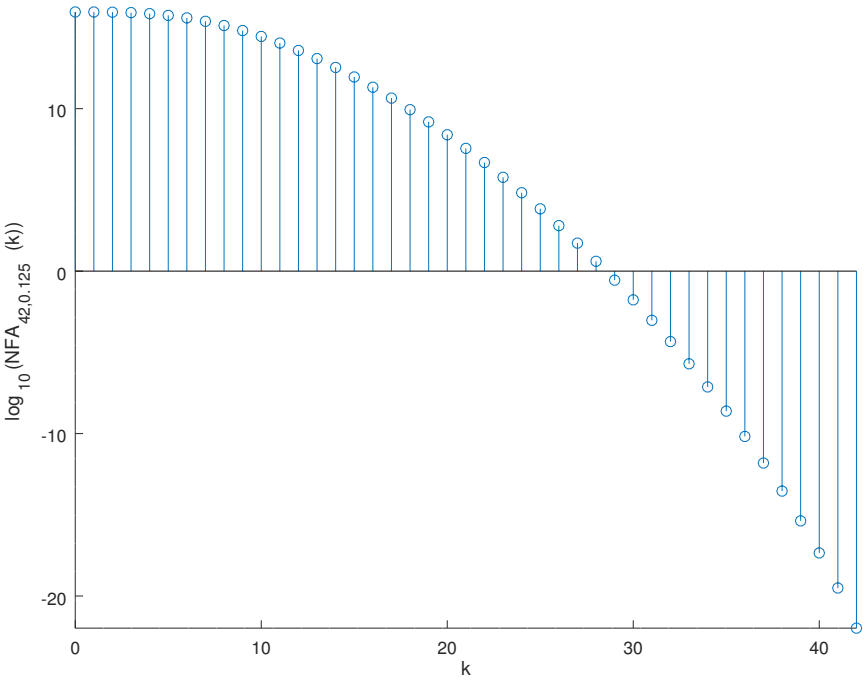


Figure 3.1: Plot of $log_{10}(\mathrm{NFA}_{n,p}(k))$, for $n = 42$, $p = 0.125$ and $N = M = 410$

3.2 Approximation approach

Our approximation approach is to split the function $log_{10}(\mathrm{NFA}_{n,p}(k))$ into three sections, with each section being approximated by a different approximation function: The first and third sections are approximated by constants, while the second section is approximated by a quadratic polynomial function (Equation 3.3).

$$\log_{10}(\mathrm{NFA}_{n,p}(k)) \approx \begin{cases} \log_{10}(t) & \text{if } k < k_{low}(n,p) \\ c_1(n,p)k^2 + c_2(n,p)k + c_3(n,p) & \text{if } k_{low}(n,p) \le k < k_{min}(n,p) \\ \log_{10}(\varepsilon) & \text{if } k \ge k_{min}(n,p) \end{cases} \tag{3.3}$$

$k_{min}(n,p)$ is the breakpoint between the second and third section, and is defined to be the lowest k so that $log_{10}(\text{NFA}_{n,p}(k)) \leq log_{10}(\varepsilon)$. For any k greater than or equal to $k_{min}(n,p)$, $log_{10}(\text{NFA}_{n,p}(k))$ is approximated to an arbitrary value smaller than or equal to $log_{10}(\varepsilon)$. This is justified by the fact that the LSD algorithm does not require the exact NFA value in case the verification of the line segment is successful.

$k_{low}(n,p)$ is the breakpoint between the first and second section. For any k below $k_{low}(n,p)$, $log_{10}(\text{NFA}_{n,p}(k))$ is approximated by $log_{10}(t)$, which is the value $log_{10}(\text{NFA}_{n,p}(k))$ converges to for $k \to 0$. For any k greater than or equal to $k_{low}(n,p)$, given it is smaller than $k_{min}(n,p)$, $log_{10}(\text{NFA}_{n,p}(k))$ is approximated by a quadratic polynomial function with coefficients $c_1(n,p)$, $c_2(n,p)$ and $c_3(n,p)$. $k_{low}(n,p)$ is picked so that the total squared approximation error in the first and second section is minimal, assuming the best polynomial is used for approximation in each case.

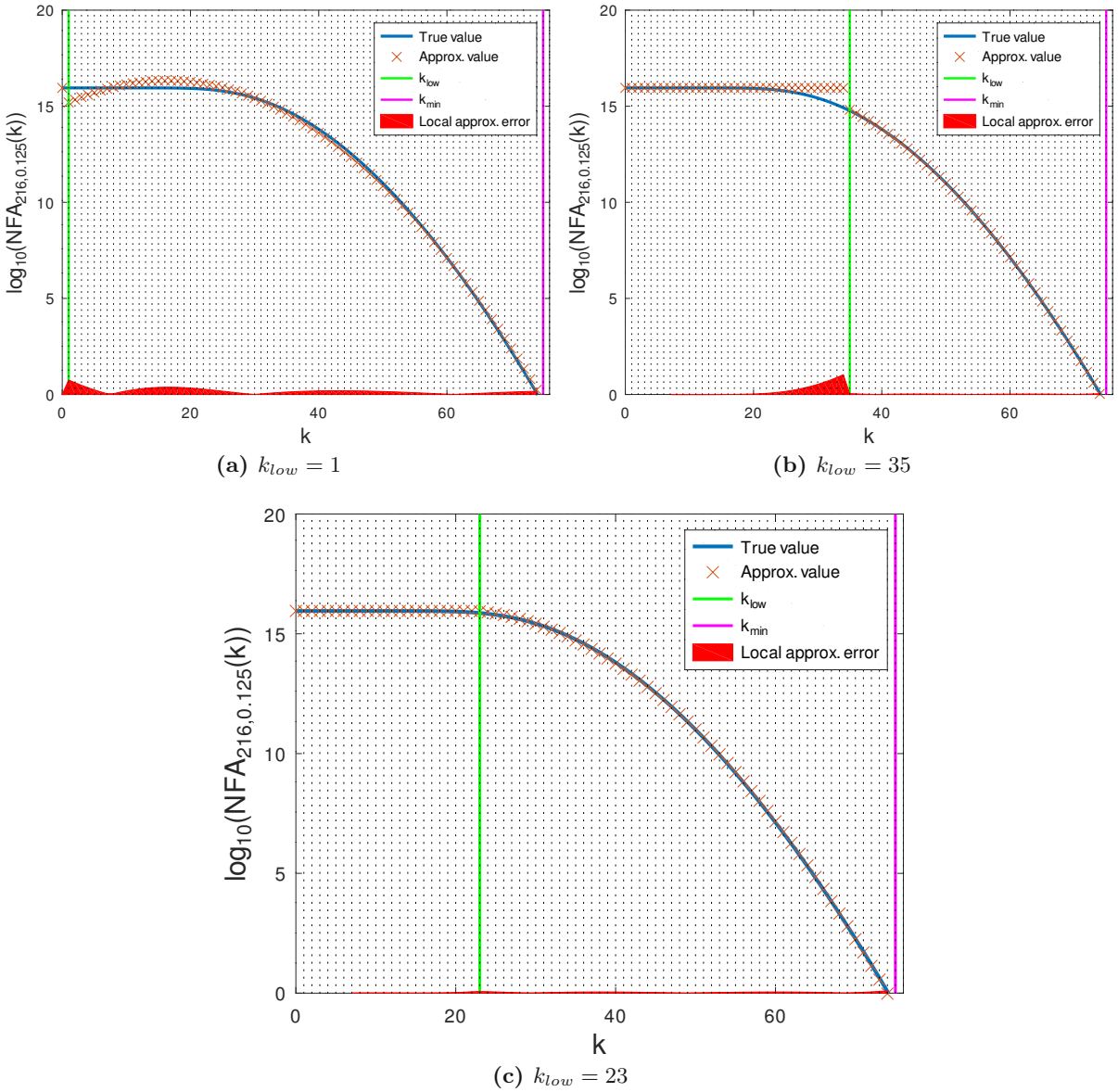
Fig. 3.2 demonstrates the impact of the selection of k_{low} on the quality of the approximation. If k_{low} is selected too small, the selected polynomial will have to account for the slowly converging area for small k-values, leading to a poor approximation especially in this area (Fig. 3.2a). A too high k_{low} will lead to a good polynomial in the range $k \geq k_{low}$, but high error in the range of k-values slightly below k_{low} (Fig. 3.2b). The total squared approximation error in dependence of k_{low}, as well as the maximum local error for the same example are shown in Fig. 3.3.

3.3 Lookup table generation

To generate the LUT, the following steps are performed in this order for every pair of (n,p):

1. Calculate $log_{10}(\text{NFA}_{n,p}(k))$ starting from $k = 1$ until the result is smaller than or equal to $\log_{10}(\varepsilon)$. The index of the last result is $k_{min}(n,p)$, which is to be stored in the LUT. All results except for the last one should be stored in an array. If no $k_{min}(n,p) \leq n$ exists, assume $k_{min}(n,p) := n + 1$.
2. For all $k_{low}(n,p) \in [1, k_{min}(n,p)]$, perform steps 3 and 4.
3. Find a quadratic polynomial to approximate $log_{10}(\text{NFA}_{n,p}(k))$ in the range $k \in [k_{low}(n,p), k_{min}(n,p)-1]$, using polynomial regression on the values in the array. Skip this step if $k_{low}(n,p) = k_{min}(n,p)$.
4. Calculate the sum of the squared error values of the approximation for all $k \in [1, k_{min}(n,p)]$. If the approximation quality improved (the total squared error decreased), update $c_1(n,p)$, $c_2(n,p)$, $c_3(n,p)$ and $k_{low}(n,p)$ in the LUT.

For each pair of (n,p), the resulting values of k_{min}, k_{low}, c_1, c_2 and c_3 are stored in a data structure (Table 3.1). Each of these structs is stored in a two-dimensional array `LUT[p][n]`, with `n` constituting the column index and an index assigned to each value of `p` constituting the row index. Our experiments have shown this way of indexing to be faster than the alternative of using `n` as the row index and `p` for the column index. These experiments also revealed that having one array with a 5-tuple of values is faster than having 5 separate arrays, one for each value, likely caused by a smaller number of cache misses in the single-array implementation.



(a) $k_{low} = 1$ (b) $k_{low} = 35$

(c) $k_{low} = 23$

Figure 3.2: Approximation using different values of k_{low}, assuming $n = 216$, $p = 0.125$, $N = M = 960$, $\gamma = 11$.

The size of the LUT depends on γ and n_{max}. The size in bytes can be calculated using the formula $\gamma \cdot n_{max} \cdot 16$. For example, for $n_{max} = 20000$ and the default $\gamma = 11$, we have a size of 3520 kB, which is a reasonable usage of memory considering that the LSD algorithm uses multiple frame buffers in the default implementation.

3.3.1 Run-time Complexity

The LUT contains $n_{max} \cdot \gamma$ entries. For each entry, the NFA value has to be calculated $k_{min}(n, p)$ times, which has approximately a linear relation to n as shown in Fig. 3.4. The time to evaluate the NFA scales approximately linearly with n. Therefore, the time to generate the NFA values

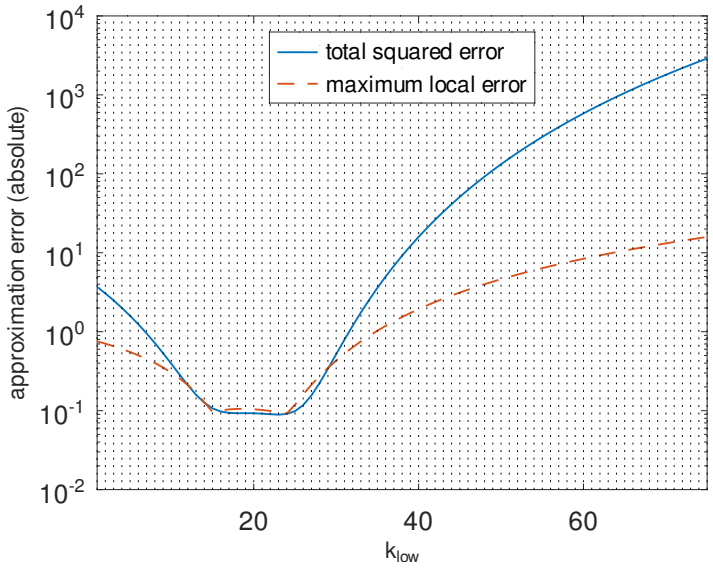


Figure 3.3: Total squared approximation error and maximum local error dependent on k_{low}, assuming $n = 216$, $p = 0.125$, $N = M = 960$, $\gamma = 11$. The plot shows that the approximation is optimal for $k_{low} = 23$.

Table 3.1: Lookup-table entry data structure

Field	Type	Size (Bytes)
`k_min`	`unsigned short`	2
`k_low`	`unsigned short`	2
`c_1`	`float`	4
`c_2`	`float`	4
`c_3`	`float`	4

for all entries is $\mathcal{O}(n_{max}^3 \gamma)$.

After the calculation of the NFA values, the polynomial regression has to be performed $k_{min}(n,p)$ times for each entry in the LUT. The run-time of the polynomial regression scales linearly with the number of data points used, which is $k_{min}(n,p) - k_{low}(n,p)$. As established, $k_{low}(n,p)$ ranges from 1 to $k_{min}(n,p)$, being $\frac{k_{min}(n,p)}{2}$ on average, which means the run-time scales linearly with $k_{min}(n,p)$ and, by extension, linearly with n. We can conclude that the time to determine the best $k_{low}(n,p)$ for each LUT entry, as well as the corresponding coefficients of the polynomial, is $\mathcal{O}(n^2)$, resulting in a complexity of $\mathcal{O}(n_{max}^3 \gamma)$ total for all entries. Therefore, the run-time complexity of the complete LUT generation is $\mathcal{O}(n_{max}^3 \gamma)$.

3.4 Lookup table usage

Algorithm 3.1 describes the optimized calculation of the NFA value. In this table;

- The function `old_NFA()` in line 4 denotes the original implementation of the NFA calculation. It is used whenever n is a value that is above n_{max}, meaning there is no entry in the LUT for it.

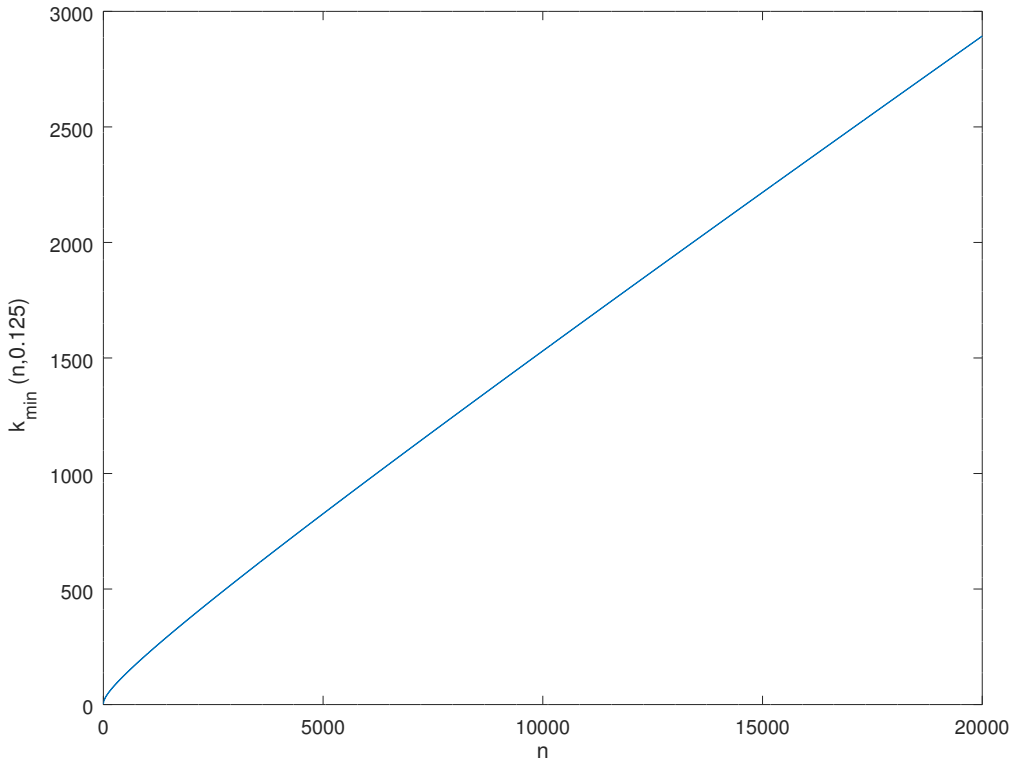


Figure 3.4: $k_{min}(n, p)$ in dependence of n, assuming $p = 0.125$, $N = M = 960$ and $\gamma = 11$

- The function `get_from_LUT()` in line 5 retrieves the required entry from the LUT. Its implementation depends on the layout of the LUT, as well as on how the index of the LUT entry is to be derived from p.

- The if clause in lines 11 and 12 prevents a false-positive verification, which might happen if the NFA value, usually with a value of k close to k_{min}, is approximated by the polynom to a value below $log_{10}(\varepsilon)$. Instead, a value of $log_{10}(\varepsilon) + \Delta$ will be returned, where Δ is the smallest positive value so that $log_{10}(\varepsilon) \neq log_{10}(\varepsilon) + \Delta$ using the given floating-point arithmetic. In our implementation, this is done using the `nextafter()` method, found in `math.h`.

3.5 Quantitative results

To measure the performance increase of the optimization over a wide array of images, the computation time of both the original and optimized LSD algorithm has been measured on the *TESTIMAGES* sampling set by Asuni and Giachetti [AG13] [Asu], which consists of 40 grayscale images, at a resolution of 1200×1200. The set of images was processed a total of 101 times, with the first processing run of each test image being omitted from the results to account for initial cache misses, branch mispredictions, and other transient effects. The mean of the processing time for every image was calculated. The measurement of the optimized algorithm was repeated for

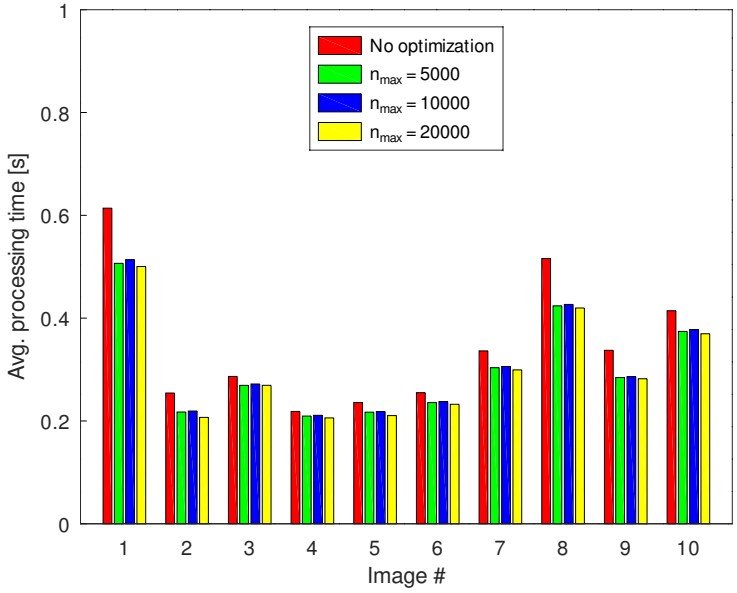
Algorithm 3.1: Optimized NFA calculation

input : n, p, n_{max}, ε, t
output: The NFA value

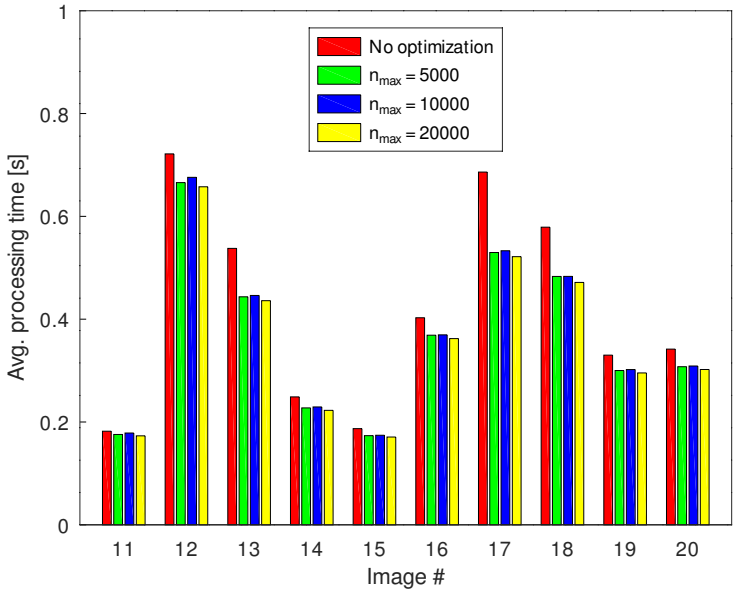
if $n = 0$ **or** $k = 0$ **then**
  return $log_{10}(t)$;
if $n > n_{max}$ **then**
  return `old_NFA`(n,p);
$(k_{min}, k_{low}, c_1, c_2, c_3) \leftarrow$ `get_from_LUT`(n, p, n_{max})
if $k \geq k_{min}$ **then**
  return $log_{10}(\varepsilon)$;
if $k < k_{low}$ **then**
  return $log_{10}(t)$;
$ret \leftarrow c_1 \cdot k^2 + c_2 \cdot k + c_3$;
if $ret \leq log_{10}(\varepsilon)$ **then**
  return $log_{10}(\varepsilon) + \Delta$;
return ret

three different values of n_{max} ($n_{max} \in \{5000, 10000, 20000\}$) to evaluate the effect the selection of this parameter has on the specific image set.

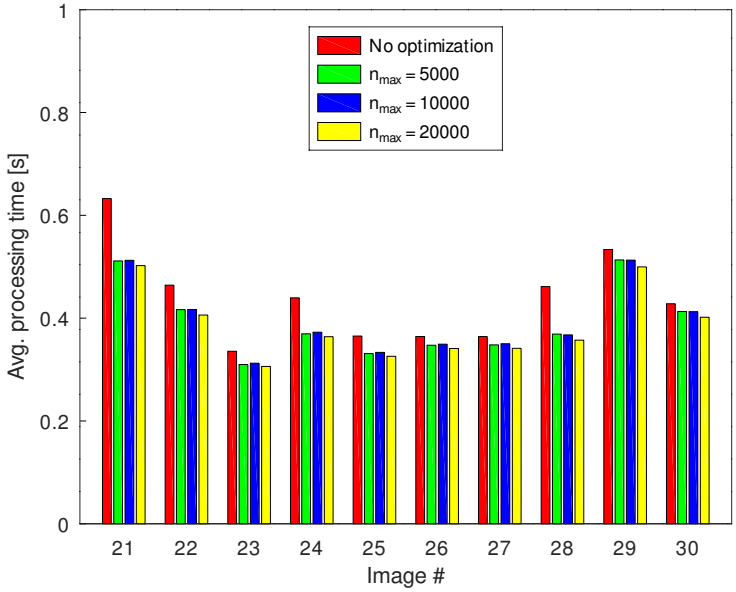
The quantitative results are shown in Fig. 3.5 and Table 3.2. For all test cases, the optimized algorithm performed better than the original one. The optimized algorithm performed best using $n_{max} = 20k$, with a mean reduction of processing time by 13.08% and offering the largest increase in 39 of the 40 test images (only Test Image #3 performed better using $n_{max} = 5000$). However, the performance difference for different values of n_{max} is relatively small, with the majority of test cases showing the performance being within ±3% of each other.



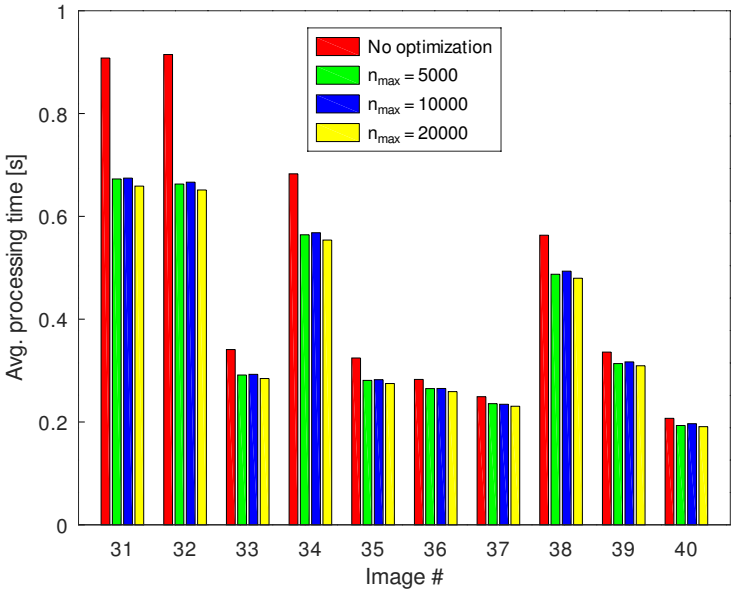
(a) Images #1 to #10



(b) Images #11 to #20



(c) Images #21 to #30



(d) Images #31 to #40

Figure 3.5: Comparison of average computation times for the original and optimized LSD algorithm on the test image set.

Table 3.2: Processing time reduction stats

	$n_{max} = 5k$	$n_{max} = 10k$	$n_{max} = 20k$
Biggest reduction	27.54% (# 32)	27.14% (# 32)	28.8% (# 32)
Smallest reduction	3.48% (# 11)	1.93% (# 11)	4.95% (# 11)
Mean reduction	11.46%	10.91%	13.08%
Median reduction	9.52%	8.76%	10.81%

3.6 Qualitative results

Due to the approximation error of the optimization, the Rectangle Refinement routine (Section 2.2.10) of the LSD algorithm potentially selects different precision or width values when advancing through the optimization steps, which can cause the detected line segments to be changed or omitted, as well as new line segments to be detected.

To measure this impact of the optimization on the detection results, our approach is to try to match each line segment detected by the optimized algorithm with an equivalent line segment detected by the original algorithm, and categorize them using the following metric:

- **Identical:** Line segments are categorized as *identical* if they appear in the results of both the original and optimized versions of LSD, with identical[1] endpoints.
- **Changed:** Line segments are categorized as *changed* if a line segment appears in the results of both the original and optimized versions of LSD, but the endpoints differ slightly in the

[1] The coordinates of the line segment endpoints were rounded to the nearest integer beforehand.

results. We qualify a pair of line segments as *changed* if the sum of distances between their endpoints[1] is smaller or equal to 4 pixels.

- **New:** All line segments in the results of the optimized algorithm that do not have an *identical* or *changed* match are categorized as *new*.
- **Lost:** All line segments in the results of the original algorithm that do not have an *identical* or *changed* match are categorized as *lost*.

This categorization was performed on the detection results of both the original and optimized algorithm, using our test image set as input. For the optimized algorithm, $n_{max} = 20000$ was used, since this version maximizes the usage of the approximated NFA value, and is therefore expected to deviate the most from the original algorithm regarding the detection results. Tables 3.3, 3.4, 3.5 and 3.6 list the results of the categorization, as well as the line segment totals detected by the original (denoted as **Total (orig.)**) and optimized (denoted as **Total (opt.)**) versions of the algorithm.

Table 3.3: Categorization of detected line segments, Test Images #1 to #10

Test Image #	1	2	3	4	5	6	7	8	9	10
Identical	3932	541	1391	179	695	912	1611	3604	1768	777
Changed	2	1	1	0	2	0	0	5	1	0
New	2	1	0	0	0	1	0	2	0	1
Lost	0	0	0	0	0	0	0	1	0	1
Total (orig.)	3934	542	1392	179	697	912	1611	3610	1769	778
Total (opt.)	3936	543	1392	179	697	913	1611	3611	1769	778

Table 3.4: Categorization of detected line segments, Test Images #11 to #20

Test Image #	11	12	13	14	15	16	17	18	19	20
Identical	210	4710	2414	619	687	1577	2722	1163	711	1073
Changed	0	10	3	0	0	4	11	0	0	0
New	0	2	0	0	0	0	2	5	1	0
Lost	0	1	1	0	0	0	0	0	0	0
Total (orig.)	210	4721	2418	619	687	1581	2733	1163	711	1073
Total (opt.)	210	4722	2417	619	687	1581	2735	1168	712	1073

Table 3.5: Categorization of detected line segments, Test Images #21 to #30

Test Image #	21	22	23	24	25	26	27	28	29	30
Identical	1789	1944	1115	2081	1024	686	754	1439	323	1950
Changed	0	4	0	5	0	2	0	2	0	0
New	0	2	0	0	0	1	0	2	0	0
Lost	0	0	0	0	0	0	0	0	0	0
Total (orig.)	1789	1948	1115	2086	1024	688	754	1441	323	1950
Total (opt.)	1789	1950	1115	2086	1024	689	754	1443	323	1950

Table 3.6: Categorization of detected line segments, Test Images #31 to #40

Test Image #	31	32	33	34	35	36	37	38	39	40
Identical	3896	2409	1217	391	1623	755	772	2641	701	801
Changed	11	16	2	0	4	0	1	3	0	0
New	3	0	0	0	0	1	0	1	1	0
Lost	0	0	0	0	0	0	0	0	0	0
Total (orig.)	3907	2425	1219	391	1627	755	773	2644	701	801
Total (opt.)	3910	2425	1219	391	1627	756	773	2645	702	801

On our test image set, the number of changed line segments was 0.11% on average, and 0.66% at most (Test Image #32). On average, the results of the optimized algorithm contained 0.05% new line segments. Only four test images suffered from lost line segments with only one each. The optimized algorithm yielded completely identical results for 19 of the 40 test images. On average, 99.84% of the line segments detected by the optimized algorithm were categorized as *identical*.

The reason for which *new* line segments are present in the optimized algorithm while missing from the output of the original algorithm is that different steps are performed in the rectangle refinement heuristic. Due to the approximation error, different sets of rectangle width and precision are used when progressing through the steps of rectangle refinement, which in turn would allow for a successful verification. The approximated verification is designed to never verify a line segment that would not be verifiable using the original calculation. Since the number of *new* line segments in our test image set is higher than the number of *lost* line segments, one could argue that the approximation slightly improves the detection quality of the original LSD algorithm.

4 Step-length algorithm

The scope of this chapter is the introduction of the step-length algorithm. The step-length algorithm was developed to provide an alternative to existing line segment detection algorithms. The main design goal was to provide a good detection quality, while allowing for an efficient implementation on FPGA and application-specific integrated circuit (ASIC) platforms. For this purpose, the algorithm avoids using a frame buffer, and operates on a stream on incoming pixels with limited buffering that can be implemented solely with block RAM (BRAM) on FPGA platforms.

This chapter is divided into three parts: In Section 4.1, the working principle of the algorithm is explained. In Section 4.2, the algorithm is described on a high-level algorithmic level. In Sections 4.3 and 4.4, details on the software and FPGA implementations are given. Section 4.5 is entirely dedicated to analyzing the results.

4.1 Background

The Brensenham algorithm [Bre65] is a commonly used algorithm to draw line segments in bitmap images. Given the two endpoints (x_s, y_s) and (x_e, y_e) of a line segment, the algorithm draws the line onto the raster by selecting pixels to be colored in between the two endpoints. The algorithm

is shown in Algorithm 4.1.

Algorithm 4.1: Bresenham's line drawing algorithm

input : Endpoints (x_s, y_s), (x_e, y_e)
output: A set D of pixels representing the drawn line segment (first octant)

```
1  x ← x_s;
2  y ← y_s;
3  Δx ← x_e − x_s;
4  Δy ← y_e − y_s;
5  D ← (x_s, y_s);
6  e ← Δx/2;
7  while x < x_e do
8      x ← x + 1;
9      e ← e − Δy;
10     if e < 0 then
11         y ← y + 1;
12         e ← e + Δx;
13     D ← D ∪ (x, y);
```

(x_s, y_s) is defined as the starting point of the line segment, while (x_e, y_e) is defined as the endpoint. x and y are variables used to iterate over the pixels. Δx and Δy represent the distance of the two points regarding either of the two directions. Note that if Δx or Δy is non-positive, signs have to be swapped. Additionally, if $|\Delta x| < |\Delta y|$, the roles of x and y in the algorithm are swapped. e is a variable depicting the current error value due to rasterization.

If $|\Delta x| > |\Delta y|$, we define x as the *fast direction* and y as the *slow direction*. If $|\Delta x| < |\Delta y|$, we define y as the *fast direction* and x as the *slow direction*. If x is the fast direction, the algorithm will draw multiple pixels in a row until the error variable e validates a change to the row above or below. If y is the fast direction, multiple pixels will be drawn in a column until the error variable forces a transition to a neighboring column. We define neighboring pixels in the same row or column as a *step*.

The *step length* (number of neighboring pixels in a step) depends on the value of e at the start of the loop. Therefore, the length of the first step is $\left\lceil \frac{\Delta x/2}{\Delta y} \right\rceil$, whereas the following steps (except for the last one) are in the range of $[\left\lceil \frac{\Delta x}{\Delta y} \right\rceil - 1, \left\lceil \frac{\Delta x - 1}{\Delta y} \right\rceil]$. For all Δx and Δy, it holds that the difference between $\left\lceil \frac{\Delta x}{\Delta y} \right\rceil - 1$ and $\left\lceil \frac{\Delta x - 1}{\Delta y} \right\rceil$ is at most 1 (Equation 4.1).

$$\left\lceil \frac{\Delta x - 1}{\Delta y} \right\rceil - (\left\lceil \frac{\Delta x}{\Delta y} \right\rceil - 1) \in \{0, 1\} \tag{4.1}$$

Therefore, all steps in a line segment drawn by Bresenham's algorithm, with the exception for the first and the last one, have identical step lengths, with a tolerance of ± 1. In the step-length algorithm, this property is used by tracking step lengths on an edge map of the image created by Canny's edge detection algorithm [Can86].

A line segment is detected when we record a connected chain of steps, with all step lengths defined to be within ± 1 of each other. In addition, the *direction* and the *orientation* of each step has to be compatible. The *direction* refers to whether the predecessor steps are on the top-left or

top-right, the former indicating the successor being on the bottom-right and the latter indicating the successor being on the bottom-left side. The *orientation* of a step specifies whether the steps are horizontal or vertical.

4.1.1 Angle tolerance

A problem with this basic approach is that the angle tolerance (the maximum change of angle within a line segment candidate) is not constant: The tolerance is the highest when the nomial step length is 2, and falling with rising step length, converging towards zero (Figure 4.1). The angle tolerance $\varphi(n)$ for nomial step lengths $n \geq 2$ can be calculated by Equation 4.2, while $\varphi(1)$ can be calculated by Equation 4.3.

$$\varphi(n) = \arctan(n+1) - \arctan(n-1) \tag{4.2}$$

$$\varphi(1) = \arctan(2) - \arctan(1) \tag{4.3}$$

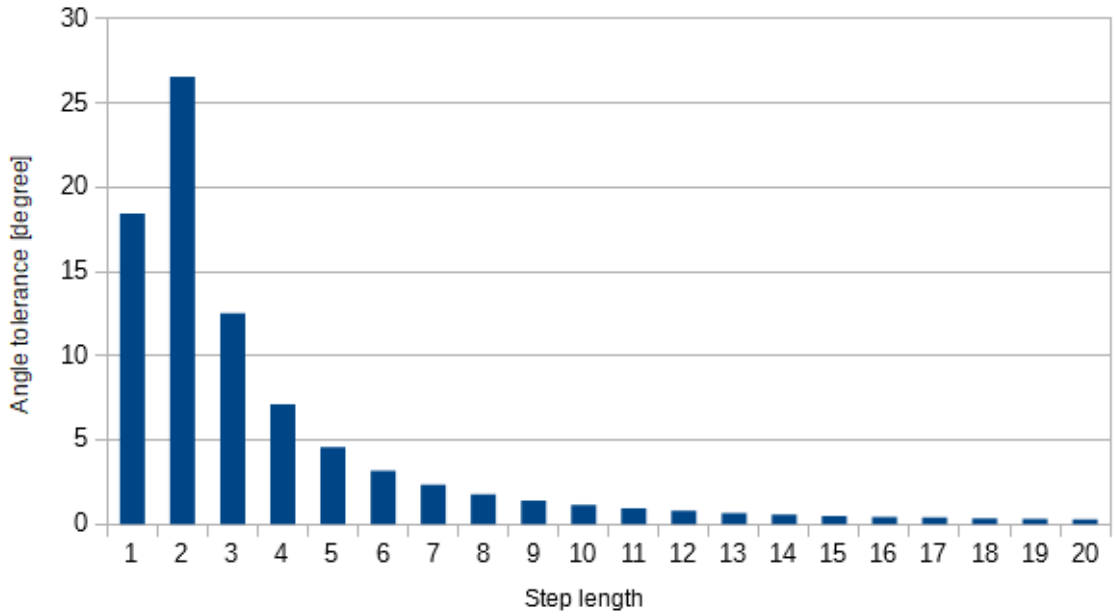


Figure 4.1: Angle tolerance for various step lengths. The tolerance for $n = 1$ refers to the angle tolerance when the orientation is already defined (which happens once the 1-pixel step is connected to at least one horizontal or vertical step with length 2)

A too small angle tolerance can hurt the detection quality since even the slightest noise or quantization error can disrupt the detection of a very flat or steep line segment. For this reason, a slightly different approach is used: Instead of limiting the allowed range to be within ± 1 of the nomial step length, the following scheme is used: If a step with length 1 is encountered, the other steps in the line segment are limited to a length of 3 or less. If a step with length of 4 or greater is encountered, the other steps in the line segment are limited to a length of 2 or greater. Making steps with length $n = 1$ and $n \geq 4$ mutually exclusive inside a line segment results in a

constant angle tolerance that matches the value of $n = 2$ using the original approach (Equation 4.4), assuming the same rules for *orientation* and *direction* are used.

$$\varphi = \arctan(3) - \arctan(1) = \arctan(\infty) - \arctan(2) \approx 26.57^\circ \tag{4.4}$$

4.2 Algorithm description

4.2.1 Canny Edge Detection

The first part of the step-length algorithm encompasses an implementation of Canny's edge detection algorithm, taking a grayscale image as an input and delivering a 1 bit edge map as output [Can86]. The implementation is similar to the one used by Zhou et al. [ZCW18]. The processing steps as well as all modifications done are summarized here:

Gaussian filtering A 3×3 filter is used for blurring the image. Unlike in the implementation from Zhou et al. [ZCW18], the filter is not decomposed into separate horizontal and vertical averaging steps. Also, the weights are chosen so that the weight multiplication can be replaced by a shift operation (Equation 4.5).

Directional gradients A sobel filter (Equations 2.3a and 2.3b) is used to calculate the gradients in x and y directions.

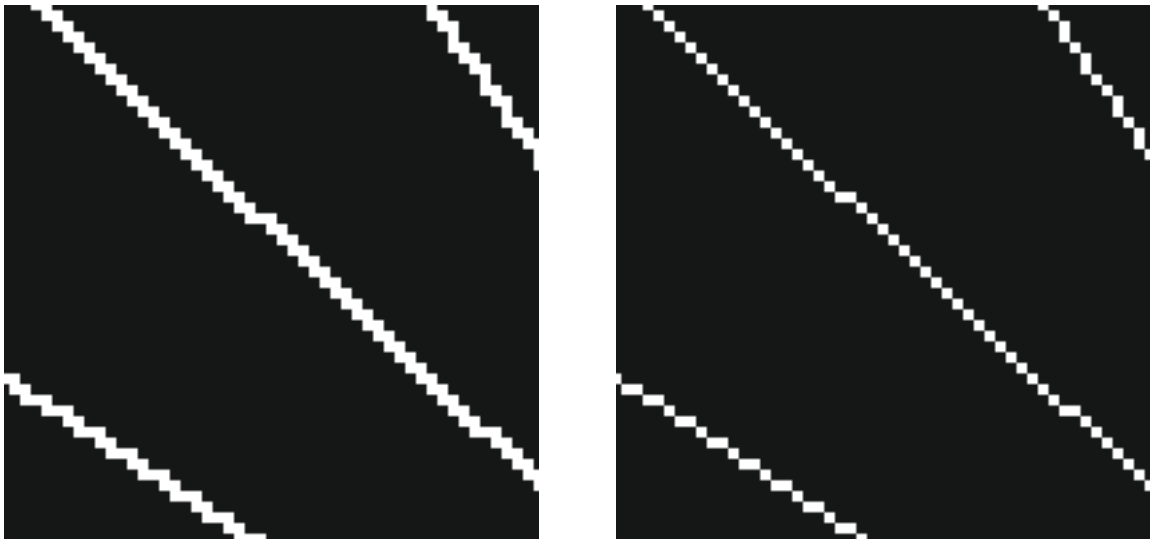
Gradient magnitude & angle The gradient magnitude is calculated by the sum of the absolute value of the directional gradients. The gradient angle is calculated as a 1 bit value, indicating whether the gradient is stronger in the horizontal or vertical direction. The significance of this change to the original implementation will be discussed later in this Section.

Non-maximum suppression To improve the detection quality of the step-length algorithm, a different thinning scheme is used, which also takes the magnitudes of surrounding pixels into account. See Section 4.4.3 for details.

Double threshold No significant changes were made to the double threshold implementation. A single pass in scan-line direction is used to update weak edge pixels, using a heterogeneous sliding window (Section 2.1.5). The selected threshold values are $t_l = 5$ and $t_h = 10$ (assuming the intensity values of the input are in the range of $[0, 255]$).

$$K_G = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{4.5}$$

As seen in Figure 4.2a and Figure 4.2b, the resolution of the gradient angle and its effect on the non-maximum suppression step has a noticeable impact on the resulting edge map. On the left image, we see the steps overlapping each other, while on the right side the steps are only connected diagonally. Overlaps would require the introduction of non-natural step-length (overlapping pixels can be interpreted as contributing half a pixel of length for each step), which would increase the complexity of the algorithm. To avoid overlaps, the gradient angle uses a



(a) 2 bit angle value **(b)** 1 bit angle value

Figure 4.2: Comparison of edge maps with different angle resolutions

resolution of 1 bit, which eliminates them for the most part. However, it is possible that straight edges with angles of 45° or 135° still have overlaps. This is caused by neighboring steps that are each two pixels wide, with identical gradient magnitudes, but alternating gradient angles. The non-maximum suppression compares the left (in case the gradient is stronger in y direction) or top (in case the gradient is stronger in x direction) neighbor with a *less-than-or-equal* operator. However, since this neighbor itself checks different pixels, both are selected as edge pixels during thinning (Figure 4.3). Since this issue cannot be avoided with the current implementation of edge thinning, steps with a length of two are a special case in the following parts of the step-length algorithm, with dedicated checks for overlaps.

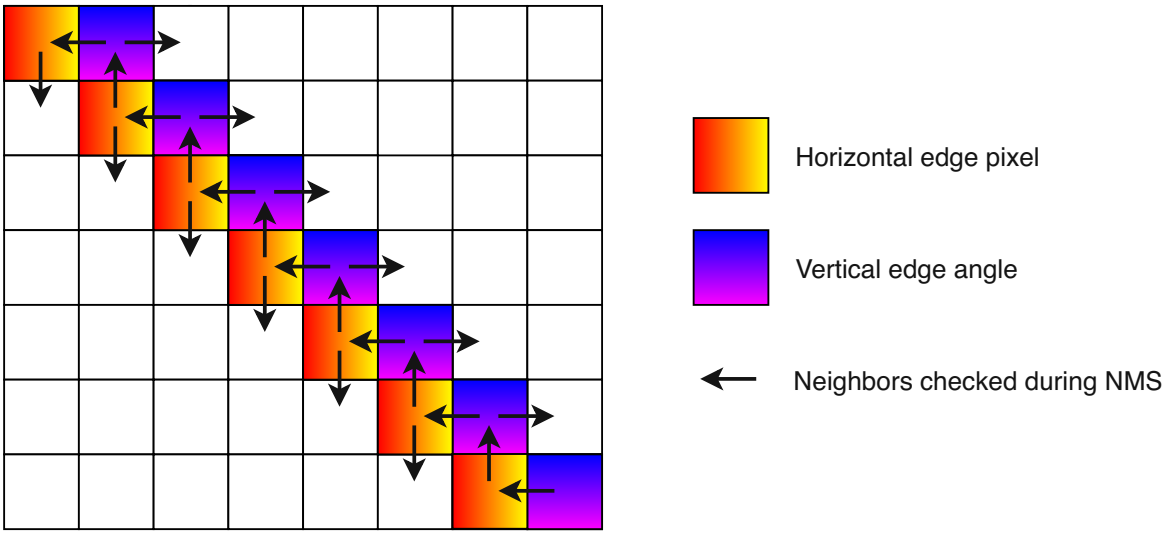


Figure 4.3: Overlaps in a 135° edge

4.2.2 Step record data structure

The core part of the step-length algorithm requires to keep track of the state of steps in the preceding row. This structure is referred to as the *step record*. Step records are created each row, usually one for vertical steps and two for horizontal steps (one for each diagonal growth direction). Table 4.1 lists the fields contained in the record, where W refers to the image width in pixels and H refers to the image height in pixels.

Table 4.1: Step record data structure

Field Name	Length (bit)	Description
`sx`	$\lceil log_2 W \rceil$	x coordinate of line segment starting point
`sy`	$\lceil log_2 H \rceil$	y coordinate of line segment starting point
`pdir`	2	parent direction
`ori`	2	step orientation
`len`	2	step length flags
`cstep`	$\lceil log_2 H \rceil$	vertical step pixel counter

`sx` & `sy`: These two fields are used to store the start point of a line segment. For vertical steps, the starting point is always the top-most pixel of the first step. For horizontal steps, it is the first step's left-most pixel (in case the line segment grows in the bottom-right direction) or right-most pixel (in case the line segment grows in the bottom-left direction).

`pdir`: This field stores the *parent direction*, which is the position of the predecessor (top neighbor) step. The encoding for this field is listed in Table 4.2. For vertical steps, this value is initialized as undefined in the first step. For horizontal steps, two records are created - one for each `pdir` direction.

`ori`: This field stores the *orientation*, which allows for distinguishing whether steps are horizontal or vertical. The encoding for this field is listed in Table 4.3.

`len`: This field is set whenever a step with length $n = 1$ or length $n \geq 4$ is encountered, to enforce the condition discribed in Section 4.1.1. The encoding for this field is listed in Table 4.4.

`cstep`: This field keeps track of the number of pixels in the *current* (vertical) *step*. This is required since the count of edge pixels in the current vertical step has to be made available to the succeeding pixel in the next row. For horizontal steps, all pixels of a step are processed consecutively in the scan-line direction, and step records are not updated until the last pixel of the edge is encountered. For this reason, this field remains unused for horizontal steps.

Table 4.2: Values of the `pdir` field

Value	Symbol	Description
`00`	`LEFT`	Parent is top-left
`01`	`RIGHT`	Parent is top-right
`1X`	`UNDEF`	Undefined

Table 4.3: Values of the `ori` field

Value	Symbol	Description
`00`	`HOR`	Horizontal steps
`01`	`VER`	Vertical steps
`1X`	`UNDEF`	Undefined

Table 4.4: Values of the `len` field

Value	Symbol	Description
00	EQ_1	Contains a step with length = 1
01	GEQ_4	Contains a step with length ≥ 4
1X	UNDEF	No steps with lengths = 1 or ≥ 4 encountered

Additionally, each step record has an additional parameter called the *index*. That is the x coordinate of the *junction point*, the pixel of a step that connects to the successor step or to the next pixel inside a vertical step. The successor step or pixel in the next row will use the index to find the step record of the predecessor. There are two design alternatives with regards to the storage and retrieval of step records, which are discussed here:

Variant 1: RAM Step records minus the index are stored explicitly in a RAM, while the index is used as the address. The RAM requires W entries, one for each possible index value.

Variant 2: FIFO A first in, first out buffer (FIFO) is used to store step records, including their indices. The FIFO needs to be able to hold $\lceil \frac{2}{3}W \rceil + 1$ entries, since that is the maximum number of entries that can be created within a row (assuming all steps are horizontal steps with a size of 2, separated by a single non-edge pixel). Storing the index requires $\lceil log_2 W \rceil$ bits in addition to the size of the step record itself.

Assuming $W_b = \lceil log_2 W \rceil$ is the number of bits required to store an x coordinate and $H_b = \lceil log_2 H \rceil$ is the number of bits required to store a y coordinate, the size of storage required for using a RAM is $(W_b + 2H_b + 6)W$. Under the same assumptions, the size of storage required for using a FIFO is $(2W_b + 2H_b + 6)(\lceil \frac{2}{3}W \rceil + 1)$. In Table 4.5, the required storage size is calculated for various common image resolutions, which shows that the FIFO variant requires less storage in any real-world scenario. The rightmost column denotes the relative saving in memory usage when using the FIFO design over the RAM design, in percent.

Table 4.5: Storage size requirements for both variants and various image resolutions and relative savings when using the FIFO variant

Resolution	Storage RAM (bit)	Storage FIFO (bit)	Storage size reduction (%)
512 × 512	16,896	14,406	14.7
640 × 480	21,760	18,832	13.5
800 × 600	28,800	24,610	14.5
1920 × 1080	74,880	64,050	14.5
1080 × 1920	42,120	36,050	14.4

4.2.3 Step linking

Line segment detection is done by linking steps with compatible step lengths (Section 4.1.1), as well as compatible values for the `ori`, `len` and `pdir` parameters. To achieve this, the algorithm runs in scan-line direction (row-wise scan from left to right, top to bottom) to look for edge pixels. Whenever we encounter an edge pixel, we proceed and count the number of edge pixels until a non-edge or the border of the image is reached. These edge pixels are either a new horizontal step or part of a vertical step.The following processing steps are performed:

1. **Check for valid predecessor steps:** We check whether the predecessor steps are compatible for linking with the edge pixels we encountered. For this reason, the step record indexed on the top or top left neighbor of the first edge pixel we encounter is captured until the last edge pixel of the current step is being being processed. This is necessary to evaluate the length of a horizontal step in the current row.

2. **Report incompatible predecessor steps:** In case predecessor steps are incompatible, we have to *report* (output) them as line segments. The start point of the reported line segment is set to be the starting coordinates (`sx`,`sy`) stored in the step record of the predecessor, whereas the endpoint of the line segment is the junction point of the predecessor. For vertical steps, this step is only performed if the bottom neighbour is a non-edge (end of a vertical step), as the length of the current vertical step is otherwise still unknown.

3. **Create new step records:** Create new step records, either by inheriting the information of valid predecessors or by initializing them if no valid predecessor exists.

4. **Report step without successors:** If there are no successors to the current step in the next row, the new step record has to be reported as a line segment, with the junction point and the stored coordinates in the step record to be used as its endpoints.

The algorithm distinguishes four different cases, depending on the number of neighboring line pixels encountered:

Case 1: One edge pixel (`v_step`)

This case is triggered once we encounter a single edge pixel during our horizontal scan of the image, with no edge pixel neighbors on the left or right side. The following processing steps are performed to update the step records and detect line segments:

1. **Check top neighbor:** If there is a step record indexed on the top neighbor pixel and that step is not horizontal (`ori != HOR`), then this pixel is part of a vertical step that has already been initialized. The new step record inherits information from predecessor. However, the `cstep` value is incremented. Furthermore, the `ori` field is set to value `VER`, as we can rule out at this point that the chain contains (short) horizontal steps. Note that the `len` field is not updated until the end of the step is reached (which is the case when the bottom neighbor pixel is a non-edge). An example for such a transaction is shown in Figure 4.4.

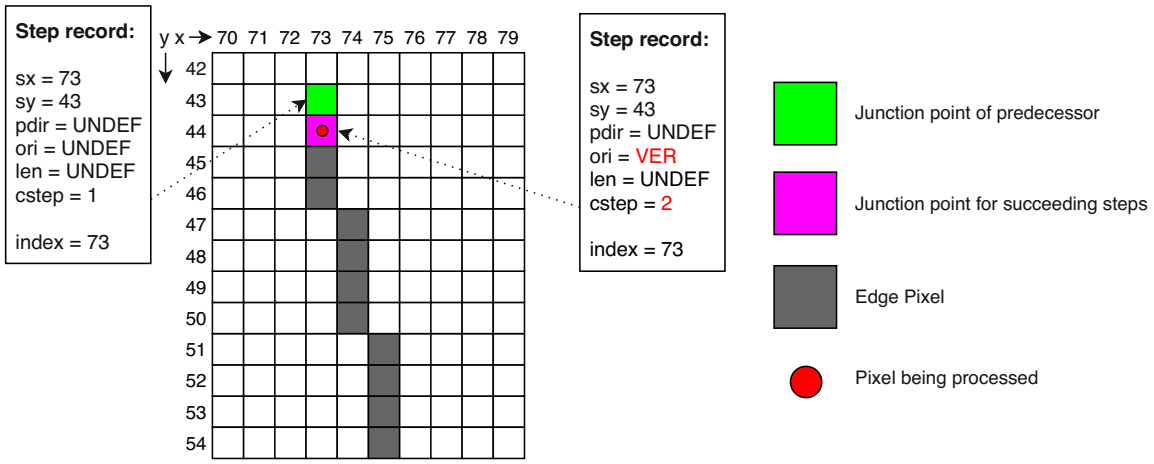


Figure 4.4: Extending a vertical step

2. **Check top-left and top-right neighbors:** If there is no (valid) step record indexed at the top neighbors position, the step records indexed at the top-left and top-right neighbor are checked. The conditions for these step records to be valid predecessors are slightly different: For one, the `pdir` value must be compatible (for example, for the step record on the top-left side, `pdir` must be `LEFT` or `UNDEF`). Additionally, `ori == HOR` is only permitted if `len != GEQ_FOUR` (to enforce the step length condition). If any of the two top-diagonal predecessors are valid, their information is inherited and their corresponding line segments are extended. In the new step record, the `pdir` value is updated corresponding to the neighbor we chose for linking, and `cstep` is reset back to 1 . If both top-left and top-right predecessors are valid, the left one is picked per default. A potential design to remove this limitation is proposed in Section 4.2.4. All predecessors which were not chosen for extension are reported as line segments, using the `sx` and `sy` values as the starting point and the junction point as the endpoint (Figure 4.5).

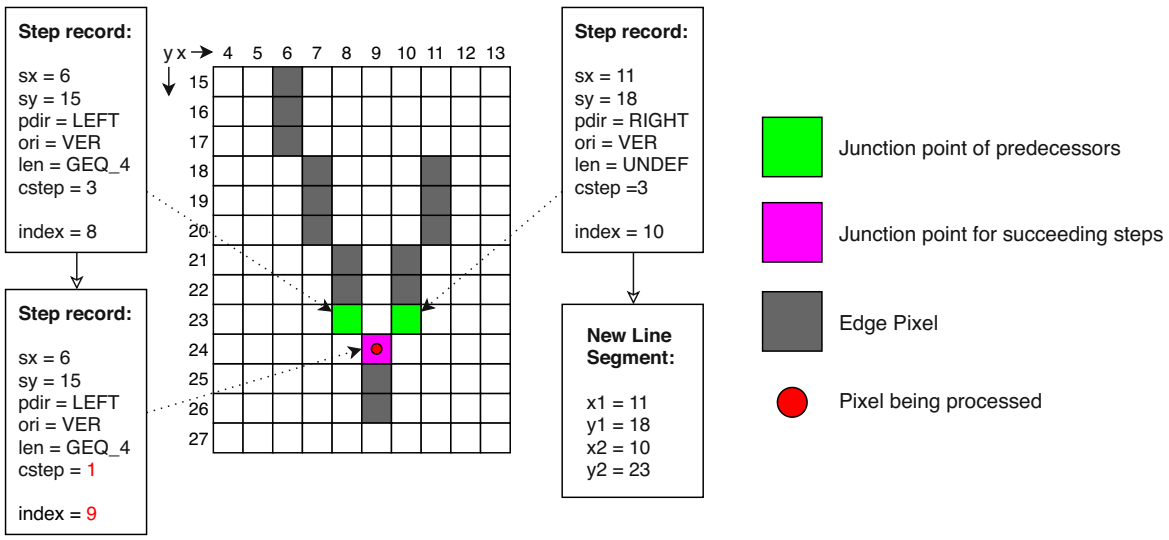


Figure 4.5: The left predecessor gets extended, while the right one terminates and gets reported as a line segment.

3. **Initialize step record:** If there are no valid predecessors at all, the step record is initialized: `sx` and `sy` are set to the coordinates of the current pixel. `cstep` is initialized to 1, and `ori` and `pdir` are set to `UNDEF`. `len` is initialized to `UNDEF` (Figure 4.6).

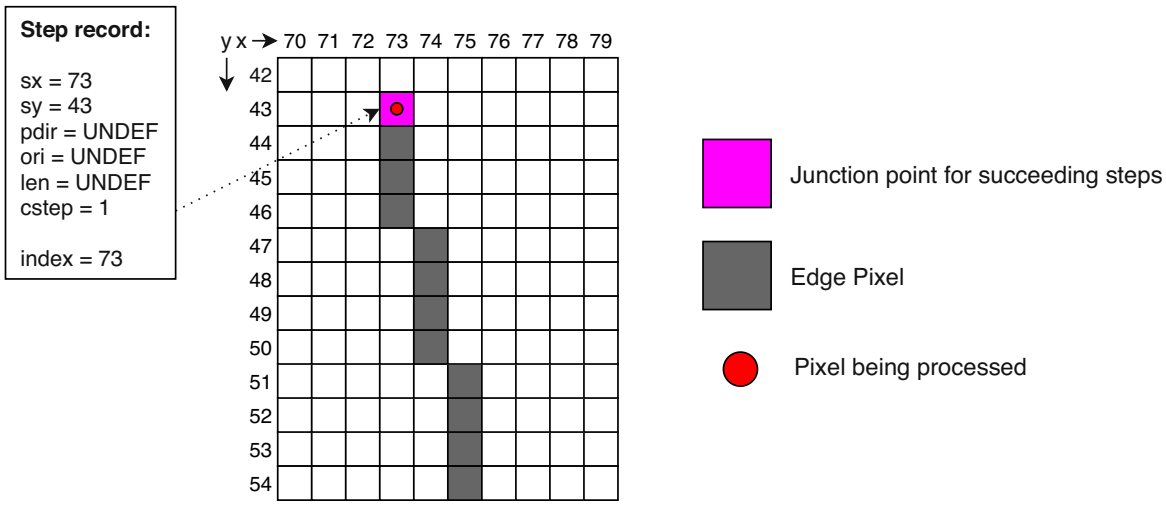


Figure 4.6: A step record without predecessor get's initialized.

4. **Check bottom neighbor:** Next, the bottom neighbor is checked. If it is not an edge pixel, the current vertical step has reached its full length, and we have to check whether its length is compatible with the predecessors'. If predecessors to the current step exist, we check if the length of this vertical step is compatible with the `len` value stored in the step record. If not, the predecessor steps get reported as a detected line segment, and the `sx` and `sy` values of the new step record get set to the top end of the current vertical step (Figure 4.7). This is also done if there is no predecessor.

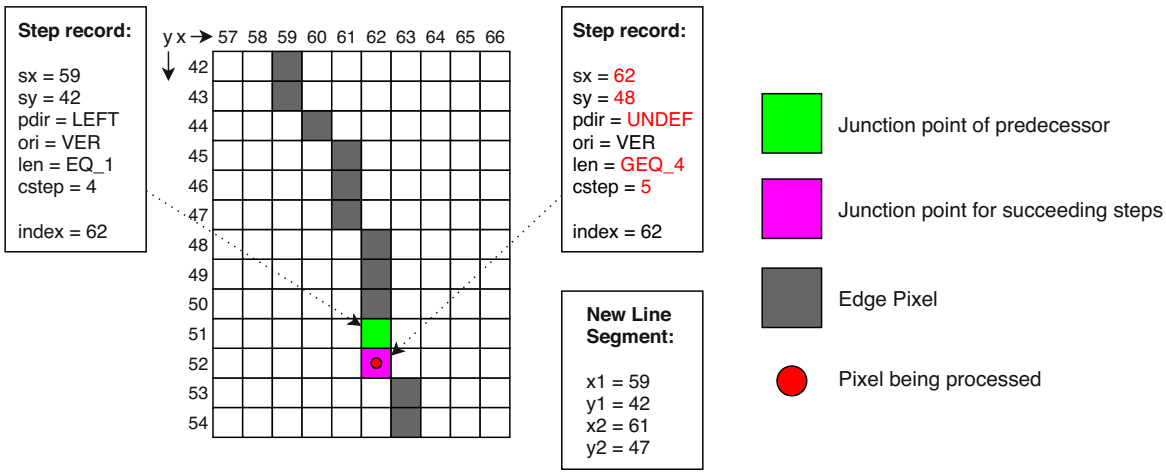


Figure 4.7: If the step-lengths don't match, the line segment of the predecessor gets reported and the step record is re-initialized.

5. **Check successors:** If no successor exists (bottom-left, bottom and bottom-right neighbors are all non-edges), we report another line segment, starting at the coordinates `sx` and `sy` stored in the step record and ending in the pixel that is currently being processed.

Case 2: Three or more edge pixels (`h_step`)

In case we encounter at least three neighboring edge pixels in a row, a new horizontal step is registered. This case is more straightforward than the first one, since we don't have to account for vertical steps. Also, we don't have to count pixels in the `cstep` field, as all pixels in a step are placed on the same row, and are therefore consecutively processed by the algorithm.

When the first edge pixel is encountered, the step record indexed at the top or top-left neighbor of the pixel is captured and stored. This information is not used until the last edge pixel is encountered, since up to this point the length of the new step is still unknown. The following processing steps are performed once the last edge pixel is encountered:

1. **Check predecessors:** Both the captured step record, if it exists, as well as the step record indexed at the top or top-right neighbor of the last edge pixel are checked for compatibility with the new horizontal step. This includes compatible `pdir` values (`pdir != LEFT` for right predecessors and vice-versa), fitting orientations (`ori != VER`) and compatible step lengths with regards to the `len` field.

2. **Create new step records:** Two step records are created for the new step, one indexed at either end. If the right predecessor is valid, the information is inherited by the left step record and information from a valid left predecessor is inherited by the right step record. If a predecessors is not valid, the corresponding new step record is instead initialized, setting `sx` and `sy` to the coordinates of the other end of the new step (Figure 4.8 and Figure 4.9).

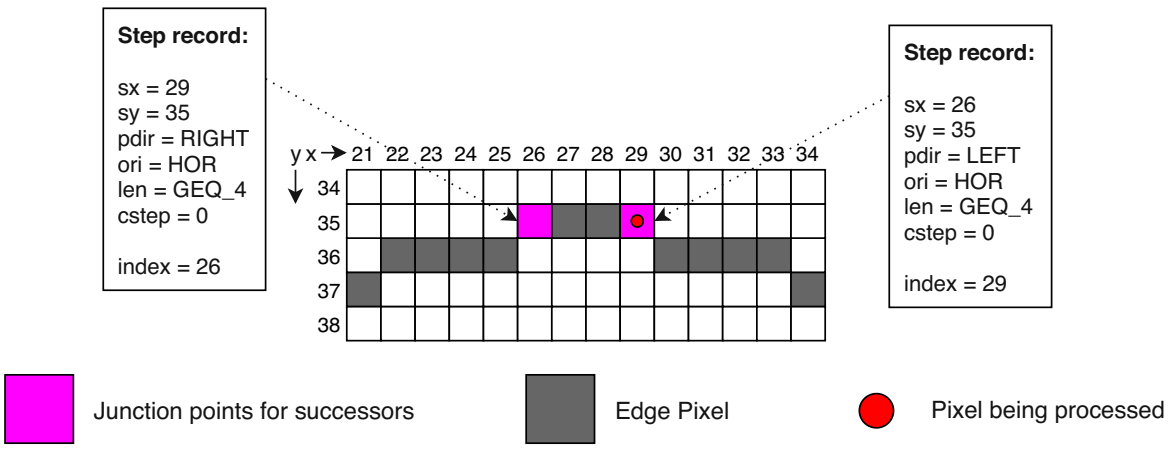


Figure 4.8: The step records of a horizontal step get initialized (no predecessors)

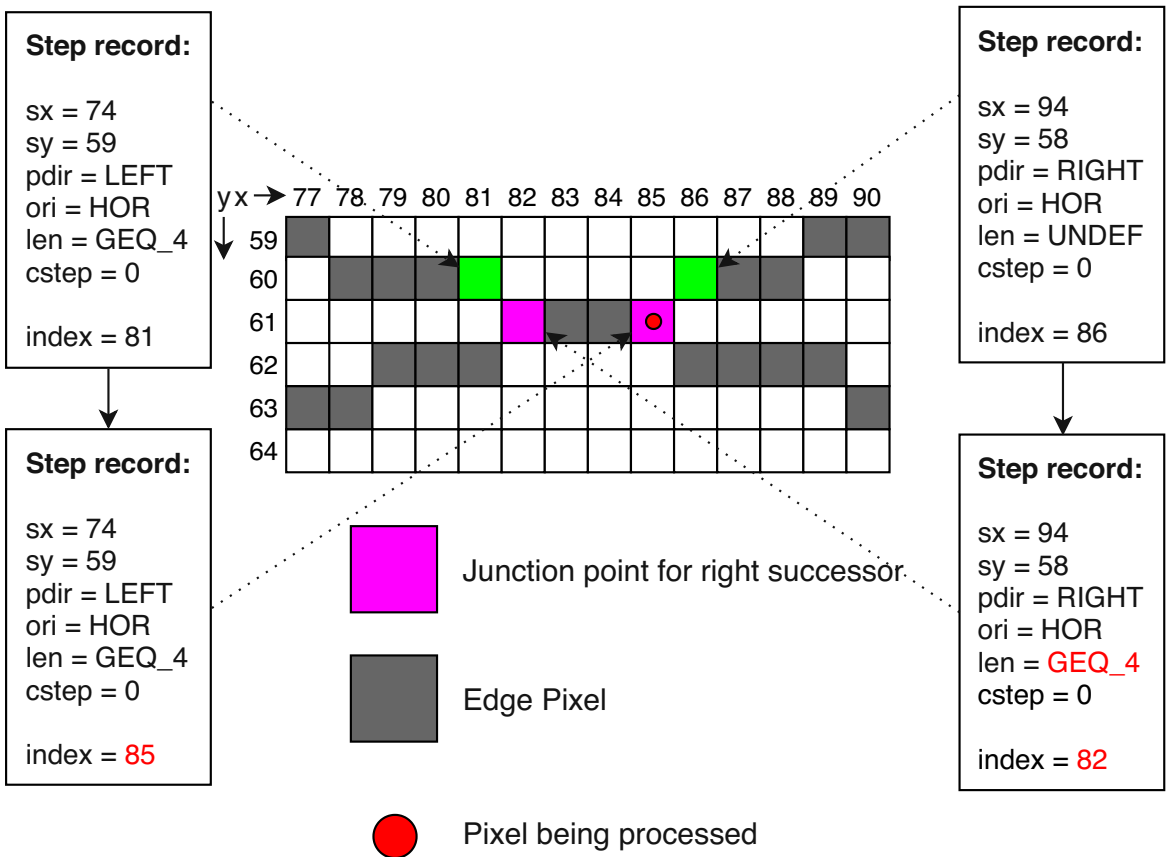


Figure 4.9: The step records of a horizontal step inherit information from their predecessors.

3. **Report line segments:** Up to three line segments have to be reported:

   (a) A line segment has to be reported if either the top-left predecessor is existent but invalid, or if there is a valid top-left predecessor but no potential successor on the bottom right side.

   (b) A line segment has to be reported if either the top-right predecessor is existent but invalid, or if there is a valid top-right predecessor but no potential successor on the bottom left side.

   (c) If there are neither valid predecessors nor potential successors, a horizontal line segment that is exactly identical to the current step has to be reported.

Case 3: Two edge pixels (`two_step`)

Two neighboring edge pixels in a row can either be a single horizontal step, or an overlap of two vertical steps, which also includes the case where overlaps are caused by 45°/135° diagonals (Figure 4.3). Depending on whether or not overlap is detected, two cases are distinguished:

- **Overlap case:** The condition for detecting an overlap is having a valid predecessor index on top of either of the two edge pixels, as well as having an edge pixel (potential successor) below the other one. The conditions for a valid predecessor are `ori != HOR` and a noncontradictory

`pdir` value (`pdir != LEFT` for the right predecessor and vice versa). If an overlap case is detected, the `cstep` variable of the predecessor is checked for compatibility with the `len` flags, following the same steps as in Case 1 if the last pixel of a vertical step is reached due to the bottom neighbor being a non-edge pixel. Then a new step record is created, either inheriting information from the predecessor (Figure 4.10) or being initialized like in Case 1.

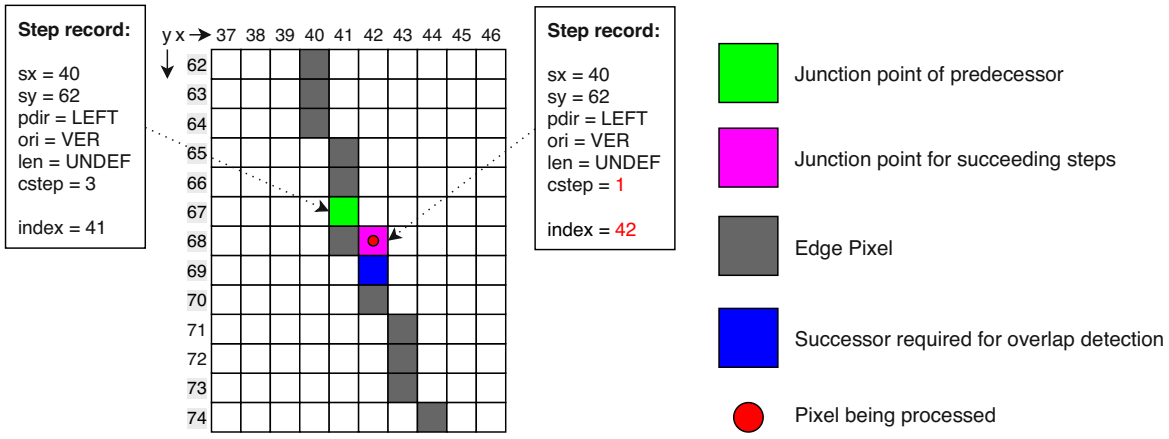


Figure 4.10: Overlap treatment

- **Horizontal step case:** If the conditions for detecting an overlap are not satisfied, the algorithm treats the two pixels as a horizontal step with a length of two, using the same algorithm as in Case 2.

Case 4: Vertical step intersecting with a horizontal step (`h_term`)

Finally, we have to account for a special case where a vertical step ends in a horizontal step (Figure 4.11). Therefore, for edge pixels inside a horizontal step, we check if there is a vertical step record indexed at the top neighbor. If so, the corresponding line segment gets reported.

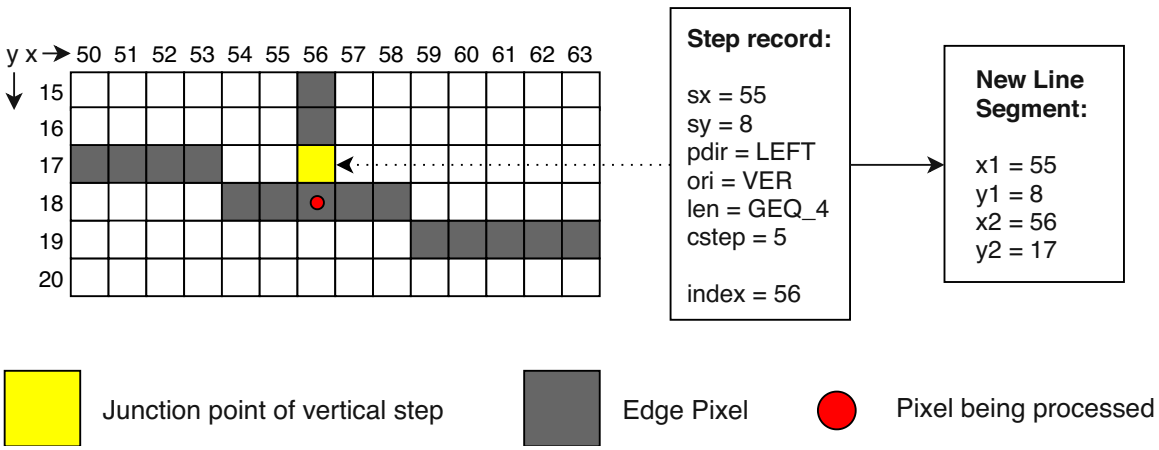


Figure 4.11: While processing edge pixels of a horizontal step, we have to check for a vertical step terminating.

4.2.4 Vertical intersections

As mentioned earlier, if two line segments intersect into a single vertical step, by default, only the top-left neighbor is selected for linking. This limitation could be removed by storing two step records (one linked with each predecessor) into the FIFO with the same `index`. Since now up to two step records are stored in a vertical step, and vertical steps are separated by at least one non-edge pixel, the number of maximum step records per line would increase from $\lceil \frac{2}{3}W \rceil$ to W, therefore increasing the amount of memory required to store step records by $\approx 50\%$ (Table 4.6).

Since this is a severe increase in memory usage for a minor quality improvement and due to the fact that other algorithms like LSD and EDLines have similar limitations, only one predecessor is stored for vertical steps.

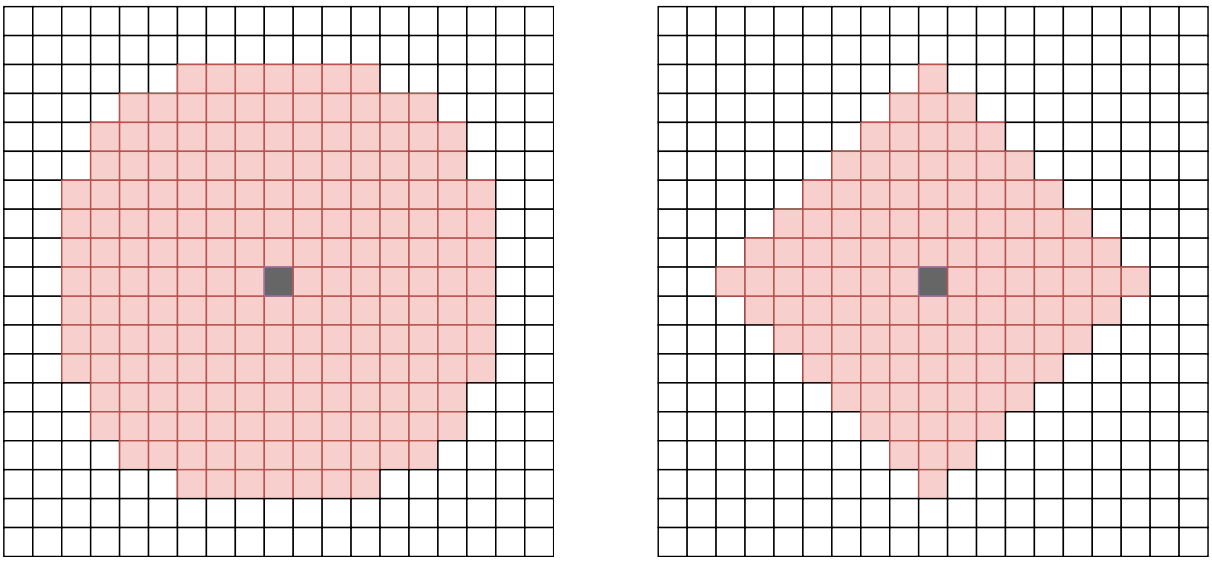
Table 4.6: Storage increase for step records required when storing both predecessors of vertical steps

Resolution	Absolute storage increase (bit)	Relative storage increase (%)
512×512	8,330	49.7
640×480	11,076	49.9
800×600	27,234	49.8
1920×1080	37,760	50
1080×1920	21,240	50

4.2.5 Length threshold

Whenever steps are incompatible for linking due to a mismatch of lengths or orientations, the algorithm tends to output a lot of short line segments and in some cases line segments with identical endpoints. Therefore, a processing step to filter short line segments is required.

The length of a line segment (p_1, p_2) is defined as the distance of its two endpoints $p_1 = (x_1, y_1)$ and $p_2 = (x2, y2)$. Different metrics can be used to specify the length of a line segment, like the *Euclidean distance* $d_e(p_1, p_2) = \sqrt{\Delta x^2 + \Delta y^2}$, with $\Delta x = |x_1 - x_2|$ and $\Delta y = |y_1 - y_2|$. Furthermore, a threshold c is defined, so that all line segments with $d_e(p_1, p_2) < c$ are too short and therefore should be removed from the line segment list. Graphically, as shown in Figure 4.12a, this can be interpreted as an area around p_1 that constitutes all positions of p_2 that would result in an invalid line segment. In the case of the Euclidean distance, this area approximates a circular area. Alternatively, if the *Manhattan distance* $d_m = \Delta x + \Delta y$ is used as the metric, the resulting area approximates a rotated square (Figure 4.12b). With the same value for c, the Manhattan distance imposes a more lenient threshold on diagonal line segments. While the Manhattan distance is very easy to calculate, the Euclidean distance is just slightly more complex. It requires only two square operations additionally, assuming that the square root operation is eliminated by performing a check of $d_e^2(p_1, p_2) = \Delta x^2 + \Delta y^2 < c^2$ instead of calculating the square root. For sufficiently small threshold values (in our implementation, 8 is the default value), the square operation can be efficiently implemented using a LUT. For this reason, and due to the fact that showing a bias towards short diagonal line segments is not desired, the algorithm uses Euclidean distance as the length metric for filtering.



(a) Using the Euclidean distance d_e and $c = 8$

(b) Using the Manhattan distance d_m and $c = 8$

Figure 4.12: Comparison of different metrics for distance. If p_1 is the gray center pixel, the line segment is only long enough if p_2 is outside the red area.

4.2.6 Duplicate removal

In some instances, the algorithm reports some line segments more than once. This can happen if a line segment without predecessors has two successors, both of which are incompatible (Figure 4.13). Since checking for duplicates in the algorithm itself would further increase the complexity of the logic involved, duplicate detection and removal is performed as a post-processing step. This is done by storing N line segments that previously passed the stage and checking them against any new line segment entering the stage. If the new line segment is different from all N stored line segments, it is output and replaces the oldest of the stored line segments. All stored line segments are marked as invalid once a new frame starts, so that no duplicates can wrongfully be detected across different frames.

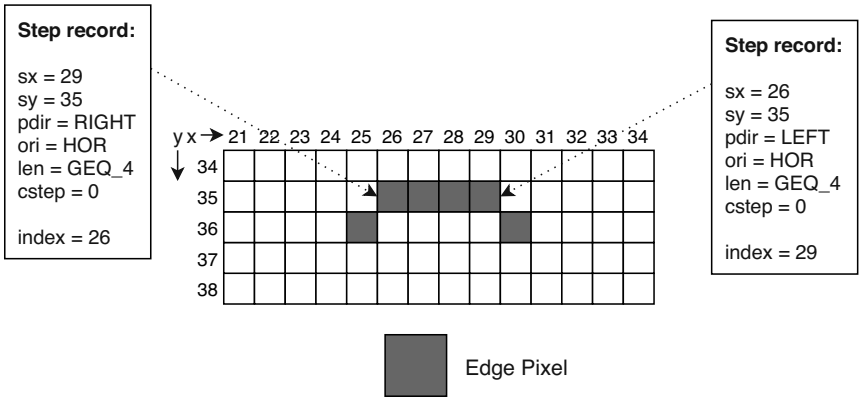


Figure 4.13: The long horizontal line segment will be reported independently by each incompatible successor, thus resulting in a duplicate line segment.

4.3 Software implementation

Most of the development on the algorithm was done using a software prototype (written in C), since it allows for quicker changes and testing. During the design phase, the suitability for a hardware implementation was always kept in mind. However, for the sake of simplicity, a few details in the software algorithm differ from the final hardware implementation:

- **No pipelining:** While the goal is to have a fully pipelined algorithm in hardware, the software implementation is not pipelined and single-threaded, which means that each processing step will not start until the previous one has finished processing the whole frame.
- **Image resolutions:** To avoid an additional layer of complexity, the supported image resolution of the hardware implementation cannot be changed during runtime. The software prototype is able to process images with any resolution.
- **Input & Output formats:** The software implementation uses `.PGM` and `.SVG` files for input and output, respectively. The hardware implementation uses a raw stream of magnitudes as input and outputs a raw stream of line segments (each encoded by a vector of the endpoints' coordinates, in unsigned integer format).
- **Floating-point arithmetic:** The implementation of the Canny edge detection uses double-precision floating-point arithmetic, whereas the hardware implementation uses integer and fixed-point arithmetic only. However, due to the simplistic nature of all arithmetic operations in the algorithm, this change has no effect on the results.
- **Step record storage:** While it was shown that using a FIFO is superior to using a RAM to store step records in terms of memory usage, for simplicity reasons, the software implementation uses a RAM indexed by the coordinates of the junction point. The RAM has therefore a number of fields equal to the resolution of the image. This design was chosen due to the software implementation not having any delays when writing or reading data to or from the FIFO, which would require special care to prevent the step record to be used in the same line it was created in. By indexing the step record by both the x and y coordinate of the junction point, the implementation of step records is straightforward, while the higher memory usage can be excused since the focus of the software prototype lies in functionality, and not on performance and efficient resource usage.

Note that even though this is a software implementation, the implementation of double thresholding (Section 2.1.4) still only uses a single pass in scan-line direction to update weak edge pixels, to keep parity with the hardware implementation of the algorithm.

4.4 Hardware implementation

The hardware implementation is based on an register-transfer level (RTL) model, written in Very High Speed Integrated Circuit Hardware Description Language (VHDL). The design is fully pipelined and processes one pixel of an input image per clock cycle. The design is able to process a continuous stream of pixels even across the borders of a frame. There is no need for horizontal or vertical line blanking. The only input signals required are clock, reset, and chip enable signals, as

well as an 8 bit input for the intensity value of the pixel. However, due to the end of frame being detected implicitly by counting the number of processed pixels, the design uses a fixed resolution that cannot be changed, unless it is resynthesized. On the output side, the design is able to generate up to one line segment per clock cycle. Additionally there is an output signal called `frame_info`, indicating whether the output belongs to an odd or even frame. This signal flips once all line segments of the previous frame have been output, therefore allowing the succeeding logic to detect the end of a frame. The algorithm is separated into multiple different stages (Figure 4.14):

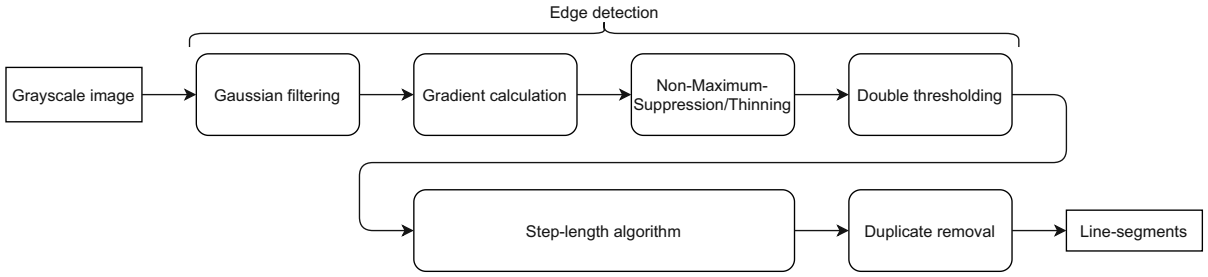


Figure 4.14: Overview of the processing steps

4.4.1 Gaussian filtering

The Gaussian filter was implemented using the sliding window technique described in Section 2.1.5. A 3×3 sliding window performs a weighted averaging on each pixel and up to 8 of its neighbors. The kernel matrix (weights) is shown in Equation 4.5, and was selected so that the multiplication can be implemented using shift operations only.

If any pixel of a sliding window would lie outside the area of the image (which is the case for all pixels on the borders of the image), the value of the nearest valid pixel is mirrored to that position (Figure 4.15).

The module's input and output are both in 8 bit unsigned integer format. The fractional part of the weighted averaging is truncated, since rounding would require an additional pipeline stage to meet the timing requirements for a minimal effect. The total pipeline delay is $W + 2$ - this is the delay until all pixels required to process the first pixel are input. No additional delay is added to this number, as the convolution result is output in an unregistered fashion.

4.4.2 Gradients

This module uses the same sliding window approach as the Gaussian filtering, including the same treatment of pixels on the outer edge of the image. A Sobel filter (Eq. 4.6a and 4.6b) is used to calculate the directional gradients. Afterwards, these values are used to calculate the gradient magnitude using the sum of absolutes (Eq. 4.7) and dividing the result by two, as well as the gradient direction, which is calculated at a binary resolution, by doing a comparison between the two directional gradients: If the horizontal gradient is higher than the vertical gradient, the edge direction is vertical, and vice versa.

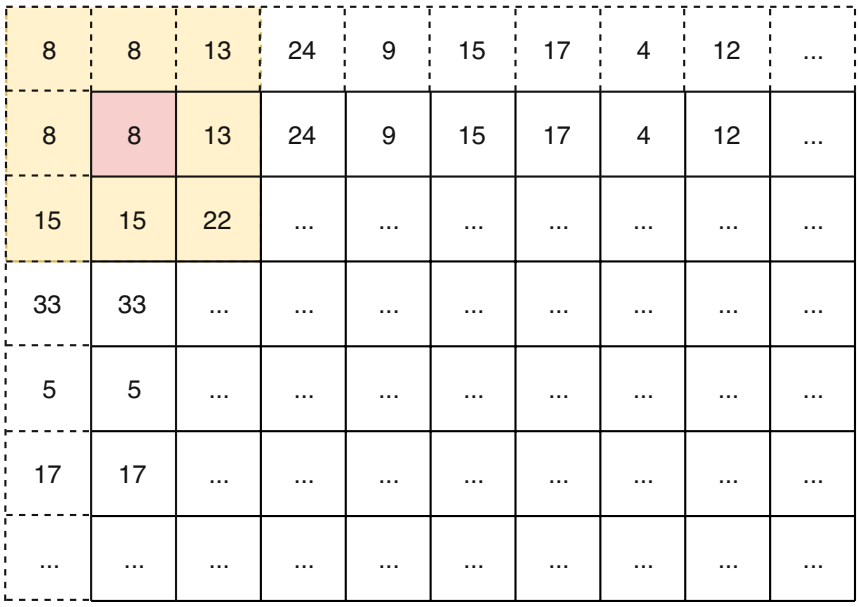


Figure 4.15: The sliding windows for the first pixel in the top-left corner of the image requires the values of 5 pixels which are outside the image range. For these pixels, the value of the nearest valid pixel is used.

The kernel uses the same mirroring strategy as the Gaussian module for edge and corner pixels, assuming the value of the nearest valid pixel for any pixel outside the image area.

$$K_x = \frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{4.6a}$$

$$K_y = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{4.6b}$$

$$G = \frac{1}{2}(|G_x| + |G_y|) \tag{4.7}$$

Internally, the directional gradients are stored in a 9 bit two's complement signed integer format, with magnitudes in the range $[-256, 255]$. The output format of the resulting gradient magnitude is 8 bit unsigned integer in the range $[0, 255]$ and the gradient angle is output as a 1 bit value.

The module needs $W + 2$ cycles to fill the sliding window for operation. Additionally, the calculation result for the directional gradients is internally registered, which was needed to meet the timing constraints. This results in a total pipeline delay of $W + 3$.

4.4.3 Thinning

This module uses the same sliding window approach as the Gaussian filtering and Sobel filtering. However, pixels outside the image boundaries are assumed to have a magnitude of zero, instead of mirroring the nearest valid pixel.

The original thinning scheme of the Canny edge detector only compared the magnitude of the center pixel with its orthogonal neighbors (relative to the edge direction). In the modified algorithm, the center pixel's neighbors in the edge direction (left and right neighbors for horizontal, top and bottom neighbors for vertical direction) are also checked against their respective orthogonal neighbors (the diagonal neighbors of the center pixel). If both those pixels have a higher magnitude than their orthogonal neighbors, and if their magnitudes as well as the magnitude of the center pixel are above the lower threshold of the double thresholding step, the center pixel is selected as an edge pixel, even if it does not have the maximum gradient magnitude among its orthogonal neighbors.

This modification of the thinning algorithm was chosen to smooth over single outlier pixels disrupting steps, since those have a negative impact on the (single pass) double thresholding as well as on the step-length algorithm. Unfortunately, it cannot smooth two or more succeeding outliers, since that would require the use of a bigger sliding window resulting in a higher pipeline delay. Figure 4.16 illustrates how the algorithm smooths the step by picking the pixel in the dead center over the outlier pixel, even though the latter has a higher magnitude.

0	0	0	0	0	0	0
11	10	13	20	9	14	11
22	25	32	15	27	30	23
12	8	10	12	11	13	9
0	0	0	0	0	0	0

(a) Gradient magnitudes before thinning

0	0	0	0	0	0	0
0	0	0	20	0	0	0
22	25	32	0	27	30	23
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(b) Thinning result using Canny-like thinning

0	0	0	0	0	0	0
0	0	0	0	0	0	0
22	25	32	15	27	30	23
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(c) Thinning result using the new thinning algorithm

Figure 4.16: Comparison of the old and new thinning algorithms' results. Gradient directions are assumed to be vertical.

The kernel has registers for intermediate results, resulting in a total pipeline delay of $W + 3$.

4.4.4 Double thresholding

This module uses a heterogeneous sliding window (Fig. 2.5). Pixels that were already processed are stored as a 1 bit value (`1` for an edge, `0` for a non-edge) and used in the computation of succeeding pixels. For a pixel to be recognized as an edge pixel, one of the three conditions has to apply:

1. The pixel's gradient magnitude is above the high threshold.
2. The pixel's gradient magnitude is above the low threshold and at least one of the succeeding neighbor pixels (in scan-line direction) has a magnitude above the high threshold.
3. The pixel's gradient magnitude is above the low threshold and at least one of the post-processed neighbor pixels is an edge pixel.

Pixels outside the image boundaries are assumed to be non-edges with a gradient magnitude of zero.

The two threshold values are parameters per default set to 8 and 12 resp., assuming the gradient magnitude lies in the range $[0, 255]$.

The output of the kernel is not registered, therefore the total pipeline delay is limited to the $W + 2$ cycles required to fill the sliding window.

4.4.5 Step-length algorithm

The step-length algorithm is implemented using a 3×2 sliding window, containing pixels from the current line and the succeeding line. The values of these 6 pixels as well as the top two elements of the step record FIFO are used for the step-length algorithm calculations. The architecture (Figure 4.17) resembles a heterogeneous sliding window, with some differences:

- The calculation result consists of up to three line segments and up to two step records. These are serialized so that one line segment and one step record is output per clock cycle. Only the step records are used in subsequent calculations.
- In contrast to the previous kernels, the computation kernel of the step-length algorithm module has an internal state (besides pipeline stages delaying the output), which is required to determine the length of horizontal steps and to generate the `frame_info` signal.
- The step record FIFO does not contain an entry for every pixel. Instead, each entry has an index value to determine the horizontal position of the junction point for that step record (refer to Section 4.2.2 for details). Additionally, the FIFO was designed so that the first two entries can be read by the kernel simultaneously (one for the left and one for the right predecessor).

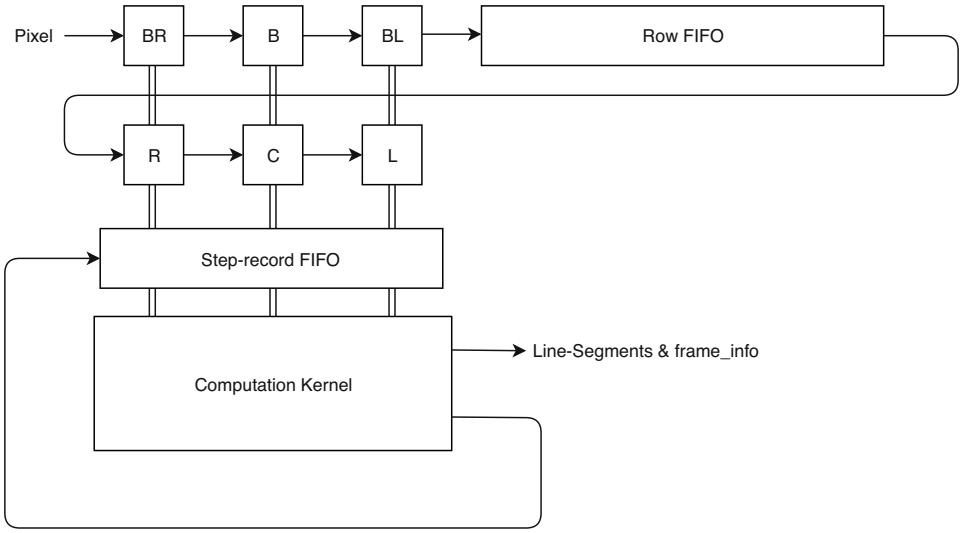


Figure 4.17: Architecture of the step-length algorithm module.

An overview of the structure of the computation kernel is given in Figure 4.18. The calculations for the four different cases discussed in Section 4.2.3 are done separately (denoted as `v_step`, `two_step`, `h_step` and `h_term`) . A 4-input multiplexer selects the result from the applicable case, which encompasses up to two step records and up to three line segments that can be generated in a single cycle, although not all modules are able to output these numbers of step records and line segments: `v_step` does not generate more than one step record per cycle (see Section 4.2.4 for details), whereas `h_term` as a special case can only generate a single line segment and no step records at all. Note that if a length threshold above one pixel is used, `v_step` can only generate up to two line segments that actually are above the threshold (although nonetheless three different signal paths are used, to prevent an increase in logic complexity). The same is true for `two_step` if the length threshold is larger than 2. The output signal `frame_info` encodes whether the current output belongs to an odd (if the signal is high) or even (if the signal is low) frame. A flip on this signal indicates the beginning of a new frame.

Length thresholding is performed on the line segments before the serialization, using three instances of the same thresholding unit in parallel. This order was chosen to reduce the load on the serializer and eliminate the danger of congestion.

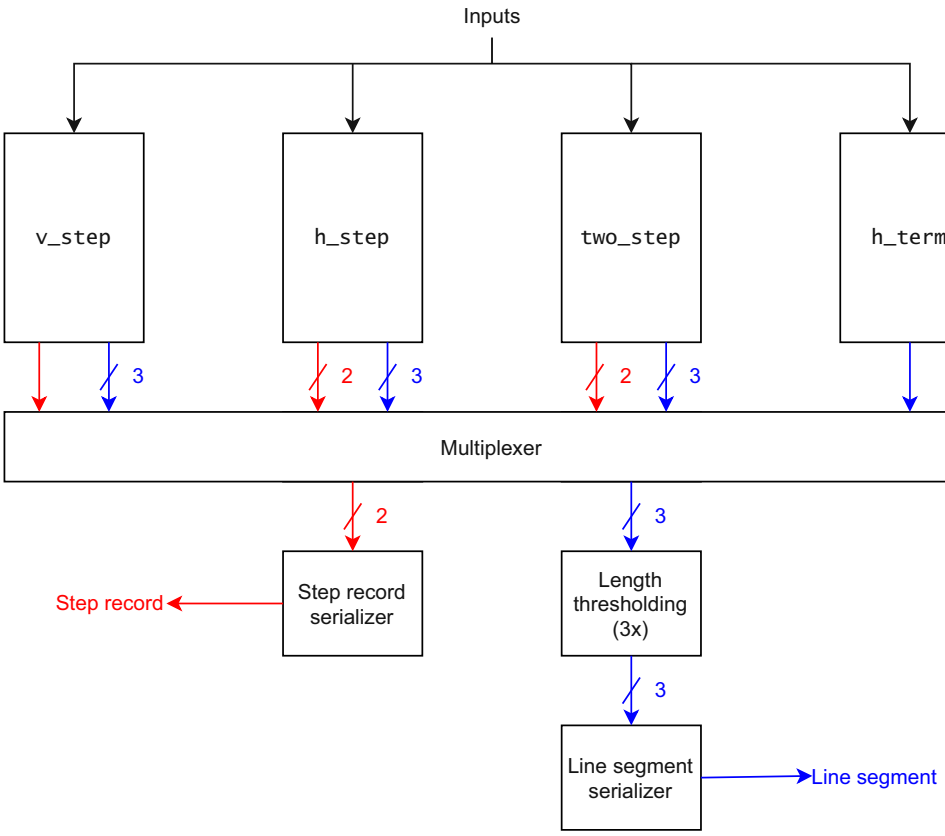


Figure 4.18: Architecture of the computation kernel. Blue arrows denote datapaths of line segments, while red arrows denote datapaths of step records. Not shown is the logic to generate the `frame_info` signal.

4.4.5.1 Pipeline delay for line segments

Like the previous modules, the step-length algorithm module needs $W+2$ cycles to fill the sliding window before the first pixel can be processed. Additionally, step records as well as line segments have a delay of their own. Due to the need for serializers, this delay is not constant.

The already multiplexed line segments are registered, which adds a cycle to the delay. The length thresholding requires an additional cycle, since intermediate results need to be registered to meet the timing constraints. The serializer requires at least two cycles. If three line segments are reported at the same time, the third one gets an additional delay of at least four cycles. These numbers apply when the serializer is idle at the time the line segments arrive. Line segments cannot be reported in consecutive cycles, except when the first one is reported when processing the last pixel of a row and the second one is reported when processing the first pixel of the next row. In all other cases, the serializer has a free cycle without any new inputs after line segments are reported, making it unlikely that a number of line segments large enough to lead to a noteworthy increase of the pipeline delay can accumulate in the FIFOs of the serializer. There is also a limit on the number of line segments that can realistically be reported within a certain number of cycles based on the selected length threshold. As mentioned earlier, a length threshold above one will result in `v_step` producing at most two line segments above the threshold per clock cycle. Similarly, a length threshold above two will result in `two_step` producing at most two line segments above the length threshold per clock cycle. Additionally, `h_step` can only produce a

valid third line segment if the step itself is longer than the length threshold, which will give the serializer enough cycles to empty its FIFOs in time beforehand (as `h_term` can only produce at most an additional line segment every second cycle while a horizontal step is being processed).

In any case, the delay is capped by the size of the FIFOs inside the serializer. If congestion would lead to a higher delay, the FIFOs would overflow instead, resulting in lost line segments. In this implementation, the maximum delay with regards to the FIFO dimensions would be 13.

Table 4.7: Pipeline delay of line segments

Processing stage	min. delay	max. delay	usual delay
Calculation & multiplexing	1	1	1
Length thresholding	1	1	1
Serialization	2	13	2-4
Total	4	15	4-6

4.4.5.2 Pipeline delay for step records

Step records have a pipeline delay of their own, and one needs to take into account that this puts limitations on the length of horizontal steps: If the length of a horizontal step approaches the row length, there is a limited number of cycles available to prepare the step record and feed it into the step record FIFO, so that its there in time for the bottom left successor of that line segment. Generally speaking, for a horizontal step of length L, the step record must be readily available in the FIFO within $W - L$ cycles. To enforce this rule, if $W - L$ grows smaller than the maximum pipeline delay of step records, the bottom-left successor is ignored during the calculation of step records and line segments. For this reason, the pipeline delay for step records has to be analyzed. The delay consists of two components: The delay from the step record serializer, and the delay from the step record FIFO. Note that unlike the line segment pipeline, the multiplexing result of step records is not registered. The reason is that the inputs of the serializer are registered (as opposed to the inputs of the length thresholding units, which are not registered).

The step record serializer is able to output one step record per cycle, either one that was generated during this cycle, or one that was buffered. Step records enter the buffer whenever two step records are generated in the same cycle, or when step records are generated while the buffer is not empty. The maximum delay of the serializer is composed of the base delay plus the maximum number of buffered step records.

We denote $s(n)$ as the number of step records in the buffer, $g(n)$ as the number of step records inserted into the buffer, and $e(n)$ as the number of step records being output/serialized when processing the pixel with index n. All of these variables are assumed to be in the domain of natural numbers (including 0).

The buffer state can be calculated according to the equation:

$$s(n) = s(n-1) - e(n-1) + g(n-1) \tag{4.8}$$

Step records can only be output if there are any present (either generated in the current cycle or available in the buffer):

$$e(n) = \begin{cases} 0 & | \quad s(n) = g(n) = 0 \\ 1 & | \quad s(n) + g(n) \geq 1 \end{cases} \tag{4.9}$$

The number of step records generated per cycle is limited to two:

$$g(n) \leq 2 \tag{4.10}$$

Furthermore, with the exception of the last pixel in each row, generating step records implies that in the following cycle, no step records will be generated:

$$g(n) > 0 \implies g(n+1) = 0 \quad | \quad n \bmod W \neq (W-1) \tag{4.11}$$

When processing the first and last pixel of each row, only one step record can be generated:

$$g(n) \leq 1 \quad | \quad n \bmod W = 0 \tag{4.12}$$

$$g(n) \leq 1 \quad | \quad n \bmod W = (W-1) \tag{4.13}$$

From Equations 4.10, 4.11, 4.12 and 4.13 follows:

$$g(n) + g(n+1) \leq 2 \tag{4.14}$$

Furthermore, we can assume the buffer to be empty at the first pixel:

$$s(0) = 0 \tag{4.15}$$

From Equations 4.8, 4.9 and 4.10 follows:

$$s(n) \leq s(n-1) + 1 \tag{4.16}$$

After applying Equation 4.15:

$$s(1) \leq 1 \tag{4.17}$$

We can now prove by induction that $s(n) \leq 1$ for all n, using Equations 4.15 and 4.17 as the base cases. The induction step Equation 4.18 can be proven using Table 4.8, which shows that if $s(n) \leq 1$, for any combination of $g(n)$ and $g(n+1)$ that satisfies Equation 4.14, $s(n+2) \leq 1$ holds. Applying the induction step to the two base cases, one can prove that $s(n) \leq 1$ holds for even and odd n respectively, and therefore for all n.

$$s(n) \leq 1 \implies s(n+2) \leq 1 \tag{4.18}$$

Table 4.8: Table to proof the induction step Equation 4.18. $s(n+2)$ is evaluated using Equation 4.8.

$s(n)$	$g(n)$	$g(n+1)$	$s(n+2)$
0	0	0	0
0	1	0	0
0	0	1	0
0	1	1	0
0	2	0	0
0	0	2	1
1	0	0	0
1	1	0	0
1	0	1	0
1	1	1	1
1	2	0	1
1	0	2	1

The base delay of the serializer is 2 clock cycles plus another cycle for every step record buffered. Since we have shown that the maximum occupancy of the buffer is one step record, the maximum delay of the step record serializer is 3 clock cycles.

The FIFO delay depends on the actual implementation of the FIFO, in particular the delay between a write operation and an update on the `empty`-flag of the FIFO. The synthesized BRAM-FIFO on our Xilinx Zynq- FPGA has no such delay (meaning the `empty`-flag will immediately be deasserted after a write operation). After the `empty`-flag has been updated, another clock cycle is needed to read the new step record from the FIFO, which requires another cycle.

As shown in Table 4.9, the total maximum pipeline delay for step records is 4 cycles. Therefore, the bottom-left successors of horizontal steps longer than $W-4$ will be ignored. Note that this check is also present in the software implementation, even though the software does not emulate a pipeline delay for step records. This was done to keep parity between both implementations.

Table 4.9: Pipeline delay of step records

Processing stage	**min. delay**	**max. delay**	**usual delay**
Calculation & multiplexing	0	0	0
Serialization	2	3	2-3
FIFO `empty`-flag update latency	0	0	0
FIFO read operation latency	1	1	1
Total	3	4	3-4

4.4.6 AXI

To be able to connect the modules to other logic, soft- or hard-core CPUs, two AXI4-Stream interfaces were implemented to handle input and output.

One interface serves as a slave and is used to deliver the input stream of 8 bit unsigned magnitude values to the Gaussian module. The stream uses a data width of 4 bytes with little-endian byte-order. Therefore, on every transaction four intensity values are delivered. A FIFO is used to buffer up 16 of such 4-byte words. The Gaussian module processes one pixel magnitude value

per clock cycle, which means every four clock cycles, an entry in the FIFO will be consumed. In case the FIFO is full, the `TREADY`-signal of the AXI slave interface will be deasserted.

The other interface is configured as a master and used to transfer line segments. The stream uses data words with a width of 8 bytes and each word is used to encode one line segment. Figure 4.19 shows how line segments are encoded in the AXI data words. The data format supports image widths of up to 65,536 pixels and image heights of up to 32,768.

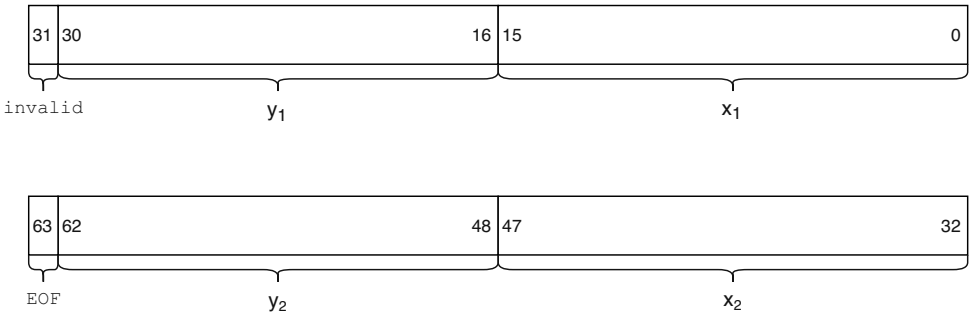


Figure 4.19: Data format of the AXI4-Stream master interface

An asserted `EOF`-flag indicates that the processing of a frame has concluded, which means all line segments to follow will belong to the next frame. Note that in many cases it is not known whether a line segment is the last of a frame. For this reason, an 'empty' word with asserted `invalid`- and `EOF`-signals can be sent to signal the end of the frame, without actually encoding a line segment.

Line segments are grouped into packets of 16 words. A transfer is initiated once at least 16 line segments are ready in the FIFO, or preemptively in case the frame switches.

4.5 Results

In this Section, we will discuss the results of the proposed step-length algorithm, including the performance of the hardware implementation as well as the detection results. In particular, we will compare the results to the Modified LSD by Zhou et al. [ZCW18] (Section 2.5) since it is the only recent work on the topic and shares a lot of similarities to the proposed algorithm, like the internal usage of Canny's algorithm and the pipelined, framebuffer-free design, making them both comparable in many regards.

4.5.1 Specifications

The hardware implementation was tested on a XC7Z015 FPGA from the Xilinx Zynq-7000 series. Grayscale input images up to a resolution of 1080p (1920×1080) are supported, with a depth of 8 bits per pixel. The design does allow for lower resolutions, although the resolution cannot be changed at run-time and requires resynthesis of the design. The synthesized logic supports clock frequencies up to 101.6 MHz. However, due to the limited precision of the FPGA's phase-locked loop (PLL), the design runs at 100 MHz. The total power consumption of our reference design is 1.744 W (Table 4.10), with 3.7% (64 mW) of that related to the step-length algorithm.

Table 4.10: Power Analysis

Component	Power Consumption [W]	Relative Power
Static Power (Device)	0.118	6.8%
Processing System[1]	1.536	88%
Step-length algorithm	0.064	3.7%
DMA & Interconnect	0.026	1.5%
Total power consumption	**1.744**	**100%**

For comparison, the Modified LSD algorithm [ZCW18] was implemented on a XC7Z020 FPGA platform with a resolution of 640×480 pixels. It also runs at 100 MHz.

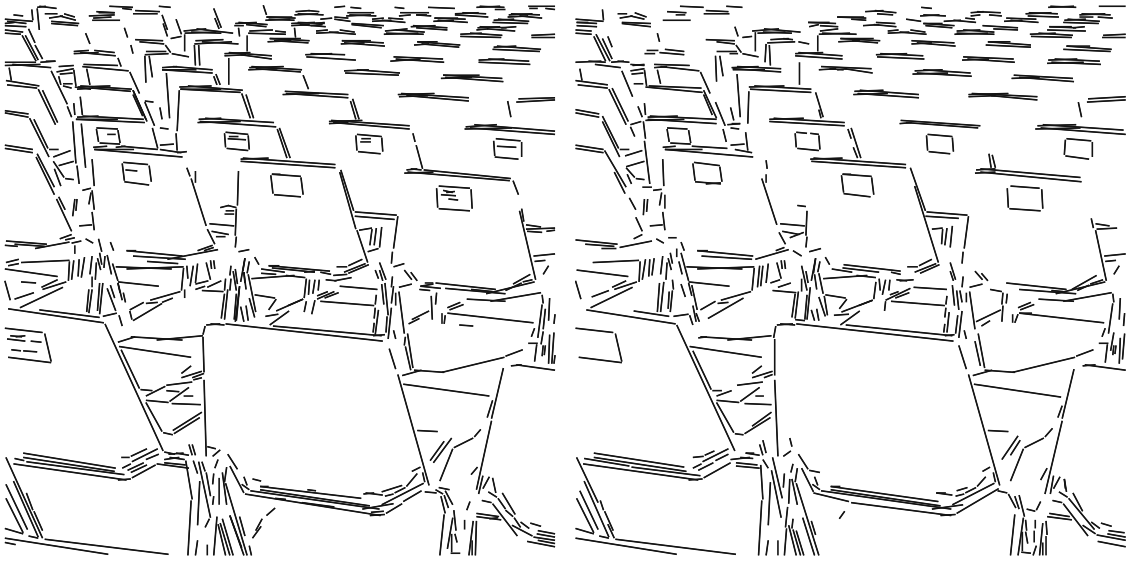
4.5.2 Processing delay

Table 4.11 provides a list of the maximum delays for each module. This shows that the total processing delay is at most $5W + 27$ clock cycles, where W is the image width. This is the number of cycles between receiving the last pixel of an image on the input and sending the last line segment on the output. In comparison, the Modified LSD algorithm [ZCW18] has a delay of $7W + 21$. Comparing both algorithms, a delay of W is saved by not having to mirror lines for the step-length algorithm. Another W is saved by using a 3×3 Gaussian filter in favor of a 5×5 filter. While the usage of the 5×5 filter does slightly improve the quality of the results (Fig. 4.20), there is a trade-off between maximizing the detection quality and minimizing the processing delay. Given the negligible difference, as seen in Figure 4.20, we opted to go for the latter. However, even if we were to employ a 5×5 filter, our solution would still outperform the Modified LSD regarding the processing delay. Table 4.12 lists throughput and delay for various image resolution standards.

Table 4.11: Processing delay of the step-length algorithm, in clock cycles. W is the image width.

Module name	max. Delay (cycles)
Gaussian filtering	$W + 2$
Gradient calculation	$W + 3$
Non-maximum suppression	$W + 2$
Double thresholding	$W + 2$
Step-length algorithm (sliding window)	$W + 2$
Step-length algorithm (kernel)	15
Duplicate removal	1
Total	$5W + 27$

[1] Part of the Zynq that contains hard ARM cores as well as various I/O logic



(a) 3×3 Gaussian filter (Eq. 4.5) **(b)** 5×5 Gaussian filter (Eq. 2.1)

Figure 4.20: Line segment detection results, using different Gaussian filters

Table 4.12: Throughput and processing delays at various resolutions (assuming clock speed of 100 MHz)

Resolution	Throughput	Processing delay
512×512	381.5 fps	$25.87\mu s$
640×480	325.5 fps	$32.27\mu s$
1024×768	127.2 fps	$51.47\mu s$
1920×1080	48.23 fps	$96.27\mu s$

4.5.3 FPGA Resource usage

A summary of the resource utilization is shown in Table 4.13. A comparison between the resource usage of the step-length algorithm and the Modified LSD algorithm is shown in Table 4.14. Note that while different FPGAs are used in our and Zhou's work [ZCW18], XC7Z015 and XC7Z020 are from the same series and mainly differ in the number of available resources, rather than architecture. Therefore, a meaningful comparison between the usage numbers of the two solutions can be made.

Table 4.13: Resource utilization of the step-length algorithm on the XC7Z015

Resource Category	Usage	Available	Utilization (in %)
Slice LUTs	3948	46200	8.55%
Slice registers	3337	92400	3.61%
DSPs	0	160	0%
36K BRAM Tiles	7.5[2]	95	7.89%

[2]Each tile consists of two 18 kbit blocks (*half-tiles*) that can be used independently

Table 4.14: Comparison of resource usage between our solution and the Modified LSD algorithm [ZCW18]

Resource Category	this work	[ZCW18]	Reduction (abs.)	Reduction (rel.)
Slice LUTs (as Logic)	3948	4437	489	11%
Slice LUTs (as Memory)	0	378	378	100%
Slice registers (as Flip-flops)	3337	4810	1473	30.6%
DSPs	0	8	8	100%
36K BRAM Tiles	7.5	17.5	10	57.14%

Compared to the Modified LSD algorithm, the step-length algorithm requires a slightly smaller number of slice LUTs, and significantly fewer slice registers. Additionally, since the step-length algorithm does not need multiplication operations for the Gaussian filtering or gradient direction calculation, no digital signal processor (DSP) blocks were synthesized in our design.

The most surprising improvement regards a cut in the usage of BRAM by more than half, even though our design operates at a much higher resolution, with larger row FIFOs required. Upon closer examination, the following reasons were found:

- The Modified LSD algorithm requires two additional FIFOs for the 5×5 Gaussian filter, as well as an additional FIFO for row mirroring.
- The FIFO depths of row FIFOs dependent on the image width. However, only FIFO depths that are a power of two are supported. Therefore, for a 640×480 image, the Modified LSD requires FIFOs with a depth of 1024, while our 1920×1080 image requires FIFOs with a depth of 2048. Therefore, even though our image has three times the width, a FIFO with only twice the depth is sufficient.
- Regardless of the depth, each FIFO always uses at least half of a BRAM tile, which is 18 kbit in size. For the Modified LSD algorithm, which uses FIFOs with a depth of 1024, the half-tiles are used to their full capacity if the word size is a multiple of 18 bits. For the step-length algorithm, the FIFOs of depth 2048 require word sizes that are multiples of 9 bit, to use the 18 kbit half-tiles optimally. This means that there is a lot less potential overhead in our solution.

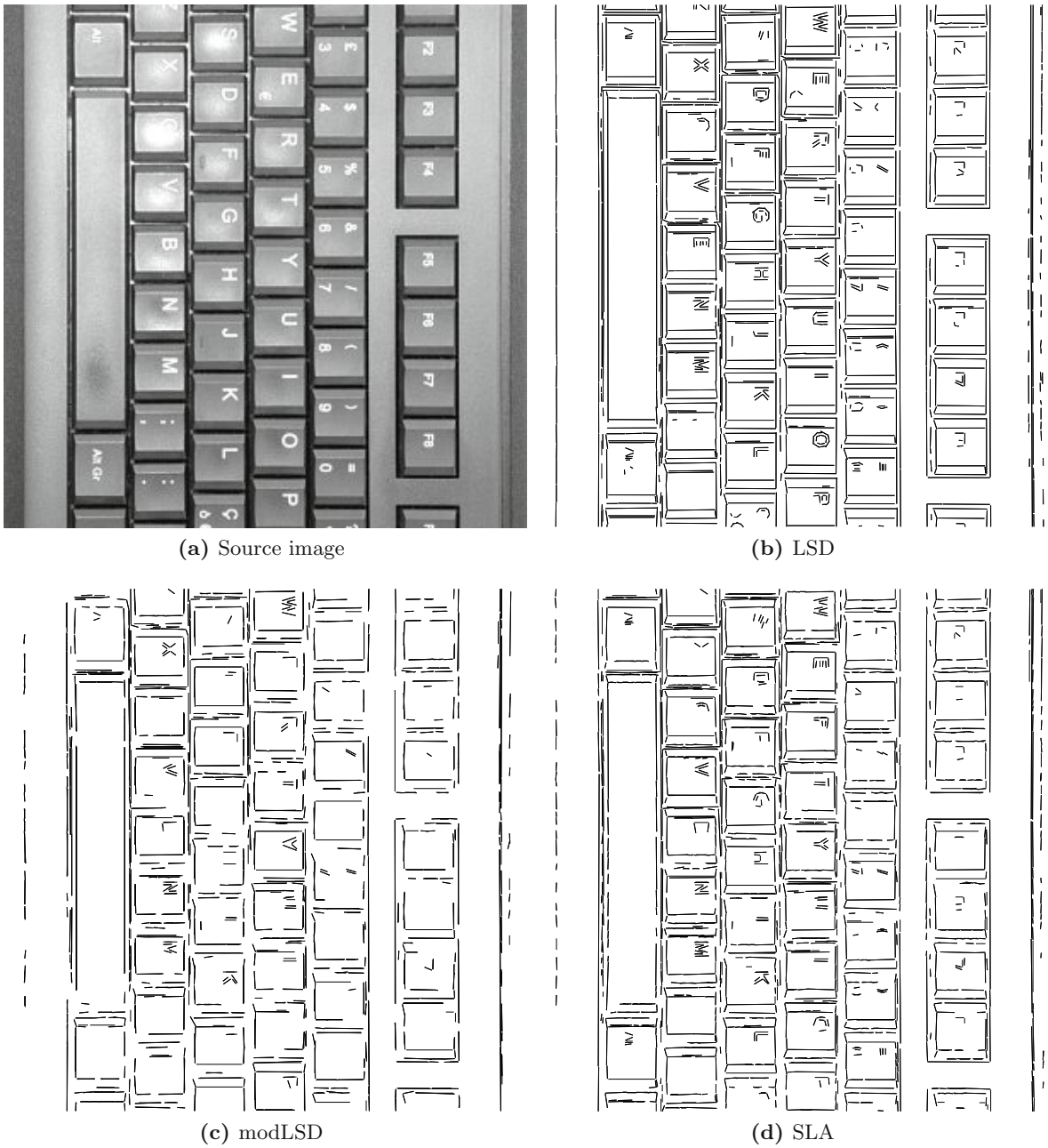
4.5.4 Detection result comparison

This Section will be divided into three parts: First, we will examine, compare, and discuss the most interesting detection results from the LSD algorithm by von Gioi et al., the Modified LSD algorithm by Zhou et al. [ZCW18] as well as our work from the *TESTIMAGES* sampling set[3]. Secondly, we introduce a method to evaluate and compare the detection quality of our work and the Modified LSD algorithm. Thirdly, we evaluate the impact of parameter selection (length threshold) using the same metric from the previous step.

Part I: Detection analysis

In the first part, we will look at the detection result of three different images from our test image set, using the LSD algorithm, the Modified LSD algorithm and our work. We selected these three images because they stood out among the rest.

[3] `https://testimages.org/sampling/` (Accessed 2020-08-12)



(a) Source image **(b)** LSD

(c) modLSD **(d)** SLA

Figure 4.21: Comparison of results - Test Image # 23

Test image # 23 shows a computer keyboard, rotated by an 90° angle clockwise. This image has a lot of straight features of various lengths, most of them horizontally or vertically aligned. All three algorithms are able to detect the outline of the keys, although they are most clearly defined in the detection results from LSD, and least clearly in the results from the Modified LSD algorithm. Vertical lines, unless geometrically perfect (e.g., computer generated), pose a challenge to our work and the Modified LSD algorithm - if the edge is detected just one pixel off from the rest, the line segment will be segmented at that point. If these interruptions happen too frequently, the length threshold in our work, or the region size threshold in the Modified LSD

algorithm will not be met and no line segment will be detected in this spot, which is especially apparent regarding the left-most vertical line segment that corresponds to left border between keyboard and background. However, even the LSD algorithm is subject to a similar issue, as can be seen on the right-most vertical line that is actually expected to span the entire height of the image, but actually does not in any of the results.
While it would be theoretically possible to make the step-length algorithm more tolerant with regard to vertical line segment detection, it would require a more sophisticated step-linking logic and more information to be stored in step records, breaking our throughput goals and resource constraints of the reference hardware. Solving this issue in the Modified LSD algorithm is even more difficult. Edges are only allowed to grow either from top-left to bottom-right or top-right to bottom-left for the detection of a line segment to occur, but not both in an alternating fashion, as both directions are processed independently in two separate module. For this reason, the issue cannot be fixed in the Modified LSD algorithm without a major redesign.

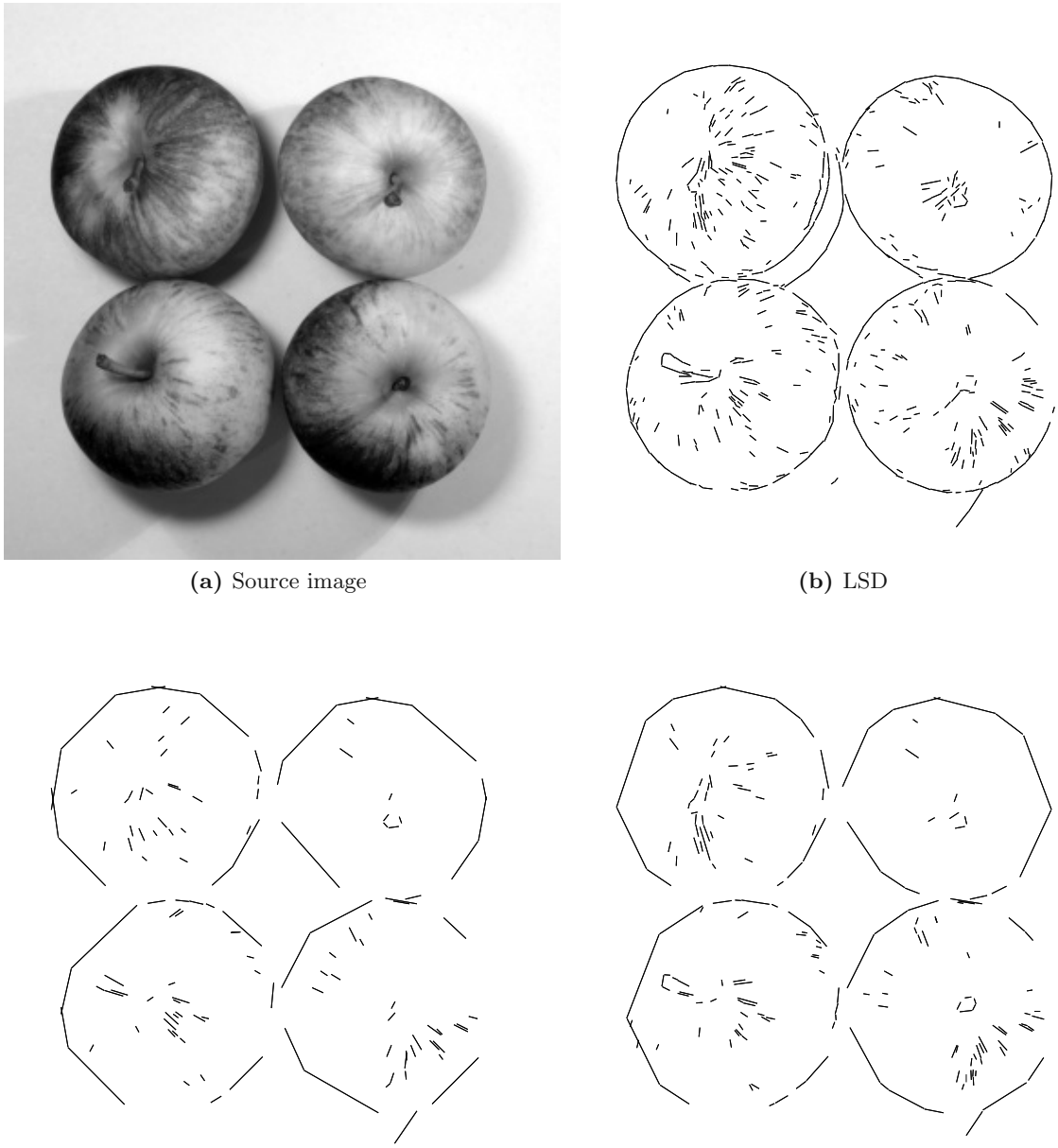
Test image # 2 shows four apples. This image does not have a lot of straight features, but many curved structures. All three algorithms approximate arcs using line segments. However, the Modified LSD algorithm and the step-length algorithm do so in a more crude fashion than the LSD algorithm, using fewer but longer line segments. This is caused by the smaller angle tolerance used in the LSD algorithm compared to the other two implementations.
Decreasing the angle tolerance in the step-length algorithm is only possible by revising the step-linking logic. However, this might negatively impact the overall detection sensitivity regarding the tolerance towards quantization errors. For the Modified LSD algorithm, Zhou [ZCW18] suggests using the region size threshold parameter as a countermeasure to this effect. However, this can only be done if the parameters of the arcs occuring in the image are known in advance, and it has effects on the global detection quality.

Test image # 34 shows a close-up of a pattern on a piece of clothing. The step-length algorithm detects more line segments than the other two. It detects a lot of horizontal ones caused by lines of thread in the bright sections of the fabric. Meanwhile, LSD almost exclusively detects the vertical borders of the stripes and horizontal borders between the different patterns. The Modified LSD algorithm struggles the most on this image, as it only detects seemingly random line segments aside from a few horizontal ones in the lower left image section that mark the pattern borders.
One explanation for the different results might be the differences in smoothing: While LSD uses Gaussian Scaling, the Modified LSD and step-length algorithms use 5×5 and 3×3 Gaussian filters, respectively. Using different blurring methods might have a different effect on the more delicate structures in the image, i.e., keeping them intact with the 3×3 filter but smoothing them out with the other two methods.

Part II: Detection quality - metric and comparison

To assess the quality of the step-length algorithm's detection, our approach was to use the *a contrario* validation of the LSD algorithm (Section 2.2.9) to check how many line segments produced by our work and the Modified LSD algorithm [ZCW18] can be verified respectively, and compare those results. This was done using the software implementations of both algorithms to process our TESTIMAGES set at a resolution of 1200×1200.

After processing a test image with the step-length algorithm or Modified LSD algorithm, Gaussian scaling and gradient calculation are performed exactly as in the LSD algorithm (Sections 2.2.2 and 2.2.3) on the same test image. Then, for each line segment detected by the step-length



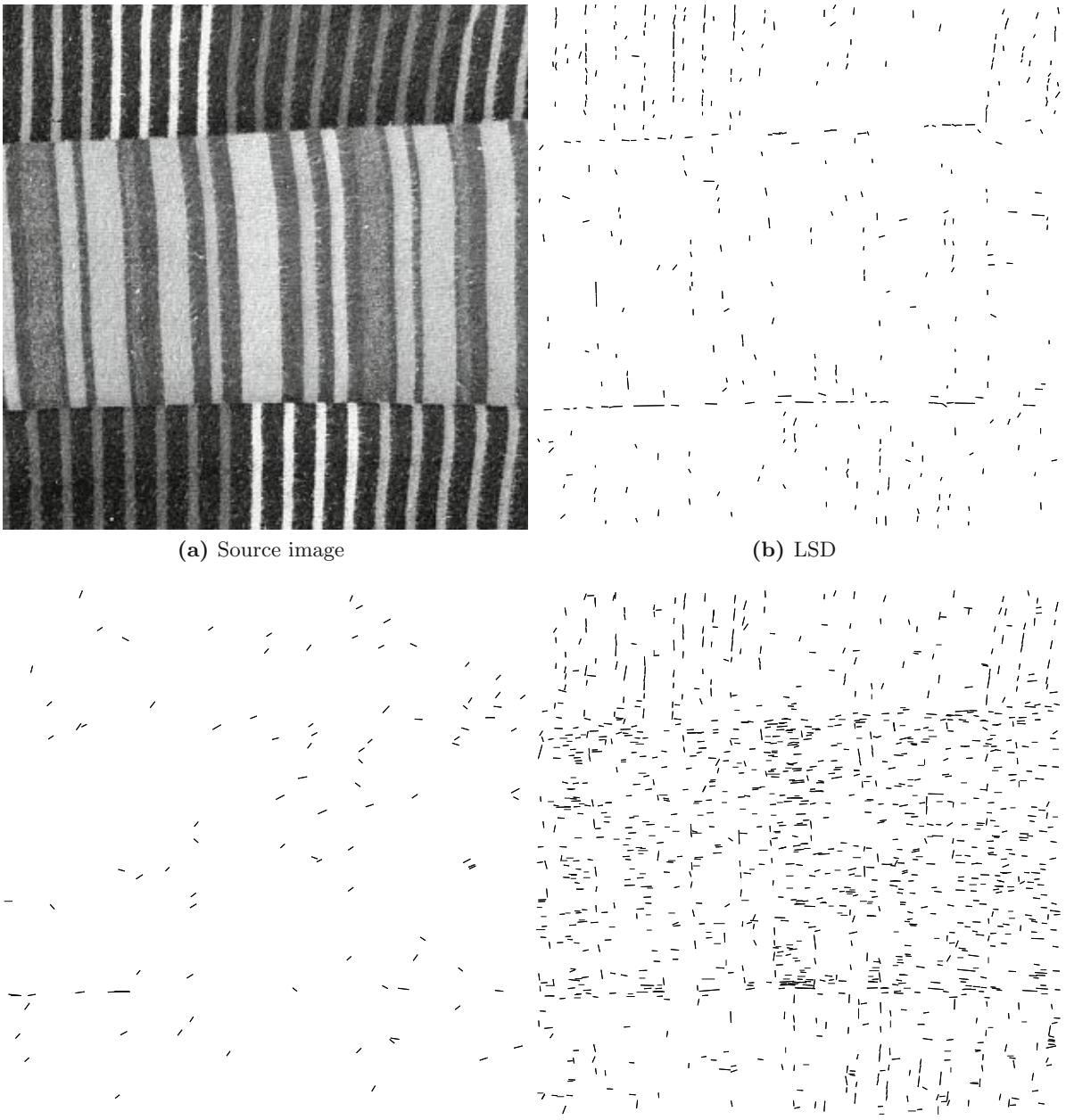
(a) Source image **(b)** LSD

(c) modLSD **(d)** SLA

Figure 4.22: Comparison of results - Test Image # 2

algorithm and Modified LSD on the respective test image, the algorithm tries to verify them on the generated gradient map using the 11 different values for p as in the LSD algorithm, with rectangle widths in the range of $W \in [0.5, W_{max}]$ using 0.5 pixel increments. W_{max} denotes the maximum width of a region in the LSD algorithm, assuming the intensity change is maximized and all pixels along the width are above the gradient threshold ρ. It can be calculated using Equation 4.19.

$$W_{max} = \frac{DR_{max}}{\rho} \tag{4.19}$$



(a) Source image **(b)** LSD **(c)** modLSD **(d)** SLA

Figure 4.23: Comparison of results - Test Image #34

In this equation, DR_{max} is the maximum dynamic range of the image format (255 assuming 8 bit intensity depth) and ρ is as defined in Equation 2.8. Afterwards, the verification is performed a second time for each line segment, with the endpoints swapped. In the step-length algorithm and Modified LSD, the indexing of the endpoints of the line segment is interchangeable - in the step-length algorithm, the first endpoint is defined by `sx` and `sy` in the step record, and the second endpoint is the junction point of the last step in the chain. In LSD, the indexing of the endpoints depends on which side of the line segment the higher/lower intensity areas are located (i.e., when processing the negative of an image, all detected line segments would have their endpoint indices

swapped). The smallest NFA value encountered is checked against ε. If it is smaller than or equal to ε, the line segment is denoted as *verified*.

Our proposed metric uses two values for comparing algorithms. The first value is a metric for the number of detections, or sensitivity of the algorithm: The *total length* d_{all} is defined as the sum of the Euclidean lengths of all line segments detected. The total length is used in favor of using the number of detected line segments, as fragmentation (as discussed during the analysis of test image 23) can easily inflate the number of line segments, making a comparison harder. The *total length* has to be compared on a per-image basis. Equation 4.20 shows the formula for the total length, where LS_{all} is the set of all detected line segments in an image, and $p_1(i) = (x_1(i), y_1(i))$ and $p_2(i) = (x_2(i), y_2(i))$ are the endpoints of the line segment with index i.

$$d_{all} = \sum_{i}^{LS_{all}} \sqrt{|x_1(i) - x_2(i)|^2 + |y_1(i) - y_2(i)|^2} \tag{4.20}$$

The second value is a metric for noise tolerance and detection quality: The *verification rate* q_v is the sum of the Euclidean lengths of all *verified* line segments, divided by the total length. Equation 4.21 shows the formula for the *verification rate*, where $LS_{verified}$ is the set of *verified* line segments in an image.

$$q_v = \frac{\sum_{i}^{LS_{verified}} \sqrt{|x_1(i) - x_2(i)|^2 + |y_1(i) - y_2(i)|^2}}{d_{all}} \tag{4.21}$$

The results are highly dependent on the selected parameters, mainly the region size threshold in the Modified LSD algorithm, and the length threshold in the step-length algorithm.

The region size threshold for the Modified LSD algorithm has been set to 19, following the recommended methodology of Zhou et al. [ZCW18]. This number is based directly on the NFA value of the LSD algorithm: Any line segments with a region size smaller than 19, given the resolution of the test image set we use, will automatically fail the line segment verification, which is the reason a region size threshold of 19 is selected as the minimum.

Length thresholding as performed in the step-length algorithm works on a different principle. For this reason, it was difficult to select the parameter in a way to produce comparable results regarding both the total length and verification rate, as the latter is per definition dependent on the former.
To solve this issue, our approach was to select the length threshold so that the *total length* values for each test image are on a close level for both the step-length algorithm and the Modified LSD algorithm, and then compare resulting *verification rates*. We found that a length threshold of 17 is a good value for a comparison, as the average ratio between the total length values of both algorithms is close to one. However, some extremes emerged as a result of this test. Test cases 31 and 34 in particular have shown great deviations regarding the *total lengths*.

The results of the verification experiment are shown in Tables 4.15, 4.16, 4.17, 4.18, 4.19.

Table 4.15: Line segment evaluation - Test Images #1 to #8

Test image #	1	2	3	4	5	6	7	8
d_{all} **(SLA)**	45088	5843	21144	7213	11696	14623	60052	68910
d_{all} **(modLSD)**	46894	6010	20948	7497	12008	14833	66140	69693
q_v **(SLA)**	0.97	0.87	0.99	0.99	0.99	1.00	0.99	0.97
q_v **(modLSD)**	0.97	0.88	0.99	0.75	0.96	0.98	0.99	0.97

Table 4.16: Line segment evaluation- Test Images #9 to #16

Test image #	9	10	11	12	13	14	15	16
d_{all} **(SLA)**	28844	15232	12503	99747	38889	9422	10210	36314
d_{all} **(modLSD)**	30384	15316	11650	102142	39062	9835	10421	47798
q_v **(SLA)**	0.97	0.98	1.00	1.00	0.93	0.99	0.94	0.99
q_v **(modLSD)**	0.96	0.98	0.97	1.00	0.92	0.99	0.96	0.99

Table 4.17: Line segment evaluation - Test Images #17 to #24

Test image #	17	18	19	20	21	22	23	24
d_{all} **(SLA)**	22522	20914	18258	36859	27157	54172	52067	30238
d_{all} **(modLSD)**	27128	18700	14248	48649	26891	58324	66813	28207
q_v **(SLA)**	0.92	0.95	0.98	1.00	0.94	1.00	1.00	0.98
q_v **(modLSD)**	0.90	0.91	0.99	1.00	0.95	1.00	1.00	0.98

Table 4.18: Line segment evaluation - Test Images #25 to #32

Test image #	25	26	27	28	29	30	31	32
d_{all} **(SLA)**	25963	58369	45682	22160	6614	98403	10532	21987
d_{all} **(modLSD)**	32614	86007	67783	24185	5878	105363	37860	24246
q_v **(SLA)**	0.98	1.00	1.00	0.95	0.99	1.00	0.86	0.95
q_v **(modLSD)**	0.99	1.00	1.00	0.94	0.98	1.00	0.88	0.96

Table 4.19: Line segment evaluation - Test Images #33 to #40

Test image #	33	34	35	36	37	38	39	40
d_{all} **(SLA)**	11967	6894	18941	14167	13746	44088	16114	15335
d_{all} **(modLSD)**	12728	1677	21460	14509	13975	53222	17406	15780
q_v **(SLA)**	0.93	0.37	0.98	1.00	0.98	0.98	0.95	0.95
q_v **(modLSD)**	0.95	0.23	0.97	1.00	0.97	0.97	0.95	0.96

The results show that using these parameters (17 as the length threshold in the step-length algorithm and 19 as the region size threshold in the Modified LSD algorithm), the resulting *total length* is, on average, 0.39% higher using the step-length algorithm, and has a slightly higher average verification rate compared to the Modified LSD algorithm (95.51% vs. 94.36%). With these values, we can conclude that both algorithms have shown a very comparable detection quality in our experiment.

One result that needs further discussion is test image 34 with mediocre verification rates on both algorithms. As we have seen in the previous part, the results on this test image are vastly different. Our current theory as to why the verification rate is so low lies in the differences of the smoothing/scaling methods used by the three algorithms. The fine structures of the thread lines cause differences in the gradient map when different smoothing algorithms are used, thus causing the verification using the gradient map extracted by a different algorithm to fail.

Part III: Relationship between the length threshold parameter and the detection quality

The metric defined in the previous part can also be used to determine the influence the selected length threshold has on the detection quality of the step-length algorithm. To do so, we will look at the *total lengths* and *verification rates* over the whole test image set, using different length thresholds values. We limited ourselves to this range, as a higher threshold will cause some test images to have zero line segments detected, making the evaluation results less meaningful.

Averaging over the total lengths in our test image set has the undesired effect of giving images with a higher total length more weight on the aggregated result. For this reason, we introduce the *relative length* d_{rel}, which we define as the total length divided by the total length when evaluating the same image using the smallest length threshold value in the experiment. We selected this length threshold value to be 1, as this is the smallest length a meaningful line segment can have - we consider line segments of length 0 as not meaningful, since they do not have any impact on d_{all} and q_v. Assuming the *total length* using the threshold c is $d_{all}(c)$, we can calculate the *relative length* $d_{rel}(c)$ for that threshold using Equation 4.22. Also, we set the upper bound of the length threshold in our experiment to 26, as higher values will cause some image in the set to have zero line segments, making the experimentation results less meaningful.

$$d_{rel}(c) = \frac{d_{all}(c)}{d_{all}(1)} \tag{4.22}$$

Figure 4.24 plots the average *relative lengths* and *verification rates* over our test image set for length thresholds in the range $[1, 26]$, using 0.05 intervals. At a threshold of one, the verification was at the minimum of 0.66, while the maximum verification rate was witnessed at length threshold 25.3, being 0.97. The overall trend of the verification rate is rising with the length threshold, although the increments become smaller and smaller. This is caused by the design of the NFA calculation, where short line segments are less likely or even impossible to verify, causing the verification rate to rise when small line segments are removed. At higher threshold-levels nearly all line segments are verifiable, thus causing the changes to become smaller and smaller.

In contrast, the relative length is monotonically falling with a rising length threshold. This is to be expected, as increasing the threshold will remove a number of line segments from the output, therefore decreasing the total length. However, it can be noticed that this rate slows down slightly at higher levels, implying that the number of line segments removed for each threshold increasing step drops.

In conclusion, this experiment has shown that the verification rate can only be increased at the cost of total length. However, the verification rate is relatively high for a broad range of length threshold values. This is useful, as it means the length threshold can be optimized for the application the algorithm is used for, without worrying too much about the noise tolerance, as

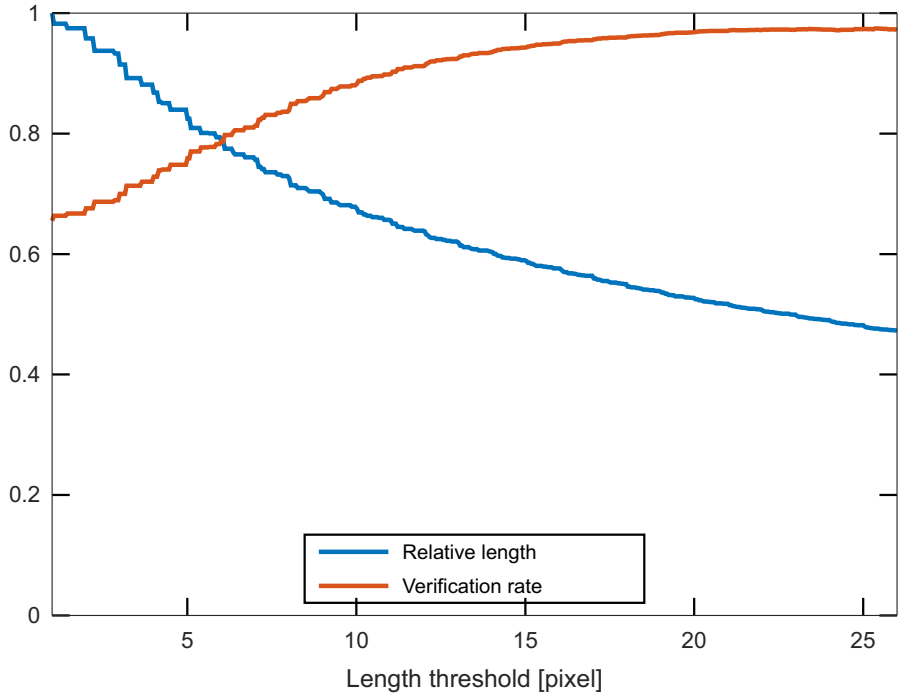


Figure 4.24: Average relative lengths and verification rates across our test image set, for different length thresholds

long as the selection is not too small. For this purpose, we recommend a length threshold of at least 8. At the same time it can be seen that the verification rate reaches a plateau around a length threshold of 20, where a further increase of the threshold will only induce a minor gain in the verification rate, while the relative length keeps falling at a more consistent rate over the range in our experiments. At this point, we discourage further increasing the length threshold.

5 Conclusion

Line segment detection is an important preprocessing step in Computer Vision applications. In this diploma thesis, various existing techniques to solve this problem were discussed and two novel methods to improve on existing solutions for line segment detection were proposed.

An optimization of the LSD algorithm [GJMR12] allows for a speedup in the line segment verification step using a LUT. In software, we could show that this algorithm decreases the processing time of the algorithm on our test image set by 13.08% on average and up to 28.8% in one test case, while the detection quality remains unchanged. Although no hardware accelerator using the LSD algorithm was developed in the scope of this work, replacing involved math operations by a LUT will be helpful for such future projects.

Furthermore, a novel line segment detection algorithm, called step-length algorithm, was proposed, designed for implementation on FPGAs. The algorithm encompasses edge detection based on Canny's algorithm[Can86] and uses an detection method based on an aliasing effect present in rasterized images, called the staircase effect. The algorithm was implemented on a Xilinx XC7Z015 FPGA. On this chip, the algorithm was able to run at a clock rate of 100 MHz and supports image resolutions of up to 1920×1080 using less than 10% of the available resources and drawing 64 mW of power. Compared to the Modified LSD algorithm [ZCW18], which was published by Zhou et al. in 2018, we could show that in comparison the step-length algorithm uses less FPGA resources and has a smaller latency.

Furthermore, a new metric to measure the detection quality of line segment detection algorithms was proposed. Applying this metric, both the step-length algorithm and the Modified LSD algorithm are shown to have a very similar detection quality. The metric shows that the total length of detected line-segments is 0.39% higher in the proposed algorithm, with a higher portion of the line-segments being verifiable (95.51% vs. 94.36%, weighted by length).

Literature

[ACA14] Abdallah, A. ; Cardarelli, R. ; Aeilli, G.: On a fast discrete straight line segment detection. In: *The 2nd International Conference on Intelligent Systems and Image Processing*, 2014, S. 89–94

[AFAC17] Abdallah, A. ; Felici, D. ; Aielli, G. ; Cardarelli, R.: FPGA implementation of resistor network for fast segment line detector. In: *2017 29th International Conference on Microelectronics (ICM)*, 2017, S. 1–4

[AG13] Asuni, N. ; Giachetti, A.: TESTIMAGES: A Large Data Archive For Display and Algorithm Testing. In: *Journal of Graphics Tools* 17 (2013), 10, S. 113–125

[Asu] Asuni, N. *TESTIMAGES/SAMPLING*. URL: `https://testimages.org/sampling` (Accessed: 10-23-2020)

[AT11] Akinlar, C. ; Topal, C.: Edlines: Real-time line segment detection by Edge Drawing (ed). In: Macq, Benoît (Hrsg.) ; Schelkens, Peter (Hrsg.): *ICIP*, IEEE, 2011. – ISBN 978–1–4577–1304–0, S. 2837–2840

[AT18] Akinlar, C. ; Topal, C.: `https://github.com/CihanTopal/ED_Lib`. August 2018. – Accessed: 2019-03-21

[ATQE17] Almazan, E. J. ; Tal, R. ; Qian, Y. ; Elder, J. H.: MCMLSD: A Dynamic Programming Approach to Line Segment Detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017

[BAS18] Budak, U. ; Alcin, O. F. ; Sengur, A.: Automatic Airport Detection with Line Segment Detector and Histogram of Oriented Gradients from Satellite Images. In: *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*, 2018, S. 1–5

[Bre65] Bresenham, J. E.: Algorithm for computer control of a digital plotter. In: *IBM Systems Journal* 4 (1965), Nr. 1, S. 25–30. – ISSN 0018–8670

[CA16] Creusot, C. ; A., Munawar: Low-computation egocentric barcode detector for the blind. In: *2016 IEEE International Conference on Image Processing (ICIP)*, 2016, S. 2856–2860

[Can86] Canny, J.: A Computational Approach to Edge Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986), Nov, Nr. 6, S. 679–698. – ISSN 0162–8828

[DH72] Duda, R. O. ; Hart, P. E.: Use of the Hough Transformation to Detect Lines and Curves in Pictures. In: *Commun. ACM* 15 (1972), Januar, Nr. 1, S. 11–15. – ISSN 0001–0782

[DMM00] Desolneux, A. ; Moisan, L. ; Morel, J.: Meaningful Alignments. In: *Int. J. Comput. Vision* 40 (2000), Nr. 1, S. 7–23. – ISSN 0920–5691

[DMM08] Delsolneux, A. ; Moisan, L. ; Morel, J.: *From Gestalt Theory to Image Analysis: A Probabilistic Approach*. Bd. 34. 2008

[EMAE20] El Hajjouji, I. ; Mars, S. ; Asrih, Z. ; El Mourabit, A.: A novel FPGA implementation of Hough Transform for straight lane detection. In: *Engineering Science and Technology, an International Journal* 23 (2020), Nr. 2, S. 274–280. – ISSN 2215–0986

[GAZ+17] Guan, J. ; An, F. ; Zhang, X. ; Chen, L. ; Mattausch, H. J.: Real-Time Straight-Line Detection for XGA-Size Videos by Hough Transform with Parallelized Voting Procedures. In: *Sensors* 17 (2017), Nr. 2. – ISSN 1424–8220

[Gio14] von Gioi, R. G.: *A Contrario Line Segment Detection*. Springer Publishing Company, Incorporated, 2014. – ISBN 1493905740, 9781493905744

[GJMR12] von Gioi, R. G. ; Jakubowicz, J. ; Morel, J. ; Randall, G.: LSD: a Line Segment Detector. In: *Image Processing On Line* 2 (2012), S. 35–55. – `https://doi.org/10.5201/ipol.2012.gjmr-lsd`

[Hou62] Hough, P. V. C.: *Method and means for recognizing complex patterns*. December 1962. – U.S. Patent 3,069,654

[HSB+20] Hagara, M. ; Stojanovic, R. ; Bagala, T. ; Kubinec, P. ; Ondracek, O.: Grayscale image formats for edge detection and for its FPGA implementation. In: *Microprocessors and Microsystems* 75 (2020), 02, S. 103056

[HWZ+18] Huang, K. ; Wang, Y. ; Zhou, Z. ; Ding, T. ; Gao, S. ; Ma, Y.: Learning to Parse Wireframes in Images of Man-Made Environments. In: *CVPR*, 2018

[KJT+08] Kim, D. ; Jin, S. H. ; Thuy, N. T. ; Kim, K. H. ; Jeon, J. W.: A real-time finite line detection system based on FPGA. In: *2008 6th IEEE International Conference on Industrial Informatics*, 2008. – ISSN 1935–4576, S. 655–660

[LAGT20] Liu, C. ; Abergel, R. ; Gousseau, Y. ; Tupin, F.: LSDSAR, a Markovian a contrario framework for line segment detection in SAR images. In: *Pattern Recognition* 98 (2020), S. 107034. – ISSN 0031–3203

[LCC+18] Liu, N. ; Cui, Z. ; Cao, Z. ; Pi, Y. ; Dang, S.: Airport Detection in Large-Scale SAR Images via Line Segment Grouping and Saliency Analysis. In: *IEEE Geoscience and Remote Sensing Letters* 15 (2018), Nr. 3, S. 434–438

[LGL16] Lin, S. ; Garratt, M. ; Lambert, A.: Monocular vision-based real-time target recognition and tracking for autonomously landing an UAV in a cluttered shipboard environment. In: *Autonomous Robots* 41 (2016), 04

[LLZZ18] Liu, S. ; Lu, L. ; Zhong, X. ; Zeng, J.: Effective Road Lane Detection and Tracking Method Using Line Segment Detector. In: *2018 37th Chinese Control Conference (CCC)*, 2018, S. 5222–5227

[LYXL16] Li, K. ; Yao, J. ; Xia, M. ; Li, L.: Joint point and line segment matching on wide-baseline stereo images. In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2016, S. 1–9

[LYY+20] Li, H. ; Yu, H. ; Yang, W. ; Yu, L. ; Scherer, S. *ULSD: Unified Line Segment Detection across Pinhole, Fisheye, and Spherical Cameras*. 2020

[MJ16] Ma, T. ; J., Ma.: A sea-sky line detection method based on line segment detector and Hough transform. In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, 2016, S. 700–703

[MWP20] Mora, L. ; Wu, X. ; Panori, A.: Mind the gap: Developments in autonomous driving research and the sustainability challenge. In: *Journal of Cleaner Production* 275 (2020), S. 124087. – ISSN 0959–6526

[NJR21] Nguyen, V. N. ; Jenssen, R. ; Roverso, D.: LS-Net: Fast Single-Shot Line-Segment

Detector. In: *Machine Vision and Applications* 32 (2021), 1

[OT21] Ossimitz, C. ; TaheriNejad, N.: A Fast Line Segment Detector using Approximate Computing. In: *2021 IEEE International Symposium on Circuits & Systems*, 2021, S. 1–5

[SMA+17] Santos, T. ; Moreira, M. ; Almeida, J. ; Dias, A. ; Martins, A. ; Dinis, J. ; Formiga, J. ; Silva, E.: PLineD: Vision-based power lines detection for Unmanned Aerial Vehicles. In: *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2017, S. 253–259

[TAG10] Topal, C. ; Akinlar, C. ; Genc, Y.: Edge Drawing: A Heuristic Approach to Robust Real-Time Edge Detection. In: *ICPR*, IEEE Computer Society, 2010. – ISBN 978–0–7695–4109–9, S. 2424–2427

[Vol59] Volder, J. E.: The CORDIC Trigonometric Computing Technique. In: *IRE Transactions on Electronic Computers* EC-8 (1959), Sep., Nr. 3, S. 330–334. – ISSN 0367–9950

[WTZZ02] Wang, X. ; Tan, H. ; Zhou, F. ; Zhao, Y.: Real-time Classified Hough Transform Line Detection Based on FPGA. In: *2015 5th International Conference on Computer Sciences and Automation Engineering (ICCSAE 2015)*, Atlantis Press, 2016/02. – ISBN 978–94–6252–156–8

[XXCT21] Xu, Y. ; Xu, W. ; Cheung, D. ; Tu, Z. *Line Segment Detection Using Transformers without Edges*. 2021

[ZCW18] Zhou, F. ; Cao, Y. ; Wang, X.: Fast and Resource-Efficient Hardware Implementation of Modified Line Segment Detector. In: *IEEE Transactions on Circuits and Systems for Video Technology* 28 (2018), Nov, Nr. 11, S. 3262–3273. – ISSN 1051–8215

[ZWL21] Zhang, Y. ; Wei, D. ; Li, Y.: AG3line: Active grouping and geometry-gradient combined validation for fast line segment extraction. In: *Pattern Recognition* 113 (2021). – ISSN 0031–3203

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct, insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirektübernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 3. März 2021

[Christoph Ossimitz]