Towards Automated and Fast Whisker Tracking in Rodents

Petros Arvanitis

Towards Automated and Fast Whisker Tracking in Rodents

by

Petros Arvanitis

to obtain the degree of Master of Science in Embedded Systems
at the Delft University of Technology,
to be defended publicly on Tuesday January 28, 2021 at 01:00 PM.

Student number:	4936736	
Project duration:	January, 2020 – January, 2021	
Thesis committee:	Dr. ir. Zaid Al-Ars,	CE, TU Delft, Supervisor
	Dr. ir. Christos Strydis,	Neurocomputing lab, Erasmus MC, Advisor
	Dr. ir. Rob Remis,	CAS, TU Delft
	Dr. Laurens Bosman,	Neuroscience, Erasmus MC

Abstract

Whisker tracking in rodents is an ongoing research in neuroscience. Neuroscientists have recorded experiments with high-speed cameras in which untrimmed, head-restrained mice are provided with air stimuli. These videos required the development of algorithms to reliably track whisker movement. Recently, a Whisker-Tracking System, WhiskEras, emerged, which is able to detect and track whiskers over the course of such videos in a more accurate way than pre-existing methods. WhiskEras is slow, processing less than 1 frame per second. Additionally, it involves a great number of parameters which need tuning for different experimental setups and recording settings which were used for this experiment. This thesis addresses these two problems. First, the algorithm was examined and its shortcomings were exposed. A more accurate whisker-point detection algorithm was suggested and implemented, among a range of alternative solutions which were studied. Furthermore, its Stitching stage was modified to replace a range of hard-to-tune parameters with more robust ones. Finally, the improved WhiskEras was accelerated by porting the MATLAB code to C++ and using advanced parallelization techniques with CUDA and OpenMP to achieve a speedup of 74.96. Overall, the improvements yielded better tracking results in our benchmarks, while the parameters were much easier to tune and remained constant under different video setups of whisking experiments.

Acknowledgements

My advisor, Christos Strydis, was instrumental in setting the path of this thesis. The project's structure and milestones were defined after consoling with him at every step of this long road. Providing a clear head, when it mattered, and being comprehensive, when setbacks were encountered, were a few of the qualities he brought to the project.

My supervisor, Zaid Al-Ars, aided me in finding my thesis topic. His advice and comments about the project are deeply appreciated.

Jan-Harm Betting was helpful in providing the resources and guidance necessary to complete this project. His knowledge and explanations often came in handy. Overall, he heavily contributed to setting the right goals for this thesis.

I would also like to express my gratitude to Dr. Laurens Bosman for his assistance in providing crucial information on neuroscientific facts which were linked to this project. Talking with him, in particular, helped me understand more about the purposes of whisker tracking.

Finally, I would like to wholeheartedly thank my family and friends who supported and, at times, tolerated me during this thesis.

Petros Arvanitis
Delft, January 2021

Contents

List of Figures

List of Tables

1 Introduction

Neuroscience is the scientific field studying how the nervous system works. This involves understanding the complex properties of neurons, as entities, and neural networks. The *ultimate challenge*, as described by Eric Kandel, is to comprehend the biological grounds behind learning, memory, behaviour, perception and consciousness [31]. Studying this field has expanded from molecular and cellular examination of neurons to imaging various sensory, motor and cognitive tasks in the brain.

An interesting ongoing research in the field of neuroscience is observing whisker movements of rodents. Mice, like other mammals, use their whiskers to obtain tactile information regarding their environment, as well as performing other functionalities [51]. This research can potentially lead to useful information concerning neuronal function in sensorimotor activities, in mammals.

1.1. Motivation and Problem Statement

Whisker movement can theoretically now be tracked reliably, thanks to the advent of high-speed cameras. These cameras are able to record at up to 1000 frames per second, thus making it possible for a human being to follow the trajectory of each whisker through the video, frame-by-frame. However, this is extremely impractical, as a human would have to manually go through 1000 frames for each second of a video and perform annotations.

For that purpose, algorithms are devised to do this work in an automated and efficient way. In the Erasmus MC Neuroscience department, experiments are conducted where untrimmed, head-fixed rodents are exposed to lab-controlled stimuli. The tool currently used to track whiskers in these experiments is BWTT (Biotact Whisker Tracking Tool), which incorporates the ViSA [41] algorithm. It was accelerated by Y. Ma et al. [37] to perform the task almost in real-time, in a recent MSc thesis which involved TUDelft and Erasmus MC. BWTT detects whiskers, frame-by-frame, with limited accuracy. A post-processing script, developed in Erasmus MC, is responsible for tracking the whiskers in the duration of the video, delivering the angle and pivot point position of all whiskers. However, the tracking part of the algorithm is unreliable. As studied by J.L.F. Betting in his more recent MSc thesis [8], the algorithm had to be redesigned from the bottom up in order to perform both accurate detection and tracking of whiskers.

As a solution, Betting devised the WhiskEras algorithm, formerly known as FWTS, a project that was also part of the collaboration between TUDelft and Erasmus MC. It has since been proven to be a strong competitor method in tracking whiskers of untrimmed, head-fixed rodents [9]. This work has by far surpassed the Biotact Whisker Tracking Tool's ViSA method [41] in terms of quality in detecting whiskers and also incorporates a tracking module.

There are two main issues, though. WhiskEras, as a new algorithm, is slow at its task. Specifically, it was reported to process less than 1 frame per second. This is inconvenient, since there is an entire database of whisker videos in Erasmus MC which await processing. In addition, *online tracking* is a desired feature which requires real-time processing performance (1000 frames per second). That would enable neuroscientists to obtain results as they run their experiments. Thus, if it is to be used in practice, WhiskEras needs to be speeded up substantially.

The other problem is *parameter tuning*. The videos recorded include different experimental settings such as lighting conditions, camera viewing angle etc. To process all these videos, a human would need to understand how all algorithm parameters affect the results and attempt to tune them for each video. This involves 34 parameters, which could theoretically take from 2 to more than 100 discrete values. This is undesirable as it would require both user expertise and time consumed between runs on different videos.

Thus, the problem can be formalized as follows: **revisit WhiskEras, making it faster and able to deal with different video conditions in a robust, automated way.**

1.2. Thesis Scope and Contributions

Following is a list of the objectives of this thesis, which was conducted from the beginning, but was also modified along the way. The most noteworthy change of plans was in the improvements of WhiskEras. The initial approach was to analyze the algorithm in depth. This involved defining the parameter space and the impact of the parameters on the results. This was done to prepare the ground for a potential Design Space Exploration (DSE), in order to discover optimal sets of parameters for different videos. However, it was found that performing a successful DSE was not trivial for this problem. Furthermore, the analysis showed that the algorithm suffers in accuracy due to other inherent limitations. Thus, a different direction was chosen, which targeted certain algorithmic aspects of WhiskEras.

1.2.1. Thesis Objectives

The objectives of this thesis can be divided in two separate parts. The first part corresponds to **improvements targeting automation and higher detection accuracy**. This involves the following steps:

(i) Analyze WhiskEras' whisker detection stages in terms of their parameters and their effect on the algorithm's results.

(ii) Pinpoint the weak spots of WhiskEras and investigate why they perform suboptimally under certain conditions or in general.

(iii) Explore ways to tackle the problems spotted in the literature and implement the solution(s) which are most prominent.

(iv) Evaluate their impact and compare to original WhiskEras.

Furthermore, the resulting algorithm was targeted for **acceleration**. This is a long process which requires good programming skills and high-performance computing knowledge. Specifically, the goal was to:

(i) Port the Matlab code to a low-level, efficient language (C++).

(ii) Profile the algorithm to locate its hot-spots, where most time is spent during processing.

(iii) Implement parallel execution on the CPU (OpenMP) and the GPU (CUDA) wherever appropriate to further speed up the algorithm and perform optimizations wherever possible.

(iv) Evaluate the speed-up obtained.

1.2.2. Thesis Contributions

During this project, the following contributions were made:

- WhiskEras was analyzed in depth in terms of its design choices and their effect. Parameter tuning was elaborated on and the most impactful parameters were singled out.
- The shortcomings of WhiskEras' whisker detection part were pinpointed. Solutions to the issues arising were studied from the literature and their applicability to the project's needs was assessed.
- A solution which was a natural build on WhiskEras' whisker-point detection stage was implemented. Other modifications to the algorithm were also carried out, including replacements of several parameters to make the algorithm more robust.
- The whole WhiskEras pipeline was profiled and accelerated by porting the code to C++, taking advantage of inherent parallelism and performing advanced optimization techniques.

- The results of the improved WhiskEras were assessed using a multitude of tracking-related criteria. The speedup achieved from the acceleration was also evaluated by running the code on different videos, on selected powerful hardware.
- A publication will be made as a by-product of this thesis.

1.3. Thesis Organization

The document is organized as follows. Chapter 2 lays the theoretical grounds for topics which will be discussed in the rest of this report. Chapter 3 mentions past projects which are related to this work. Then, Chapter 4 analyzes the WhiskEras algorithm and discovers its shortcomings. Also, it makes a special mention to Design Space Exploration and its deficiency to the problem at hand. The solutions which were examined and adopted to counter these issues are given in Chapter 5. Furthermore, the acceleration techniques used to speed up the improved WhiskEras are mentioned in Chapter 6. The results are illustrated in Chapter 7, where a direct comparison to the original WhiskEras in quality and execution time is made. Finally, the conclusions and future work considerations are reported in Chapter 8.

2 Background

This chapter introduces important concepts and topics which will form the basis for the rest of this thesis. First, the anatomy of whisker movement in rodents is described, as well as the experimental setup used to record whisker videos in Section 2.1. Following is a brief introduction to Design Space Exploration in Section 2.2. Section 2.3 makes a short introduction to Computer Vision and outlines the problem of Curvilinear Centerline Extraction. Then, Section 2.4 provides a summary of Radon Transform, specifically used for curve detection. Last, a recount on using parallelism to boost computing performance is made, with a focus on Amdahl's Law and GPU acceleration in Section 2.5.

2.1. Rodent Whisker Movement

Whiskers, also known as *macrovibrissae*, are long facial hair that are found on numerous mammals, including rodents. They can be distinguished from normal hair by their long length, hair follicle, and by having an identifiable representation in the somatosensory cortex of the brain. Rats and mice use their whiskers to explore their environment through multiple tasks such as localisation, tactile discrimination and locomotion [51]. Thus, they are particularly interesting for understanding patterned motor activity in mammals.

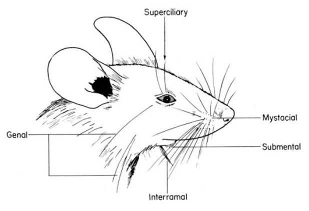
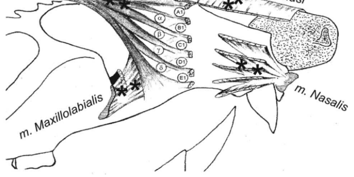


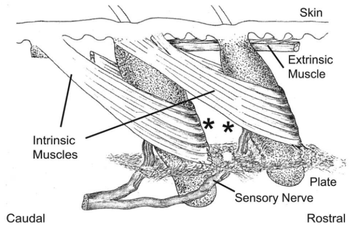
Figure 2.1: Rodent vibrissal groups [17]

Neuroscientists mostly observe the behaviour of whiskers rooted in the mystacial pad (Figure 2.1). These form an ordered grid of rows, where each whisker grows from a hair follicle which is heavily innervated by sensory nerves.

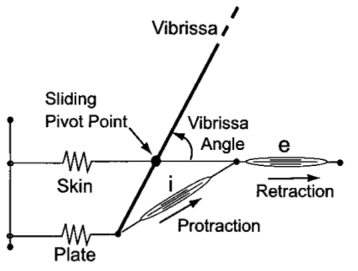
Whisker movement is controlled by the extrinsic and intrinsic muscles. Extrinsic musculature shifts the surface of the mystacial pad backwards, while the intrinsic muscles rotate individual whiskers forward [7]. It can occur in a rhythmic pattern, called *whisking*. During whisking, rodents sweep their whiskers in frequencies up to 25 Hz and at reported speeds up to 1106 deg./sec [13]. The parameters of interest that describe the state of a whisker are its pivot point (attachment point of follicle) and its angle. The mechanical model of whiskers is depicted in Figure 2.2.



(a) Extrinsic musculature [27]



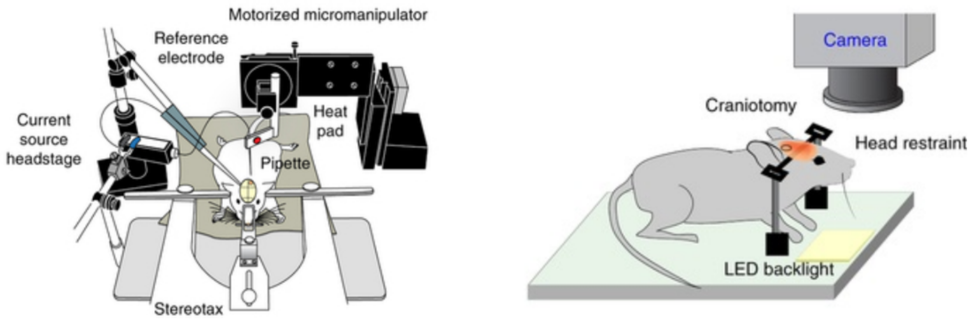
(b) Intrinsic musculature and follicular anatomy [27]



(c) Mechanical model of whisker movement [7]

Figure 2.2: Vibrissal mechanical model

2.1.1. Whisking Experimental Setup



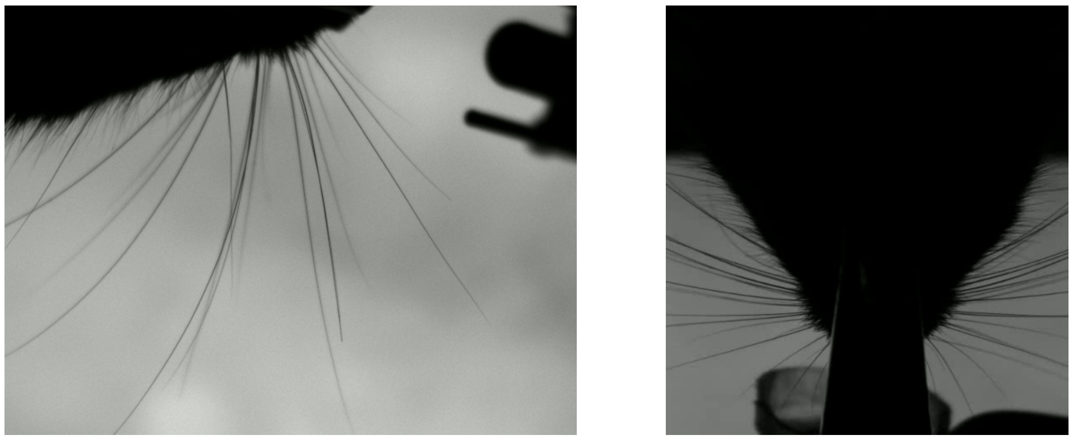
(a) Operating setup (b) Head constraint and camera setup

Figure 2.3: Whisking experimental setup [8]

The videos containing whisker movement which were used to develop, test and validate the improved WhiskEras algorithm originate from the Erasmus MC Neuroscience Department in Rotterdam, the Netherlands. These videos were recorded at 750-1000 Hz, using a bright LED panel as backlight, in grayscale.

Adult C57BL/6J mice were used as test subjects. These mice have dark fur, which contributes to lower noise in the videos. The mouse's snout appears as a black silhouette which can later be filtered out easily. The animal is head-restrained in order to be able to reliably track its whiskers. There is no prior trimming to its fur or whiskers, so that the whisker movements are as close to real behaviour as possible. The experimental setup is illustrated in Figure 2.3 and is the same as in [47].

In general, two different camera-angle views have been used during the recording of the videos, depicted in Figure 2.4. In 2.4a, the camera is zoomed in towards one side of the snout. The whiskers that can be seen mostly belong to the top row of the mystacial pad, while intersections and occlusions between whiskers are common. In 2.4b, only the snout's right-side whiskers (left in the image) are tracked. Because there is only a small fraction of pixels dedicated to the area of interest, less whiskers are visible.



(a) Camera view 1 (b) Camera view 2

Figure 2.4: Whisking experiment camera views

The neuroscientists of Erasmus MC are interested in tracking the angle of the whiskers relative to the snout. Due to the mystacial pad moving backwards (Figure 2.2a), the pivot point of each whisker along the snout also changes. This is not clearly visible on the videos because the mystacial pad is in vertical view.

2.2. Design Space Exploration

Design Space Exploration (DSE) is the process of discovering and assessing design alternatives during system development [32]. It is commonly used as an optimization method in order to go through millions or even billions of different design choices and pinpoint the optimal ones according to some metric.

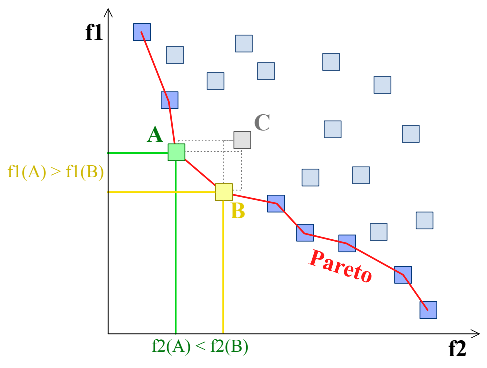


Figure 2.5: f1 and f2 scores for different design points [38]

The red line crosses the set of the Pareto frontier points.

In multi-objective problems, there are several objectives that are targeted by the optimization process. One way to find the optimal solutions to these problems is by locating the Pareto fronts. An optimal solution is a non-dominated solution, i.e if none of the objective functions can be improved in value without degrading some of the other objective values [38]. An example is given in Figure 2.5, where there are two objective functions f_1 and f_2 and the Pareto frontier is the set of optimal solutions.

Apart from objectives, the design space has to be defined in terms of scale and constraints. When there exist a number of parameters $p_1, p_2, ..., p_N$, each parameter p_i has to be assigned lower, upper limits and a traversal step. The more parameters and the more values they are allowed to take, the more complicated becomes the design space to explore. Often, the design space is so vast that this procedure cannot be performed by the user in a manual fashion. This is where DSE frameworks come in handy.

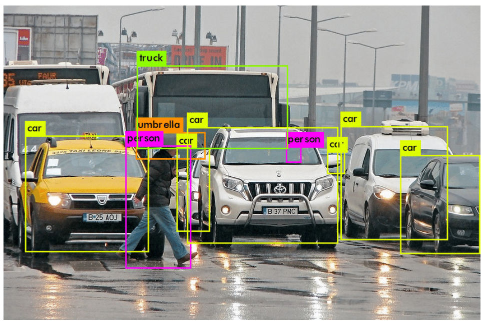
There is a variety of different approaches to DSE, such as genetic (evolutionary) algorithms (example [20]) or Monte Carlo methods. Different frameworks use different approaches and deciding on one usually comes down to efficiency, convergence to optimal solutions, diversity of solutions and ease of use.

2.3. Computer Vision

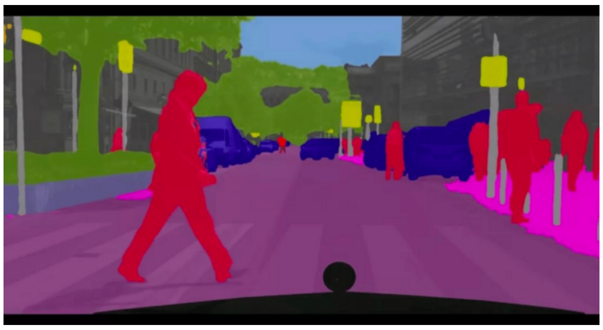
Human vision is an extraordinarily complex task, even though it seems trivial to humans. Computer vision is a field that studies how a computer can *see*, i.e how to interpret image data to infer information about the world [12].

It involves a wide variety of techniques ranging from simple image processing to sophisticated Artificial Intelligence techniques. Computer Vision applications involve **tasks**, such as:

- Object Detection: Where are the objects in the image?
- Object Classification: In which broad category does an image's object fall into?
- Object Segmentation: Which image pixels belong to the object?



(a) Object Detection: drawing boxes around objects. Object Classification: labelling objects. [33]



(b) Object Segmentation: each object's pixels are coloured uniquely for each class. [53]

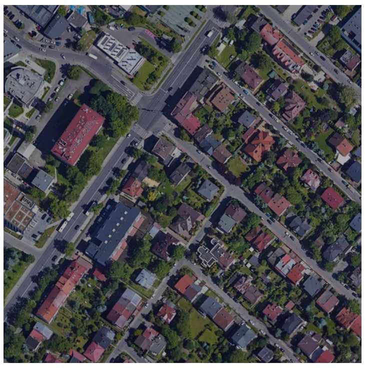
Figure 2.6: Computer Vision task examples

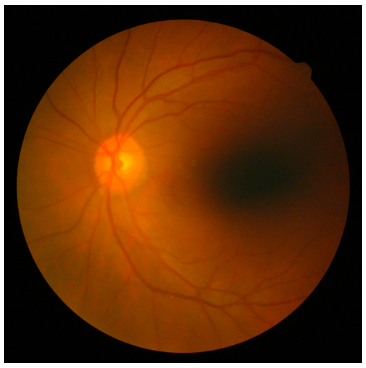
2.3.1. Curvilinear Centerline Extraction

A challenging Computer Vision task is curvilinear centerline extraction. Whiskers are essentially curvilinear objects. Knowing that, WhiskEras detects whiskers by extracting curvilinear *centerlines*, i.e 1-pixel wide curved lines crossing the centre (width-wise) of each whisker. It performs this task with sub-pixel precision. In other words, it is not limited to the discrete space of image pixels, but can find curve points between pixels. This is the main stage of WhiskEras that was targeted for improvement in this thesis.

A few **applications** which employ the curvilinear extraction technique are:

- Road detection in aerial road map images (Figure 2.7a).
- Blood-vessel detection in retinal images (Figure 2.7b).
- Neuronal detection (Figure 2.7c).


(a) Aerial suburban road map (taken using Google maps)


(b) Retinal image [29]

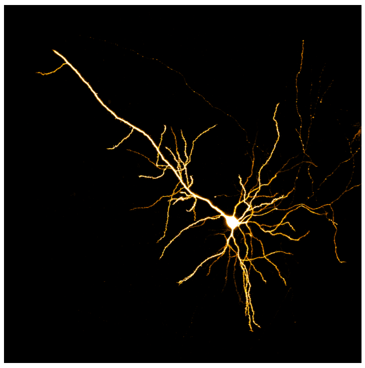

(c) Neuron confocal fluorescence image [4]

Figure 2.7: Images where curvilinear extraction may provide useful information

2.4. Radon Transform

The Radon Transform, mathematically, is an integral transform [42]. It takes as input a function f on a 2D plane and outputs a function Rf on the (also) two-dimensional space of lines in this plane. For each line, it takes a value which is equal to the integral of that line over f. Suppose $f(x, y)$ is the function representing an object in Figure 2.8. Also, α is the angle of the line with the x axis, s is its distance from the origin and z is the arc length. Then,

$$Rf(\alpha, s) = \int_{-\infty}^{\infty} f(x(z), y(z))\, dx = \int_{-\infty}^{\infty} f((z\ sin\alpha + s\ cos\alpha), (-z\ cos\alpha + s\ sin\alpha))\, dx$$

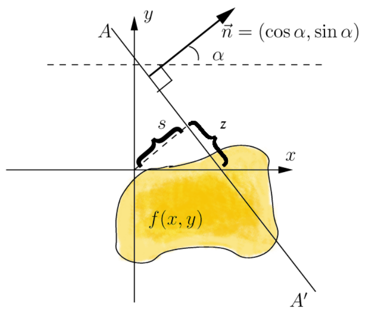


Figure 2.8: Radon Transform of a function f [43]

The Radon Transform is widely used in tomographic reconstructions [19]. In particular, when an object which is represented as $f(x, y)$, mathematically, is scanned by a detector, from all projection angles, we can then get the full representation of that object in the projection domain, parametrized by (α, s). In other words, the full description of the object in terms of the parameters (α, s) is available. Then, in order to recover the function of the object in the (x, y) domain, an Inverse Radon Transform can be used. This is the method deployed to interpret the output of CT (Computed Tomography) scans.

2.4.1. Radon Transform in Curve Detection

Images consist of an integer grid of pixels. In that case, the Discrete Radon Transform can be applied by summing over its pixels, rather than computing an integral. In this work, the Radon Transform is used to detect lines. This has already been studied in [50], [44]. Lines can be detected, globally, by performing Radon Transforms across the entire image. This involves computing the inner products of lines with the image. In other words, each line, in the (α, s) space, can be quantified, using Radon Transform, as the summation of the intensities of all pixels belonging to that particular line. If that sum is large enough, then it can be considered as a valid line in the image. Furthermore, the Generalized Radon Transform can be used to detect curves, in general, using different curvilinear profiles (linear, quadratic, etc). In this thesis, the **Localized Radon Transform** is used, as discussed in Chapter 5, which is the Radon Transform over a limited number of pixels (specific line length), rather than the entire image.

2.5. High Performance Computing and Parallelism

We live in the multicore era of High Performance Computing. Computer engineers are no longer just grinding the hardware, architecture and compiler of a single processor core to enhance performance. Instead, the programmers have to write their code in such a way that they take advantage of multiple cores for parallel execution [40].

2.5.1. Amdahl's Law

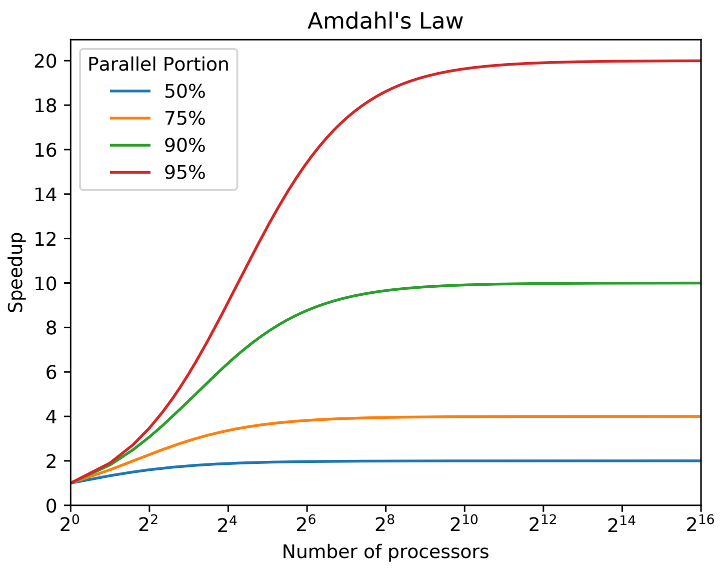


Figure 2.9: Theoretical speedup of the execution time of a program for varying parallel fractions according to Amdahl's Law

Suppose that a program's fraction f is infinitely parallelizable with no scheduling overhead, while the remaining fraction $1-f$ is entirely sequential. Then, the speedup obtained by using n processor cores to run the program, as opposed to running on a single core would be:

$$S = \frac{1}{(1-f)+\frac{f}{n}} \tag{2.1}$$

Equation 2.1 provides a theoretical upper boundary to the max speedup that can be obtained from parallelization. Needless to say, this is an optimistic prediction, as it does not account for any overhead associated with shared caches, interconnection networks, memory controllers etc. Still, it does provide the programmer with a useful indication whether parallelization would be worthwhile. Furthermore, Figure 2.9 shows that programmers should always seek to increase the parallelizable fraction f of their program [28].

2.5.2. Heterogeneous Parallel Computing

A different approach to parallelization is the *many-thread* trajectory, which focuses on the execution throughput of parallel applications [34]. Following this trajectory, GPUs support thousands of threads running in parallel, as opposed to a few threads currently supported by CPUs. GPUs take advantage of data-level parallelism to perform substantially more Floating Point Operations (FLOPS) per second than CPUs. This translates to significantly lower execution times for some applications. The reason is that CPUs are more oriented towards sequential execution. GPUs, on the other hand, have a larger memory bandwidth as well as hiding thread latency while always keeping their Streaming Multiprocessors (SMs) busy with work. This is why they are better suited for performing massively parallel operations. Now, the focus is on the programmer to be able to reformulate their algorithm's most expensive stage in such a way that it can benefit from GPU execution, as illustrated in Figure 2.10.

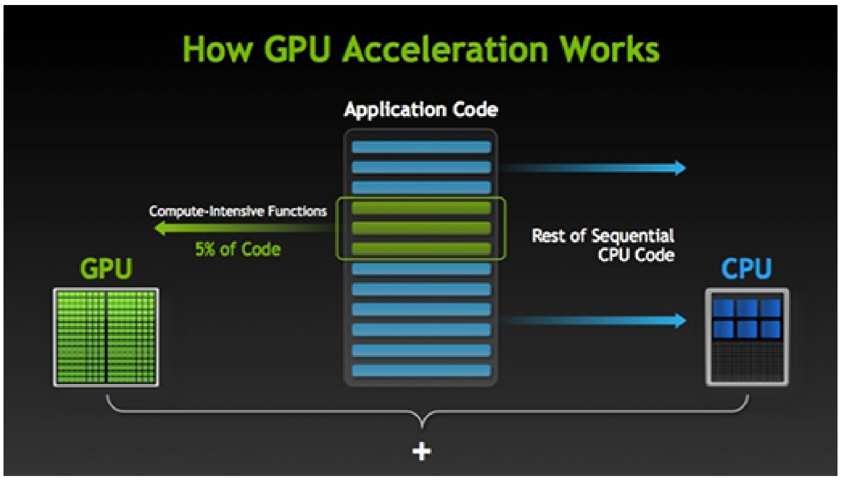

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential CPU Code
GPU
CPU
+

Figure 2.10: How GPU acceleration works [21]

2.5.3. GPU: Accelerating Image Processing

Today, GPUs are no longer just a luxury to play video games. Most research labs are equipped with the most expensive GPUs to facilitate their experiments. For example, Neural Networks, a concept that has been around for many decades [2], started being realized in a large scale only in the last two decades, largely due to the breakthroughs in GPUs. Their presence has changed research in a way that was unimaginable in the past.

Image processing can heavily benefit from data-level parallelism. Since images are dataframes with millions of elements, performing parallel operations on multiple pixels at once is desirable. This is where GPUs thrive. In particular, using the **CUDA** parallel programming language, an experienced programmer can assign memory-aligned pixels to consecutive threads and perform fast calculations in the GPU's Single Instruction Multiple Data (SIMD) processing units. For example, Figure 2.11 shows how 2D blocks of 16x16 threads can be formed to process all pixels of an image in parallel.

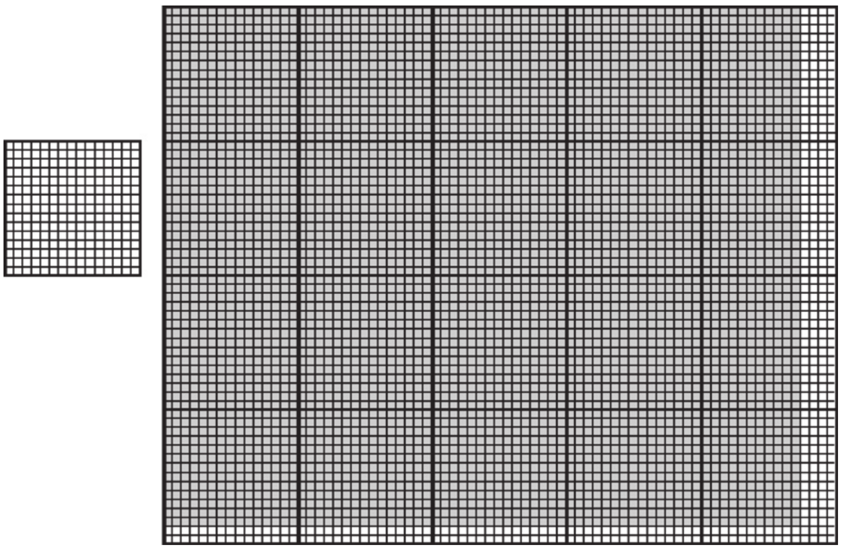


Figure 2.11: Image processing using thread blocks on the GPU [34]
2D grid of blocks of threads to process an image. Left: a single 16x16 block of threads. Right: image segments assigned to different blocks. Threads that are outside the bounds of the image (light grey) are handled by control logic to not perform any work.

3 Related work

Many whisker-tracking algorithms have been developed over the years. BWTT and Janelia Whisk are some of the most popular ones. These, along with WhiskEras, will be presented briefly in this chapter. WhiskEras was the algorithm chosen to work on because it was considered to have the most potential and is also relativaly new, so it still has margin for improvement. Then, some of the Curvilinear Centerline Extraction methods will be mentioned, which were considered as alternatives to Steger's Unbiased Curvilinear Detector, currently used in WhiskEras.

3.1. Y. Ma's ViSA Implementation

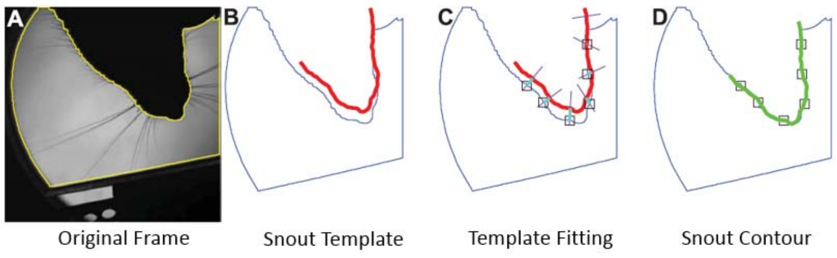


Figure 3.1: Contour extraction and snout template fitting to detect head [37]

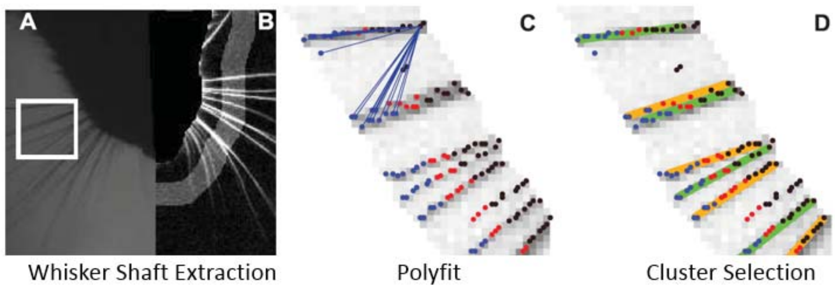


Figure 3.2: Whisker shaft detection and parametrization [37]

The ViSA algorithm is the basis of BWTT. It is an algorithm which detects whisker shafts in video frames. It does so in the following stages:

1. It detects the head of the mouse (Figure 3.1) by:
    (a) extracting the contours of the frame using simple image processing and morphological operations and
    (b) fitting a user-defined snout template to the contour.

2. It detects a narrow band around the snout and parametrizes the whisker shafts around it (Figure 3.2) by:
   (a) performing background subtraction and masking to select pixels within a radius of the snout,
   (b) creating a whisker tree of points which will form whiskers,
   (c) clustering using polynomial fitting on the sink, source and central nodes of the line segments detected and
   (d) selecting candidate segments in an iterating process to form the whisker shafts.

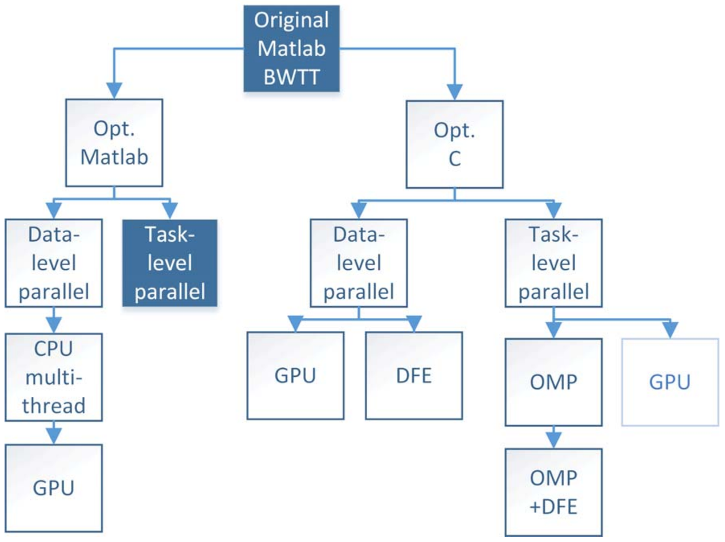


Figure 3.3: Acceleration of ViSA algorithm workflow [37]

Ma accelerated the ViSA algorithm in several different ways, as shown in Figure 3.3. First, he modified the MATLAB code to utilize parallel execution in the form of multithreading in the CPU and the GPU. The result was significantly faster but still not fast enough to be used in practice for processing high-speed whisker videos. Following, the code was ported to C, a low-level language which is much faster than MATLAB. From there on, several different routes were explored, where either *data-level parallelism* and *task-level parallelism* were emphasized. The data-level parallelism was taken advantage of both on the GPU using CUDA and a Maxeler Dataflow Engine (DFE), which is based on FPGA technology and is suitable for stream-based processing. However, some parts of the code which were non-parallelizable were becoming the bottleneck in performance. Hence, an OMP implementation was developed which provided batches of sequential frames to multiple CPU threads. This resulted in the greatest speedup, yielding a 1.21 ms/frame peak performance. This is really close to real-time processing, which requires 1 ms/frame. Ma mentions a final attempt to reach it by using DFEs together with OMP. Unfortunately, at the time, the DFEs did not support initialization from different processor threads. Thus, they did not deliver better performance.

This work really showcases the different approaches one may take in High-Performance Computing (HPC) to achieve high-speed processing and is an inspiration to future acceleration projects, such as this one. In this thesis, both *task-level* and *data level* parallelism were exploited using CUDA (GPU) and OMP (CPU multi-threading) to speed up WhiskEras.

Most whisker-tracking algorithms have a prerequisite for whisker clipping, labeling and/or trimming to facilitate the task. This is rather invasive from a behavioural point of view. BWTT was the only whisker-tracking algorithm, before WhiskEras, with the capability of detecting unclipped and unlabeled whiskers. Still, it lacks a tracking module, meaning it can only detect whiskers. The post-processing script which is used to perform tracking is not very accurate, due to the nature of BWTT, as discussed in [8]. Despite its problems, Ma's implementation of ViSA is the algorithm currently being used by the neuroscientists in Erasmus MC to process high-speed videos of the *whisking* experiments and has already led to important findings.

3.2. Janelia Whisk

Janelia Whisk [16] is another whisker-tracking algorithm which works well for trimmed, head-fixed mice, at a moderate processing speed of around 35 frames / second. Contrary to BWTT, however, it does incorporate the ability of tracking whiskers.

It uses three stages to perform whisker tracking:

1. Tracing: curvilinear objects are located using a linear detector based on profiling the intensity of whiskers as a rectangular valley in the image, with varying position, width and angle.
2. Linking: curves found during tracing are identified as whiskers using statistical methods which either require providing the number of whiskers to identify, or estimate it.
3. A Naive Bayes classifier is employed to trace the trajectory of whiskers over the course of an experiment/video.

Although this is a competitive whisker-tracking algorithm, it can only track a few whiskers accurately. Consequently, trimming of hair and whiskers is needed as a pretreatment. This limits the information that can be distilled from whisking experiments considerably.

3.3. WhiskEras: a Whisker-Tracking System

WhiskEras is a whisker tracking algorithm which detects and tracks whiskers on untrimmed, head-constrained rodents. It was inspired by the Biotact Whisker Tracking Tool (BWTT) [41], but was developed to incorporate a tracking module which was lacking in BWTT. Its recent paper publication [9] made a strong case of its superiority to other methods such as BWTT and Janelia Whisk. In particular, it has been shown to reliably follow the movement of more than 10 whiskers over the course of a video, whereas older trackers relied on more invasive experimental setups such as marking whiskers with clips and trimming to the point where 2-4 whiskers are visible.

Furthermore, it is an unsupervised algorithm, meaning it does not need to be provided with labelled data to do its work. Collecting a sufficient amount of annotated data to develop a supervised algorithm would require great amounts of resources. Although modern supervised AI algorithms dominate applications in different areas, WhiskEras makes good work of the task without having an enormous labelled dataset to back it up.

The algorithm itself was written in MATLAB in a fairly efficient way, taking advantage of inherent parallelism wherever possible. However, it is still slow due to MATLAB being a high-level, prototyping programming language. In addition, it requires a lot of parameter tuning to work accurately in different experimental and video recording setups. Further details on WhiskEras will be given in Chapter 4.

3.4. Curvilinear Centerline Extraction Methods

Whisker-point detection in WhiskEras is formulated as a curvilinear centerline extraction problem. As such, it is addressed using the Unbiased Curvilinear Detector from Steger [49]. This algorithm performs well in general, but suffers from several issues which will be described in Section 4.3. A high-level summary of this algorithm will also be provided in Section 5.2.

Curvilinear extraction (manually) is illustrated in Figure 3.4. It can easily be inferred that producing the output image on the right, given the input image on the left, is no trivial task, even for a human being. A general categorization of the methods to perform this task was provided by [55]. In fact, these methods can demonstrate the variety of ways with which one can approach a Computer Vision problem.

Intuitive Methods In general, they look for different characteristics of curvilinear structure and background in local regions (hence, their name). They can be further divided into Hessian-based and model-based approaches [24]. **Hessian-based** methods rely on the eigenvectors and eigenvalues of the Hessian matrix in order to segment the filamentary structure. The drawback is that they need a Gaussian Blur preprocessing which may lead to errors when the local structure is too wide or located in a branch region. Steger's unbiased curvilinear detector [49] falls into this category. On the other hand, **model-based** methods instead focus on fitting filaments with known geometric shapes. The *optimally oriented flux* (OOF) [35] localizes the spherical region to compute the gradient flux by minimizing the inward flow, a work that was further improved [52]. Despite the long-year progress of these methods, their performance was reported to deteriorate substantially when dealing with small and thin filaments, in cluttered and ambiguous backgrounds [24].

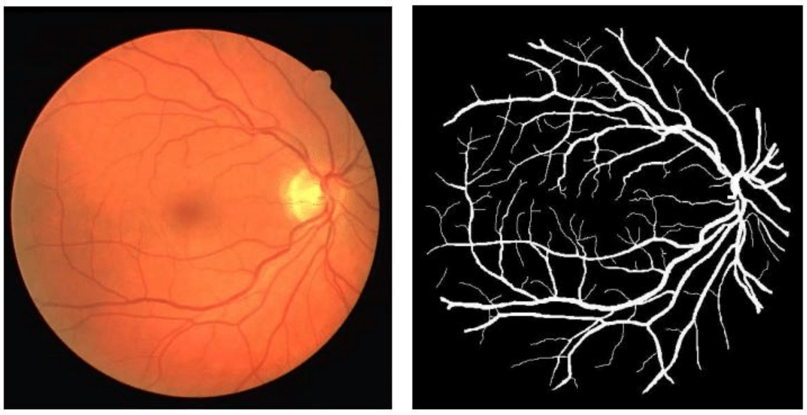


Figure 3.4: Left: retinal image input. Right: manual curvilinear extraction output. [29]

Learning-based Methods With Machine and Deep learning seeing tremendous progress during the last two decades, these methods are definitely cutting edge in this task. They employ supervised algorithms which build a learning/clustering model and then make decisions based on either classification or regression. In this task, they typically output a binary image declaring pixels either as belonging (or not) to a curvilinear structure. *Multi-Centered Hough Forest Method* (MCHF) [55], *Learning to Boost Filamentary Structure Segmentation* (LSF) [24] and other algorithms have reported results unparalleled by intuitive methods. However, the lack of labelled data is prohibiting us from using any of these methods, unless annotation of (a lot of) data precedes it.

Graph-based Methods Curvilinear structure is often characterized by the tree-like or net-like filamentary structure. Thus, there are methods that locate it by extending graphs/networks. Chai et al. [14] developed a method which samples junction points using a Monte Carlo mechanism. The disadvantage of this approach is that it is more suitable towards a network with piecewise straight lines. De et al. [18] reformulated the problem into label propagation over directed graphs. This method handles the *crossover problem* (intersection between curves) well, but it also integrates a supervised classifier, which requires training over an existing labelled dataset. More importantly, its segmentation module, which discovers the curvilinear structure, is not necessarily more accurate than the intuitive methods.

The method which was used, in this work, to replace Steger's Curvilinear Detector was K. Raghupathy's *Improved Curve Tracing* [45], which is an intuitive approach. The reasons behind this choice, as well as an extensive presentation of this method will be given in Chapter 5.

4
WhiskEras: Algorithm Analysis

In this chapter, an analysis of WhiskEras is given. Section 4.1 provides an overview of the algorithm. Then, in Section 4.2, the design choices in terms of parameters in WhiskEras are explored. Section 4.3 pinpoints the weak spots of WhiskEras. Finally, an analysis as to why a Design Space Exploration was not pursued is given in Section 4.4.

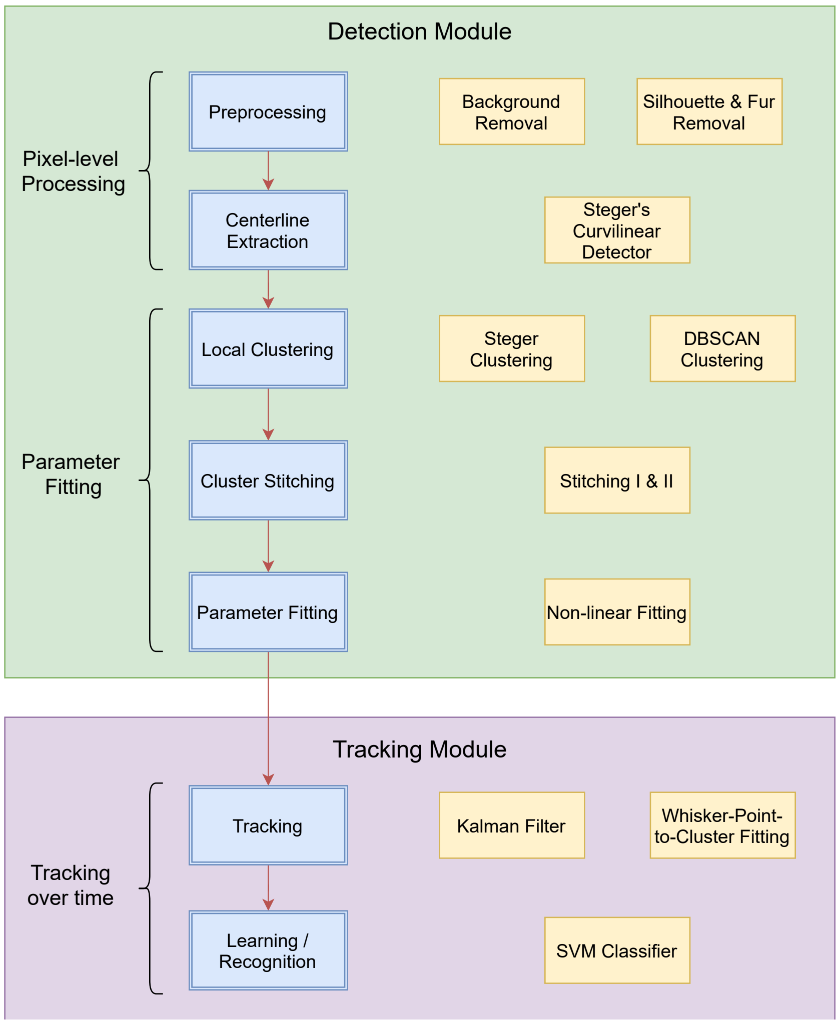


Figure 4.1: WhiskEras overview

4.1. Overview

WhiskEras comprises of two main modules: Detection and Tracking. The Detection module is responsible for detecting whiskers and fitting them into an abstract representation of several parameters. The Tracking module's duty, on the other hand, is to *follow* these whiskers, frame-by-frame. The system can also be divided into three components, summarized below. The complete Whisker-Tracking System is illustrated in Figure 4.1.

Pixel-level Processing It involves *preprocessing* of the frame in order to remove the background, silhouette and fur. We are then left with the light whiskers in a dark background. *Centerline extraction* (i.e whisker-point detection, as will be referred to later) is the process of locating the whisker points, with the help of Steger's curvilinear detector [49], as discussed in Section 3.4.

Parameter Fitting It takes as input the centerline positions of the whisker points in the image and performs *local clustering* in order to form groups of points which belong to the same whisker. This clustering, however, does not result in a perfect grouping where each cluster forms an entire whisker. Rather, cluster *stitching* is required to unify clusters which belong to the same whisker. Afterwards, *parameter fitting* takes place, where each whisker is *compressed* to a set of parameters (position along snout, angle, bending and length).

Tracking over time This stage is responsible for matching the newly found whiskers with the whiskers found previously. The *Tracking - Learning - Detection* (TLD) technique was adopted, as described in [30], to ensure consistency across a long sequence of frames. Tracking is performed using either a Kalman filter or by fitting whisker points to whiskers found in previous frames. A Support Vector Machine (SVM) Classifier is employed as a learner to recognize the correct matches between current and previous whiskers. Tracking is necessary to collect a minimum number of data to train the SVM. Then, the SVM is relied upon to identify whiskers in a consistent way, while tracking is used to potentially fit whiskers that were not recognized by the SVM.

This thesis mainly focuses on:

- Enhancing the Detection module. This is the module which requires more intense parameter tuning and is considered to be the bottleneck of the algorithm.
- accelerating the whole Whisker-Tracking System.

Naturally, providing optimal detection results to the Tracking module can facilitate its work substantially. In the next sections of this chapter, we only refer to the Detection module of the algorithm.

4.2. Analyzing Design Choices

WhiskEras is an extremely accurate whisker tracking method, being able to track a large number of whiskers in untrimmed, head-fixed mice way more consistently than other methods [9]. The downside is that it needs significant tuning in order to perform well and it is unknown whether design choices so far have offered optimal results. Specifically, the detection pipeline contains 34 parameters in total. The first goal of this thesis is to explore the impact of these parameters on the results and define a bounded discrete space for each one of them.

To facilitate this task, each substage of the algorithm has been analyzed in isolation, for the most part. This is a fair treatment, since the results generated by each stage have a clear linear relation to subsequent results. In other words, the Computer Vision techniques used do not justify any non-linear effects. The effect of the parameters on the results of each stage will be made clear below. The results of each substage can be seen in Figure 4.2. Note that *Stitching* is subdivided into parts I and II in the upcoming analysis.

The first system variable that has to be defined is the line which defines the position of the snout. This needs to be specified by the user, according to the input video. The line (Figure 4.3) is specified by the X and Y coordindates of two points on the line, for a total of 4 parameters. This will ensure only curves below the snout line are detected and in the case of Figure 4.3b, only on one side of the snout.

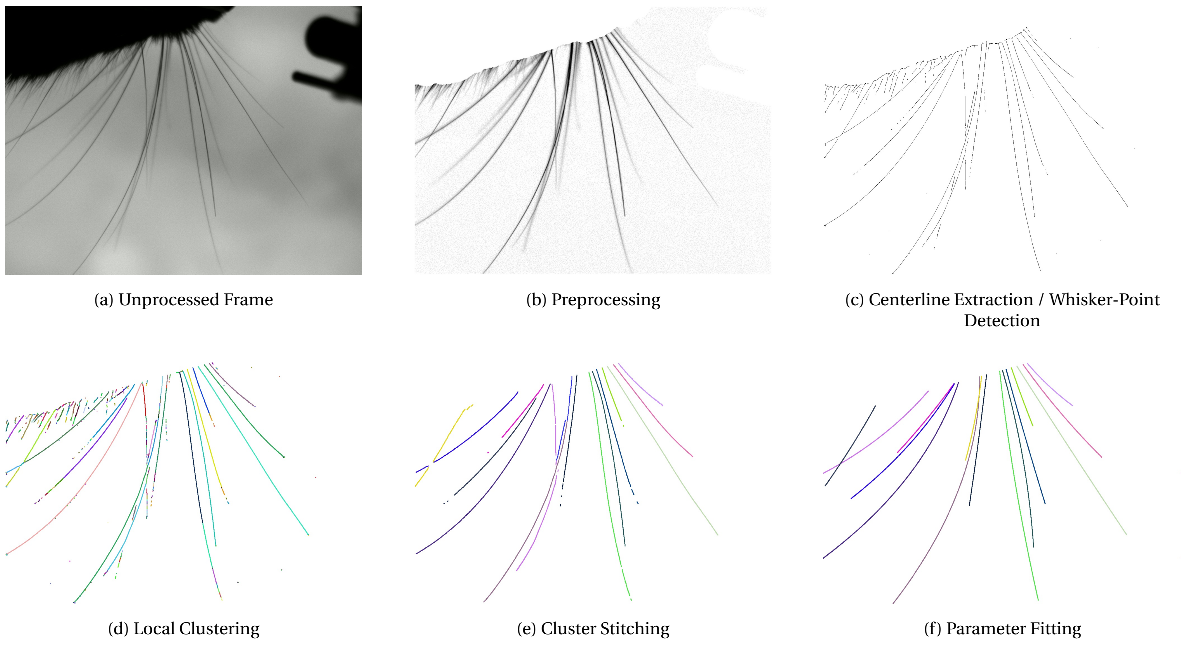


Figure 4.2: Results from each substage of the WhiskEras Detection module

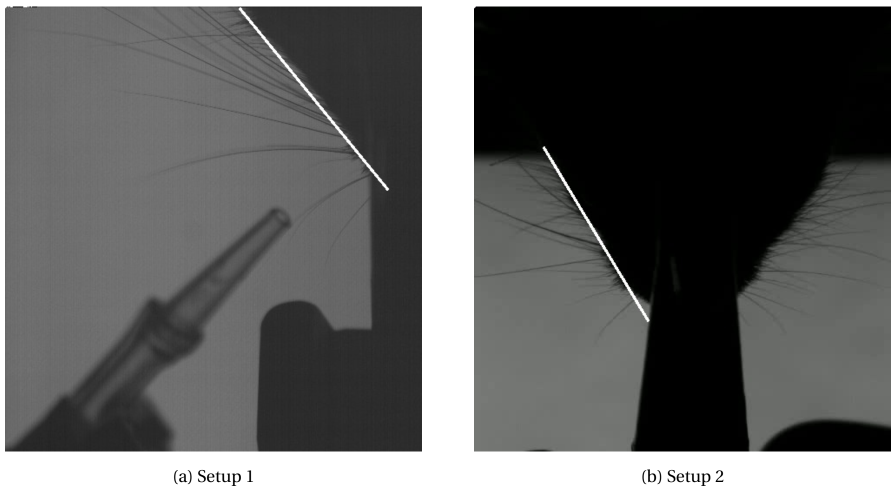


Figure 4.3: User-specified snout line (in white).

4.2.1. Preprocessing

The first preprocessing step is to subtract the unmoving background as well as the snout silhouette which is constant for the entire experiment. Additionally, the frame goes through a contrast-adjustment step to highlight whiskers more than noise. Then, deinterlacing or a Gaussian blur filter can be used interchangeably to remove the artifacts created by the interlacing effect, which is present in the Erasmus MC videos. The noise can further be reduced by a mild thresholding operation. Because there is a timestap in some of the videos, a few edge pixels can also be removed without any loss of vital whisker parts.

All parameters associated with preprocessing are described in Table 4.1. Effect refers to the result of increasing a parameter's value. A useful parameter range has been specified, as well as a traversal step. When these specify a single value, this means that the user does not need to bother tuning that specific parameter, in which case the traversal step is indicated as a "−".

Most of these parameters do not have a margin for improvement, so they can remain fixed. The *imdilate_value* and *cutoff* have a similar role. Ideally, these should be kept to small values. This way, no whisker parts are lost. However, the algorithm may have trouble, particularly during the stitching stage, if much of the hair is retained. For that reason, in certain videos, it helps to increase those values. Regarding contrast

Name	Functionality	Effect	Range	Step
bwThreshold	thresholds to obtain mouse silhouette	less noise and whisker points	1	–
imdilate_value	shaves off hair/whiskers near the snout	less hair and shorter whiskers, more distant from snout	0 – 40	5
cutoff	filters out area below snout line	less hair and shorter whiskers, more distant from snout	0 – 40	5
histEqMinT	zeroes out intensities below this value	less noise and whisker points	3	–
histEqMaxT	saturates intensities above this value	less noise and whisker points	50 – 120	10
histEqGamma	weights either bright or dark pixel intensities more	less noise and whisker points	0.7 – 1	0.05
do_deinterlacing	negates interlacing effect	introduces noise, distorts angles	false	–
initialGaussianSigma	negates interlacing effect	introduces noise distorts angles	0	–
remove_lowlevels	removes noise	removes noise and whisker points	10	–
framing	removes time stamp	removes edge pixels	5	–

Table 4.1: Preprocessing parameters

enhancement, either decreasing *histEqMax* or *histEqGamma* can potentially increase detection sensitivity, up to a critical value where decreasing them further only adds considerable noise. Because their effect is not additive, just adjusting one of those two values is sufficient, while keeping the other fixed. Furthermore, the interlacing effect, present in the videos, does not hurt the algorithm as much as its countermeasures in *deinterlacing* or *Gaussian Blur*. The last two parameters of Table 4.1 are not that significant to the algorithm's functionality.

4.2.2. Centerline Extraction

Steger's unbiased curvilinear detector [49] is used during this step. First, the image is upscaled. This step is necessary because adjacent whiskers are hard to distinguish by Steger's algorithm. Then, the Hessian Matrix's second order derivatives are computed using a Gaussian Kernel. Following that, whisker points are discovered with subpixel accuracy by using the eigenvalues and eigenvectors of the Hessian Matrix and a sensitivity value *tr2*.

Name	Functionality	Effect	Range	Step
upscaling	increases image dimensions	better response near intersections, worse elsewhere	2	–
sigma	adjusts gaussian sigma	less noise and whisker points	2.3	–
gaussianKernelSize	adjusts gaussian window size	better detection quality up to a critical value	20	–
tr2	adjusts detection sensitivity	less noise and whisker points	0.8 – 1.5	0.1

Table 4.2: Whisker point detection parameters

The effect of *upscaling*, mentioned in Table 4.2, is depicted in Figure 4.4. An additional downside (to the one mentioned in Table 4.2) of upscaling is that it heavily increases the execution time because there are more pixels to process during this stage and more whisker points during the subsequent stages. A good trade-off was found for a value of 2.

The *sigma* and *gaussianKernelSize* parameters were found to produce good-quality, stable results for a specific value under different video conditions. In particular, increasing *gaussianKernelSize* to more than 20 does not bring any further benefits. In addition, it also heavily impacts the computational cost of this stage, so

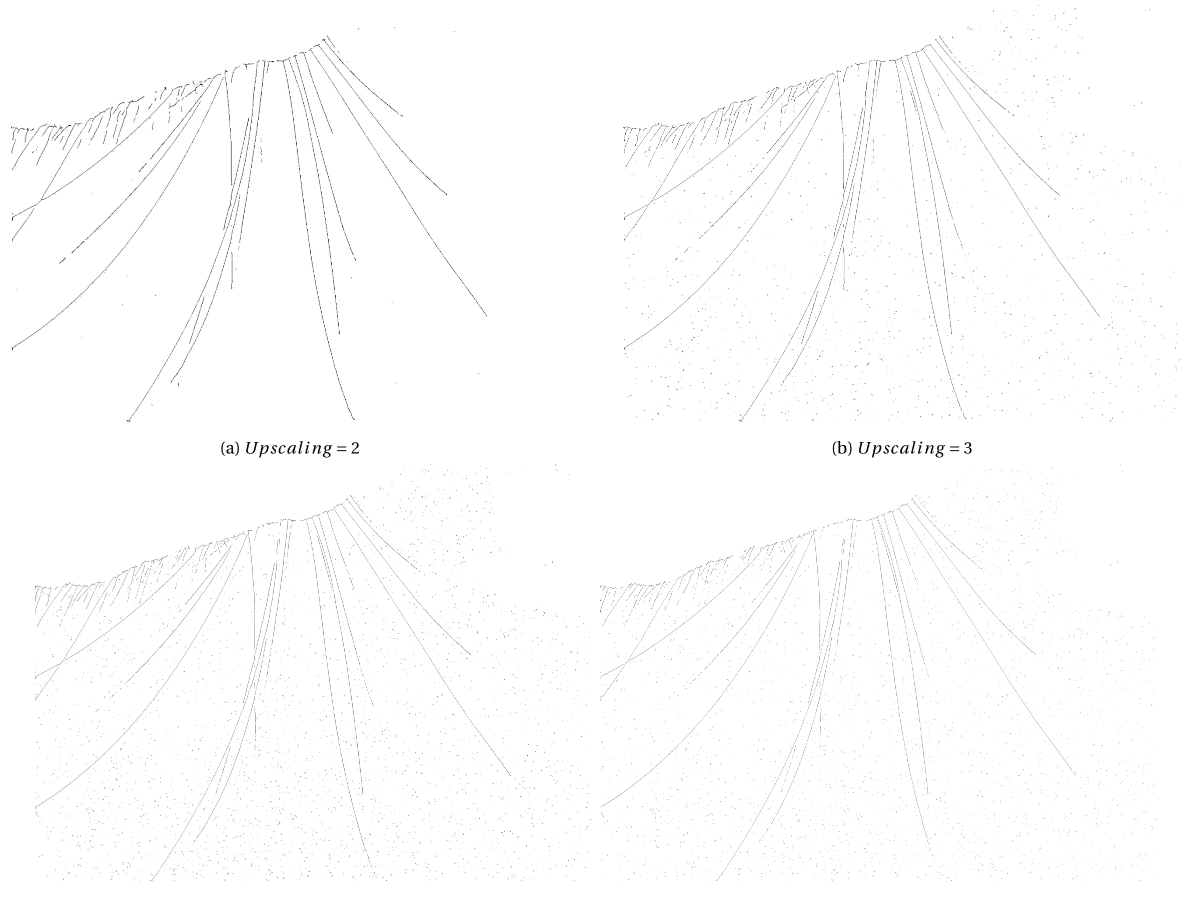


Figure 4.4: Whisker points detection: varying upscaling

it should not be increased more than needed. *tr2* is an important parameter that should be tuned depending on each video.

4.2.3. Local Clustering

Having detected the whisker points, the algorithm forms groups of points which belong to the same whiskers. Again, Steger's algorithm [49] offers a solution to this task. In this step, a graph-connectivity matrix is constructed, which connects neighbouring points together using distance and orientation as factors. Each point's orientation is computed using Steger's algorithm. Betting [9], in fact, modified Steger's algorithm to only form connections among doubly-linked points. He also increased maximum distance between neighbours to from 1 pixel to 2 pixels radius.

Name	Functionality	Effect	Range	Step
steger_c	adjusts orientation weight, relative to distance	more accurate connections, but more clusters are formed	$1-20$	2

Table 4.3: Clustering parameters

Aside from choosing the *steger_c* value in Table 4.3, an alternative clustering method option was also provided in WhiskEras, the well-known DBSCAN method [46]. It was reported that, in certain cases, DBSCAN achieved better results [9]. However, experimentation performed in this work indicates that Steger Clustering has more potential in whisker clustering. Figure 4.5 illustrates clustering results obtained using DBSCAN. In configuration 4.5a too many clusters are formed, whereas in 4.5b certain errors have been introduced, although its results seem better overall. This has to do with the fact that DBSCAN forms groups with respect to point density, not lines. Let us compare these results with the results of Steger Clustering in Figure 4.6. It

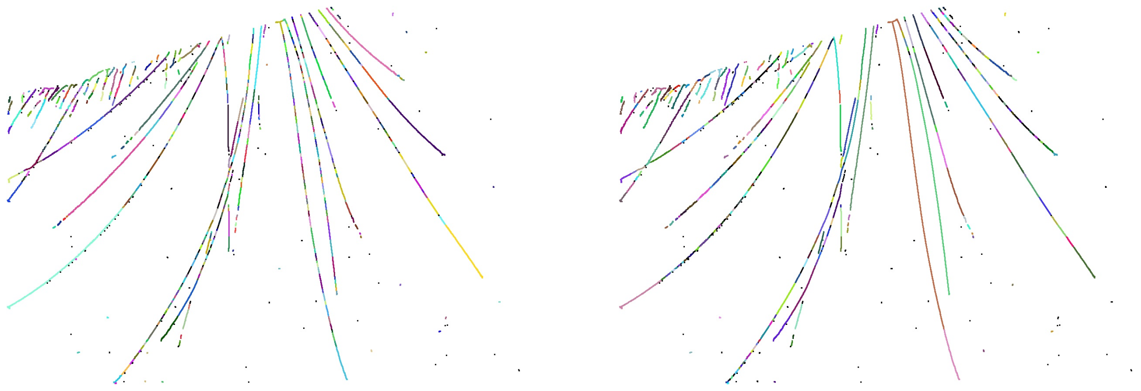

(a) $epsilon = 4, minPoints = 3$ (b) $epsilon = 5, minPoints = 4$

Figure 4.5: DBSCAN whisker clustering for two sets of parameters

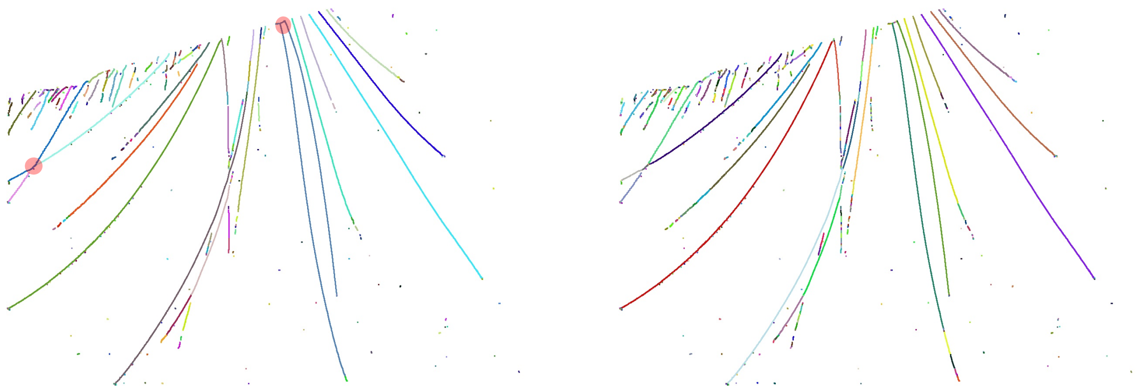

(a) WhiskEras original implementation (b) Using an angle difference limit

Figure 4.6: Steger Clustering

is clear that this method is much more suitable for identifying lines. Also, by modifying the Steger Clustering implementation used in WhiskEras and providing an angle-difference limit between neighbour points (set at 7 degrees), errors of clustering parts of different whiskers together (Figure 4.6a), no longer occur (Figure 4.6b). These results demonstrate the superiority of Steger Clustering for the task at hand.

4.2.4. Stitching I

Stitching I is responsible for connecting clusters that belong to the same whisker but are seperated either due to intersections between whiskers or noise. For a cluster, all other clusters which are rooted (their closest point to the snout) further than its tip (with regards to to the snout) are rotated towards its tip. If the two clusters are close enough in terms of distance and angle, a stitching score is computed. The cluster with the best (lowest) score is then stitched to this cluster. This procedure is repeated for all clusters in a frame.

The parameters in Table 4.4 are all very difficult to tune. There are always trade-offs to be made: increasing *maxX*, *maxY* and *maxAD* may allow some additional correct stitchings but can also force erroneous stitchings. Furthermore, parameter *c* can be increased to address intersecting whiskers but that can potentially split some whiskers into smaller segments. It appears that some issues are unavoidable, particularly in frames with multiple intersections. More details on this are given in Section 4.3.

4.2.5. Stitching II

This stage addresses cases of whiskers being hidden behind other whiskers. Whiskers which are rooted far from the snout are stitched to parts of other whiskers which are closer to the snout. For example, Figure 4.7 shows a part of the pink whisker in the middle of the frame that, after Stitching II, also belongs to the dark

Name	Functionality	Effect	Range	Step
minClusterSize	filters clusters with few points	less noise and clusters	4 – 10	2
lengthOfTipAndBottom	sets length of cluster edge to extract angle from	favours straight-line whiskers	10 – 30	5
maxX	sets max cluster distance parallel to whisker line	favours intersections, but prone to errors	100 – 600	50
maxY	sets max cluster distance perpendicular to whisker line	negates noisy edges, but prone to errors	5 – 20	2
maxAD	sets max cluster angle difference (degrees)	negates noisy edges but prone to errors	7 – 20	2
c	adjusts angle weight, relative to distance	better for intersections, but worse elsewhere	0 – 1000	50

Table 4.4: Stitching I parameters

Name	Functionality	Effect	Range	Step
minClusterSize	filters clusters with few points	less noise and clusters	15 – 30	3
lengthOfTipAndBottom	sets length of cluster edge to extract angle from	favours straight line whiskers	10 – 30	5
highestRoot	sets distance from snout which allows stitching II	better for videos with obstructed part of snout	20 – 200	20
acceptableXdistance	sets max cluster distance on x axis	more stitching, prone to errors	30 – 80	10
acceptableYdistance	sets max cluster distance on y axis	more stitching, prone to errors	7 – 14	2

Table 4.5: Stitching II parameters

green whisker.

Stitching II parameters are a bit easier to tune than Stitching I. However, in the case of *highestRoot*, some video setups can be tricky. Figure 4.8 illustrates such a case, where part of the snout is obstructed. When $highestRoot = 60$, two whiskers on the right of the image are filtered out, contrary to when $highestRoot = 200$. This high value could spawn an error in different cases though.

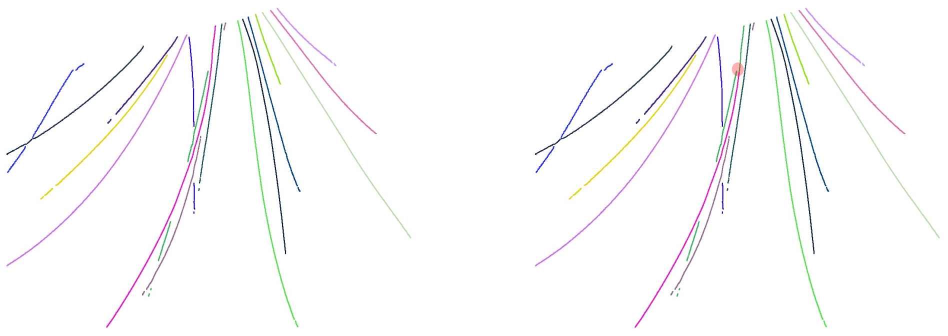
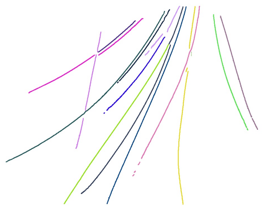

(a) Input (b) Output

Figure 4.7: Stitching II effect

(a) Unprocessed Frame


(b) Stitching II output, $highestRoot = 60$

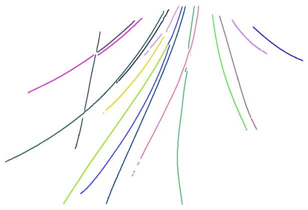

(c) Stitching II output, $highestRoot = 200$

Figure 4.8: Stitching II results with varying highestRoot

4.2.6. Parameter Fitting

This final detection step gets each cluster's points and outputs a parametrization for each of them, by fitting them on a 2nd degree curve $ax^2 + bx + c$. Each whisker can now be represented by only 4 parameters: pivot-point position c, angle b, bending a and length L. The length is computed by integrating over the whisker.

Parameter fitting parameters, shown in Table 4.6, only have to do with filtering clusters that are either likely not whiskers or the result of wrong clustering/stitching. They are fairly standard, meaning they can mostly make right decisions in rejecting clusters for a general configuration, even under different video setups.

The *minLength* parameter requires more delicate handling. For example, in Figure 4.9, many whiskers which are rooted to the mystacial pad have comparable length to hair from the animal's fur, rooted higher on the snout. It is impossible to keep every whisker while discarding the hair, in the current algorithm. One solution would be to keep both and let the neuroscientists discard the hair from the final tracking results.

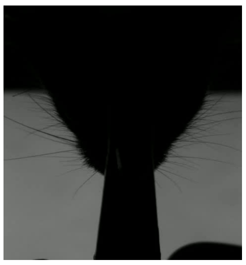


Figure 4.9: Frame in which whiskers have comparable length to hair

Name	Functionality	Effect	Range	Step
minNrOfDots	filters clusters with few points	less hair and whiskers	30 – 50	5
minLength	filters clusters with small length	less hair and whiskers	50 – 150	20
minAngleToSnout	filters hair (almost) parallel to snout	less hair and whiskers	2 – 10	2
maxBend	filters erroneous clusters	less whiskers	0.01	–
maxMSE	filters erroneous clusters	less whiskers	80 – 400	40

Table 4.6: Parameter fitting parameters

4.3. Whisker Detection Limitations

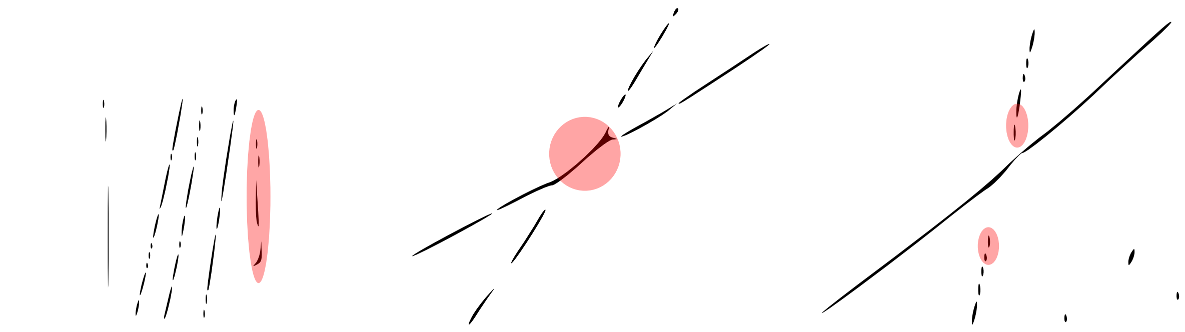
As has become apparent from the above analysis, WhiskEras has a few weaknesses, particularly revolving around intersecting whiskers. Overall, the preprocessing and parametrization stages do a reasonably good job without much parameter tuning. Most issues stem from the whisker-point detection stage (Section 4.2.2) which feeds an imperfect input to the clustering and stitching stages, further down the pipeline.

The main issues of Steger's algorithm, used to extract the whisker centerlines are the following:

1. Faint whiskers are only partially recovered and they can be noisy (Figure 4.10a).

2. Intersecting whiskers that are not interrupted have a unified part of noisy points near the junction centre that does not clearly belong to either of the two whiskers (Figure 4.10b).

3. Intersecting whiskers of lower intensities are being interrupted. This results in whiskers being continued as far as hundreds of pixels away in the frame. To make matters worse, the edge parts of the interrupted whiskers are often noisy (Figure 4.10c).


(a) Right line is not fully recovered because it is faint. It is also extremely noisy.

(b) Noisy points around the junction of two intersecting whiskers. A segment seems to belong to both whiskers.

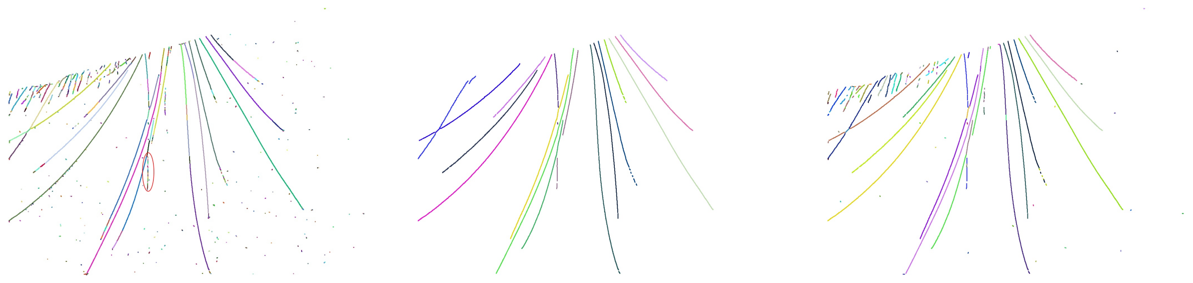
(c) Interrupted whisker. The edge on top is noisy, i.e it does not point to the other section of the whisker.

Figure 4.10: Whisker point detection issues.

Note that Figure 4.10 is not the real image of detected whisker points. It is rather the real image's conversion from *raster* to *vector* graphics, using *Inkscape*. This was done for clarity, as the real image's quality is repulsive. This also reveals another issue, perhaps the root of all problems: the centerlines detected are far from smooth. That is probably caused by the compression algorithm which was used to shrink video sizes for convenient storing, in Erasmus MC. This compression will not take place for future videos, but nothing can be done for older videos.

Hence, the input to the clustering stage is cumbersome. Then, the issue with clustering is that it forms too many groups of points, typically from 100 to 1000. This means that stitching will have to do significant work in order to form groups of complete whiskers. However, it was already mentioned that stitching requires specifying a handful of parameters that are extremely hard to generalize even for a video alone, let alone multiple videos with different setups.

Stitching I, in particular, has to deal with a multitude of complex cases. For example, it has to stitch together whiskers that are split into multiple parts and interrupted parts of whiskers due to intersections. In the first case, the distance factor has to be weighted more heavily. However, in the second case, the cluster's orientation has to be the prime factor. This is illustrated in Figure 4.11, where for $c = 1$, an interrupted whisker part (circled in red in 4.11a) is wrongly stitched to the brown cluster (4.11b). For $c = 700$, the algorithm seems to *begin to understand* where it should be stitched to, but now multiple errors have occurred due to close whisker parts not being stitched to each other.

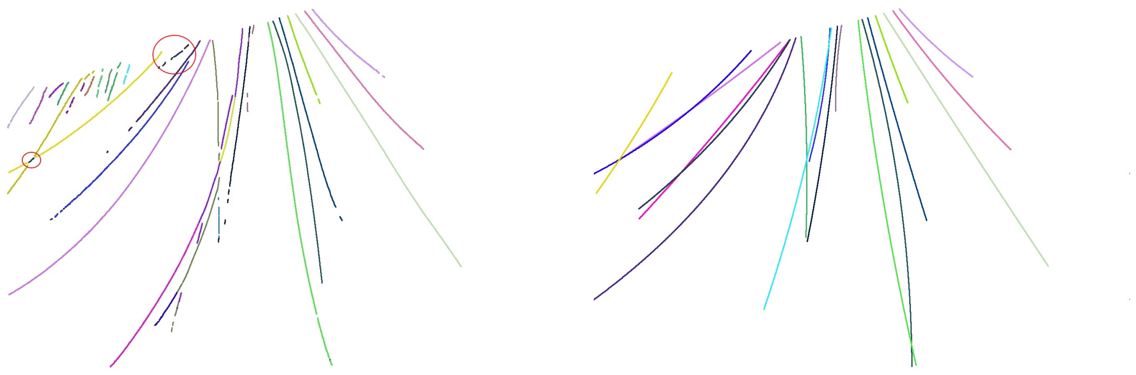

(a) Local Clustering results

(b) Stitching I results for $c = 1$

(c) Stitching I results for $c = 700$

Figure 4.11: Stitching I tuning issue 1

Another issue is demonstrated in Figure 4.12. Specifically, the small cluster circled in red in 4.12a is stitched to the other cluster circled in red which corresponds to hair. After parameter fitting, the result (4.12b) shows a pink whisker which is clearly wrong. This issue could have been avoided if the allowed distance between clusters to stitch $maxX$ was lowered. However, there may be a case where $maxX$ indeed needs to be large enough to stitch together two clusters which belong to the same whisker. This problem gets worse as the parameters would have to be tuned differently for different video dimensions and whisker sizes in general. Similar problems can occur with $maxY$ and $maxAD$ values. To sum up, Stitching I has to take a lot of difficult decisions which demand different parameter tunings. The conclusion is that, the current form of Centerline Extraction - Local Clustering - Cluster Stitching will inevitably make some mistakes over the span of a video.



(a) Results after stitching I & II (b) Results after parameter fitting

Figure 4.12: Stitching I tuning issue 2

4.4. Regarding Design Space Exploration

Design Space Exploration (DSE) could theoretically be performed at this stage. This could be done by :

1. choosing a DSE framwork for multi-objective optimization,
2. supplying it with the parameters, mentioned as impactful to the results in Section 4.2, with their corresponding boundaries and variation step and
3. choosing a good optimization criterion that would maximize whisker-detection accuracy and sensitivity across runs over different videos.

There are two issues with this approach:

1. It must have become apparent by now, that WhiskEras suffers from severe algorithmic issues, more than parameter-tuning ones. It is unlikely that a DSE could discover an optimal set of parameters, even for a single video, which would be able to deal with the problems mentioned in Section 4.3.
2. Choosing an optimization goal in this type of problem is extremely difficult. Several optimization objectives were considered such as: variation (σ) in the number of detected whiskers during the course of the video, measuring the quality of the third stage's feedback (*tracking*). For the first objective, there is no guarantee that the algorithm would not find a way to ignore certain troublesome parts of whiskers. This could make the variation of the number of detected whiskers more stable but at the same time decrease sensitivity. The second objective seems more promising, but it does rely on the assumption that tracking works well. Also, measuring the quality of tracking involves taking into account only certain parameters which affect tracking. A perfect whisker-detection algorithm would need to be accurate in terms of all the parameters set during Parameter Fitting, a goal which seems hard to achieve just by *steering* the algorithm with regards to tracking results.

To conclude this chapter, it seems more appropriate to revise certain aspects of the algorithm in order to deal with issues, such as intersecting whiskers, in a more robust way. After all, the goal is to standardize and/or automate the selection of parameters in WhiskEras so that it can be used conveniently. At its current state, it is difficult to do so. Even for a single video file, visual inspection of results showed that altering specific parameters favoured detection of certain whiskers over others, which is essentially a compromise.

5 Improvements

As shown in Section 4.3, WhiskEras has a number of shortcomings which mainly stem from the *whisker-point detection* stage. Steger's algorithm, which performs both *curvilinear detection* and *clustering* is the target for improvement. If these two steps were enhanced, then the work of *stitching* would be made easier. Furthermore, stitching also has some room for improvement, particularly related to its parameter selection.

In the rest of this chapter, the improvements which were made to WhiskEras are reported. First, the intuition behind the decision for a new detection algorithm is given, in Section 5.1. Next, the algorithm of our choice, *Improved Curve Tracing* is discussed in Section 5.2 along with its adaptation in WhiskEras and our thoughts on it. Finally, the stitching stage's enhancements are explained in Section 5.3.

5.1. Choosing a Detection Algorithm

Whisker-point detection can be placed into the broader category of *curvilinear centerline extraction.* As described in Section 3.4, there are three main approaches to address this problem.

Learning-based approaches are the most advanced ones, but as clarified before, they require a non-negligible amount of labelled data which is simply not available. **Graph-based** approaches, on the other hand, have their own shortcomings (see Section 3.4). The only remaining choice is, then, **intuitive** methods.

Going for an alternative Hessian-based approach such as multiscale vessel enhancement [23] would seem naive as it is based on the same principles as Steger's unbiased curvilinear detector [49]. Another option is to adopt a model-based approach, such as the optimally oriented flux (OOF) [35]. However, there is no guarantee that a different intuitive method would provide substantially better results than Steger's. As a reference, both multiscale vessel enhancement and OOF are reported to have comparable results in terms of accuracy, sensitivity and specificity on retinal vessel, neuron and aerial road map datasets in [55]. Furthermore, these methods do not provide a clustering method such as Steger's, to group whisker points into different clusters. This task would then have to be resolved by looking into other methods, if DBSCAN is deemed incompetent as a clustering choice (such an argument is made in Section 4.2). However, this would mean adapting a brand-new algorithm to detect whiskers and also searching for an additional clustering method. Considering the uncertainty of such an approach having major improvements over the existing algorithm and this thesis' time constraints, it was decided not to pursue this path. After all, WhiskEras has proven to be significantly better at whisker tracking than its competition [9]. It is therefore better to invest time and resources into improving some aspects of the algorithm and accelerating it, rather than completely reinventing its core stages.

The method that was chosen is K. Raghupathy's *Improved Curve Tracing in Images* [45]. This solution builds on Steger's algorithm and expands it to deal with the problems below:

1. Noisy images.
2. Multiple and possibly intersecting curves.
3. Disappearance and re-emergence of a curve.
4. Tracing curves that fade out.

These are more or less the issues with Steger's algorithm in whisker detection which were encountered in Section 4.3. *Improved curve tracing* makes some modifications to Steger's localized algorithm, while incorporating global features to address the problems above.

5.2. Improved Curve Tracing

Steger's original algorithm is summarized in Algorithm 1. There are two user-defined threshold parameters $tr1$ and $tr2$, which have to satisfy $|tr1| \leq |tr2|$. Low values of the second directional derivative S indicate high curvature. Pixels with $S < -tr1$ are valid curve points, though they are not necessarily present in the clustering results, unlike points with $S < -tr2$. The latter are points from which curve extension starts taking place. Each curve point has an orientation/angle (computed using Steger's algorithm) which dictates which pixels in a 1-pixel radius are potential neighbours. Linking of curve points is illustrated in Figure 5.1.

Algorithm 1: Steger's original algorithm

```
/* Curve point detection                                                */
for $(x, y)$ in Image pixels do
    Compute, perpendicular to curve, direction $n(t)$ using Hessian Matrix;
    Compute first $(F)$ and second $(S)$ directional derivatives along $n(t)$;
    if $F$ vanishes within the current pixel AND $S < -tr1$ then
        $(x, y)$ is a valid curve point;
    end
end
/* Clustering                                                           */
for pixels with second directional derivative $S$ in $[S_{min}, S_{max}]$ do
    Assign point a new cluster;
    if $S \geq -tr2$ then
        break; // when S is larger than some threshold tr2, stop creating new curves
    end
    Check for neighbours in a 1-pixel radius in the orientation of the current point;
    while there is a neighbour that is a valid curve point do
        Find best neighbour;
        Assign neighbour to existing cluster;
        Neigbour point becomes current point;
        Check for neighbours in a 1-pixel radius in the orientation of the current point;
    end
end
```

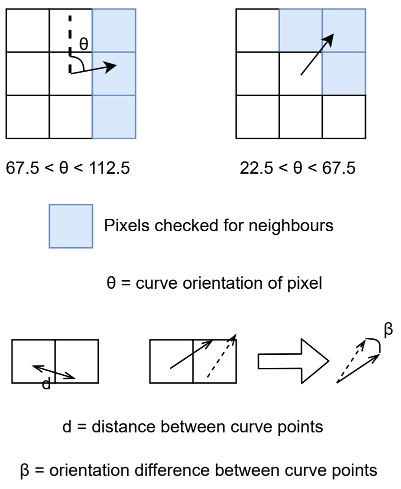

θ
67.5 < θ < 112.5
22.5 < θ < 67.5
Pixels checked for neighbours
θ = curve orientation of pixel
d
β
d = distance between curve points
β = orientation difference between curve points

Figure 5.1: Linking of curve points
Criteria to determine best neighbour are β and θ.

In **WhiskEras** [9], Steger's detection algorithm was modified (Algorithm 2). The key difference is that all points detected in the first stage are present after the clustering stage, because only one threshold, $tr2$ is used. Additionally, every curve point can peek 1 pixel further to locate a neighbour. This is beneficial because sometimes curve points of the same whisker can be more than 1 pixel apart. Finally, neighbours have to be doubly-linked to ensure reliable clustering. This means that two points can be neighbours only if both points' best neighbour is the other one.

Algorithm 2: WhiskEras adaptation of Steger's algorithm

```
/* Curve point detection                                              */
for $(x, y)$ in Image pixels do
    Compute direction $n(t)$ using Hessian Matrix;
    Compute first $(F)$ and second $(S)$ directional derivatives along $n(t)$;
    if $F$ vanishes within the current pixel AND $S < -tr2$ then
        $(x, y)$ is a valid curve point;
    end
end
/* Clustering                                                         */
for pixels in valid curve points do
    Check for neighbours in a 2-pixel radius in the orientation of point;
    Find best neighbours in both directions of the orientation;
end
Connect doubly-linked neighbours in parallel to form clusters;
```

Improved curve tracing, as opposed to WhiskEras, maintains the *curve point detection* part of Steger's algorithm as is. All improvements are incorporated into the Clustering stage. How would it then do a better job at detecting points? Remember that not every valid curve point appears at the clustered results. By improving the way curve points are connected, more curve points can be found, while leaving out most of the noise. This method complements Steger's algorithm. It is the same as Algorithm 1, but now, there are 3 more ways in which the neighbour curve points are searched, in addition to plain *Steger linking*. These are *i) curve following at junctions*, *ii) tracing faint curves* and *iii) disappearance and re-emergence of curve*. The algorithm attempts to find a neighbour by first attempting to *follow curve at junction*. If that fails, it attempts plain *Steger linking*. If this is also unsuccessful the next two options are applied. Note that Raghupathy leaves the sequence of the last two options up to the programmer. Algorithm 3 describes Improved Curve Tracing from a high-level. The introduced linking steps are described in detail below. Figure 5.2 demonstrates the superiority of Imporved Curve Tracing, compared to the Steger's algorithm, for a certain image.

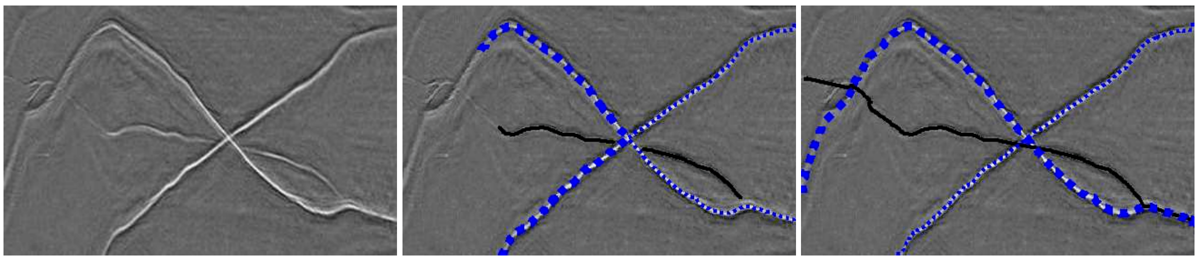

(a) Input image (b) Steger's algorithm output (c) Improved curve tracing output

Figure 5.2: Comparison of Steger's algorithm and Improved Curve Tracing

Curve following at junctions As shown in Figure 5.2, two curves can intersect with each other and sometimes Steger's algorithm fails to recognize the natural continuation of each curve (5.2b). This is solved by introducing an angle difference limit β_0, such that the angle difference $\Delta\theta$ between two neighbour points is less than β_0: $\Delta\theta < \beta_0$. This is exactly the countermeasure which was used in Section 4.2 to enhance Steger Clustering. In addition, the current point (x_1, y_1) is looking for neighbours in the direction of its angle at a

Algorithm 3: Improved Curve Tracing

```
/* Curve point detection                                                     */
// Same as original Steger's algorithm
⋮
/* Clustering                                                                */
for pixels with $S$ in $[S_{min}, S_{max}]$ do
    Assign point a new cluster;
    if $S \geq -tr2$ then
        break;
    end
    do
        Follow curve at junction;
        if no neighbour has been found then
            steger_linking; // linking that was applied in original Steger's algorithm
        end
        if no neighbour has been found then
            Trace faint curve;
        end
        if no neighbour has been found then
            Search for re-emergence of curve;
        end
    while there is a neighbour that is a valid curve point;
end
```

longer distance, determined by a factor L. In other words, possible neighbours are located as far as

$$(x_2, y_2) = (x_1 + Lcos(\theta(x_1, y_1)), y_1 + Lsin(\theta(x_1, y_1)))$$

where L decides the maximum distance to which the point looks ahead for the next curve point. The most suitable neighbour is decided as the point with the minimum $\Delta\theta$ with the current point. This ensures the most natural orientation change in the curve.

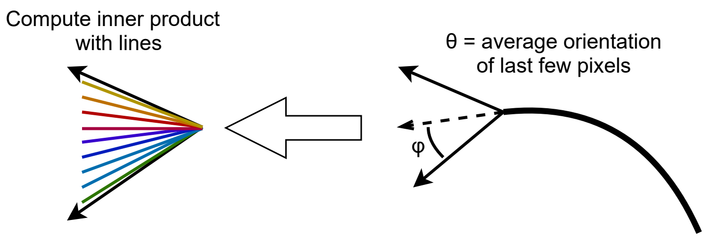

Compute inner product with lines
θ = average orientation of last few pixels
φ

Figure 5.3: Tracing curves that fade
Inner products are computed with lines oriented between $\theta - \phi$ and $\theta + \phi$.

Tracing faint curves This is the stage that incorporates global information into the Steger algorithm. It does that by using the Radon Transform, a technique described in Section 2.4. This is a more global approach, relative to the method used in Steger's algorithm, that extracts curvilinear structures by modelling their profile (linear, quadratic etc) and then calculating the inner product of the image with all possible curves. This stage uses the first-order (linear) Radon Transform. When a curve has ended, i.e no valid neighbours were found using *curve following* and *Steger linking*, we search for a potential faint curve, by computing the inner product of the image with various linear curves, with the last curve point as the starting point. This inner product is the intensity sum of the image over the faint curve. Suppose θ is the average orientation of the last part of the curve. Then, we limit our search to the 2ϕ cone $[\theta - \phi, \theta + \phi]$, as shown in Figure 5.3. We then find the angle in which the inner product is maximum:

$$\theta_0 = arg \max_{\psi \in [\theta-\phi, \theta+\phi]} \sum_{k=0}^{l-1} I(x_1 + k \cdot cos\psi, y_1 + k \cdot sin\psi)$$

If this inner product is larger than a pre-defined threshold f, this line segment can be added to the curve. The parameters introduced in this stage are: M the number of last curve points to calculate average angle θ from, ϕ the cone in which we look for faint curves, l the length of the line segment and f the threshold to accept a line segment as a valid faint curve. A different curve profile can also be used (e.g quadratic), other than the linear one.

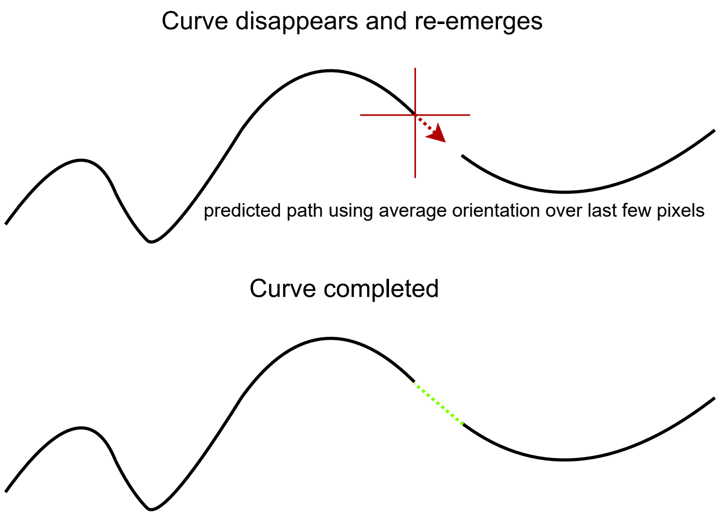

Curve disappears and re-emerges
predicted path using average orientation over last few pixels
Curve completed

Figure 5.4: Tracing re-emergent curves

Disappearance and re-emergence of curves If the curve is neither terminated nor fading, there is a third possibility: the curve is disappearing and then re-emerging some pixels further (Figure 5.4). A re-emergent curve is searched in the average orientation of the last M points of the curve, up to K pixels away:

$$(x_2, y_2) = (x_1 + Kcos(\theta(x_1, y_1)), y_1 + Ksin(\theta(x_1, y_1)))$$

Note that this is the same procedure as in *curve following*, except the orientation is averaged over the last pixels and $K > L$. However, instead of choosing the point with the minimum angle difference $\Delta\theta$, the best neighbour is chosen using a weighted score of both angle difference and strength μ (image intensity):

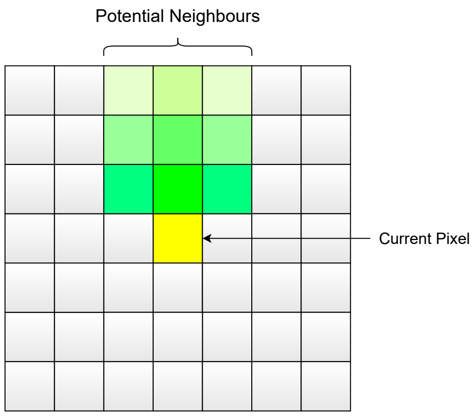
$$(x_2*, y_2*) = arg \min_{(x,y)\in S} (w \left|\theta(x, y) - \theta\right| + (1 - w) \left|\mu(x, y) - \mu\right|)$$

where S is the set of all possible neighbours and w is the weight attributed to angle difference relative to strength.

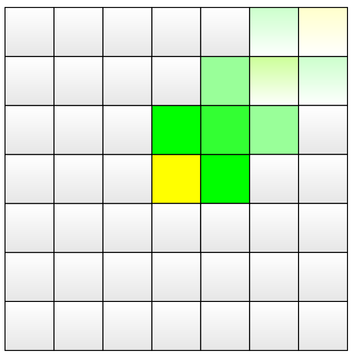
To compare Steger's algorithm to Improved Curve Tracing, the input image of Figure 5.2a was processed by both in [44]. It can be seen that Steger's algorithm makes a clustering error after the junction point of the big blue dotted and the small blue dotted clusters, while curves that fade out, disappear and re-emerge are not fully detected (Figure 5.2b). Improved curve tracing alleviates these issues, as shown in Figure 5.2c.

5.2.1. Adaptation

The actual implementation is largely inspired by Improved Curve Tracing but has some distinctions. The final modifications to Steger's algorithm were derived after thorough evaluation and testing. Following, the key elements of our adaptation of Improved Curve Tracing are explained.



(a) Upward neighbour search ($-22.5^\circ < \theta < 22.5^\circ$)



(b) Diagonal neighbour search ($22.5^\circ < \theta < 67.5^\circ$)

Figure 5.5: Pixel peeking ($L = 3$) - Current pixel (yellow) is looking for neighbours.

Pixel peeking *Steger linking* was modified to have a variable radius of pixels, L. This was a combination of Betting's implementation in WhiskEras (where $L = 2$) and the *curve following at junction* step of Improved Curve Tracing. Angle difference $\Delta\theta < \beta_0$ between neighbour points is also limited. It was discovered that a value of $L = 3$ is sufficient in the videos seen, as the gap between two whisker points which belong to the same whisker part can be as large as 3 pixels. This modified version of Steger linking was named *pixel peeking*. *Pixel peeking* is demonstrated in Figure 5.5, for $L = 3$ and two different cases of the angle θ of the current point.

Beam scanning This step is a mix of *curve following at junction* and *disappearance and re-emergence of curves*. It was named *beam scanning*, because the current pixel is scanning for neighbours in a beam-like way, i.e in a certain angle θ and up to a specific range K, as illustrated in Figure 5.6, where the current pixel (yellow) searches for a neighbour in the red pixels. The angle θ is the average angle of the last $M = 15$ points of the curve, just like in *disappearance and re-emergence of curves*, while $K = 15$. In addition, if a neighbour is found, it is checked that the new point does have a close neighbour in the angle θ, to prevent connecting to noisy points. The procedure of choosing the ideal neighbour is the same as in *pixel peeking* – a weighted score of distance and angle difference. As in *pixel peeking*, points with $\Delta\theta > \beta_0$ are discarded from candidates.

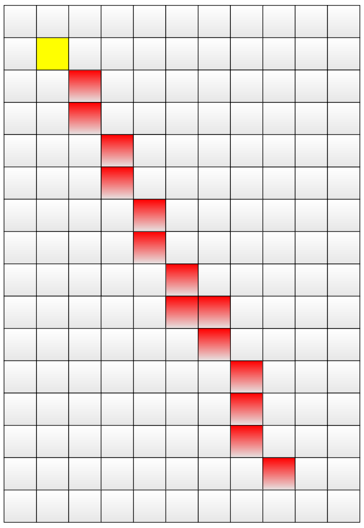


Figure 5.6: Beam scanning for $K = 15$ and $\theta = 152.6^\circ$

Algorithm 4 summarizes the logical structure of our adaptation of Improved Curve Tracing. Notice that there are two stages and two sides. When a point with a large negative S (second directional derivative) value is chosen to start forming a cluster, it is usually located somewhere in the midst of a curve. Hence, the curve needs to be extended both upwards (and right) and downwards (and left) to take its full shape. *Side* refers precisely to which direction the curve is currently being extended to. In both stages, the extension is first done upwards and when there is no further neighbour in that direction, the side is reverted. In Stage 1, only *pixel peeking* takes place. When no further connection is possible from both sides, Stage 2 begins, where both *pixel peeking* and *beam scanning* are performed. This is actually a precaution against extending noisy curve points. In particular, if the number of points (i.e cluster size) of the curve so far is lower than some pre-defined threshold, then this curve is possibly a group of noisy pixels and will not be extended.

5.2.2. Discussion

The adaptation of Improved Curve Tracing, described in Section 5.2.1 certainly leaves a lot to be desired. *"Where are the global characteristics incorporated by using Radon Transform in tracing a faint curve? How*

Algorithm 4: Adaptation of Improved Curve Tracing

```
/* Curve point detection                                          */
// Same as original Steger's algorithm
⋮
/* Clustering                                                     */
for pixels with $S\ in\ [S_{min}, S_{max}]$ do
    Assign point a new cluster;
    if $S \geq -tr2$ then
        break;
    end
    Stage = 1;
    Side = UP;
    while true do
        pixel_peeking;
        if Stage == 2 AND there is no valid neighbour AND cluster size > threshold then
            beam_scanning;
        end
        if there is no valid neighbour then
            if Side == UP then
                Side = DOWN;
                Current point = lowermost point of curve;
            else if Stage == 1 then
                Side = UP;
                Current point = uppermost point of curve;
                Stage = 2;
            else
                break;
        else
            Current point = neighbour;
    end
end
```

can a curve re-emerge after it disappears, using this implementation?" one might ask. And these are valid questions.

First of all, *Improved Curve Tracing* is used as a complete clustering technique. This means it does not need any *Stitching* thereafter, to form the complete curves. Thus, another good question would be *"Can Improved Curve Tracing surpass the performance, in quality, of the algorithm currently used by WhiskEras?"* The short answer is *no*. Careful testing showed that the results of using just *Improved Curve Tracing* are not adequate. Not only that, but using each and every aspect of *Improved Curve Tracing* would hurt *Stitching* afterwards. The details are given below.

Tracing faint curves proved to be a tall task in the whisker videos, possibly because of significant noise, with which Steger's algorithm has a hard time coping [44] [24] [55]. Although faint curves may actually be beneficial, a whisker that disappears before an intersection might start extending on the other intersecting whisker. This is an issue, as the re-emerging whisker on the other side of the intersection will never be able to reunite with its counterpart, neither during this stage nor during *Stitching I*. On a positive note though, instead of here, the Radon Transform is used to improve Stitching I, as explained in Section 5.3.1.

Handling the case of **disappearance and re-emergence of curves** is also tough. In the whisker videos, it is not rare that the curve re-emergence happens more than a 100 pixels away. Note that Raghupathy suggested using $K = 35$ in *Improved Curve Tracing*, a value far smaller than what is needed in this application. There are two dangers in increasing the value of K to such amounts. i) The angle in which we search might not be quite right in locating the next curve point, the further away we search for (remember the name used here – *Beam scanning*). ii) The noisy points which are detected near intersections, as mentioned in Section 4.3, also have a distorted angle. Thus, it is possible that the disappearing whisker gets connected to a different whisker, if K is large. Using curve strength (image intensity) in choosing the right neighbour did not help much either, as the whiskers have relatively comparable strength.

Having analyzed that, the adaptation of *Improved Curve Tracing* might come off as a huge failure. However, this implementation improves the quality of Steger Clustering significantly. It does that, largely by using *beam-scanning*, which contributes to forming complete whisker parts, where they do not disappear. As a result, the clusters formed after this stage are fewer than in WhiskEras, closer to the real number of whiskers. This is great for *Stitching I*, as it does not need to take so many difficult decisions, as seen in Section 4.3. Furthermore, it is easier to filter out noise during Stitching I by increasing the *minClusterSize*, which discards clusters with less points than this value, since whisker parts are usually large already, after this stage.

Name	Functionality	Effect	Range	Step
L	pixel peeking radius	not very impactful	2 – 5	1
β_0	prevents steep angle changes in whiskers	more reliable linking, but more clusters	5 – 20	2
M	number of points to compute average angle	not very impactful	5 – 20	2
K	beam scanning range	less clusters but more errors	10 – 40	5
beamThreshold	minimum cluster size permitting beam scanning	less noise and whisker points	3 – 15	2

Table 5.1: Improved curve tracing parameters

Improved curve tracing also introduces some parameters, shown in Table 5.1. Although it might seem that a whole lot more of tuning needs to be done, it was found that fixing these values, with only small oscillations, did not harm performance in any of the videos seen. Nevertheless, a suggested tuning range is given, in case new video settings are introduced, or the user desires to explore their impact.

5.3. Stitching

Stitching proved to be a tricky stage, as there are multiple parameters that cannot be tuned to handle all cases. Hence, the focus is towards altering conditions that allow stitching to be more robust. The ultimate goal is to automate the process of stitching so that there is no need to alter a handful of parameters when dealing with different videos.

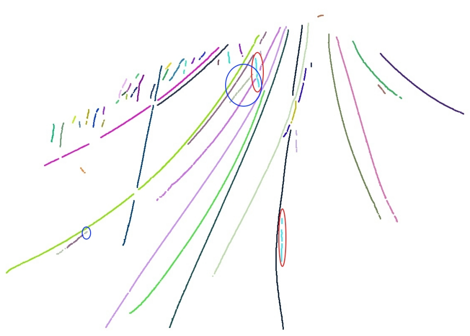
5.3.1. Stitching I

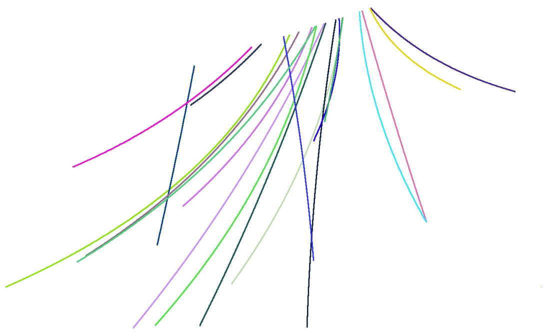
Improvements to this stage focused on eliminating parameters that were hard to tune, such as *maxX* and *maxY*, which define the maximum stitching distance, in the X and Y axis. This was done by replacing them with more sturdy parameters, as well as engineering new methods to generalize better to the desired clustering result.

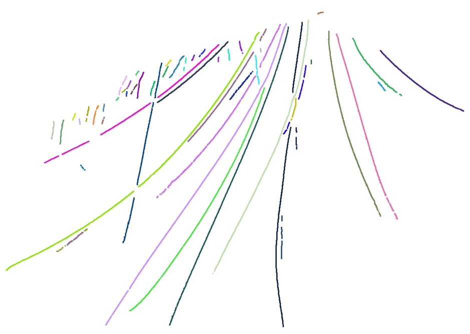
Inspired by the concept of Radon Transform, a powerful approach to reject erroneous stitchings was developed. Specifically, when two clusters are to be stitched together, an additional check is performed that involves performing a linear Radon Transform between the close edges of the two clusters that are to be stitched. This practically means that, if no whisker part is present between the two clusters, the result of the Radon Transform will be less than some pre-defined threshold $radonThreshold = 30$. Therefore, the stitching will be halted. The effect of the Radon-Transform check is demonstrated in Figure 5.7. This essentially allows us to discard the troublesome parameters *maxX* and *maxY*.

Radon-Transform check will typically not be useful for close-distance clusters. In order to further ensure that stitchings are accurate, a cone condition was added. Specifically, stitchings are only accepted within a cone of the edge of a cluster. This is depicted in Figure 5.8a.

Another issue is that the edges of a disappearing and re-emerging whisker are often noisy, as discussed in Section 4.3. In many cases, one of the two edges points towards the other, while the other is not. WhiskEras was using the parallel and perpendicular distances dx and dy from the tip cluster (closer to the snout) to the bottom cluster. These were previously compared to *maxX* and *maxY* to check if a stitching was valid. In order to compensate for the above phenomenon, these distances are now computed from each of the two clusters to the other cluster and the minimum distances are kept. The difference in distances depending


(a) Stitching I results, no Radon Transform check. The wrongly stitched clusters are circled in red and blue.


(b) Parameter fitting results, no Radon Transform check. The previously stitched clusters form erroneous whiskers together.


(c) Stitching I results, with Radon Transform check

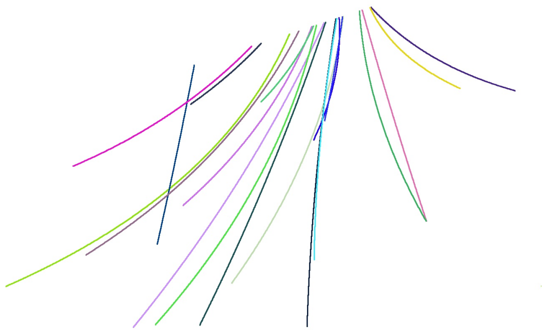
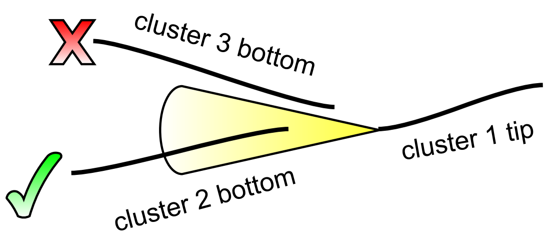
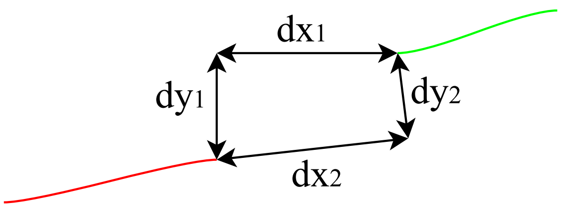

(d) Parameter fitting results, with Radon Transform check

Figure 5.7: Radon Transform check effect on Stitching I and Parameter fitting results

on the reference cluster is illustrated in Figure 5.8b. In practice, this means that there are two cones drawn from both clusters' edges. If none of the two cone conditions allows the stitching to be performed, only then it is invalidated. This was found to be advantageous in the cases mentioned, where only one of the two edges points towards the other. As for *maxAD*, it can be safely fixed to a large value, e.g $maxAD = 20°$. That is because the maximum angle which allows stitchings to take place is now largely controlled by the cone condition.



(a) Stitching I cone condition. Cluster 2 can be stitched to cluster 1, while cluster 3 cannot.



(b) The parallel and perpendicular distances between two clusters. Depending on the reference cluster, these are different.

Figure 5.8: Stitching I illustrations of cone condition and distances between clusters

Finally, the score is weighted differently than in WhiskEras. In WhiskEras, the stitching score was given by

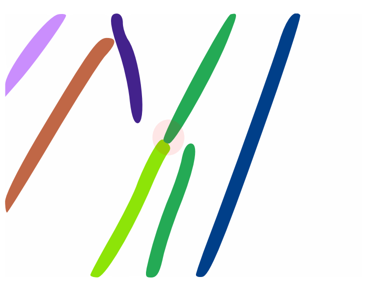
$$score = \sqrt{dx^2 + dy^2} + c \cdot \beta$$

where β is the angle difference of the edges of the two clusters and c is its weight. However, dy should be weighted more heavily than dx, especially at longer distances (larger dy means that the cluster is towards one of the edges of the cone in Figure 5.8a). The same case can be made about β. On the other hand, close whisker parts that are disconnected have noisy edges (that is why they were not connected during clustering). Thus, we should leave a little *breathing room* for dy and β to be a bit larger when two clusters are really close to each other. Taking all these into consideration, the new score is

$$score = \begin{cases} dx + dy + \beta, & \text{if } d \leq d_0 \\ dx + \frac{c}{dx} \cdot dy + c \cdot \beta, & \text{if } d > d_0 \end{cases}$$

Notice that dy and $beta$ are weighted as much as dx for small distances d_0 (e.g $d_0 = 5$). When the distance indicates that the clusters are not two close whisker parts, dy and β should be the prime factors. This necessitates the use of a large value for c (e.g $c = 1000$).

5.3.2. Stitching II


(a) old Stitching II: dark green whisker is stitched to light green whisker in an unnatural way

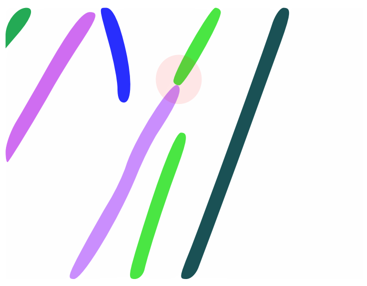

(b) new Stitching II: light green whisker is stitched to violet whisker in a natural way

Figure 5.9: Stitching II cone effect

Stitching II addresses the issue of whiskers being hidden behind other whiskers (see Section 4.2) by adding points of one cluster to another. Thus, these points end up belonging to both clusters. The improvements that were made to this stage also focused on deriving parameters that were robust and applying more natural stitchings. Specifically:

1. The parameter *acceptableXdistance* was kept, as it makes sense that a disappearing whisker should be hidden in a whisker part nearby. However, *acceptableYdistance* was not retained. Instead, a *cone* parameter was introduced, which has the same role as in Stitching I (see Figure 5.8a). The effect of this change to Stitching II is shown in Figure 5.9. New Stitching II feels much more natural.

2. A *maxAD* parameter was also added, much like in Stitching I. Here, though, the angle difference is between the *floating* whisker's edge (whisker which is partially hidden) and the whisker part's edge which is stitched to it. This rules out stitching for clusters that are very differently oriented.

3. The *highestRoot* is fixed to a small value (e.g. 10), so all whiskers that are *floating* a little further from the snout are stitched, if there is a valid candidate. To deal with obstructed snout parts (see Section 4.2), whiskers that are still *floating* after Stitching II are not discarded, but are transferred to the next stage of *Parameter fitting*. Instead, their distance from the snout is added to their length, to compensate. This is done because it was discovered that whiskers that are far from the snout are always stitched if they are indeed hidden, while noisy curve parts are just filtered out with *minClusterSize*. Thus, when a whisker part is still floating at the end of Stitching II, it is due to an obstructed snout part.

5.3.3. Discussion

Improvements to Stitching I and II introduced some new parameters, while discarding other, inconvenient parameters, as shown in Table 5.2. It was found that the new parameters can generalize much easier to the whisker videos in the Erasmus MC dataset.

Furthermore, the *Preprocessing* parameters *imdilate_value* and *cutoff* can be fixed to small values (e.g 10) as small hair is not an issue anymore in the Stitching stages. This breakthrough can be attributed to the introduction of the Radon-Transform check.

Name	Functionality	Effect	Range	Step
		Stitching I		
cone	sets cone condition's angle	more stitchings	7 – 20	2
radonThreshold	sets radon transform intensity threshold	less stitchings	20 – 40	5
radonLength	sets curve length of Radon Transform	relative to radonThreshold	10 – 30	5
d_0	sets close clusters' distance	more stitchings	2 – 10	2
~~maxX~~				
~~maxY~~				
		Stitching II		
cone	sets cone condition's angle	more stitchings	7 – 20	2
maxAD	sets acceptable angle difference	more stitchings	12 – 25	2
~~acceptableYdistance~~				

Table 5.2: Improved stitching parameters

6 Acceleration

Another serious drawback of WhiskEras is performance in terms of speed. It is coded in Matlab, a high-level programming language which is great for developing algorithms and exploring multiple approaches. However, when performance is desired, a compiled programming language has to be used.

It was decided to port the Matlab code, together with the improvements from Chapter 5, into C++. The reason is that C++ can be low-level with great compiler-optimization support, while also incorporating some fast libraries like STL and BOOST, which can speed up the application further. In a sense, C++ is a superset of C [48]. Although writing the code in C was also an option, some of the libraries that were used during development were only available in C++. Hence, the code was mostly written in C-style, while taking advantage of some of the options that C++ offers.

Using C++ provides a big boost to the performance of WhiskEras, but, still, the code is run by a single core on the CPU. Given that the goal is to reach real-time processing performance, parallelism is a feature that our application must take advantage of. Thus, multi-core processing using OpenMP, as well as GPU acceleration using CUDA were deployed for parallel execution.

Coding is an art by itself and writing a well-structured, sustainable, error-free and efficient code is a very demanding task. This chapter does not aim to describe each and every detail of this long procedure, but rather focuses on the most interesting and successful acceleration techniques which were used.

In the rest of this chapter, the algorithm's parallelism and cost overview is summarized in Section 6.1. Section 6.2 refers to acceleration techniques used in Improved Curve Tracing. Then, the alternative options which were examined to speed up the *Learning* stage are pointed out in Section 6.3. Finally, some other libraries which were used during the project are mentioned in Section 6.4.

6.1. Overview

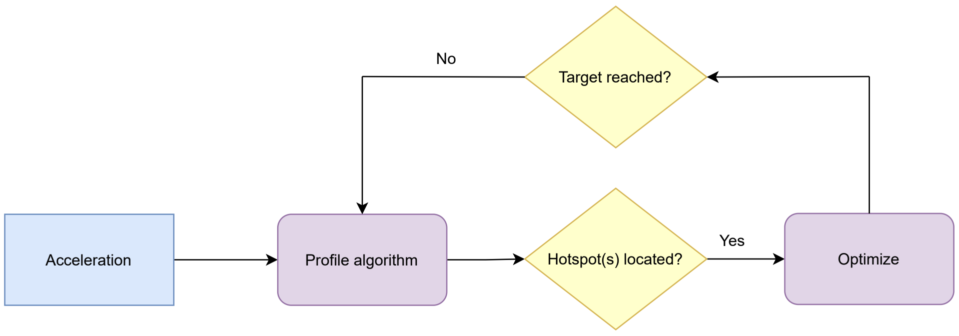


Figure 6.1: Acceleration process of WhiskEras

Accelerating an algorithm requires thorough profiling at all steps. After porting the whole code to C++, the algorithm was profiled and the stages with the heaviest workload were targeted for speedup. This is a repetitive procedure: the algorithm is profiled, its hotspots are located and then optimized, as depicted in

Figure 6.1. A good *breaking* point (i.e point to end the acceleration process) is either when the programmer has reached their goal or further speedup is hard to achieve. At this point, there ought to be multiple stages which take comparable time to complete.

This section focuses on how much parallelism can be extracted over the algorithm's different stages as well as their complexity. These are probably the most important aspects when it comes to speeding up an algorithm and recognizing its limits.

6.1.1. Parallelism

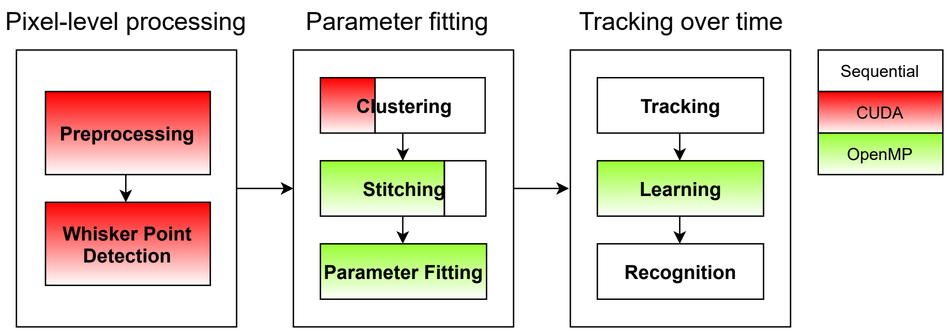


Figure 6.2: Overview of Improved WhiskEras acceleration in each of its each stages and substeps.

Figure 6.2 depicts the acceleration performed in Improved WhiskEras in each of its stages. Pixel-level processing consists of image processing operations. As expected, it involves massive parallelism which is taken advantage of, using CUDA. The previous version of Clustering could be almost fully parallelized on the GPU. The improved version, sadly, requires sequential code due to the way clusters are formed. By reformulating part of the problem, though, it was possible to move some of the costly operations to the GPU. In Stitching, multiple cores are used to compute rotation data, stitching scores (Stitching I) and also rotate clusters (Stitching II). Next, Parameter fitting uses *whisker-level parallelism* to perform non-linear fitting on each of the whiskers, using OpenMP. Tracking and Recognition are not particularly costly substeps, so they were not investigated for further speed-up. Learning though, involves training SVMs, using a a one-vs-one classification strategy [6]. This constitutes an especially heavy workload, so we ended up using an efficient library (*libLinear*) as well as multicore parallelism to train each instance of the one-vs-one classifiers.

The CUDA kernels used to compute results on the GPU parallelized the image-processing operations to N threads, where $N = Width \times Height$ of the image. This was done by using 2D blocks of threads. The only exceptions are the separable convolutions (Section 6.2.1) where the optimized code from NVIDIA uses less threads to compute more elements per thread.

6.1.2. Complexity

The computational complexity of each substep is given in Table 6.1. This is not necessarily indicative of how much execution time each stage takes, as some steps involve more costly operations. It does, however, reveal how the problem scales. Important sizes are the image dimensions n, the *guassianKernelSize* m (whisker-point detection parameter), the number of whisker points found u, the number of clusters formed before stitching I p and before stitching II q and the number of whiskers found w. Indicative values, for the above sizes, are: $n = 307200$, $m = 20$, $u = 6000$, $p = 70$, $q = 30$ and $w = 15$. The Whisker-Point Detection complexity can be lowered using special techniques, as will be explained in Section 6.2.1.

The image dimensions usually also indicate a larger number of whisker points and clusters. Note that this is true for the setups in Figure 2.4. However, this might not always be true: it largely depends on the resolution and how much the image is focused on the whiskers. In general, the more whiskers showing in the video, the larger the above sizes will get.

Naturally, stages that are performed in parallel decrease the time complexity, to a theoretically smaller one. For instance, Preprocessing and Whisker-Point Detection would have a complexity of $\mathcal{O}(1)$ and $\mathcal{O}(m^2)$, respectively, if the GPU was able to run N threads at once. The same applies to the Learning step: if K CPU threads are available, the complexity would be $\mathcal{O}(\lceil \frac{w^2}{K} \rceil)$. This is unrealistic though because: i) GPUs cannot run that many (millions) threads in parallel at once even at full occupancy and ii) there are other limiting factors such as memory bandwidth, throttling, control hazards etc. that apply to both the CPU and the GPU.

Stage	Complexity
Preprocessing	$\mathcal{O}(n)$
Whisker-Point Detection	$\mathcal{O}(n \cdot m^2)$
Clustering	$\mathcal{O}(u \cdot log(u))$
Stitching	$\mathcal{O}(p^2 + q^2)$
Parameter fitting	$\mathcal{O}(w)$
Tracking	$\mathcal{O}(u \cdot w)$
Learning	$\mathcal{O}(w^2)$
Recognition	$\mathcal{O}(w)$

Table 6.1: Stage complexities

In these cases, the true complexity lies somewhere in between.

Consequently, it is fair to say that the algorithm's cost is not too sensitive to increasing image dimensions, because image processing is performed on the GPU entirely. On the other hand, more whisker points, clusters and whiskers would impact cost more heavily, since the corresponding stages do not exhibit the same amount of parallelism.

6.2. Improved Curve Tracing

Improved Curve Tracing consists of the Whisker-Point Detection and Clustering steps. It is essentially Steger's algorithm with the improvements described in Section 5.2. Following are the most crucial techniques which were used to speed up these two steps.

6.2.1. Separable Convolution

Extracting the whisker centerlines requires computing the Hessian Matrix. To do this, we need to convolve the image with 5 different Gaussian kernels: G_x, G_{xx}, G_{xy}, G_y, G_{yy}, where G_i is the Gaussian derivative along the i axis and G_{ij} is the Gaussian derivative along the i axis, followed by the derivative in the j axis. This is the most expensive part not just of this stage, but of the entire algorithm. As shown in Table 6.1, it yields a $\mathcal{O}(N \cdot M^2)$ complexity, where $m = 20$ is the Gaussian Kernel size. In comparison, the rest of the steps in center-line extraction have an $\mathcal{O}(N)$ complexity and were found to be about 100 times faster, when run on the GPU, compared to the convolutions, on the tests run.

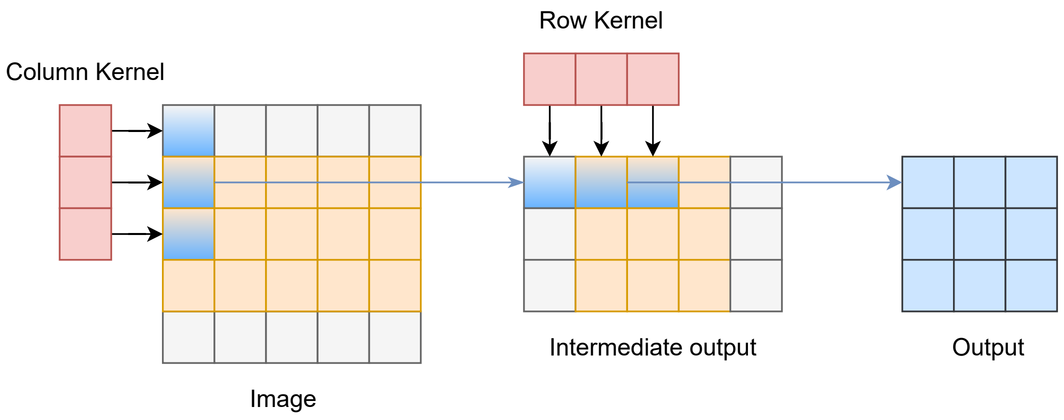


Figure 6.3: Separable convolution of a 5 × 5 image with a 3 × 3 kernel

When a matrix (in our case, the Gaussian kernel) has a rank of 1, then by using SVD (Singular Value Decomposition), two vectors can be found whose outer product is equal to this matrix. In this case, the kernel is separable. A separable kernel example is

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

All kernels G_x, ..., G_{yy} used to compute the Hessian Matrix are separable. This means that instead of carrying out normal convolutions, we can perform spatially separable convolutions and obtain the exact same results [54]. In separable convolution, the input image is first convolved with the row vector across all its rows to produce an intermediate image. Then, the new image is convolved with the column vector across all its columns to output the final result. The order (column-row convolution) can be reversed. This is demonstrated in Figure 6.3. The total complexity is now reduced to $\mathcal{O}(n \cdot 2m)$ (for square kernels - as in our case).

These separable convolutions are performed in the GPU using CUDA. The Separable Convolution sample from the NVIDIA CUDA Toolkit Samples [39] was adopted to carry out all 5 of the separable convolutions. The toolkit provides extremely optimized code such as loading *halo* elements into the GPU's shared memory. These are the elements of the image that are used multiple times during convolution and, thus, can benefit from instant memory accesses. Shared memory provides faster accesses than the GPU's default global memory. Using this toolkit saves the programmer's valuable time, not having to come up with these optimizations on their own. In addition, the sample was modified to handle variable-sized shared memory and even-sized vector-kernels due to the application's needs.

6.2.2. Sorting on the GPU

According to Algorithm 3, there is a loop, iterating over the second directional derivative values S of the whisker points, from the largest negative value to the largest positive value. Initially, this was done by locating the minimum at every step of the loop. Naturally, this approach proved to be really slow. Hence, it was decided to sort these values before entering the loop. Then, the minimum could be accessed by simply accessing the next element of the sorted array, at each iteration. This is why the complexity of this stage (Clustering) is characterized by the radix-sort (average) complexity of $\mathcal{O}(u \cdot log(u))$, where u is the number of whisker points. Then, this was taken a step further by carrying out sorting on the GPU. For that, the extremely powerful and efficient *CUB library* [1] was used. CUB's *DeviceRadixSort* class was used to sort the S values for all whisker points which were candidate starting points to form a cluster. In other words, for all points which satisfy $S < -tr2$. Overall, this provided a significant speedup to the Clustering stage.

6.2.3. Transfering Data Between Host and Device

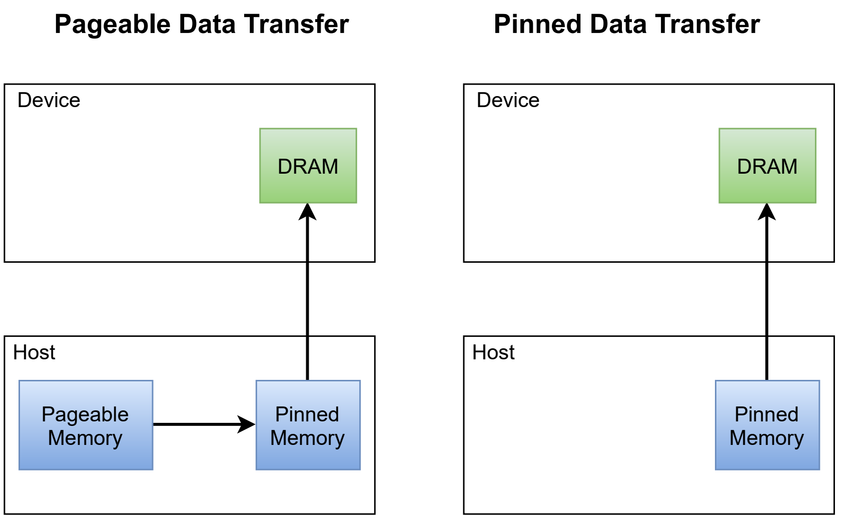


Figure 6.4: Pageable data transfer vs Pinned data transfer [26]

In order to optimize some of the transfers between the Host (CPU) and the Device (GPU), pinned memory on the side of the Host was used [26]. Specifically, some fixed-sized structures, like the preprocessed image, were allocated in pinned memory on the side of the Host. This increases the bandwidth of repetitive transfers between the Host and the Device considerably. As illustrated in Figure 6.4, when data is allocated on the Host's pageable memory, an additional step is required to transfer it to the Device. This extra step is eliminated when the data is in the Host's pinned (*page-locked*) memory.

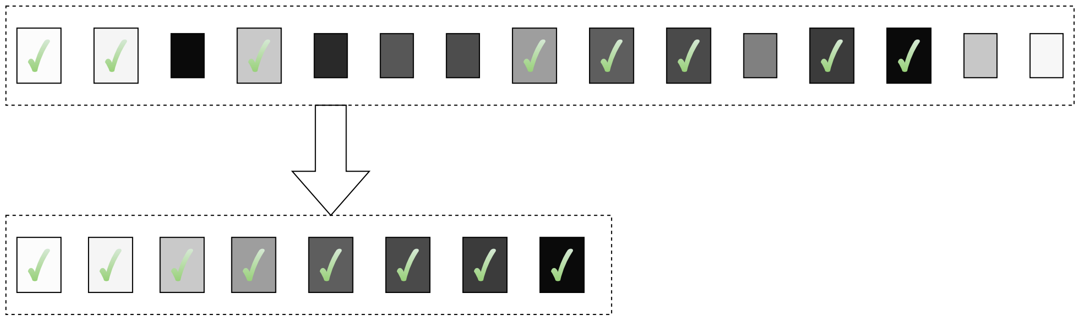


Figure 6.5: CUB DeviceSelect functionality [1]
Array elements are discarded according to the binary values of another equally sized array of *flags*.

Furthermore, the whisker points' angles and second directional derivative values are required in the Clustering stage. Originally, these were transferred to the Host in arrays with as many entries as image pixels. Non-whisker point pixels were marked with some pre-defined value e.g -1, for distinction. Again, this was a naive approach. Instead, another image-sized array was used on the GPU to indicate the presence of whisker points. Then, CUB's *DeviceSelect* class was used to discard non-whisker points from the arrays of interest and keep whisker points only. The functionality of this class is illustrated in Figure 6.5. Consequently, the size of these results, which are transferred from the GPU to the CPU, decreased dramatically.

6.3. Training SVMs

During the Learning step, the SVM is trained over a randomized subsample of the data (frames) seen so far. This does not occur at every frame. Its frequency is selected by the user. Testing and evaluation so far has shown that once every 5 frames is a good choice, but it could potentially be lowered to every 10-20 frames. As expected, training is costly in terms of execution time, as the SVM could be using data from about 500 previously seen frames (again, this is user-selectable). Four different libraries were employed for training (and testing) the SVM. Results indicate there is a clear winner in terms of performance, the *libLinear* library. Furthermore, it provides the most accurate results, only contested by *libSVM*.

libSVM This is one of the first state-of-the-art libraries for SVMs [15]. It provides several options such as kernel type and one-vs-one multi-class classification during training. The radial-basis function (RBF) kernel was extensively tested for a wide range of parameters, but was not found to provide better results than a linear-function kernel, used in WhiskEras. Furthermore, this library is fairly inefficient, even when modified to perform multi-class training in parallel, compared to other options.

gpuSVM This library [36] performs training (and testing) on the GPU. However, it was quickly realized that this approach is more suitable towards training large datasets and also testing a lot of data at once. The online method that is required in WhiskEras demands training multiple binary classification problems, since there are many classes/whiskers. As a result, running all these trainings on the GPU, one after another, is inefficient. It is rather desired to split the multi-class problem into multiple one-vs-one binary problems that are trained in parallel (*task-level parallelism*).

sgdSVM A very different approach to solving the SVM classification problem is followed by Stochastic Gradient Descent SVM. Stochastic Gradient Descent (SGD) is traditionally used in Neural Networks. This library performs optimization over a loss function by using a learning rate and SGD to construct the SVM decision boundaries. A good explanation of this method can be found in [5]. It was chosen because of its efficiency, reported to surpass even the performance of libLinear [10]. However, the quality of the results was far from satisfactory. It should be noted that this method could be tested more extensively to see if it could indeed be a good alternative option. For this project, libLinear was found to be good enough.

libLinear From the authors of libSVM, libLinear [22] offers more options to SVM Classifiers which use linear kernels, while being much faster. This is the best choice that was tested and it does not lag behind the other WhiskEras stages in performance. Note that the out-of-the-box multi-class solution offered is one-vs-all, which gave substantially worse results. Instead, multiple one-vs-one classification problems were created manually and were trained in parallel using OpenMP.

6.4. Other Libraries

Code efficiency often comes down to using fast libraries. A few other libraries which were used during the project are mentioned below.

6.4.1. OpenCV

The OpenCV libary [11] is a popular Computer Vision and Machine Learning library, with a C++ interface, among others. It was used in this project to read video frames and perform image processing both sequentially and in parallel on the GPU (it has a CUDA module), particularly during Preprocessing. Its performance and ease of use make it a an excellent choice, although it does have a slight overhead when reading frames from a video. Even though this can probably be dealt with using clever tricks, it was not that significant to the total algorithm's cost.

6.4.2. Eigen

The Eigen library [25] is an efficient and easy to use linear-algebra library for C. It performs vectorized operations on vectors and matrices, while using cache and loop-unrolling optimizations. It also provides useful numerical solvers and related algorithms.

Eigen was extensively used in this project during multiple stages. Mostly, it was employed to perform operations on the clusters' whisker points, such as computing rotation data during Stitching. Some of its out-of-the-box algorithms were also used during non-linear fitting (Parameter fitting), e.g QR decomposition using the *householder* function.

7 Evaluation

The improvements on WhiskEras look good both on paper and on visual inspection of a limited number of frames. However, such an evaluation, which was used during development, is insufficient. A more systematic way of testing the performance of the Improved WhiskEras is required. This is the topic of Section 7.1. Then, in Section 7.2, the performance of the new version of WhiskEras, accelerated as described in Chapter 6, is evaluated on powerful hardware. Where appropriate, comparisons to previous versions are made and timing profiles are given.

7.1. Algorithmic Improvements

Overall, the improvements are believed to have made the algorithm more consistent in locating whiskers. This is expected to have a positive impact on accuracy (less false positives) and sensitivity (less false negatives). This does not mean that the improved version outperforms the original version at every single frame. On the contrary, it is often the case that the improvements output similar results to before, especially for videos of the setup depicted in Figure 7.1b. However, when more whiskers are available for tracking (Figure 7.1a) and intersections are frequent, results are much more consistent in the improved version. Furthermore, the algorithm appears to perform well under a general set of parameters, for different video settings.

During development, visual inspection was used to evaluate the enhanced quality of the results. However, in order to quantitatively verify and report the improvements, an automated way had to be devised to scan through large video segments and provide strong indication of the superiority of the advanced detection module. Fortunately, Betting [9] has already come up with a few criteria. These make use of the tracking module to verify that WhiskEras is correctly following whisker movement. Thus, we will use them to evaluate the improved Whiskeras' performance.

7.1.1. Video Selection and Parameter Tuning

The videos used to evaluate the improvements are shown in Figure 7.1. Video A's setup makes it feasible to track up to about 18 whiskers. This presents challenges, as whiskers often intersect or hide each other. At the same time, the whisker that is shown intersecting with multiple whiskers in a very different angle than neighbouring whiskers, at the middle of the frame, is probably a superciliary whisker. These whiskers stem from the eyes (see Figure 2.1) and they are not of particular interest to the neuroscientists. However, it was deemed important that they are located accurately, otherwise their parts may be mixed up with other whiskers, in the Stitching stage. In the final results, they can be easily removed or disregarded.

In Video B, we only track the whiskers on the right side of the snout (left on the image). The camera is not focused on our area of interest and as expected, less whiskers are visible. This makes for a less compelling candidate to the improvements made, as intersections and occlusions are less frequent. Regardless, it is still interesting to report whether there is any meaningful difference between the two versions, the original and the improved one.

The detection parameters for both the improved and the original WhiskEras were tuned using i) visual inspection and ii) tracking results over a short duration of the video. Visual inspection provides a good basis for choosing good parameters. Tinkering those base values using tracking did not affect the results significantly, as there are usually trade-offs: a whisker is detected more frequently over another. This makes a strong case

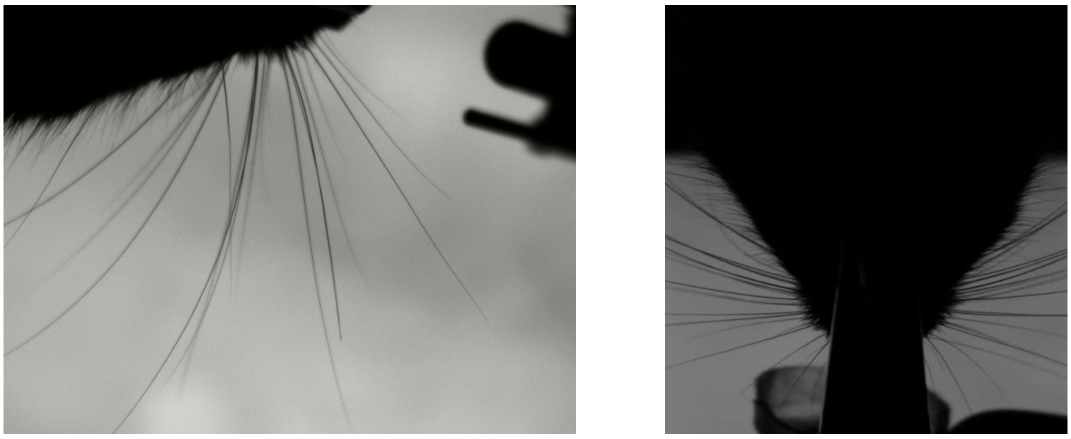

(a) Video A (b) Video B

Figure 7.1: Videos used for evaluation

for the limited potential in tuning parameters. In particular, just a *small* amount of tuning to a few parameters which can be affected by different video settings is sufficient. Note that the tracking parameters were not explored at all. Although this could yield better results, it was outside the scope of this thesis. Nevertheless, both versions will be equally evaluated using the same tracking parameters.

For the most part, the parameters used were the same for both versions. Exceptions are parameters that are not present in both versions or parameters that are tuned differently with the introduction of the improvements. In the improved version, the parameters used were the same both for Video A and Video B. On the other hand, the original version required more fine-tuning, i.e slightly different parameters for Videos A and B. For the full list of the parameters selected for each video, for which evaluation was performed, see Appendix A.

It is worth mentioning that the Mean Squared Error (MSE) criterion, in Parameter Fitting (see Section 4.2.6, maxMSE parameter), was withdrawn in both versions. The MSE is the difference between the coordinates of the points found during Clustering and the coordinates of the corresponding points in the fitted, 2nd degree curve, obtained during Parameter Fitting. Even though it was a counter-measure against erroneous whiskers (weirdly shaped), it was found to often be filtering out legitimate whiskers, making tracking harder. Additionally, in the improved version, WhiskEras rarely makes outright mistakes during detection, as was previously frequent.

7.1.2. Setting up Tracking

The algorithm's tracking module is particularly sensitive to the detection results of the first video frame. It sets up a number of whisker *tracks*, i.e certain whiskers to follow during the video, based on the whiskers found in the first frame and then attempts to rediscover the same whiskers in the rest of the video, updating these tracks. For example, if 15 whiskers are discovered in the first frame, WhiskEras will only attempt to match whiskers found in consecutive frames to the 15 pre-constructed tracks. Even if 17 whiskers are found on a following frame, at least 2 whiskers will be discarded (more if they are not matched to the existing tracks). This is a design flaw because the algorithm i) does not guarantee 100% accuracy and sensitivity in detecting whiskers, ii) a whisker may be hidden in the first frame and emerge in a subsequent frame.

A solution that was tested was establishing a small *bootstrap* period in which the algorithm would create new whisker tracks every time a newly detected whisker was not matched to a track. At the end of this period, the algorithm would discard all tracks with low detection rate and keep the rest. This solution, however, created new problems as the tracking algorithm used in the first period of the video is weak. That is because the SVM classifier is not launched before acquiring a minimum amount of data, i.e going through a minimum number of frames. Thus, more tracks are created which are difficult to track later on.

This tracking issue creates the following problems in the comparison of the original WhiskEras and the improved version:

1. The WhiskEras instance which does a better job at the first frame may have the edge, because it does not create faulty tracks or produces more accurate tracks. The improved WhiskEras is believed, based on visual inspection, to be more consistent in locating more whiskers which have the correct shape and

also filtering out erroneous whiskers. However, this does not mean that there are no single frames in which it can get outperformed by the original version.

2. Comparing two different numbers of tracks makes direct whisker-to-whisker comparison difficult. Tracks may be mixed up in the duration of the video and corrupt detection rates etc, which will be used as evaluation criteria. It is much easier to make a comparison on the same whisker tracks (i.e the whisker tracks begin from the same starting point) and deduce which WhiskEras version performs better whisker-by-whisker.

3. It is unclear whether following more tracks is an advantage. A track can begin from a whisker that is not actually there. Then, it could be a faulty track for the rest of the video (occasionally mixed up with true whiskers) or at some point be matched to a previously hidden whisker.

This first frame limitation is a drawback at the core design of WhiskEras's tracking module which should be resolved in future work. For now, though, we would feel that our results can reflect a more fair comparison if the starting point is the same for both versions and that the created tracks are valid. This way, we can measure how the new instance of WhiskEras follows those tracks, relative to the original one. The improvements, do not increase the number of whiskers found, for the most part anyway, so this method will not miss out on any important aspects. Thus, for each experiment, the tracks were set up using the improved WhiskEras on the first frame (having checked that the whiskers found were valid) and then were transfered to the original version for the same run.

7.1.3. Criteria

The criteria used to compare the quality between the original and the improved WhiskEras are the following:

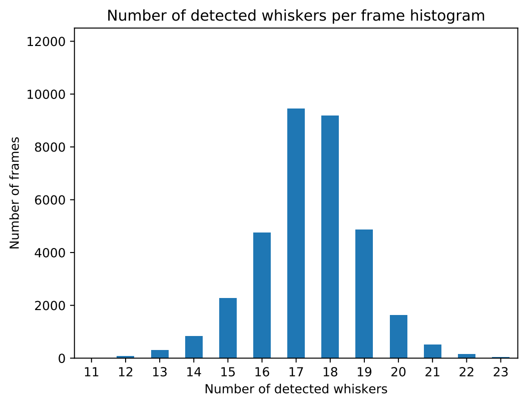
1. Detected whiskers per frame.
2. Detection ratio.
3. Angle tracking quality.
4. Signal-to-noise ratio.

Detected Whiskers Per Frame

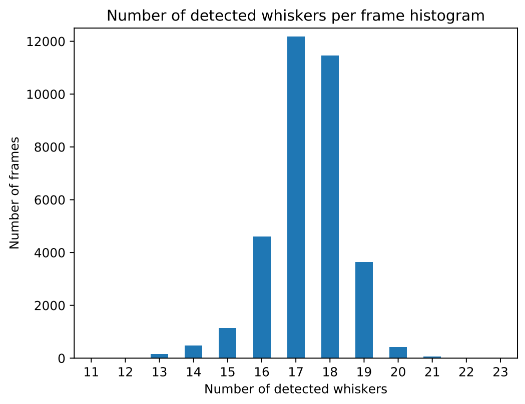
The first criterion to measure success is the number of detected whiskers per frame. This would ideally be fixed during the course of a video, but whiskers are often hidden behind other whiskers or detection fails. However, it should be fairly stable and not oscillate too much, particularly when there is only slight whisker movement. Thus, two metrics are measured with this norm: i) the variance of whiskers detected and ii) the mean number of whiskers detected, during the course of the video. The improvements made to WhiskEras mainly target consistency. In other words, detecting a constant number of valid whiskers, while minimizing the number of erroneous whiskers. This corresponds to less variance. Although a larger number of whiskers would indicate greater sensitivity, oscillating much higher than the average number of whiskers found usually means false positives. This was visually observed for the original WhiskEras algorithm. False positives can make the job of tracking more difficult, as they can potentially get confused with valid whisker tracks.

Figure 7.2 shows the histogram of the number of detected whiskers per frame, from the first to the last frame of video A. In other words, it depicts in how many frames a certain number of whiskers were detected. It is apparent that the Improved WhiskEras performs better at detecting whiskers, as the detected whiskers number is narrower around 17-18 whiskers. The average number of whiskers detected is 17.33 and the standard deviation is 1.13 for the entire course of the video, for the improved version. Compared to 17.38 and 1.54, respectively, for the original version, the improved version shows a 26.7% decrease in standard deviation. Again, this translates to more robust detection. The slight decrease in the mean number of whiskers should not be concerning, as higher variance dictates higher levels of noisy whiskers being detected.

Different results were observed for Video B. Figure 7.3 shows that, on average, more whiskers seem to be detected in Improved WhiskEras, as the distribution is centered around a higher number of whiskers. This was also confirmed by actual numbers: average number of whiskers detected per frame was 13.67, with a standard deviation of 1.19 in Improved WhiskEras. The respective numbers for Original WhiskEras were 12.56 and 1.19. Though there is no impact on variance, indeed more whiskers are detected in the improved version. This means the additional whiskers that are detected are probably valid. More can be inferred by looking at the rest of the criteria, as will be discussed below.

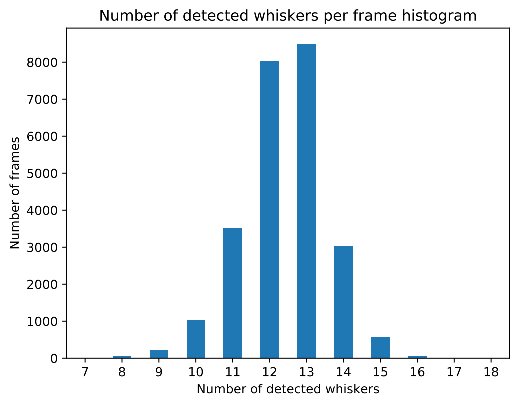


(a) Original WhiskEras

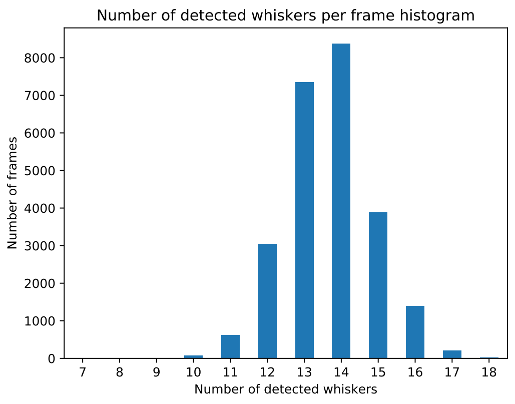


(b) Improved WhiskEras

Figure 7.2: Histogram of detected whiskers per frame in Video A, entire video segment



(a) Original WhiskEras



(b) Improved WhiskEras

Figure 7.3: Histogram of detected whiskers per frame in Video B, frames 5021-30000

Detection Ratio

The detection ratio of each whisker indicates in what percentage of the frames a whisker was successfully tracked. 18 whiskers were tracked in Video A and 13 in Video B. The results for each individual whisker and the total average are depicted in Table 7.1.

For Video A, although there are significant differences between individual whiskers, the total average difference (0.89 to 0.87) is not substantial. These differences can be attributed to different detection results which cause the whisker tracks to take very different trajectories in the two versions. Thus, whisker 1, for example, is detected less than half of the time in the original version, while it is detected almost in every frame, in the improved version. On the other hand, whisker 3 is detected much more consistently in the original version. This could have to do with how these whisker tracks evolve over time. Even though their starting points are the same, they may end up representing a very different whisker.

In Video B, the detection ratio is 5% better overall, which means that the increased number of whiskers found, per the previous criterion, are successfully matched to more tracks. While this could be a good thing, not much can be said about the accuracy of the detections or tracking quality without diving deeper. This will be done using the next criterion.

Angle Tracking Quality

The previous criteria provide a quantative way of evaluating improvements. However, it is still unclear whether the algorithm does fit the correct whiskers to the tracks. The angle is the most important parameter for the neuroscientists, representing each whisker. Hence, its trajectory was used to evaluate the tracking quality. In particular, the angular change of the whiskers was examined, over the course of the videos. When this

Whisker ID	Video A		Video B	
	Original	Improved	Original	Improved
1	0.48	0.97	0.54	0.92
2	0.99	0.97	0.99	0.87
3	0.92	0.72	0.83	1.00
4	0.89	0.81	0.98	0.86
5	0.96	0.77	0.82	0.95
6	0.99	0.96	0.96	0.86
7	0.88	0.91	0.60	0.91
8	0.82	0.94	0.94	1.00
9	0.86	0.72	0.92	0.87
10	0.69	0.86	0.82	0.50
11	0.89	0.82	0.99	0.95
12	0.82	0.99	0.99	0.94
13	0.70	0.95	0.77	0.98
14	0.93	0.76	-	-
15	0.84	0.95	-	-
16	1.00	0.97	-	-
17	1.00	0.99	-	-
18	0.95	0.98	-	-
Average	0.87	0.89	0.85	0.90

Table 7.1: Detection ratio on videos A and B, original and improved versions of WhiskEras

change is too vast and sudden, it is clear that detection and/or tracking fail. During *whisking*, in particular, the whiskers quickly oscillate and provide an extra challenge to the algorithm. These are also the most interesting movements, for the neuroscientists, so it is important that the whiskers are followed accurately.

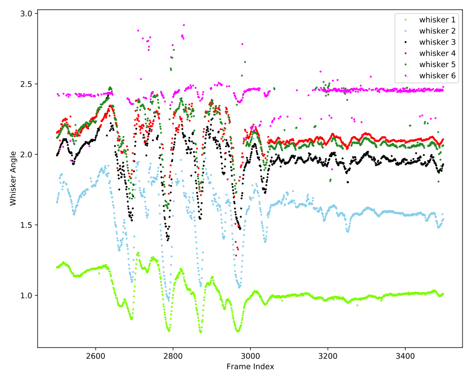
The angular tracking plots of a segment (2500-3500 frames) from Video A are provided below. For clarity, the 18 whisker tracks were divided into three separate plots for each version of WhiskEras. To further facilitate inspection, the whiskers were sorted in terms of their average angle and proximal whiskers, in terms of angle, were put in different plots.

In Figure 7.4, the trajectory of all whiskers' angles are depicted. For whiskers 1-6, the results of the original version are shown in Figure 7.4a, whereas the results for the improved one are shown in Figure 7.4b. The improved version showcases better tracking quality, in general. Specifically, whisker 6 is tracked more reliably in Figure 7.4b. Furthermore, whiskers 4 and 5 seem to be mixed up in the original version.

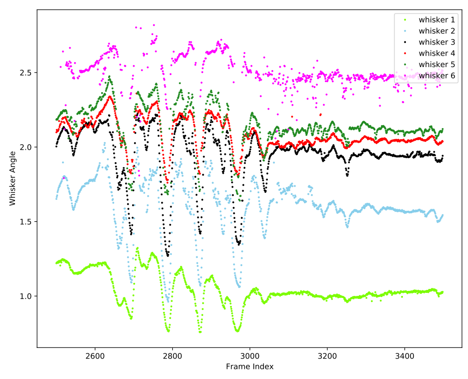
For whiskers 7-12, the results also favour the improved version. In particular, whiskers 11 and 12 are tracked better in Figure 7.4d, compared to Figure 7.4c. However, whisker 10 appears to be very noisy in the improved version. As mentioned before, the improved version does not necessarily guarantee that every single whisker would be located and tracked better. Also, the tracks that correspond to a certain whisker might have diverged for the two different versions.

Finally, Figure 7.4e demonstrates even worse problems for the original version. Whisker 16's angle seems smoother in Figure 7.4f, compared to Figure 7.4e. In addition, whiskers 17 and 18 are tracked quite well in the improved version, while the original version contains a lot of noisy points which do not appear to be connected at all.

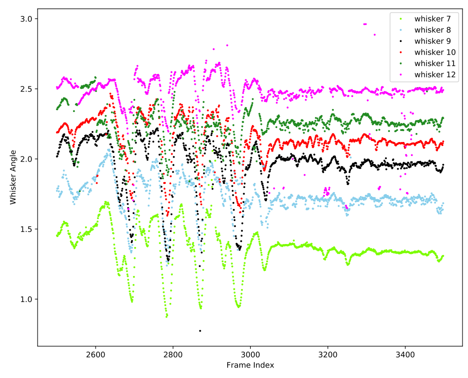
As a final remark, the improved WhiskEras does enhance tracking majorly for a few of the whiskers. This does not mean that it is necessarily better at every single point of tracking, nor that it makes tracking perfect. It certainly helps though, particularly for difficult to trace, intersecting whiskers. Tracking quality results were also obtained for Video B and can be viewed in Appendix B. Enhanced tracking quality was also observed there, limited to fewer whiskers, but fewer whiskers are tracked in Video B in any case (13 as opposed to 18 in Video A).



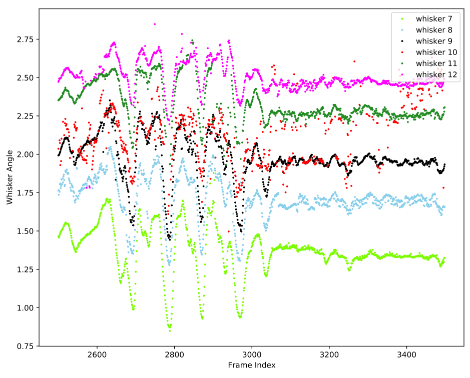
(a) Original WhiskEras, whiskers 1-6



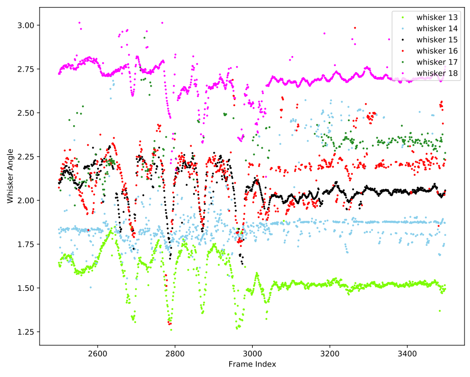
(b) Improved WhiskEras, whiskers 1-6



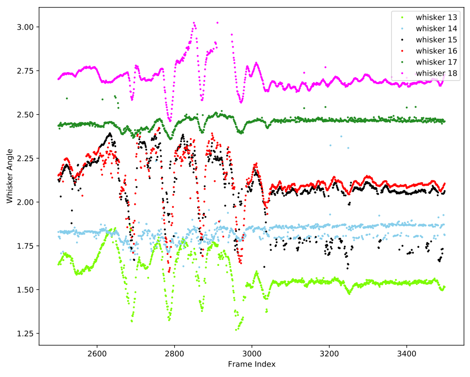
(c) Original WhiskEras, whiskers 7-12



(d) Improved WhiskEras, whiskers 7-12



(e) Original WhiskEras, whiskers 13-18



(f) Improved WhiskEras, whiskers 13-18

Figure 7.4: Angle tracking in Video A

Signal-to-Noise Ratio

An additional criterion to measure tracking/detection quality is the signal-to-noise ratio of the whisker angles. This was also estimated by tracing the whisker angle. As mentioned before, vast and sudden angle variations indicate the presence of noise. As in [9], this was computed as follows: i) The angle parameter of each whisker was smoothed over time using the *smoothdata* function of MATLAB. A window size of 10 frames was used, with a Savitzky-Golay filter, which is appropriate for quick variations of data, since whisker angles change rapidly during whisking. ii) The smoothed data was subtracted from the actual data. iii) The outcome was considered as the *noise*, while the smoothed data was the *signal*. Then, the *snr* function was used to compute the SNR of each whisker.

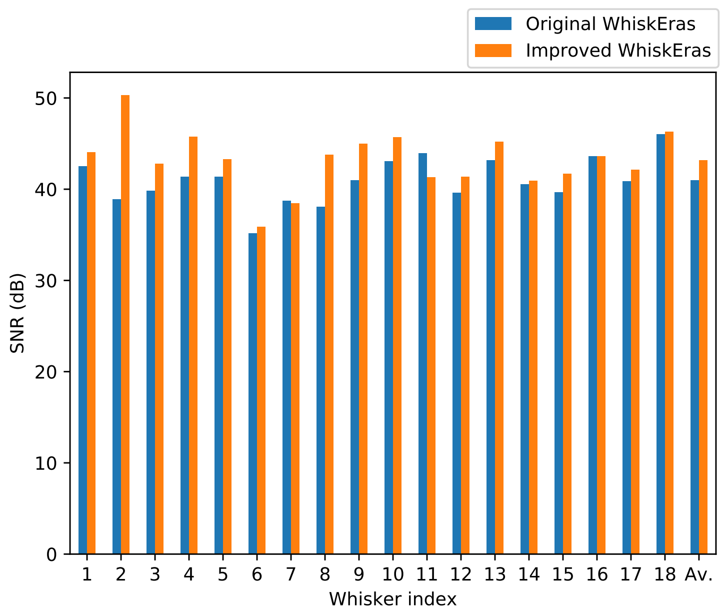


Figure 7.5: Bar diagram of SNR (in dB) of each whisker over video A

For Video A, the SNR of every tracked whisker is depicted in Figure 7.5. The Improved WhiskEras outputs a higher SNR for most of the whiskers, with only two exceptions. For some of the whiskers, e.g 2, 4, 7, 8, the improvement is significant. For others, it is only slight. The overall average SNR is 43.20 dB for the improved version and 40.97 for the original version, as shown in the rightmost column of Figure 7.5. The benefits are not groundbreaking, but as expected, whiskers which are probably intersecting are tracked much more consistently. The tracking quality was also enhanced for almost all whiskers, according to the criterion. This does not necessarily show a revolutionary breakthrough to the existing algorithm. Instead, it illustrates how the existing algorithm was fine-tuned through the improvements made in this thesis. Furthermore, both Video A and Video B were processed using the same parameters in the improved version. This makes for a tracking system which can conveniently be used in practice.

7.1.4. Discussion

After the extensive evaluation of Improved WhiskEras, some afterthoughts are in order. Specifically, the following findings are reported:

Finding 1 The improvements made to WhiskEras which targeted its shortcomings in video setups such as A (Figure 7.1a) did pay off. Specifically, results shown in the previous subsections illustrated superior detection and tracking performance. The modifications also enhanced the performance in the other commonly used experimental setup in Video B (Figure 7.1b), though to a smaller extent. This is natural, as less whiskers are visible in Video B. Overall, Video-A-style setups pose more challenges to the detection algorithm.

Finding 2 The different design choices of the whisker-detection module can be limited to a few parameters. Especially the new parameters which were introduced to replace others are much more robust to different video conditions. Results for both Video A and Video B were obtained using the same configuration in the Improved version (see Appendix A). This is great news for whisker-tracking users who want to get the best out of WhiskEras without spending too much time tinkering. Although exhaustive parameter exploration was not performed, manual tuning, as well as algorithmic analysis indicates that perfect detection is not within the reach of WhiskEras, due to the limitations of Steger's algorithm. For that, other methods should be explored, such as learning-based approaches. However, WhiskEras has already showcased great tracking performance, compared to competition [9], through its powerful Tracking - Learning - Detection technique. It is time it was used in practice by neuroscientists, who can then expertly evaluate its adequacy. The acceleration of WhiskEras provides the opportunity for that, as will be shown next.

7.2. Algorithm Acceleration

The videos used to evaluate performance are, again, the ones from Figure 7.1. The hardware utilized consisted of an AMD EPYC 7551 32-Core Processor @ 2.0 GHz, with 64 threads CPU and an NVIDIA Tesla V100-PCIE GPU. The specification of the GPU is given in Table 7.2. The C++ compiler was run with the *-O3* flag to perform aggressive optimizations such as loop unrolling and vectorized operations, which vastly improve performance.

	NVIDIA Tesla V100-PCIE
On-Board DRAM	32.5 GB
Memory Bandwidth	1134 GB/s
Max Clock Rate	1.38 GHz
Memory Clock Rate	877 MHz
Memory Bus Width	4096-bit
L2 Cache Size	6.29 MB
Multiprocessors	80
CUDA cores	5120
Max Threads/MP	2048

Table 7.2: GPU Specifications

7.2.1. Speedups

Execution times were obtained by running the algorithm on Video A and averaging the time taken between frames 3000 to 4000 to extract the processing time per frame. The reason is that by frame 3000, the SVM Classifier already has plenty of data to train on (more data means more training time). Results for each stage are shown in Table 7.3. Along with MATLAB, three different versions of the accelerated code were run: a completely sequential (C++), a C++ with CUDA version and the final version which also incorporates OpenMP. The corresponding speedups per stage are depicted in Table 7.4 and Figure 7.6.

	Prep.	Point D.	Clust.	Stitch.	Param.	Track.	Train.	Transfers	**Total**
MATLAB	12.3	23.0	39.7	520.2	421.3	123.0	140.6	-	**1284.7**
C++	23.4	232.1	3.9	4.0	12.1	2.9	9.1	0	**3081**
+ CUDA	0.9	1.5	3.2	3.9	12.0	1.9	9.9	2.1	**36.6**
+ OMP	0.9	1.5	2.5	2.7	3.3	1.9	2.0	1.2	**17.1**

Table 7.3: Execution time (milliseconds) per frame for different versions of WhiskEras

Video	Prep.	Point D.	Clust.	Stitch.	Param.	Track.	Train.	**Total**
C++	0.53	0.10	10.09	128.79	34.78	43.15	15.43	**4.17**
+ CUDA	13.47	15.38	12.38	134.39	35.17	64.02	14.16	**35.06**
+ OMP	14.00	15.58	16.02	193.71	126.94	65.44	69.63	**74.96**

Table 7.4: Speedups of accelerated versions over MATLAB

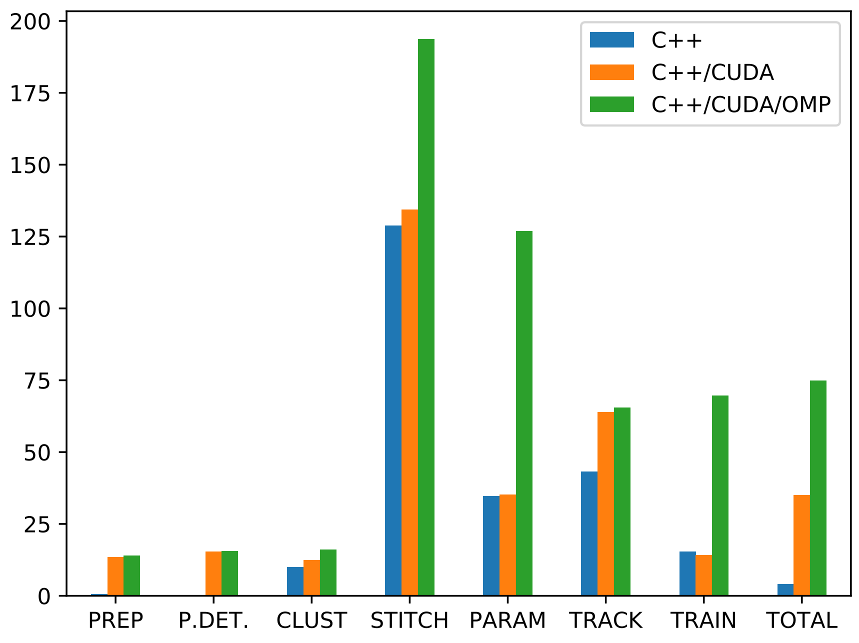
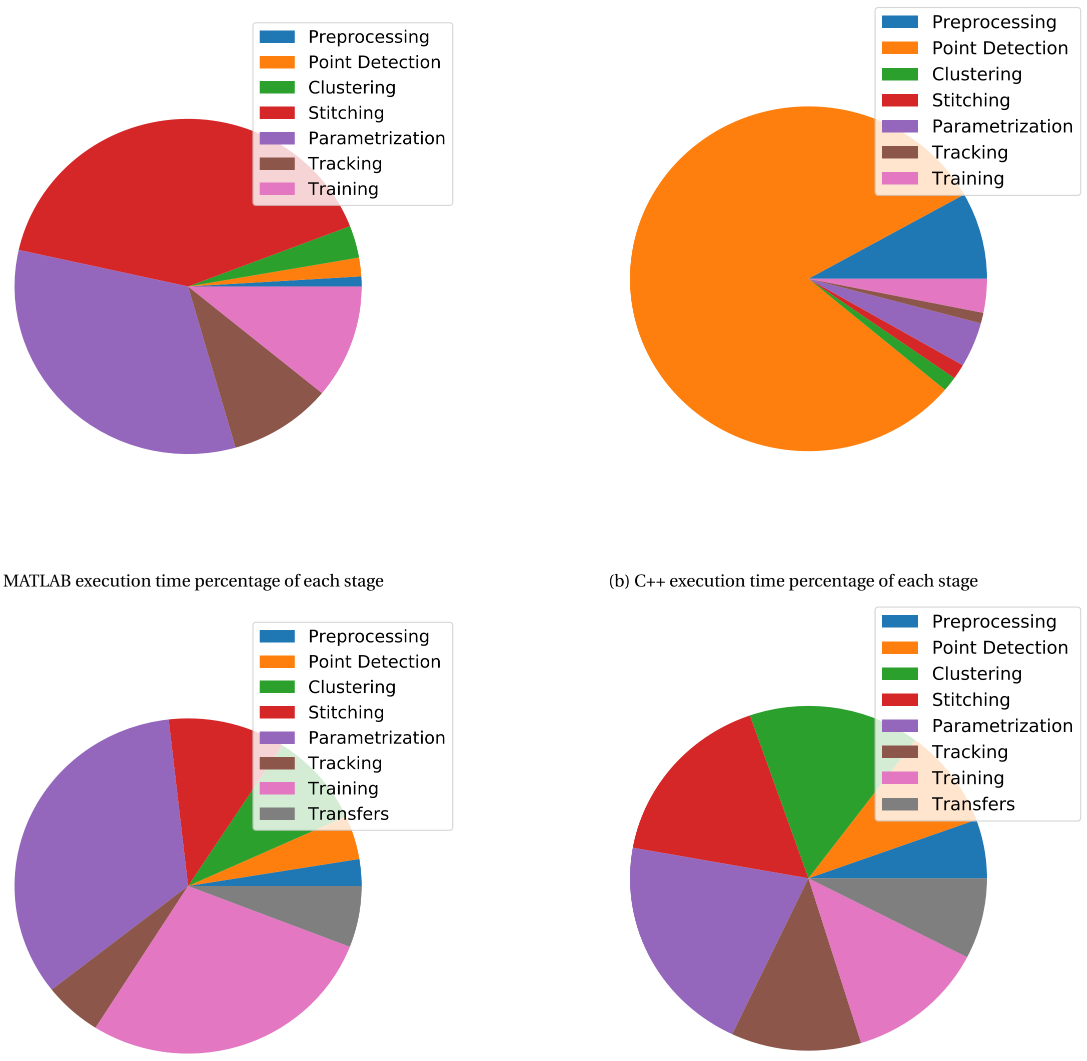


Figure 7.6: Bar diagram of the accelerated versions' speedups over MATLAB

First of all, it is observed that the speedup obtained from C++ alone is not massive (**4.17x**). Observing Table 7.3, it can be seen that both Preprocessing and Whisker-Point Detection stages are slower. In particular, the Whisker-Point Detection stage is 10x faster in MATLAB. That is because MATLAB was already optimized to use parallel execution both on the CPU and the GPU. By executing everything sequentially on the CPU, we are not taking advantage of the data-level parallelism, present in the image-processing operations which are performed in these stages. This is reflected in Figure 7.7b, where Point Detection heavily dominates total execution time. The good news is that Stitching and Parametrization, which previously were accounting for more than 70% of the execution time (see Figure 7.7a), are now substantially faster, just from porting the code to C++.

Introducing CUDA into the pipeline speeds up the stages which exploit data-level parallelism profoundly, compared to the pure C++ version, for a total speedup of **35.06x**. Furthermore, the Whisker-Point Detection step was originally running at comparable speed to MATLAB, even with CUDA. This was significantly improved (about 10x) by the Seperable Convolution optimization, seen in Section 6.2.1. By observing the new execution time profile (Figure 7.7c), it is clear that the algorithm's bottleneck is once again moved to Parameter Fitting and Training. Note that the *Transfers* stage refers to the transfer of data between the CPU and the GPU. This is an unavoidable overhead, also present in MATLAB, but is not measured independently there, hence the " – " in Table 7.3. Rather, it is accounted for in other stages which either use the GPU or read from the GPU.

The final version of the accelerated Improved WhiskEras also makes use of OpenMP to take advantage of *task-level parallelism* on the CPU. The speedup now improves further for the previously troubling stages, to a total of **75.96x** over MATLAB. It should be mentioned that the choice of the *libLinear* library was also pivotal as it makes training about 10x faster than using, e.g *libSVM*. Although the *Parameter Fitting* and *Stitching* stages appear to be slightly more expensive than the rest, the percentage pie chart is now much more balanced (Figure 7.7d). In other words, to further accelerate the algorithm, we would need to speed up multiple stages by a ~ 10x factor. For example, even if the *Parameter Fitting* and the *Stitching* stages were two times faster, the total execution time would still be around 0.014 seconds per frame (down from 0.017), which is still far from real-time processing. From this point, it is significantly harder for the programmer to extract meaningful speedups.



(a) MATLAB execution time percentage of each stage

(b) C++ execution time percentage of each stage

(c) C++/CUDA execution time percentage of each stage

(d) Final version's execution time percentage of each stage

Figure 7.7: Execution time percentage of each stage

7.2.2. Problem Scaling

Different videos yield different processing times. A comparison of the execution times of the videos shown in Table 7.5 is visualized in Figure 7.8. It can be seen that processing Video B is much faster than Video A.

This comparison can be useful in understanding how performance scales with problem size and how algorithmic complexities, introduced in Section 6.1.2, affect performance. To put the problem into context, Video A frames consist of 640×480 pixels, compared to the 480×512 pixels of Video B. By analyzing the results, it was found out that about 8000 whisker points are detected in every frame of Video A, compared to about 4000 points in Video B. Furthermore, the difference in number of clusters before Stitching I and II is not that vast to have a clear impact, while the number of whiskers found on average is only 10-20% lower in Video B.

Beginning with image dimensions, the number of pixels (of the *upscaled* image) which are processed during Preprocessing and Whisker-Point Detection are about 20% reduced in Video B, compared to Video A. As expected, this is hardly noticeable in their execution times, since these stages are massively parallelized on the GPU.

The number of points discovered is roughly double in Video A. Although we were not surprised to see the Clustering cost drop significantly, the Stitching, Parametrization and Tracking stages also benefited. That

Video	Prep.	Point D.	Clustering	Stitching	Param.	Tracking	Training	Transfers	**Total**
A	0.9	1.5	2.5	2.7	3.3	1.9	2.0	1.2	**17.1**
B	0.8	1.4	1.6	2.0	2.0	1.0	1.6	0.8	**12.2**

Table 7.5: Execution time (milliseconds) per frame for two different video setups (accelerated)

is because there are multiple linear-algebra operations on these points during the aforementioned stages. Even though the difference in number of clusters is not worth mentioning, the number of points alone does have a substantial impact on execution time. Thus, less pixels dedicated to the area of interest, as in Video B contributes to faster processing. This is also an argument against increasing image *upscaling* by too much, as it would heavily affect the compute load.

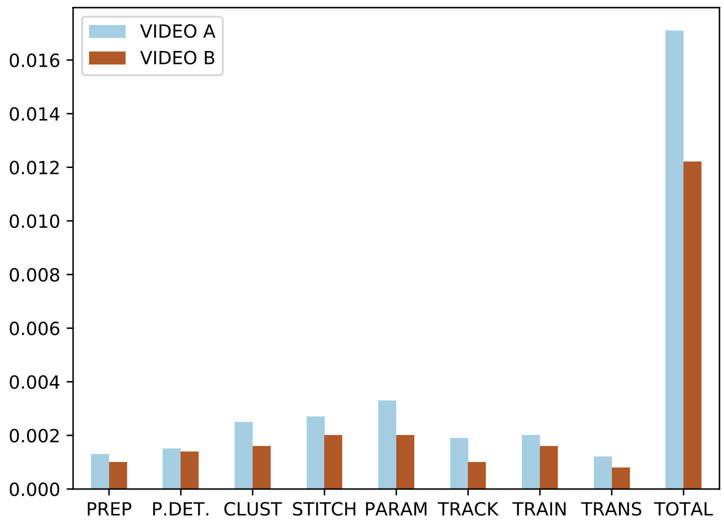


Figure 7.8: Execution time comparison for videos A and B

Lastly, the number of whiskers detected should have an effect on both parametrization and training. However, it was found (Figure 7.8) that parametrization is more strongly affected by the number of whisker points, since non-linear fitting is executed in parallel on the CPU. As for training, its cost is determined by the number of whisker tracks followed, squared. As explained in Section 7.1.2, this number is only dependent on the first frame of the video. Training is indeed 20% more costly for a 27.7% increase in whisker tracks, which is probably more than anticipated, since this stage is also parallelized on the CPU cores.

7.2.3. Real-time Processing

Online whisker tracking requires the algorithm to go through frames at least as fast as they are provided to it from a camera. In Erasmus MC, 750-1000 Hz cameras are used, which means 750-1000 frames per second. This high frequency is necessary, in order to reliably follow whisker movement. After acceleration, WhiskEras is still far from reaching this target. It can only process about 50-120 frames per second, depending on the video, as analyzed in Section 7.2.2. Further speedup would require considerable efforts (see Section 7.2.1) and it would still be uncertain whether it would be enough to reach real-time performance.

Even Ma, in his work [37], could not quite reach the goal of 1000 frames per second definitively. A 1.21 ms/frame peak performance was reported, which translates to 826 frames per second. Furthermore, ViSA is a much simpler algorithm than WhiskEras and does not provide a tracking over-time module. In particular, BWTT has reached these levels of performance just by porting the code to C and processing frames in parallel using OpenMP. This would certainly be inadequate for WhiskEras, as it contains some intensive image processing which would be slow on the CPU. More importantly, it would be infeasible, as the tracking module requires from the algorithm to go through the frames sequentially.

Finally, online tracking does not just require an algorithm with real-time processing power. Another issue is parameter tuning. An algorithm whose design choices need to be explored **after** video recording, to discover an optimal configuration for the specific setup, is unfit for online whisker tracking. During this work, the algorithm was studied in detail and altered where necessary to provide robust parameter tuning without the need to change a great number of parameters to achieve optimal results. In addition, a short calibration period was conceptualized where a few parameters could be automatically configured. However, the time constraints of this thesis did not allow to fully investigate its practicality, so it is left as future work. Therefore, some steps have already been made towards online tracking.

8 Conclusions

In this thesis, WhiskEras, a whisker-tracking system on head-fixed rodents was studied. The algorithm's whisker-detection stages were analyzed and its design choices were investigated. The system's potential was found to be limited by algorithmic flaws, rather than unexplored parameter options. Improvements were made to tackle those issues which were aiming for higher accuracy and robustness across different whisker videos. In addition, the whole system was accelerated by porting the code to C++ and exploiting massive data-level and task-level parallelism.

This chapter concludes this work. In Section 8.1, a thesis overview is given. Section 8.2 summarizes the contributions made to WhiskEras through this work. Finally, Section 8.3 offers some thoughts and recommendations regarding future work.

8.1. Thesis Overview

The objectives of this thesis were to speed up WhiskEras and improve its accuracy under different video settings. The acceleration task was straightforward and required advanced coding skills and high-performance computing knowledge. The second task, however, involved exploring its potential, its limits and some way to breach them. This proved to be substantially more challenging.

The real difficulty in making WhiskEras robust in processing different videos was identifying the core problem. Coming into this thesis, the incentive was to enable WhiskEras's whisker detection module to perform its job flawlessly by tuning its parameters. This is a well-known problem with an also common solution: Design Space Exploration. WhiskEras offers a handful of parameters which indeed can deal with many different issues during whisker detection: stitching far-distant clusters, stitching close-distant clusters, detecting more points etc. These parameters can be restricted to a sensible range in which tuning them results in trade-offs. The problem was that they could not address all the different challenges without making compromises. These compromises did not just degrade performance on some videos, but on every video. The problem was split into smaller problems: we were able to analyze the algorithm's stages in isolation and find the root of the problem. This was Steger's algorithm, which performed curvilinear centerline extraction, i.e whisker-point detection.

Steger's unbiased detector was performing subpar in detecting whisker curves, particularly in the presence of intersections. Several other algorithms were considered, also fulfilling the same purpose: curvilinear detection. These methods included intuitive, learning-based and graph-based approaches, as discussed in Section 2.3.1. Only the learning-based algorithms were presumed to offer certain gains over Steger's algorithm (Section 5.1). However, these require supervision, i.e a large amount of labelled data to train on. This restriction is prohibiting us from migrating to a detection algorithm which would be far superior. Instead, the route that was chosen was that of Improved Curve Tracing [45], which was an expansion of Steger's algorithm, showcasing promising results.

Improved Curve Tracing was also an uphill battle in its implementation. Although its structure was clear, the whisker-videos' peculiarities prohibited the use of many of its strong aspects. Hence, the resulting adaptation of the algorithm, which performed both curvilinear detection and clustering, was not as impressive as expected. However, it did provide certain virtues which could help the next stage of the algorithm-Stitching. Stitching completes the work left unfinished by Local Clustering: it receives the incomplete clustering results of Steger's Clustering and finalizes them by forming the complete whiskers. This was an especially troubling stage, as it has to make a lot of decisions when pairing up clusters. These clusters' local characteristics often did not help as they would not point towards each other when they should etc. Nevertheless, a modification of the Stitching stage followed, which targeted different issues such as the one mentioned. In particular, there was a focus towards devising new, more robust parameters which would make sense algorithmically, under any video conditions.

The improvements performed proved to be valuable to the tracking module of the algorithm, as shown in Section 7.1. Seamless, automated whisker tracking is the ultimate goal of WhiskEras and, although it is not there yet, the modifications made to the algorithm certainly steered it in the correct direction. Furthermore, the parameters introduced were much easier to tune: the same set of parameters were used for both Videos A and B (Figure 7.1) in the Improved WhiskEras and still offered better results overall than the original version, which required more delicate tuning. Although it is not claimed that this is the best performance that can be obtained, it is strongly believed that there is not much more potential to exploit by exhaustively tuning the parameters. This can be a relief to candidate users of WhiskEras, which will not need to channel their efforts in altering the parameters, in search of a massive quality upgrade.

Besides the improvements made, the algorithm was also speeded up substantially. First the MATLAB code was ported to C++, a low-level language which provides significantly faster execution on the CPU. However, the lack of parallel execution was prohibitive in achieving massive speedups. This is where standard parallelization techniques were used to split processing on the GPU threads, as well as the CPU threads, depending on the level of parallelism. Both CUDA and OpenMP were instrumental in achieving the final speedup of **74.96x** over MATLAB. In addition, some key optimizations were implemented, the most significant being the one of Separable Convolution and the use of an efficient library (libLinear) in SVM training.

The accelerated Improved WhiskEras is capable of processing around 50-120 frames per second, depending on the input video. This should be sufficient for practical post-processing of whisker videos in Neuroscience labs. At the moment, online tracking is not possible. This requires real-time processing (1000 frames per second) capabilities which demand speeding up most of the algorithm's stages significantly more. Although this seems out of reach for the time being, the advancements in hardware platforms and high-performance computing could help future endeavors gain further acceleration. Perhaps other routes could be explored as well, for example FPGAs.

8.2. Contributions

The following contributions were made throughout this project:

- The WhiskEras algorithm was extensively studied. The potential of its whisker detection module was explored through its design choices.
- Its stages were examined in isolation, through their intermediate results, to simplify the problem. Non-linear relation between the results of different stages was deemed infeasible, due to the Computer Vision techniques used.
- The weaknesses of WhiskEras were pinpointed in the Whisker-Point Detection stage, which employed Steger's unbiased curvilinear detector.
- Alternative solutions in the broader category of Curvilinear Centerline Extraction were examined. The most prominent solutions were considered to be learning-based, which at the moment are inaccessible due to the lack of labelled data.
- Improved Curve Tracing, an algorithmic expansion of Steger's algorithm was adapted into WhiskEras' detection module.
- The Stitching stage was modified to make room for more robust parameters under different video setups.
- The MATLAB code was initially ported to C++. Then, CUDA and OpenMP were utilized to move the most expensive parts of the code into parallel execution on the GPU and the CPU threads, respectively.
- Some more advanced acceleration techniques were used to increase the parallelizable portion of the code or decrease the time complexity of certain stages.

- The Improvements to WhiskEras' detection module were evaluated on different video setups by using the feedback of its tracking module. Several criteria were adopted from Betting's first evaluation of the algorithm, such as detection ratio and tracking quality.

- The speedups obtained from each level of parallelization were assessed, compared also to the MATLAB code. The timing profile of each was given as well as an analysis on problem scaling on different whisker videos.

- The work done during this thesis led to important breakthroughs in whisker tracking. A publication will be made, as a result.

8.3. Discussion and Future Work

Concluding this work, we have presented a working version of a fast and even more accurate WhiskEras. This will enable researchers in Neuroscience around the globe to actively use it to process their whisker videos from head-fixed rodents. Its true capabilities will then be explored further, as well as limitations or issues which did not come to light during the limited time of this thesis. However, hands-on experience on WhiskEras has given us the confidence to make some recommendations for future work. These pointers are presented in a *question - answer* form, for illustration purposes.

What is next for WhiskEras?

WhiskEras was devised more than 2 years ago [8]. Its paper [9] was only recently published and it has already received attention in the field. It has been shown to be miles ahead of other popular methods such as Janelia Whisk [16] and BWTT [37]. The objective of this work was to make it practical to use in the Neuroscience lab of Erasmus MC. A GUI may need to be integrated in this work, along with a manual, so that it can be used by a neuroscientist with ease.

In addition, improving the tracking module was not the focus of this thesis. However, it was explored and a number of limitations were discovered there as well, such as the sensitivity to the tracking results of the first frame, as explained in Section 7.1.2. This should be addressed in future work. Furthermore, the intensity of each whisker, as a function of the distance from the snout, was found to remain relatively constant. This could be integrated as a feature in the SVM classifier and confirm if it can indeed increase tracking accuracy. Finally, there is a large number of parameters, also in this module. Future research should consider studying the effect of these parameters on the different whisker videos and potentially find an automated way of tuning them.

Are the tracking results reliable, so that the neuroscientists can trust them in their research?

This question can be answered from multiple perspectives. From a Computer Vision point of view, the algorithm is estimated (not measured) to be about 70-80% accurate, depending on the challenges posed by each whisker video. Certain whiskers will consistently be detected, some will be detected most or some of the time and a few will not be detected at all. WhiskEras' tracking module is very robust in following whiskers which are not detected for some of the time. *Macrovibrissae* which are never detected are usually short or intersect with multiple hair along a line nearly parallel to the snout. It is the place of an expert in neuroscience to judge whether these whiskers could provide valuable information to justify further work into improving the detection module of WhiskEras. If this is deemed necessary, the research conducted during this thesis has pointed towards learning-based methods. Adopting one of the countless methods mentioned in Section 2.3.1 and combining it with Transfer Learning to minimize the amount of labelled data necessary could be promising. However, this would go into a completely different direction (supervised method) and would require frame-by-frame annotation, which, in this application, is extremely time-consuming.

Regarding different approaches, deep learning is currently leading the research industry in Computer Vision. Convolutional neural networks, in particular, are great candidates for a wide variety of image-processing applications. Theoretically, an end-to-end neural network could be devised, which obtains the input image and outputs a compressed representation of whiskers (e.g the output of the Parameter Fitting stage). This could combine a conv-net (CNN) which takes as input each frame's pixels and outputs a squeezed output. Then a recurrent neural network (RNN) further down the pipeline, which processes this output and *decides* how many and what whiskers are in the frame. This could constitute the whole detection module, replacing the entire pipeline from Preprocessing to Parameter Fitting. Needless to say, such a project would require huge resources to come to realization, starting from the collection of data.

A much simpler route would be to build on the current implementation by performing more modest modifications to the algorithm. For example, tracing faint curves could prove to be effective after Stitching II. This was unfortunately not tested in this work due to time constraints. Also, improving the tracking module, as already mentioned, could lead to significant improvement.

Parameter tuning seems to be easier than before. However, there are still lots of parameters. How can we trust that the best values have been discovered, let alone for any future video to be processed? Is it not necessary to perform a Design Space Exploration?

From an empirical standpoint, searching for a better parameter configuration could help, but only slightly. The videos examined, which involved two very different setups showcased that near-optimal results can be achieved, where optimality is dictated by the inherent limitations of Steger's algorithm. However, it is not claimed that every whisker video will fall into the same category. There can be videos from different labs which use a very different setup and/or recording settings. These could indeed require different design choices.

A Design Space Exploration could theoretically help the detection module of WhiskEras, although it is not strongly believed, given that the algorithm can be manually tuned stage-by-stage, to a reasonable extent. Hence, finding a unique combination of parameters that yields tremendous improvements, without paying attention to intermediate results, seems unlikely. Furthermore, as explained in Section 4.4, a DSE would impose practical difficulties in its implementation which mostly have to do with the optimization criterion. Therefore, it is not recommended to go into that direction.

Can we speed up WhiskEras further?

The answer to this question is a plain, simple *yes*. However, one would have to consider the target. The algorithm can already process videos at a reasonably good pace. If real-time processing is the goal, then a great amount of time and high-performance computing skills would need to be invested. As already explained in Section 7.2.3, the current timing profile of the algorithm suggests that multiple stages (specifically, all of them) of the algorithm have to be speeded up and in some cases, by more than 10 times. This is a significant speedup for an algorithm that already uses advanced parallelization techniques, both on the CPU and the GPU. The programmer would have to come up with even more advanced optimization techniques and possibly make compromises or completely reformulate the most costly parts of the code. For a better grasp of the situation, consider that only the separable convolutions performed very efficiently on one of the best GPUs on the market (NVIDIA Tesla V100) take about 1 ms per frame or more. That means that one part of one stage takes about all the time that is available to perform real-time processing. And this is also one of the fastest stages. Hence, it seems unlikely that the current iteration of WhiskEras can be used for online tracking. One possibility would be to use multiple GPUs and attempt to move more stages of the problem into CUDA. However, that would require an even more expensive hardware setup. Perhaps a different accelerator platform, such as an FPGA could be used, but since this falls outside our field of expertise, it is left for future research.

A
Selected Parameters for Detection

In Table A.1, the complete list of parameters selected to process Videos A and B (Figure 7.1) for both the Improved and the Original WhiskEras is provided. Notice that for the Improved WhiskEras, the parameters selected are identical in both Videos. Oscillating around this specific design point did not offer any meaningful quality gain. On the other hand, the Original WhiskEras did have some margin for improvement through parameter alteration between videos, particularly for the hard to tune parameters in Stitching I and II.

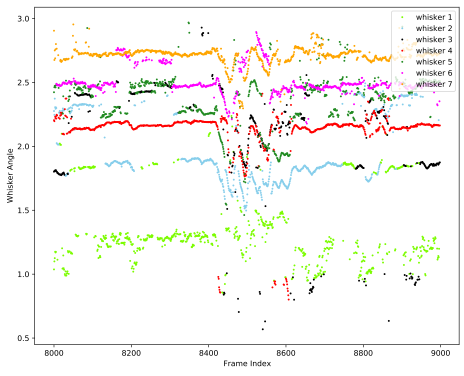
Original WhiskEras			**Improved WhiskEras**		
Name	**Video A**	**Video B**	**Name**	**Video A**	**Video B**
Snout Line Position			*Snout Line Position*		
tip	(764,70)	(420,840)	tip	(764,70)	(420,840)
middle	(28,276)	(120,370)	middle	(28,276)	(120,370)
Preprocessing			*Preprocessing*		
bwThreshold	1	1	bwThreshold	1	1
imdilate_value	20	20	imdilate_value	10	10
cutoff	10	10	cutoff	0	0
histEqMinT	3	3	histEqMinT	3	3
histEqMaxT	100	100	histEqMaxT	100	100
histEqGamma	0.95	0.95	histEqGamma	0.95	0.95
do_deinterlacing	0	0	do_deinterlacing	0	0
initialGaussianSigma	0	0	initialGaussianSigma	0	0
remove_lowlevels	10	10	remove_lowlevels	10	10
framing	2	2	framing	2	2
Whisker Point Detection			*Whisker Point Detection*		
upscaling	2	2	upscaling	2	2
sigma	2.3	2.3	sigma	2.3	2.3
gaussianKernelSize	20	20	gaussianKernelSize	20	20
tr2	1.2	1.2	-	-	-
Clustering			*Clustering*		
steger_c	1	1	steger_c	1	1
USE_DBSCAN	0	0	tr2	1.5	1.5
-	-	-	β_0	7	7
-	-	-	L	3	3
-	-	-	M	15	15
-	-	-	K	15	15
-	-	-	beamThreshold	10	10
Stitching I			*Stitching I*		
minClusterSize	5	4	minClusterSize	7	7
maxAD	20	20	maxAD	20	20
lengthOfTipAndBottom	20	20	lengthOfTipAndBottom	15	15
c	1	1	c	200	200
maxX	100	100	cone	7	7
maxY	10	7	radonLength	20	20
-	-	-	radonThreshold	30	30
-	-	-	d_0	5	5
Stitching II			*Stitching II*		
minClusterSize	30	30	minClusterSize	30	30
highestRoot	20	40	highestRoot	10	10
lengthOfTipAndBottom	30	30	lengthOfTipAndBottom	30	30
acceptableXdistance	50	50	acceptableXdistance	100	100
acceptableYdistance	10	8	cone	20	20
-	-	-	maxAD	20	20
Parameter Fitting			*Parameter Fitting*		
minNrOfDots	40	40	minNrOfDots	40	40
minLength	150	150	minLength	150	150
minAngleToSnout	5	5	minAngleToSnout	5	5
maxBend	0.01	0.01	maxBend	0.01	0.01
maxMSE	Inf	Inf	maxMSE	Inf	Inf

Table A.1: Parameter selection for videos A and B, original and improved versions of WhiskEras

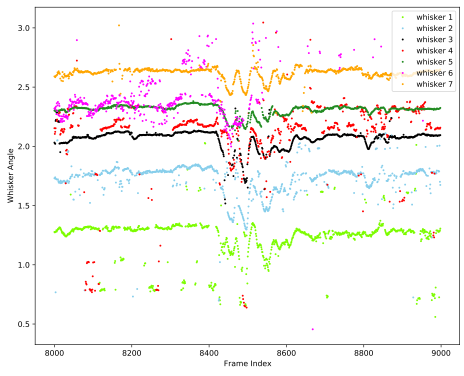
B

Tracking Quality - Angle Plots

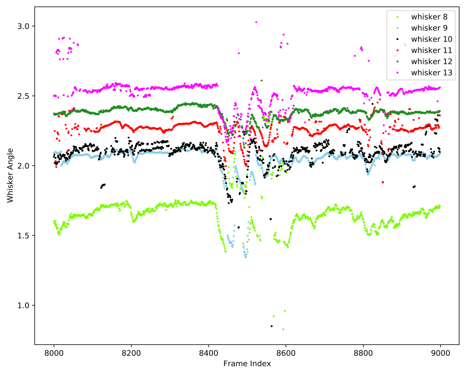
Figure B.1 illustrates angle tracking in Video B (Figure 7.1b) for frames 8000-9000. Two plots are provided with different whiskers, for clarity. It can be seen that, particularly for whiskers 1, 2 (Figures B.1b - B.1a), Improved WhiskEras has the edge over the Original one.



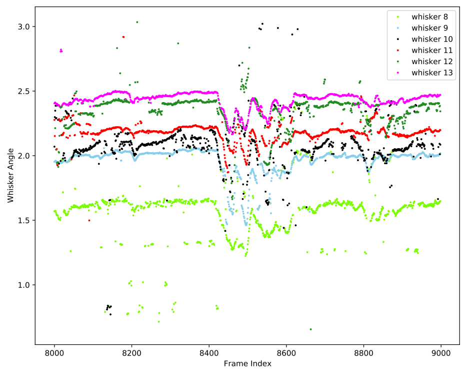
(a) Original WhiskEras, whiskers 1-7



(b) Improved WhiskEras, whiskers 1-7



(c) Original WhiskEras, whiskers 8-13



(d) Improved WhiskEras, whiskers 8-13

Figure B.1: Angle tracking in Video B

Bibliography

[1] The cub documentation. URL https://nvlabs.github.io/cub/.

[2] Neural networks: History. URL https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html.

[3] Mouse cover image. URL https://wallhere.com/en/wallpaper/109260.

[4] 2015 cold spring harbor summer course: "imaging structure and function in the nervous system". URL https://meetings.cshl.edu/galleries/galleries.html.

[5] William Arliss. Solving the svm: Stochastic gradient descent and hinge loss, 2020. URL https://towardsdatascience.com/solving-svm-stochastic-gradient-descent-and-hinge-loss-8e8b4dd91f5b.

[6] Amey Band. Multi-class classification — one-vs-all & one-vs-one, 2020. URL https://towardsdatascience.com/multi-class-classification-one-vs-all-one-vs-one-94daed32a87b.

[7] Rune W. Berg and David Kleinfeld. Rhythmic whisking by rat: Retraction as well as protraction of the vibrissae is under active muscular control. *Journal of Neurophysiology*, 89(1):104–117, jan 2003. doi: 10.1152/jn.00600.2002.

[8] Jan-Harm L. F. Betting. Full-whisker tracking system: A new algorithm for accurate whisker tracking in untrimmed head-fixed mice. Master's thesis, Delft University of Technology, Sep. 2018.

[9] Jan-Harm L. F. Betting, Vincenzo Romano, Zaid Al-Ars, Laurens W. J. Bosman, Christos Strydis, and Chris I. De Zeeuw. WhiskEras: A new algorithm for accurate whisker tracking. *Frontiers in Cellular Neuroscience*, 14, nov 2020. doi: 10.3389/fncel.2020.588445.

[10] Léon Bottou. Stochastic gradient svm, 2007. URL https://leon.bottou.org/projects/sgd#references.

[11] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[12] Jason Brownlee. A gentle introduction to computer vision, March 2019. URL https://machinelearningmastery.com/what-is-computer-vision/.

[13] GE Carvell and DJ Simons. Biometric analyses of vibrissal tactile discrimination in the rat. *The Journal of Neuroscience*, 10(8):2638–2648, aug 1990. doi: 10.1523/jneurosci.10-08-02638.1990.

[14] Dengfeng Chai, Wolfgang Forstner, and Florent Lafarge. Recovering line-networks in images by junction-point processes. jun 2013. doi: 10.1109/cvpr.2013.247.

[15] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[16] Nathan G. Clack, Daniel H. O'Connor, Daniel Huber, Leopoldo Petreanu, Andrew Hires, Simon Peron, Karel Svoboda, and Eugene W. Myers. Automated tracking of whiskers in videos of head fixed rodents. *PLoS Computational Biology*, 8(7):e1002591, jul 2012. doi: 10.1371/journal.pcbi.1002591.

[17] Margaret J. Cook. *The Anatomy of the Laboratory Mouse*. Academic Press, Elsevier, 1965, Revised 2008 for the web. URL http://www.informatics.jax.org/cookbook/index.shtml.

[18] Jaydeep De, Li Cheng, Xiaowei Zhang, Feng Lin, Huiqi Li, Kok Haur Ong, Weimiao Yu, Yuanhong Yu, and Sohail Ahmed. A graph-theoretical approach for tracing filamentary structures in neuronal and retinal images. *IEEE Transactions on Medical Imaging*, 35(1):257–272, jan 2016. doi: 10.1109/tmi.2015.2465962.

[19] Stanley R. Deans. *The Radon Transform and Some of Its Applications.* Courier Corporation. ISBN 0486462412.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002. doi: 10.1109/4235.996017.

[21] NVIDIA Documents. How gpu acceleration works. URL `http://www.nvidia.com/docs/IO/143716/how-gpu-acceleration-works.png`.

[22] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[23] Alejandro F. Frangi, Wiro J. Niessen, Koen L. Vincken, and Max A. Viergever. Multiscale vessel enhancement filtering. pages 130–137, 1998. doi: 10.1007/bfb0056195.

[24] Lin Gu and Li Cheng. Learning to boost filamentary structure segmentation. dec 2015. doi: 10.1109/iccv.2015.80.

[25] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[26] Mark Harris. How to optimize data transfers in cuda c/c++, 2012. URL `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/`.

[27] D. N. Hill, R. Bermejo, H. P. Zeigler, and D. Kleinfeld. Biomechanics of the vibrissa motor plant in rat: Rhythmic whisking consists of triphasic neuromuscular activity. *Journal of Neuroscience*, 28(13):3438–3455, mar 2008. doi: 10.1523/jneurosci.5008-07.2008.

[28] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, jul 2008. doi: 10.1109/mc.2008.209.

[29] A.D. Hoover, V. Kouznetsova, and M. Goldbaum. Locating blood vessels in retinal images by piecewise threshold probing of a matched filter response. *IEEE Transactions on Medical Imaging*, 19(3):203–210, mar 2000. doi: 10.1109/42.845178.

[30] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1409–1422, 2012. ISSN 0162-8828. doi: 10.1109/TPAMI.2011.239.

[31] Eric R. Kandel. *Principles of Neural Science, Fifth Edition. McGraw-Hill Education.* 2012. ISBN 978-0071390118.

[32] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. pages 33–54, 2011. doi: 10.1007/978-3-642-21292-5_3.

[33] Kedar Potdar, Chinmay Pai, and Sukrut Akolkar. A convolutional neural network based live object recognition system as blind aid. 2018. doi: 10.13140/RG.2.2.34494.54085.

[34] David B. Kirk and Wen mei W. Hwu. Programming massively parallel processors: A hands-on approach. In Todd Green and Nathaniel McFadden, editors, *Second Edition*, chapter 1, pages 2–13. Morgan Kaufmann, 225 Wyman Street, Waltham, MA, 02451, USA, 2013.

[35] Max W. K. Law and Albert C. S. Chung. Three dimensional curvilinear structure detection using optimally oriented flux. pages 368–382, 2008. doi: 10.1007/978-3-540-88693-8_27.

[36] Qi Li, Raied Salman, Erik Test, Robert Strack, and Vojislav Kecman. GPUSVM: a comprehensive CUDA based support vector machine package. *Open Computer Science*, 1(4), jan 2011. doi: 10.2478/s13537-011-0028-7.

[37] Yang Ma, Prajith Ramakrishnan Geethakumari, Georgios Smaragdos, Sander Lindeman, Vincenzo Romano, Mario Negrello, Ioannis Sourdis, Laurens W.J. Bosman, Chris I. De Zeeuw, Zaid Al-Ars, and Christos Strydis. Towards real-time whisker tracking in rodents for studying sensorimotor disorders. jul 2017. doi: 10.1109/samos.2017.8344621.

[38] Multi-objective optimization. Multi-objective optimization — Wikipedia, the free encyclopedia, 2007. URL `https://en.wikipedia.org/wiki/Multi-objective_optimization`. [Online; accessed 22-12-2020].

[39] NVIDIA. Cuda samples. URL `https://github.com/NVIDIA/cuda-samples`.

[40] David A. Patterson and John L. Hennessy. Computer organization and design: The hardware/software interface. In Todd Green and Nate McFadden, editors, *Fifth Edition*, chapter 1, pages 43–46. Morgan Kaufmann, The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB, 2014.

[41] Igor Perkon, Andrej Košir, Pavel M. Itskov, Jurij Tasič, and Mathew E. Diamond. Unsupervised quantification of whisking and head movement in freely moving rodents. *Journal of Neurophysiology*, 105(4): 1950–1962, apr 2011. doi: 10.1152/jn.00764.2010.

[42] Johann Radon. On the determination of functions from their integral values along certain manifolds. *IEEE Transactions on Medical Imaging*, 5(4):170–176, dec 1986. doi: 10.1109/tmi.1986.4307775.

[43] Radon transform. Radon transform — Wikipedia, the free encyclopedia, 2004. URL `https://en.wikipedia.org/wiki/Radon_transform`. [Online; accessed 16-01-2021].

[44] K. Raghupathy. Improved curve tracing in images. Master's thesis, CORNELL UNIVERSITY, 2004.

[45] K. Raghupathy and T.W. Parks. Improved curve tracing in images. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*. IEEE. doi: 10.1109/icassp.2004.1326611.

[46] Anant Ram, Jalal Sunita, Anand Jalal, and Kumar Manoj. A density based algorithm for discovering density varied clusters in large spatial databases. *International Journal of Computer Applications*, 3, 06 2010. doi: 10.5120/739-1038.

[47] Vincenzo Romano, Licia De Propris, Laurens WJ Bosman, Pascal Warnaar, Michiel M ten Brinke, Sander Lindeman, Chiheng Ju, Arthiha Velauthapillai, Jochen K Spanke, Emily Middendorp Guerra, Tycho M Hoogland, Mario Negrello, Egidio D'Angelo, and Chris I De Zeeuw. Potentiation of cerebellar purkinje cells facilitates whisker reflex adaptation through increased simple spike activity. *eLife*, 7, dec 2018. doi: 10.7554/elife.38852.

[48] Vijay Singh. Difference between c and c++. URL `https://hackr.io/blog/difference-between-c-and-cplusplus`.

[49] C. Steger. An unbiased detector of curvilinear structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(2):113–125, 1998. doi: 10.1109/34.659930.

[50] P.A. Toft. Using the generalized radon transform for detection of curves in noisy images. doi: 10.1109/icassp.1996.545862.

[51] Ben Mitchinson Tony J. Prescott and Robyn Anne Grant. Vibrissal behavior and function. *Scholarpedia*, 6(10):6642, 2011. doi: 10.4249/scholarpedia.6642.

[52] Engin Turetken, Carlos Becker, Przemyslaw Glowacki, Fethallah Benmansour, and Pascal Fua. Detecting irregular curvilinear structures in gray scale and color imagery using multi-directional oriented flux. dec 2013. doi: 10.1109/ICCV.2013.196.

[53] Kanan Vyas. Object segmentation, 2020. URL `https://medium.com/visionwizard/object-segmentation-4fc67077a678`.

[54] Chi-Feng Wang. A basic introduction to separable convolutions, 2018. URL `https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728`.

[55] Hanjin Zhang, Yang Yang, and Hongbin Shen. Detection of curvilinear structure in images by a multi-centered hough forest method. *IEEE Access*, 6:22684–22694, 2018. doi: 10.1109/access.2018.2823726.