NTNU
Norwegian University of
Science and Technology

Parallel Algorithms for Neuronal Spike Sorting

Thomas Stian Bergheim
Arve Aleksander Nymo Skogvold

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Ian Bratt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

State-of-the-art algorithms will be studied for sorting spike data recorded via tetrodes from living rats. Parallel algorithms will be implemented on state-of-the-art parallel CPU hardware in order to improve performance of existing algorithms, and increase the range of tractable algorithms.

Emphasis will be placed on tuning the algorithms to achieve maximum performance from the hardware platform, and grasping a deep understanding of programming practices needed to achieve high performance on parallel hardware.

Detailed analysis will be performed to understand the trade-offs and quality of the algorithms, with regards to clustering quality and execution time.

The project should culminate in a working application with a graphical user interface which will be used to perform spike sorting, and to experiment with the different implemented algorithms.

Assignment given: 21. January 2011
Supervisor: Ian Bratt

Abstract

Neurons communicate through electrophysiological signals, which may be recorded using electrodes inserted into living tissue. When a neuron emits a signal, it is referred to as a spike, and an electrode can detect these from multiple neurons. Neuronal spike sorting is the process of classifying the spike activity based on which neuron each spike signal is emitted from.

Advances in technology have introduced better recording equipment, which allows the recording of many neurons at the same time. However, clustering software is lagging behind.

Currently, spike sorting is often performed semi-manually by experts, with computer assistance, in a drastically reduced feature space. This makes the clustering prone to subjectivity. Automating the process will make classification much more efficient, and may produce better results. Implementing accurate and efficient spike sorting algorithms is therefore increasingly important.

We have developed parallel implementations of superparamagnetic clustering, a novel clustering algorithm, as well as k-means clustering, serving as a useful comparison. Several feature extraction methods have been implemented to test various input distributions with the clustering algorithms. To assess the quality of the results from the algorithms, we have also implemented different cluster quality algorithms.

Our implementations have been benchmarked, and found to scale well both with increased problem sizes and when run on multi-core processors.

The results from our cluster quality measurements are inconclusive, and we identify this as a problem related to the subjectivity in the manually classified datasets. To better assess the utility of the algorithms, comparisons with intracellular recordings should be performed.

Acknowledgements

This thesis was written during spring 2011, as part of the course *TDT4900 – Computer and Information science, master thesis*. The project was carried out in collaboration between Thomas Bergheim and Arve Skogvold.

We would like to thank our supervisors, Ian Bratt and Professor Lasse Natvig at the Department of Computer and Information Science, Norwegian University of Science and Technology, for invaluable feedback throughout the project.

Our thanks also go to Albert Tsao at the Kavli Institute for Systems Neuroscience for helpful insight on the manual spike clustering process.

Finally, we would like to thank our fellow students at the *Fiol* computer lab, for interesting conversations and Stiga table hockey matches to help us through the many long work days.

Trondheim, June 2011
Thomas Bergheim and Arve Skogvold

Contents

List of Figures

CHAPTER 1

Introduction

This chapter starts with briefly explaining the basic prerequisites needed to understand this thesis. The background is described in further detail in Chapter 2. We then extract the goals for our thesis. Finally, we describe the outline of the rest of the report.

1.1 Spike sorting

In an effort to better understand how the brains of mammals work, researchers try to map the cells in the hippocampus by recording the electrophysiological activity. When the cells communicate, they "activate", emitting what is called an *action potential* or a *spike*, in the form of an electrical discharge.

A visualization of a typical spike can be seen in Figure 1.1 on the following page. The characteristic signature begins at around zero, where the cell is idle. It them rises to a positive peak amplitude, where the action potential is released. Finally, the discharge makes the cell reach a negative voltage state before rising back to its idle state again.

A cell which communicates in this way is called a *neuron*. The chain of events formed when multiple neurons communicate is of great interest to neuroscientists. To record these spikes, one has to insert sensors directly into the brain, which then record these signals. The location of such a sensor is often called a *recording site*. Section 2.1 gives a more detailed introduction to neuroscience.

There are two ways to record the signals that the cells emit. The most accurate way is *intracellular* recording, which means inserting an electrode for every cell that you want to monitor. This way you can be certain which cell

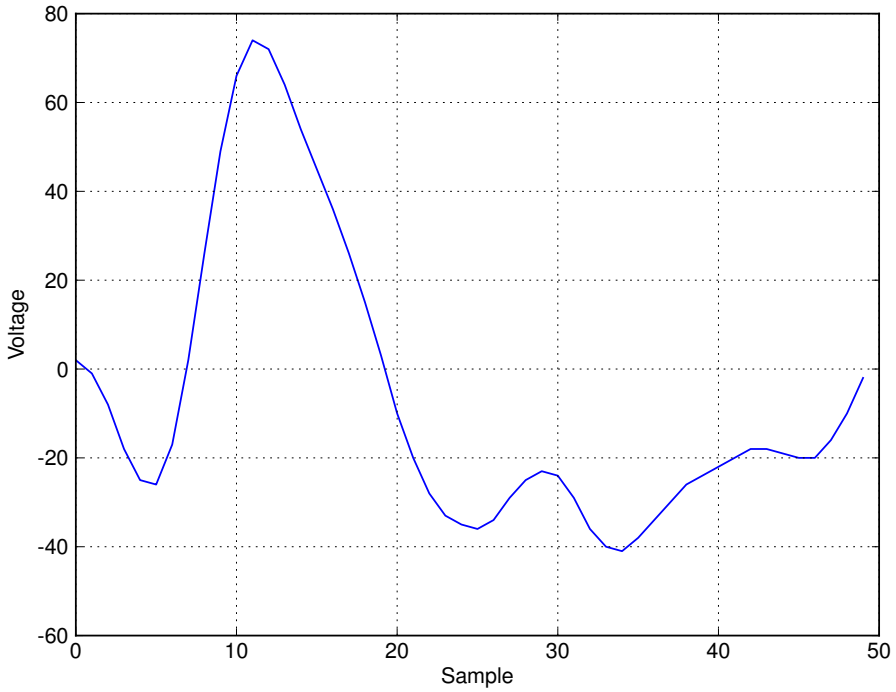


Figure 1.1: A visualization of the waveform of a single spike. The x axis represents time, while the y axis represents the voltage values recorded from the system. Each step on the x axis corresponds to $20\mu s$. The recording system is calibrated to represent amplitudes within a range of -127 and 128, so the actual voltages are not shown.

emitted which signal. However, given the number of cells, this is impractical, as each electrode introduced inflicts damage to the surrounding tissue.

The much more common way to record signals is by *extracellular* recording. The electrodes are implanted in close proximity to the firing cells. These then record signals from several cells simultaneously. Recording from multiple cells poses a challenge as it might be difficult to identify the source of a signal. If only one sensor is used, it can become impossible to identify different cells if they fire within a similar distance from the sensor. To help overcome this problem, the recording equipment has several electrodes in close proximity. An example of this can be seen in Figure 1.2 on the next page.

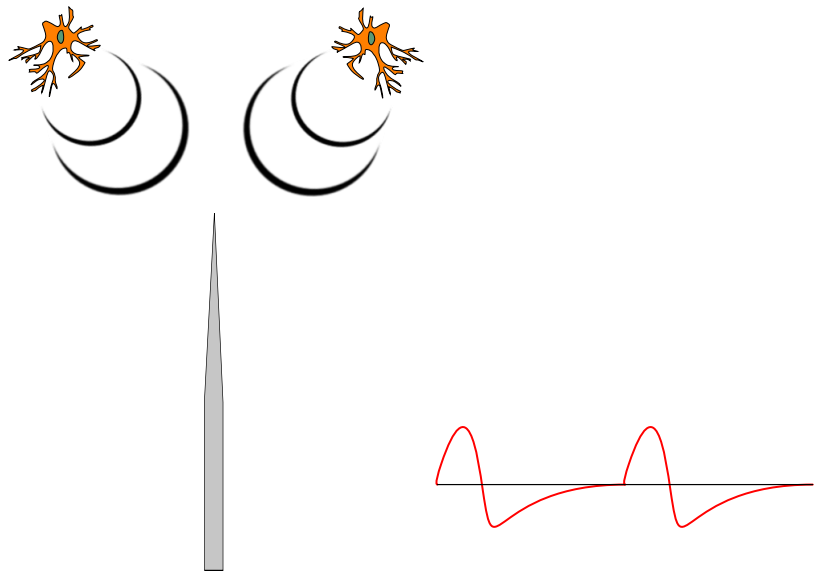
At the present time the sensors used usually contain four coupled electrodes allowing for a much better precision when detecting signals. Sensors containing four such electrodes are referred to as *tetrodes*. Using four seperate channels allows for the triangulation of a signal in a three dimensional space, because different electrodes will detect the same signal but with different amplitudes (Buzsáki [20]). Figure 1.3 on page 5 displays two representations of the same signal recorded by a tetrode. The recording equipment used is described in Section 2.4.

Measuring biological signals is a challenging task, because the signals are not completely reliable – the amplitude and frequency may vary slightly from time to time. Adding to the complexity, the sensors can move slightly during the experiment, introducing further distortions to the signals.

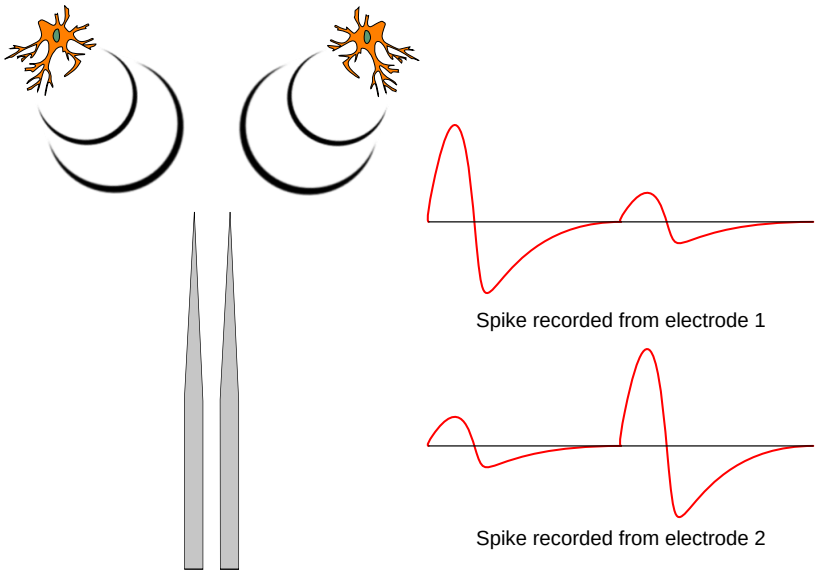
The sensors can only reliably record activity within a limited range. Everything outside of this range will add noise to the signal. Because the sensors only see electrical signals in the form of voltage amplitudes, one has to determine how many cells were recorded, and from which cell every activity was detected.

A spike lasts for a limited time, and, in most scenarios, is the only interesting part of the signal. Thus it is needed to determine when a neuron is spiking, and filter the spikes out from the rest of the signal. This is often referred to as *spike detection*, which is described in Section 2.3.2.

When the spikes have been recorded, we have to choose a way to represent it, a process referred to as *feature extraction*. This may be the whole array of samples from the raw signal, which leads to high dimensionality, or the output of a reduction algorithm, which strives to extract only the important information of the signal. Feature extraction is further explained in Section 2.3.3.

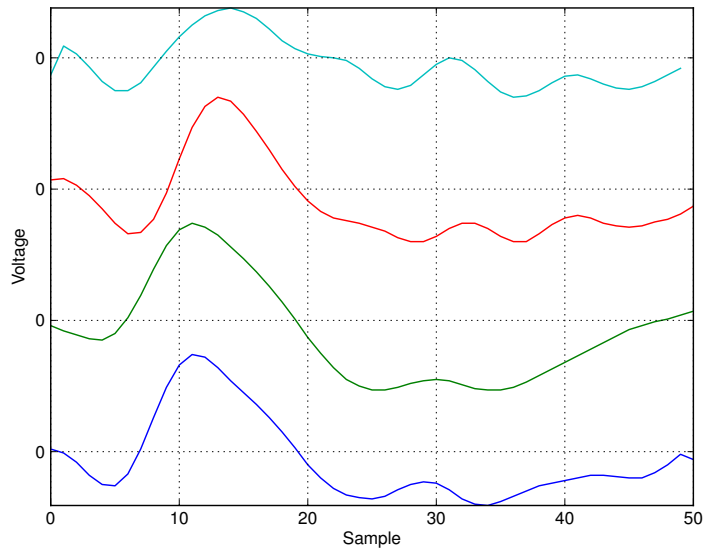


(a) Single recording site – notice that the two spikes on the right appear similar

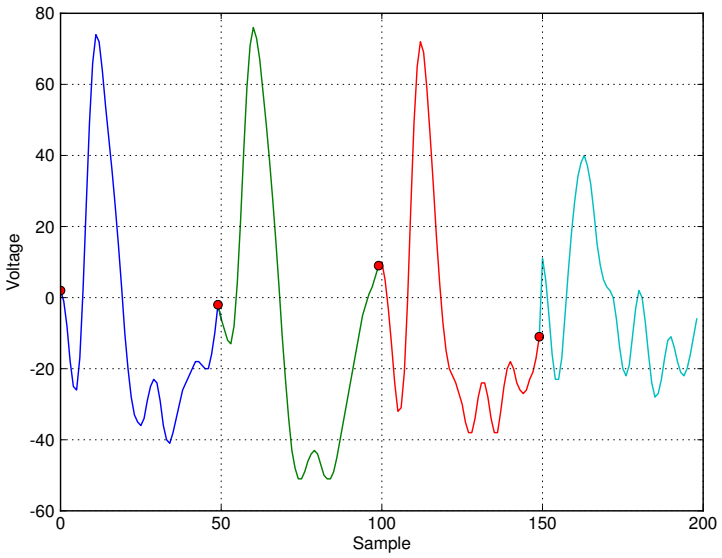


(b) Double recording site (stereotrode) – notice that the two spikes on the right appear different, depending on which fires first

Figure 1.2: A sequence of two spikes from two similar neurons, at equal distance from the recording site. With a single recording site, the spikes seem to origin from the same neuron. With a stereotrode, the distances are dissimilar for each electrode, and the difference becomes detectable.



(a) One spike, four channels, separated vertically



(b) One spike, four channels, concatenated horizontally

Figure 1.3: An illustration of a sample as recorded by a tetrode. Notice that they are both the same signal. (a) shows how they are actually recorded – simultaneously. These four channels are then concatenated into one signal as illustrated in (b). The red dots are there as visual aids only.

The next task is to classify the spikes according to which neuron was the source of which signal. This is done on the presumption that a neuron will emit a signal with a characteristic signature. The similar signals are classified in groups called *clusters*, a process often called *spike clustering*, described in Section 2.3.4.

Spike clustering can be performed either manually or automatically. The motivation for our thesis is the Kavli Institute for Systems Neuroscience at St. Olav's Hospital (KI), who currently do this in a computer assisted manual way. Using a program called Tint, they manually identify the clusters by selecting a center and a radius, or drawing a polygon. Various techniques are used to reduce the dataset down to two dimensions so that it can be presented to the operator. This is explained further in Section 2.7.6.

The KI have provided us with spike data, as well as manually clustered results, allowing us to compare our automated efforts with the results of skilled experts. We also use other datasets to test the different algorithms. All datasets are described in Section 3.1.

Our focus for this thesis is thus investigating algorithms for performing spike clustering quickly and automatically, and to see if this approach may give results comparable to those of skilled professionals. It is important that it performs clustering fast, and should make use of parallel CPU hardware where available. To verify the quality of the results from the different algorithms, we evaluate the clustering results using cluster quality algorithms, as well as comparing with the results of the manually performed clustering. Cluster quality measurement is described in Section 3.2.

1.2 Problem interpretation

The problem description is included on the first page in the report. Spike sorting may be considered as a pipeline consisting of four independent, but all important, steps: signal filtering, spike detection, feature extraction (sometimes referred to as spike modeling or dimensional reduction) and spike clustering. The hardware used to record the data used in this thesis performs signal filtering and spike detection in hardware. These steps are explained briefly in the thesis, but are not included in our implementation.

This leaves feature extraction and spike clustering. There are many ways to perform both, and there is currently no agreed-upon best choice for how to do this.

Because this is a cross-disciplinary problem, touching on both neuro-

science and computer science, we aim to make this thesis understandable for both audiences. We therefore include basic theory for both fields.

Implementing all of the interesting state-of-the-art algorithms is simply not possible within the scope of this thesis. After an extensive literature study, we have therefore selected the most promising algorithms in terms of spike sorting quality. We then apply them to the datasets provided by the Kavli Institute for Systems Neuroscience (KI), comparing the results of these to human expertise.

To utilize state-of-the-art hardware, it is necessary to develop parallel algorithms for the spike sorting process, which depending on the algorithm, can be a challenge. The problem description states we use CPUs, and focus on the ability to utilize the processing power made available with the increasing number of processor cores. Thus, using specialized hardware such as GPUs is considered beyond the scope of our thesis.

We extract the following goals:

G1 Develop a graphical application for experimenting with the different algorithms, and to enable visual inspection of results.

G2 Provide an introduction to the relevant aspects of neuro- and computer science.

G3 Identify state-of-the-art algorithms for feature extraction and spike clustering.

G4 Develop parallel implementations of the selected algorithms.

G4.1 Evaluate and optimize the performance of the implemented algorithms.

G4.2 Evaluate the quality of the clustering results performed by the implemented algorithms. This includes comparing to results of human experts, as provided by the KI.

The real-world datasets provided by the KI are bigger than what is normally described in the literature – usually at least ten times the size. How our algorithms will perform on such large sets, especially given that the classification should be done quickly, is very interesting. This leaves the following additional goals:

G5 Given the size of the KI datasets, determine if our algorithms can perform spike sorting in near real-time. With real-time we consider that

the operator should not be required to wait for an extended period of time for the results to appear. This means that the entire process must be completed within seconds – we have set an arbitrary limit of one minute.

G6 Some of the datasets received from the KI have been labeled as either easy or difficult by Tsao [69]. Our clustering efforts must classify the easy sets near-perfect, while it must perform well on the difficult sets.

1.3 Report outline

Chapter 2 describes the general background, including neuroscience, spike sorting, the need for parallelism, and the equipment used. The implementation relies on many external libraries for functionality such as task based programming and visualization, and these are explained in detail here. Finally, related work in the field of spike sorting are explored. **G2** and **G3** are covered in this chapter.

Chapter 3 describes the methodology, where we describe the datasets used and the methods of how **G5** and **G6** are solved.

Chapter 4 describes the implementation of our application, covering **G1** and **G4**. We also discuss optimization and make brief comments on our development process.

Chapter 5 describes the results of our experiments, covering **G4.1**, **G4.2**, **G5** and **G6**. We also report on the challenges we met.

Finally, Chapter 6 wraps up the project with a conclusion and thoughts about further work. We sum up how we have dealt with the goals specified in the previous section.

CHAPTER 2

Background

In this chapter, we describe the prerequisites for understanding our work. We begin with a superficial coverage of neuroscience, for readers who do not work within this field. We then describe theory of parallel programming, to set a context for further analysis, as well as providing an introduction for readers who are unfamiliar with parallel programming. After introducing these necessary concepts, we continue by describing the recording equipment, computer hardware and software libraries which are used by the implementation. Finally, we describe related work in the field of spike sorting.

2.1 Neuroscience introduction

The following is a very brief introduction to the vast field of neuroscience. It is only meant to give the reader a basic idea about why we are doing spike sorting. For a much more comprehensive introduction, Fred Rieke and Bialek [25] have written a well-known book on the subject.

Neuroscience forms the field of scientific study which tries to understand how the nervous system carries out its functions. The nervous system is a vastly complex system made up of about 100 billion neural components, with hundreds of trillions of interconnections, and many thousand kilometers of cabling (Koch and Laurent [45]).

A neuron is an excitable cell which communicates by electrical and chemical signals. It is a principal component of the nervous system, which includes the brain, the spinal cord, and the peripheral ganglia. This makes it a very interesting cell type, because it plays a part in how behavior is formed (Roberts [57]). From perceptions and movement to thoughts and behavior, the nervous system makes up a big part of what defines a consciousness.

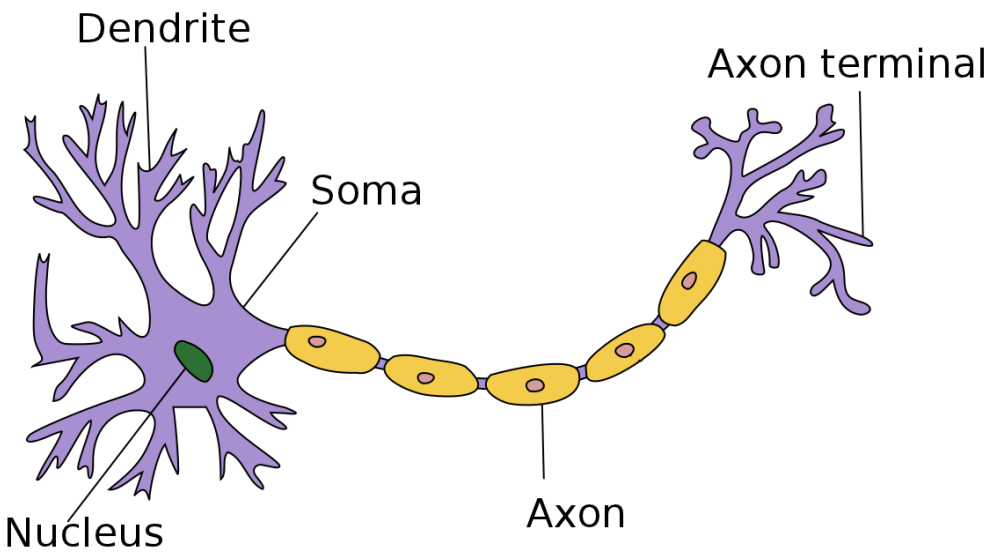


Figure 2.1: An illustration of a neuron. (Derivative work. Creative Commons, by Quasar Jarosz at en.wikipedia).

A typical neuron is shown in Figure 2.1. A number of specialized neurons exist. A sensory neuron responds to stimuli effects from the sensory organs, such as sound and touch. Motor neurons cause muscle contractions based on electrochemical from the brain and spinal chord. A neuron consists of a cell body, *dendrites* and *axons*. The cell body is called the *soma*. The soma is the central part of the neuron and contains the nucleus of the cell as well as most of the genomic expression and synthetic machinery. The dendrites are cellular extensions with many branches – often called the dendrite tree. This is where most of the input to the neuron occurs. The axon usually forms as a single cable-like structure from the soma. It can be tens or even tens of thousands times the diameter of a soma in length, and propagates electrochemical signals termed *action potentials* (nerve impulses) away from the soma. This signal can excite or inhibit other neurons. The signals are communicated between the neurons via connections called *synapses*, and this is where the neurons are closest to each other – the neurons do not touch each other directly.

It is these electrochemical signals communicated between neurons that scientists capture and want to label. The action potential is a rapid all-or-nothing signal and can travel long distances. It is a very stereotyped signal, so discrimination can be based on the firing patterns. By sorting these signals,

it is possible to isolate cells and look at their behavior. Neurons do not work in isolation, but form circuits of similar cells. This means that it is of interest to observe how networks of neurons interact. This is the goal when using tetrodes – observe and identify as many neurons as possible. Two ways to record this neural activity is by performing extra- and intra-cellular recording.

With intracellular recording, an electrode is placed inside a neuron. It can measure smaller graded action potential changes, and can accurately label the neurons.

Extracellular recording is performed by placing an electrode near a neuron, which then can capture its action potentials. With current technology, one can reliably separate signals within a radius of $50\mu m$ from the tetrode, containing on the order of 100 neurons in a rat cortex [20]. With larger distances, the signal-to-noise ratio decreases, making it hard to separate signals from background noise. Buzsáki [20] states that signals within $140\mu m$ from the tetrode can be detected, and could be classified by improving recording and clustering methods. This area corresponds to an order of 1000 neurons in the rat cortex. A potential problem with extracellular recording is that sometimes multiple neurons firing in close proximity might appear as a single signal to the electrodes. This superimposed signal is referred to as *multi-units*. Many automatic spike sorting algorithms make no distinction between the two however, and there are no clear, objective and agreed-upon criteria for making the distinction (Tankus et al. [67]), which is a problem for quantifying the accuracy of different classifiers.

Sometimes, the observed signal can deviate from its stereotypical signature. This is usually the result of muscle twitches, for instance from chewing.

From the KI, we learned that they never record more than 20 neurons in each recording, and that contribution from neurons farther away only contribute to background noise. This could possibly be increased by using other algorithms for signal filtering and spike detection (see Section 2.3), but as filtering and spike detection have already been performed on the datasets we use, we do not pursue this aspect further.

2.2 Parallelism introduction

In 1965, Moore [52] noted that the number of components in integrated circuits had doubled every year since it was invented 7 years earlier. Moore predicted that this rate would continue for at least ten years. He later mod-

ified his expectation to a doubling every two years. This has later been referred to as *Moore's law*, and has turned out to be a good estimation of the increase in capacity of microprocessors. A bi-effect of this was that one could expect computer programs to run with twice the performance, if one upgraded the hardware two years later, without altering the software.

Today, this is no longer the case. The power consumption in a standard CMOS multiprocessor is proportional to the cube of the frequency, and the heat production is proportional to the power consumption. Increasing frequency therefore gives a disproportional increase in power consumption, meaning that as the frequency is increased, performance per watt decreases. Doubling the frequency results in an order of eight times the power consumption. The power consumption has reached a top of a few hundred watts per chip level that can practically be dissipated in a mass-market computing system. This has resulted in a growth stop for single-core processor performance [26].

To solve the problem, the focus today is on parallelization of computation. We then increase the number of processors, rather than simply increasing single processor performance. In addition to increasing performance, this may lower the power consumption. Consider exchanging one processor with two processors of 0.8 times the frequency. In theory, this will give 1.6 times the performance, with $2 \cdot 0.8^3 = 1.024$ times the power consumption, i.e. a 60% performance increase, with only 2.4% increase in power consumption. Fully utilizing the two cores in a program is however usually non-trivial, so the maximal theoretic performance gain is rarely achieved.

The approach where two or more identical (homogeneous) processors are used in parallel is referred to as *symmetric multiprocessing* (SMP). Today, it is common to fit several processor cores on a single chip, which is referred to as *chip multiprocessing* (CMP). Many-core systems are scheduled for release in 2011, such as the Tilera TILE-Gx100 [12], which will have 100 identical general purpose processor cores on a single chip.

An alternate approach is to combine different processors or cores, which is referred to as *heterogeneous* systems. An example of this is the Cell Broadband Engine (CBE) [3], which consists of one *power processing element* (PPE) and eight *synergistic processing units* (SPU). The most well-known system running the CBE is the PlayStation 3.

Performing scientific calculations on graphics cards may have seemed strange ten years ago, but the graphics processing unit (GPU) has proved to be a very capable compute platform, often outperforming the traditional (x86) processor architectures in the order of magnitudes. Programs must be

explicitly programmed to run on these architectures, as they are not compatible with traditional processors. The GPUs are designed to be massively parallel and are equipped with many cores. The Fermi architecture is a recent GPU platform from NVIDIA, which supports up to 512 cores [4].

Traditional computer programs, which are not programmed to use several processors or cores simultaneously, are unable to take advantage of the increased number of processors. Their performance might even decrease, as a single core may have lower performance than older single-core processors. This means that these programs must be redesigned – often a non-trivial task. Therefore, it is now important to write software which may scale automatically with the increased number of processors [22].

Heterogeneous systems may introduce even further challenges, to optimize parts of the programs for different parts of the architecture. This also applies to GPU programming. We will not describe these in further detail, as it is beyond the scope of our project.

2.2.1 Measuring parallelism

To measure parallelism, two quantities are commonly used: parallel speedup and parallel efficiency.

Parallel speedup is the measure of how many times faster an application finishes, when executed in a parallel context. It is defined as:

$$S_p = \frac{T_1}{T_p} \tag{2.1}$$

Here, S_p is the speedup achieved when using p processors. T_p is the execution time when using p processors, meaning that T_1 is the serial execution time. T_1 refers to the execution time of the best sequential implementation. As the algorithm itself should not be altered by the parallelization, one often considers T_1 as the execution time of the parallel implementation, limited to using one processor.

Parallel efficiency is related to the total utilization of resources.

$$E = \frac{T_1}{T_p \cdot p} = \frac{S_p}{p} \tag{2.2}$$

In this formula, p still represents the number of processors, and efficiency is thus a measure of how much speedup each added processor contributes. An

efficiency of 100% means that all processors are completely utilized throughout the execution time. This is what is referred to as *linear speedup*, where all processors can do their work simultaneously. A speedup which is less than linear is referred to as *sub-linear*. One may also achieve super-linear speedup, where parallel efficiency is greater than 100%. This is often the case in searching algorithms, where the entire search is stopped once the key is found by one of the processor. Another common reason is that smaller problem sizes fit the caches of the processor cores, reducing the cost of memory operations.

2.2.2 Limits to parallelization

Parallel programming is not the solution to every problem. Some tasks have to be done serially, and cannot benefit from parallelization. Disk I/O usually has to be done sequentially, meaning that it cannot be parallelized. Consider a program which spends 20% of it's execution time reading a file from disk. No matter how efficient the remaining 80% is performed, the first 20% will dictate a minimum execution time.

This was formalized by Gene Amdahl [14], and is commonly referred to as *Amdahl's law*, as shown in Equation (2.3).

$$S(p) = \frac{p}{1 + (p-1)f} \tag{2.3}$$

$$\lim_{p \to \infty} S(p) = \frac{1}{f} \tag{2.4}$$

$S(p)$ means the speedup with p processors. f is the fraction of the program which has to be performed sequentially. This means that no matter how fast the 80% are executed, our maximum speedup is $\frac{1}{0.2} = 5$.

Amdahl assumes that the problem size is constant. Instead of assuming a constant problem size, Gustafson argued that a constant time constraint was more realistic. When the number of available processors increases, the problem size usually increases as well – keeping the total run time the same as before. He also argued that the amount of sequential work is normally fixed, and does not increase with a larger problem size.

This more optimistic outlook can be formulated as shown in Equation (2.5), and is called Gustafson's law [73].

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = p + (1-p)ft_s \tag{2.5}$$

Here, f is the fraction of the program which has to be performed sequentially. Note that it is not dependent on p. If we assume that the sequential part has lower complexity than the parallelizable part, it is clear that the fraction will decrease with larger problem sizes. This leads us to the following limit:

$$\lim_{s \to \infty} S_s(p) = p \tag{2.6}$$

This tells us that we get increased speedup with increased problem sizes. As an example, a program with a 10% serial fraction running on a system with $p = 12$ cores will get no more than a 5.71x speedup according to Amdahl's law. Using Gustafson's law and its time constraint assumption, we end up with a 10.9x speedup however. A compute node in Kongull has 12 cores, and is described in Section 2.5.

2.3 Spike sorting

Spike sorting is the process of modeling spikes using signal-processing methods in such a way that spikes which belong together, are isolated and labeled as the same type, forming clusters of related spikes that originate from the same neuron. The process consists of ordered steps, which will be explained in the following sections:

1. Signal filtering
2. Spike detection
3. Feature extraction
4. Spike clustering

All of these steps are important for the final result – an efficient and reliable clustering algorithm is of little use if one is unable to extract relevant features from the relevant spikes.

Quiroga et al. [55] state that development of reliable and efficient spike sorting algorithms is lagging behind the recent developments in recording hardware, increasing the importance of improving the algorithms. Hundreds of electrodes can now record signals simultaneously, allowing the recording of many neurons, which helps in providing a bigger picture of the activity. However, being able to classify all the recorded neurons is non-trivial, especially when they are being recorded simultaneously.

A potential problem with neuronal activity recording is making a distinction between one or several close neurons. This is sometimes not possible, and clusters are labeled either as single neurons or multi-units – collections of several neurons which fire together. Many automatic spike sorting algorithms make no distinction between the two, and there are no clear, objective and agreed-upon criteria for making the distinction [67].

Finally, perhaps the biggest challenge in spike sorting is the lack of any "ground truth". How many spiking neurons are near the recording sites, and which of the spikes belong to which neuron, is impossible to know for certain without using both intra- and extracellular recording equipment for all the experiments, such as done by Harris et al. [30]. The intracellular data can realistically only be used in limited situations because of the complicated setup. This means that it works as something to benchmark against – since in these cases you *do* know the correct answer – but is not used in normal recording sessions. Unfortunately, we were not able to obtain intracellular recordings when working with our thesis.

2.3.1 Signal filtering

The first step is to process the signal, applying a band-pass filter to avoid the low frequency activity. A narrow filter makes the spikes easy to visualize, since some of the frequencies are removed, making the main spikes stand out. On the other hand, this could remove important features of the different spike shapes. Because of this, there is no universal solution to the problem, and what is best depends on the situation. The Axona Dacq acquisition system [1] that provides the recordings used in this project report does this in hardware, enabling the users to tweak the filter window. Because all the datasets we are working on have already been recorded and classified manually, this is not explored further. We are comparing our automatic efforts to their manual efforts, so we have to compare using the same data.

However, it should be noted that it is likely that the recorded samples from the systems are not ideal. Looking at Figure 2.2 on the facing page, it seems clear that the scale is not properly aligned to the most extreme peaks, since the signal appears to be cropped at the highest peaks. We learned that this is an intentional trade-off done to achieve a higher resolution for most of the spike recording. This indicates that 8 bits is not really enough to capture the signal. This is not as important when they do manual cluster cutting, but could matter for unsupervised algorithms.

Closer inspection of the datasets shows that the problem looks worse in

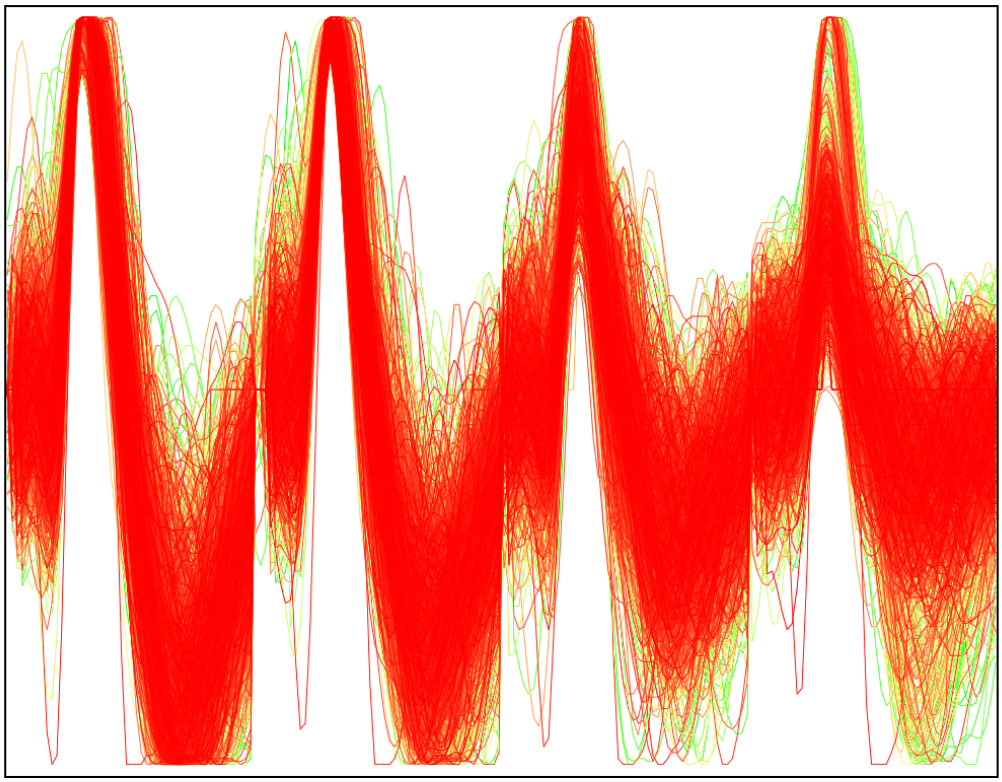


Figure 2.2: The spikes from a single cluster. Notice how some of the recorded spikes appear be cropped at the extreme amplitudes, where they flat out.

the figure, than it really is. Overflowing spikes are a low fraction of the total spikes, and it is usually only 1-3 samples which overflow the limit. On dataset 180501, described in Section 3.1, the maximum number of samples overflowing the limit in each spike (all channels in total) is 16. The fraction of spikes with four or more overflows is only 0.15%, and we therefore do not consider this a problem in the dataset. The other datasets show the same tendency.

2.3.2 Spike detection

To be able to classify and cluster spikes, it is important that the recordings contain the relevant parts of the spike, and preferably nothing else. To remove as much as possible from the waveform except the spike, it is necessary to define what constitutes a spike start and spike end. Depending on the shape of the spike and the background noise, this can be a challenge in itself. The

idle parts of the signal, where no spikes are detected, is discarded in this step.

A common way to identify the spikes is to check whether a signal crosses a given amplitude threshold, and record the signal from the channels for the duration of the spike. How to decide this threshold then becomes a challenge. If the threshold amplitude is set too low, noise signals will be considered to be spikes, resulting in false positives (type I error). If the threshold is set too high, one will fail to detect spikes with lower amplitudes (type II error). In many systems, the threshold is specified manually.

A fixed manually set threshold may be attractive for real-time applications, because of computational simplicity. However, it requires user input, and may be sensitive to noise [68]. If the electrode moves during a recording, this will also lead to an alteration of the signal, so that the chosen threshold performs differently. Another common approach is to base the threshold on statistical properties of the signals. Quiroga et al. [55] calculate the threshold using an estimation of the standard deviation of the background noise, by using the median of the filtered signal. Traver et al. [68] perform preprocessing of the signal, and compare the preprocessed signal to an adaptive threshold. They also describe how this could be used in real-time applications by considering $50ms$ frames at a time. Brychta et al. [19] describe the use of wavelet decomposition to de-noise the signal before the next step.

Chelaru and Jog [21] perform additional filtering which discards every spike recording that contains an amplitude larger than three standard deviations from the mean amplitude. They also discard any signals having a peak at a time moment greater than one standard deviation compared to the mean time moment. Their argument is that the first filtering eliminates most of the artifacts caused by external electrical disturbances and rat muscular activity, and that the second filtering eliminates most of the recordings caused by superimposed spikes (multi-unit) or noise from distant units. This results in discarding approximately $\frac{1}{4}$ of the spike recordings.

Other notable filtering contributions include unsupervised amplitude discriminator, power or energy detectors, and the matched filter [19]. The unsupervised amplitude discriminator typically discards samples that deviate from the mean signal by some multiple of the standard deviation. Power detectors use a sliding window to compute the sum of squared amplitude, and creates a threshold based on the standard deviation of the signal. The matched filter uses template waveforms identified from the signal. Although effective, this requires a manual identification of these waveforms, and is therefore not relevant for us.

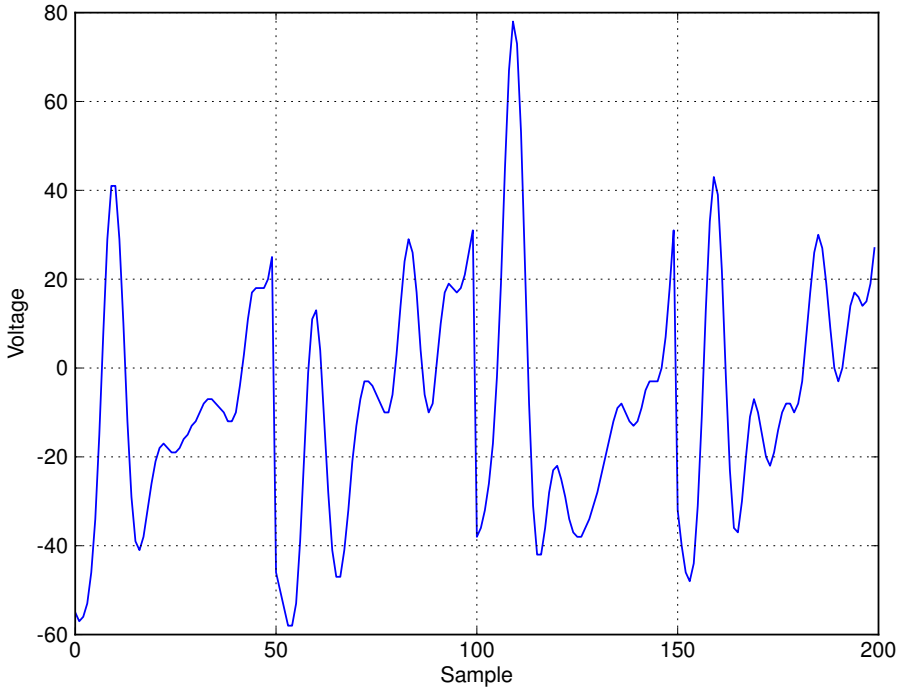


Figure 2.3: An example of a spike recording that contains noise. The stereotypical spike signature with four similar spikes is not obvious. This could be the effect of muscle twitches or multi-units.

The Axona Dacq system, used to provide the data used in this thesis, uses manual hardware amplitude discriminators. When the signal in any of the channels exceeds a given user specified threshold, the system stores the signal from all channels for the $200\mu s$ preceding the trigger event, and the following $800\mu s$. This results in 50 samples per channel for each spike, i.e. 200 samples using all four channels. For more information about the recording equipment, see Section 2.4.

To see how a spike sample that contains noise can look like, see Figure 2.3. Clearly, the spikes there are not as well isolated as in Figure 1.3 on page 5.

2.3.3 Feature extraction

Feature extraction (FE) refers to selecting the important features which identify a given spike. It is also referred to as *spike modeling*, as this decides

how the spike will be represented (modeled) before clustering. The detected spike contains a given number of samples for each channel. Feature extraction consists of representing the signal as robustly as possible, by extracting the relevant information from the input data. Ideally, the spike should be represented by as few features as possible, while remaining discriminative enough to classify the different neurons. Fewer data points means less work for the clustering algorithms, and may at the same time abstract away noise which does not help identify the spike. Indeed, too many identifiers can actually lead to overfitting, where similar spikes are not detected as similar because of many small variations. If the feature space is reduced, it is important to keep in mind that a dataset that was well-separated in its original high-dimensional form might lose this property. This could require a different, more capable clustering algorithm, which in turn might require more computational effort.

Finally, being able to visualize the results is a desirable feature. Visualization can be helpful not only for inspecting how well the clustering performs, but also for fine-tuning the clusters, moving and merging points (spikes) as desired. This introduces a need for *dimensional reduction*, as it is challenging to visualize a feature space that exceeds three dimensions. There are various tricks to add additional dimensions to the 3D visual representation, including having colors, time or shape of points represent dimensions. However, it is clear that this adds complexity to the image, especially if several of these tricks are combined at the same time. This complexity could impact the ability of a human operator to make a well-separated clustering. Thus it is desirable to use few features for visualization as well.

Some of the options for feature extraction are:

- All the samples. Using the signal as-is (no reduction)
- Peaks of the available channels (four for a tetrode)
- Peak-to-peak amplitudes
- Principal component analysis (PCA)
- Domain transformations, such as
  - Wavelet transform (frequency-time-domain)
  - Fourier transform (frequency-domain)

To model the spikes, several approaches are investigated. The easiest approach is to just use all the discrete samples as the basis for a 200 dimensional feature vector. Another option is to use the peaks of each channel as a

four dimensional vector. As expected, this is much faster to work with than the 200 dimensional case. More sophisticated methods are also explored. Principal component analysis (PCA) is a tried and tested method which also has a few known drawbacks, as described later in this chapter. These are compared to the discrete wavelet transform, one of the current state of the art ways to represent a spike.

Letelier and Weber [47] state that contrary to Fourier analysis, which represents signals with functions bounded in frequency, wavelet functions also bound the signal in time. This applies better to spikes, as each spike lasts for only one period. Because of this, and to limit our scope, we do not consider Fourier analysis further.

All the samples

The easiest approach is to use the signal as-is. The four channels are concatenated into one feature vector with 200 samples. With so many dimensions, this is a very demanding representation for advanced clustering algorithms such as SPC, which is described in Section 2.3.4. In addition to a high computational demand, it has other drawbacks.

First, it ignores the time dimension. This means that if spikes are time-skewed (shifted), but still similar, they may not be identified as similar. One way to minimize the effect of this is to align the recordings according to their highest peak. This means that some of the signal will be lost if it is shifted. The resulting gap can be zero-filled as shown in Figure 2.4(b) on the following page, or the signal can be interpolated in some way to try and keep as much of the signal as possible [55]. This helps, but because the peaks themselves can curve slightly differently, the problem is still present.

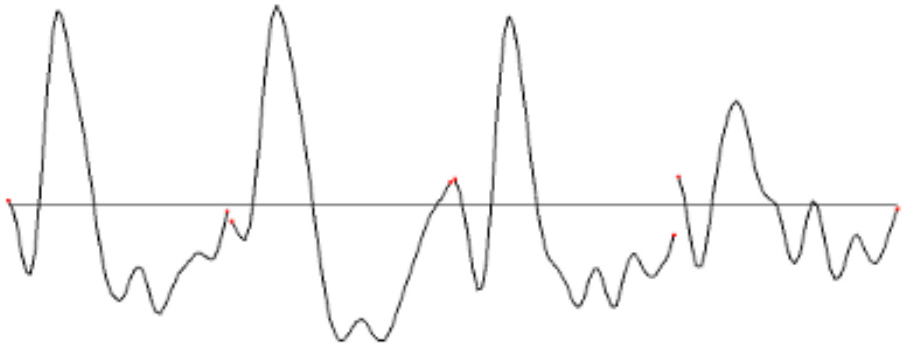
Second, comparing spikes sample-for-sample gives equal weight to all the samples, even though high amplitude peaks are more important. However, this is not as noticeable because the absolute difference between the high peaks tends to be larger than the difference between the low amplitudes.

Finally, the samples at the beginning and the end of the signal are much more likely to be unimportant noise, but they are given equal significance as the rest of the signal.

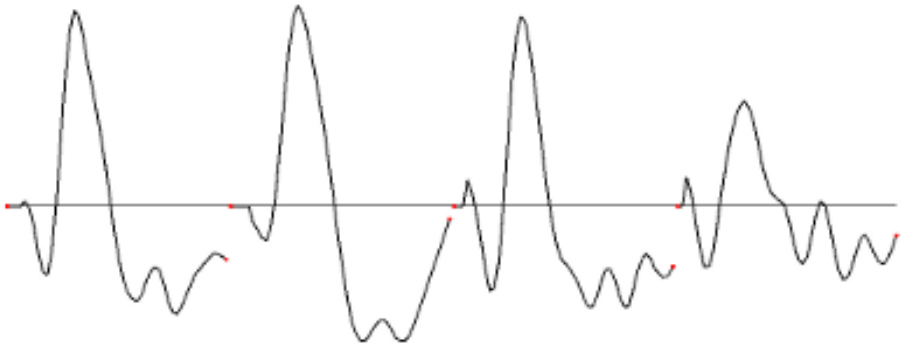
Figure 1.3(b) on page 5 shows an example of such an unprocessed spike. Using that many dimensions is computationally inefficient, the distances become less meaningful (sometimes called the Hughes phenomenon, or the curse of dimensionality [62]), and it is very hard to visualize the clustering. However, on an efficient clustering algorithm such as k-means (described in Sec-

tion 2.3.4), this performs surprisingly well.

This representation is the only one which guarantees retaining all the information, although some of the information might be of less interest.



(a) A single spike visualized



(b) A single spike with peaks aligned

Figure 2.4: (a): An unprocessed sample of a neuron spike with the four channels listed in turn, as visualized in the program. The Axona Dacq system records the four channels simultaneously, and the red dots are just visual aids here to show the distinct channels. (b): The same spike, this time with the highest peak centered at position 15 in each channel. Notice that the translation of the signal means that some information is lost (here replaced with zero).

Peaks of channels

Another simple representation is to use the peaks of each channel, which gives four samples for a tetrode. This means we search through each spike, and extract the highest amplitude in each channel. We reduce the number of

dimensions from 200 to 4, and may remove information which is important for the signal.

However, this is one of the representations used by experts at the KI for visualizing and performing manual cluster cutting. Therefore, we find it interesting to include this representation, especially for comparing our results to the ones of the experts. To visualize, they match each pair of peaks of a tetrode, giving 6 2D plots.

Principal Component Analysis

Principal component analysis (PCA) is a useful technique for performing dimensional reduction. A nice introduction is provided by Smith [64]. PCA can be used for many things, including image analysis and image compression. It is a statistical method which seeks to extract the most prominent features by maximizing the variance of the reduced dataset.

To compute the PCA of a dataset, first the mean of each dimension is subtracted from all the points. Then the covariance between all the dimensions are computed. The covariance indicates how two dimensions are related. A positive covariance means they are increasing together, a negative covariance means that as the first dimension is increasing, the second is decreasing. A higher value means a stronger relationship. If the variance is zero, or close to it, it suggests the two dimensions are unrelated. The results of the covariance between all the dimensions are stored in a covariance matrix. The eigenvectors and eigenvalues from the covariance matrix are then extracted. The eigenvectors are per definition perpendicular, and these form an orthogonal basis for the dataset. If they are sorted by the eigenvalues, they are ordered in decreasingly important dimensions. After sorting, the first component is called the principal component, and so on.

In a 2D example, the first component would be the line which fitted the data points the most, while the second component would indicate each point's variance relative to that line.

To derive the new dataset, the transposed matrix of eigenvectors is multiplied by the transposed of the original mean adjusted data. If all the eigenvectors are used, the result is simply the original dataset rotated so that the eigenvectors are the axes. It is a reversible operation, by simply multiplying both sides from the left by the inverse matrix of eigenvectors.

To achieve dimensional reduction, we must discard the least important eigenvectors. If we start removing these eigenvectors (sorted using the eigenvalues), we are left with a reduced dataset, containing most of the energy of

the signal and hopefully still representing the spikes accurate enough to be used as a discriminator. Because the most important vectors appear first, PCA is often used as an approximation tool for visualizing higher dimensional data in 2D or 3D.

Although the first few PCA components the data is projected on maximizes the variance of the data, it does not necessarily provide an optimal separation of the clusters. In the comparisons done by Quiroga et al. [55], PCA features actually gave slightly worse results than the clustering using all the samples from the spike recordings. This could be compensated for by using more PCA components, or a more computationally demanding clustering algorithm which classifies better, since the number of features in the dataset could still be drastically reduced from the original 200 features. Quiroga et al. [55] also state that although PCA always uses the eigenvectors corresponding to the highest values, these may not be the directions which best separate the spike classes. The information which separates the classes may also be represented by the components corresponding to lower eigenvalues, which are disregarded.

Wavelet transform

The wavelet transform (WT) is an advanced signal representation algorithm that performs a time-frequency decomposition of the signal. In addition to providing resolution in both the time and the frequency domains, it eliminates the requirement of periodic signals [55]. The addition of time resolution means that it should be better at determining similarities between spikes which are time-skewed, than for instance the Fourier transform [70]. Compared to PCA, it contains more information about the shape of the spikes in several coefficients [55], which could be advantageous for cluster identification (but not necessarily for visualization).

Technically, wavelets are a set of non-linear bases. A basis is a set of linearly independent vectors which span a vector space – they define a coordinate system. It is advantageous to match the bases to the dataset, because that means the datapoints will require less information to be accurately represented. A static basis in a two-dimensional space $\mathbb{R}^2$ could be $v_1 = (1, 0)$ and $v_2 = (0, 1)$. A dynamic basis could be $v_1 = (f, 0)$ and $v_2 = (0, g)$, which will depend on a state. Wavelets use dynamic bases functions that represent the input in an efficient manner. This means that it is a great way to compress a signal, and it is therefore popular in image and signal processing, perhaps most notably in JPEG 2000 [6].

Wavelets are generally categorized as either continuous wavelet transforms (CWT), discrete wavelet transforms (DWT) or multi-resolution discrete wavelet transforms. The signals we work with in this thesis are already discretized. Because it is desirable to be able to see the signal at different resolutions, and it is difficult to know beforehand which set of coefficients will prove most useful, a multi-resolution DWT is used in this thesis.

The wavelet transform is defined as the convolution between the signal $x(t)$ and a number of wavelet functions:

$$W_\psi X(s,t) = \langle x(t) \mid \psi_{s,\tau}(t) \rangle \tag{2.7}$$

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{|s|}} \psi(\frac{t-\tau}{s}) \tag{2.8}$$

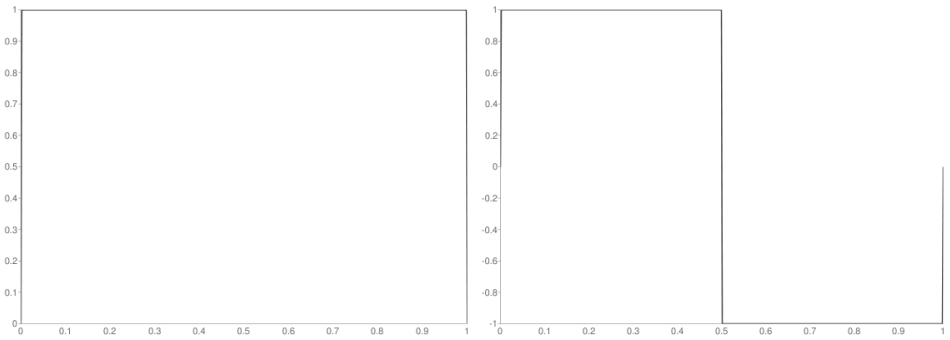
where the basis functions $\psi_{s,\tau}(t)$ are the decomposed (transformed and shifted) versions of the so-called mother wavelet function $\psi(t)$. s is the scale of the transform, τ is the translation parameter. The mother function is an integrable function which is dilated and shifted.

A contracted wavelet function (often called dilated) represents the high frequency components, while an expanded function represents the low frequency components. The expanded function creates a low pass result which is a smoother version of the original signal. In the case of Haar wavelets, they represent the average. This result is used recursively to input the next wavelet step, where another set of high and low pass results are computed, until only a single low pass result (2^0) is calculated. By correlating different sizes of the wavelets with the original signal, you can obtain details at several scales [55]. A small scale will show the details, a large scale will show the "big picture". The scales can be combined to perform a hierarchical clustering, called a multiresolution decomposition.

The wavelet transform has proved to be quite capable at representing spikes, and is used successfully in a number of published articles. Quiroga et al. [55] use Haar wavelets, a four-level hierarchical decomposition using rescaled square functions. The Haar wavelets enable a representation of the spike recordings with few wavelet coefficients and make no assumptions about the spike shape. For more information about the stationary wavelet transform (SWT), [19, 36, 37, 44] provide details of implementations, and demonstrate better results than conventional methods.

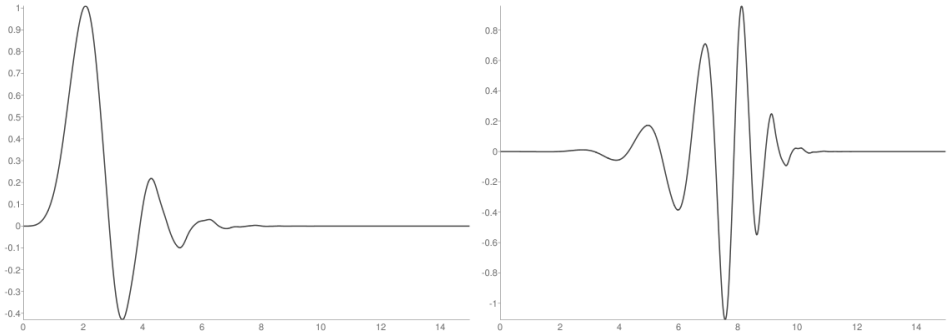
In addition to Haar wavelets, we use Daubechies wavelets. These exist for different numbers of coefficients, named *D2* and upwards. D2 is the same as Haar wavelets. Scaling functions and wavelet functions for Haar and

Daubechies are shown in Figure 2.5 and Figure 2.6



(a) Haar scaling function (b) Haar wavelet function

Figure 2.5: Haar wavelet [72]

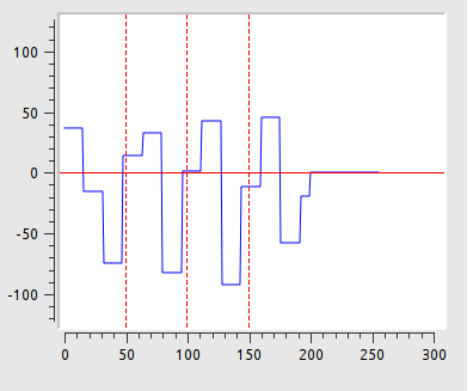


(a) Daubechies D8 scaling function (b) Daubechies D8 wavelet function

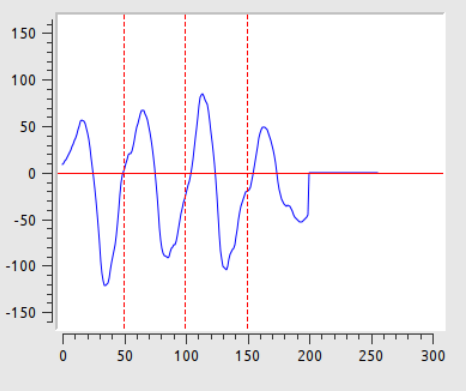
Figure 2.6: Daubechies D8 wavelet [72]

As Daubechies wavelets contain more coefficients than Haar wavelets, signals going through a Daubechies transform and back again will look smoother than when using Haar. This is demonstrated in Figure 2.7 on the facing page, where a concatenated spike signal (200 samples) is sent through an 8-level transform and reconstructed from the first 16 components (4 levels), and 32 components (5 levels). The step function shape from the Haar wavelet is recognizable throughout the reconstructed signal.

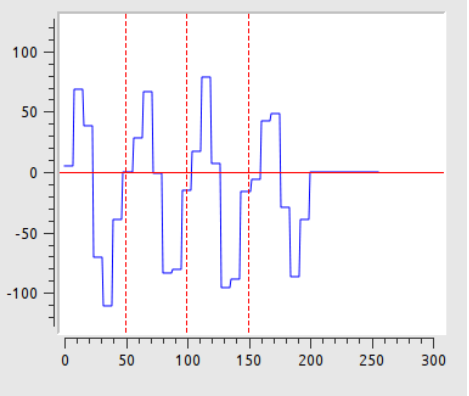
After performing a wavelet transform, we have to choose the components which represent the important features of the spike. Extracting the first n components for each channel of the signal will represent the low-frequency transitions for each channel well. Extracting the first n components for a



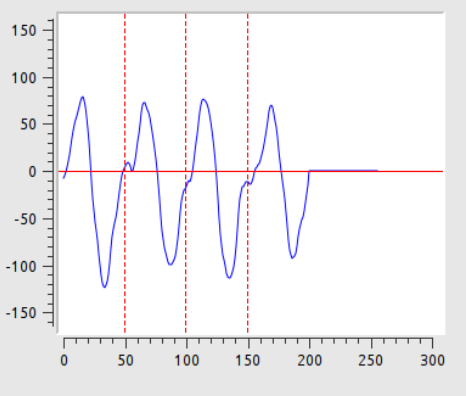
(a) Haar (16 components)



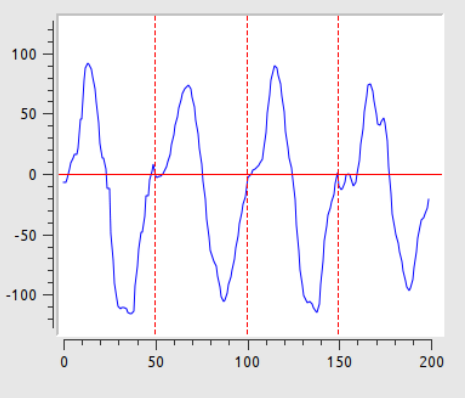
(b) Daubechies D8 (16 components)



(c) Haar (32 components)



(d) Daubechies D8 (32 components)



(e) Original signal

Figure 2.7: Result of transforming signal of 200 samples through an 8-level wavelet transform (Haar and D8), and applying the reverse transform on the first 16 or 32 components.

concatenated signal will represent the low-frequency transitions for the signal in total.

A possibly better approach is using the Lilliefors modification of the Kolmogorov-Smirnov test for normality, as explained by Quiroga et al. [55]. It is based on finding the dimensions which to highest degree deviate from a normal distribution. This should identify the dimensions which best separate the different clusters. We first calculate the cumulative distribution function for each dimension, and compare it to the Gaussian distribution with the mean and variance equal to the samples in the dimensions. Then we extract the dimensions with the highest difference between these functions for any spike.

We have not dived into the depth of the mathematics for Wavelet transforms, as the implementation is available to us through the GNU Scientific Library, as described in Section 2.6, and for more information, see [29] and [70].

2.3.4 Clustering

Cluster classification is the process of organizing a set of observations into groups (clusters). Clustering is important for many fields of research, including machine learning, data mining, statistics, pattern recognition and image analysis. Spike clustering aims to group similar spikes together, based on the assumption that similar spikes originate from the same neuron, so that each cluster represents a neuron (or similar multi-unit).

Generally, there are three ways to classify a dataset. The first class assigns all the points to a cluster. The second only assigns a cluster to points that have a strong similarity to other points – high density regions. The last class has degrees of membership to clusters; a single point can be 30% in cluster A and 70% in cluster B, for instance. Such a fuzzy clustering is not relevant to our problem and is not explored further; a spike can only originate from a single neuron.

A *parametric* clustering makes assumptions about the structure of the clusters. This could be the number of clusters present or how the clusters are distributed. K-means, described in Section 2.3.4, is an example of a parametric clustering algorithm. A *nonparametric* algorithm does not make any assumptions and can thus be used when there is no a priori knowledge about the data structure. SPC, described in Section 2.3.4, is an example of a nonparametric algorithm.

Usually, what constitutes a good or bad clustering classification is highly

problem dependent. This is demonstrated in Figure 2.8 on the next page, where all of the classifications appear perfectly valid. Which partitioning is the better one is impossible to answer without domain-specific knowledge of the problem.

A variety of clustering algorithms have been developed. Generally, it is desirable that the algorithm does not assume a predefined number of clusters. However, if the clustering algorithm is fast enough, it can be run multiple times with a different amount of clusters each time. The different clusterings can then be benchmarked against each other using a variety of cluster quality metrics, of which we will describe some in Section 3.2. Many also assume a typical Gaussian distribution, which depending on the dataset can make a significant difference. If a dataset is linearly separable, it means that there is a way to separate two groups of points by drawing a hyperplane between them.

Tan et al. [66] group the different data distributions into the following categories:

Well-separated a distribution where the distance between any two points in different groups are larger than the distance between any two points within a group.

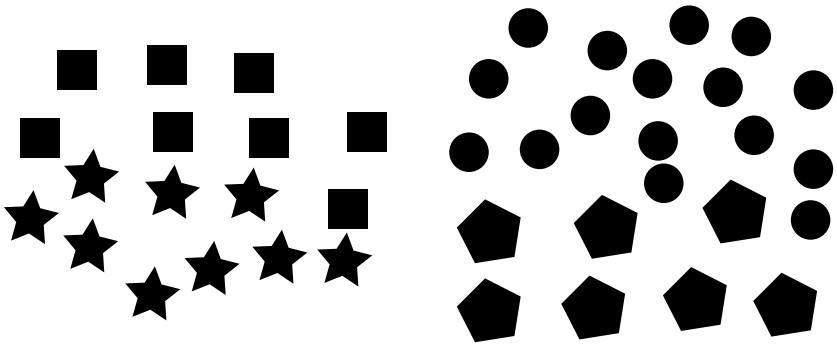
Prototype-based a distribution where every data point is closer to the prototype that defines the cluster than the prototype of any other cluster. K-means is a prototype-based clustering algorithm.

Graph-based each point is closer to at least one point in its own cluster than to any point in other clusters. SPC is a graph-based algorithm.

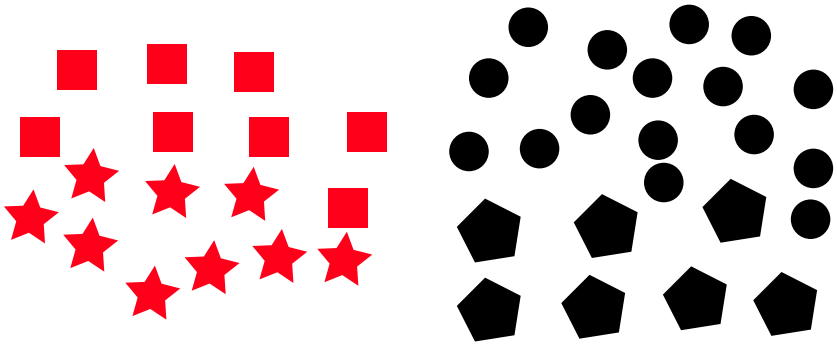
Density based clusters are defined by partitioning high density points that are separated by regions of low density.

Conceptual clusters where points in a cluster together make up a conceptual figure.

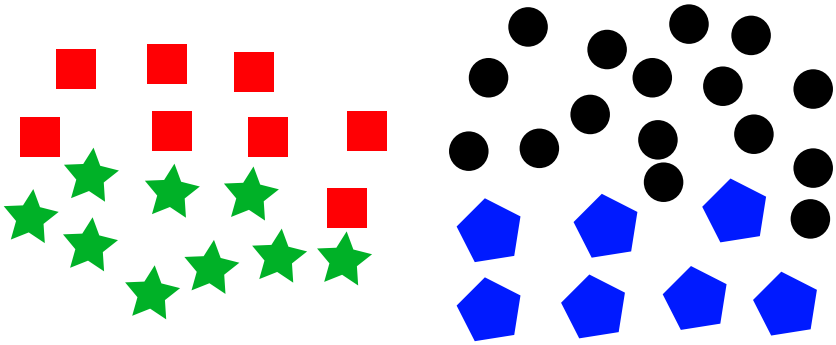
Which algorithm is best will depend on the distribution. In this project, two clustering algorithms are explored. The k-means clustering algorithm was chosen due to its simplicity and computational efficiency, and that it is a class of the partitioning clustering algorithms that classifies everything. Superparamagnetic clustering (SPC) is a clustering algorithm which only classifies points it thinks belong together. This means it will find the number of clusters at runtime, and it should be less susceptible to outliers and noisy



(a) The original datapoints



(b) After setting separate colors to the two “islands”, two clusters appears to be a good discriminator



(c) Adding more colors, and 4 clusters starts to look like a good clustering.

Figure 2.8: Three examples of why clustering is a highly subjective discipline. To the human eye, features such as shape, color and distance can make you biased when choosing clusters. The significance of these features will depend on the problem.

data. What is and is not noise is as mentioned a very subjective decision, but we will evaluate how SPC handles noise, both with synthetic and real datasets. After an extensive literature search, and advice from our supervisor, this algorithm appears to be the most promising overall for this problem, and was thus chosen for implementation.

The rest of this section will describe the two implemented clustering algorithms, k-means and SPC.

K-means clustering

K-means clustering is a fast clustering algorithm which partitions P points in D dimensions into K clusters, minimizing the sum of squares within each cluster. With a large dataset, and when $K > 2$, it is not practical to require a global minimum where the solution has the minimal sum of squares across all clusters, because a local minimum in one cluster will affect another cluster negatively. Instead, a local optimum is expected, where no movement of points from one cluster to another will reduce the sum of squares within the cluster [31]. In k-means, the mean of each cluster is called a centroid. If the centroid is required to be on an actual datapoint, and not the true mean, it is called k-medoids clustering. To compute similarity, some form of distance metric must be used. We have used Euclidean distance in our implementation, but other measures such as the Manhattan distance could make more sense for other problems.

K-means is a fast, well-known algorithm. A high-level overview is shown in Algorithm 2.1.

Algorithm 2.1 High-level serial k-means

```
1: Distribute K centroids according to some heuristic, as explained later in
   this section.
2: repeat
3:    for all point in points do
4:       calculate which centroid is nearest, and assign point to that centroid
5:    end for
6:    for all centroid in centroids do
7:       recalculate centroid position as a mean of all the current points in
         its cluster
8:    end for
9: until stop criteria (number of iterations, number of membership changes,
   distance moved, etc)
```

Algorithm 4.1 on page 77 lists the psuedo-code of our implementation of algorithm, which uses two stop criterias – fraction of membership changes per iteration, and a max total iteration threshold. The latter is there in case the algorithm converges too slowly, something we rarely experienced.

Because each iteration only consists of a distance measure and comparison between the P points and K centroids in D dimensions, and finally a position update of just a few centroids (relative to the number of points in the dataset), this is a very fast clustering algorithm. The distance calculation requires an addition, two subtractions and a multiplication, $N \cdot K \cdot D$ times, every iteration. The centroid updates requires an addition, $N \cdot D$ times, also every iteration. Every iteration is $\mathcal{O}(N \cdot D \cdot K)$. The parallel implementation is $\mathcal{O}(\frac{N \cdot D \cdot K}{P})$, where P is the number of cores. Inaba et al. [38] found that the entire algorithm was $\mathcal{O}(N^{DK})$. This is, however, a very hard thing to estimate, because the number of iterations required for convergence depends so much on the distribution of data, as well as how the centroids are placed initially. Our implementation will be benchmarked to see how it scales in practice.

Most of the demanding parts of the algorithm can be parallelized. The expensive part of k-means is the distance calculation between points and centroids - or N points times K centroids operations. It is clear that these operations can be handled independently, so it also maps well to the parallel paradigm.

However, k-means clustering comes with several drawbacks. The number of K clusters has to be specified a priori, it is dependent on where the centroids are placed initially, it tends to create equally-sized clusters, it is unduly influenced by points far away, and it can not be expected to converge to a global minimum [63].

The need to specify the number of K clusters initially can be partly averted by running the algorithm with $K = \{N \dots M\}$ clusters, and then using a cluster quality metric to automatically determine which clustering appears to be the best. This of course makes the algorithm slower.

There are various heuristics to initially distribute the centroids, which can help to give a good solution more often. A random distribution which ignores actual point position and only regards the bounding box that the datapoints create is fast but unfair. It completely ignores point density, and several centroids could be placed in peripheral areas defined only by a few datapoints. A random distribution which can only be placed where actual datapoints lie should be better, because it will be influenced more by point density. Another clever trick is to randomly place the first centroid,

then place each successive point on a datapoint as far away from the closest centroid as possible. This should make the centroids more evenly spaced out, and could make it converge faster (however, this is a consuming process in itself). While it will distribute the clusters more evenly, it will be susceptible to outliers. These initialization algorithms are implemented in our thesis.

Bisecting k-means [66] is an alternative which is less prone to the initial centroid placement, which performs a series of recursive bisections until K clusters have been produced. It works by performing several trial bisections, computing the sum of squared errors between the centroids and the points, and then selecting the the clustering which minimized this. This would make it a more demanding algorithm, but it could certainly make the initial centroid placement more robust.

The k-means implementation in this project is very fast, so multiple clusterings can be performed instead, making the initial centroid placement less important. Although fast, k-means is sensitive to the initial centroid distribution, so the convergence time will vary on the same dataset.

Because the more fundamental problems with k-means are not addressed, such as how it tends to create equally sized clusters, this variation of k-means is not explored further.

Superparamagnetic clustering

Superparamagnetic clustering (SPC) is a non-parametric clustrering algorithm, presented by Blatt et al. [17, 18], and has been successfully applied to spike sorting by Quiroga et al. [55]. It is based on theory from the Potts model [74], which describes interaction between spins on a crystalline lattice, and is used to describe behavior of ferromagnets in varying temperatures.

A spin is simply a point on the lattice with a state ("spin") q, ranging from 1 to Q. The points on the lattice are called spins because of the way they will rotate to align with an external magnetic field. When two points have the same spin, they are said to be aligned. Increasing the temperature will increase the entropy, which reduces the influence of neighboring spins. This is related to the Curie temperatures (T_c) of ferromagnets, which is where the ferromagnet becomes paramagnetic. Above this temperature, alignment will be random, and a ferromagnet no longer be influenced by a magnetic field (magnetic susceptibility tends to 0). As an example, the Curie temperature of iron (Fe) is $770°C$, and above this temperature, iron will no longer be attracted to a magnet.

Blatt et al. [17] give a good description of the process, in an example with

three dense regions:

> At high temperatures, the system is in a disordered (paramagnetic) phase. As the temperature is lowered a transition to a *superparamagnetic phase* occurs; spins within the same high density region become completely aligned, while different regions remain unordered. As the temperature is further lowered, the effective coupling between the three clusters (induced via the dilute background spins) increases, until they become aligned. [...] we call this "phase" of aligned clusters ferromagnetic.

Hence, the algorithm consists of two major steps: locating the superparamagnetic phases, and determining how the points cluster together in these phases. As some values used throughout the algorithm are determined by the input data, and not dependent on other variables, we can consider preparing these as a first step, giving us three parts of the algorithm:

1. Calculate values depending only on input
2. Locate the superparamagnetic regions
3. Perform clustering

Calculating values depending only on input The input to the algorithm is a vector of size N, containing D-dimensional points, $P_1 ... P_N$. As the calculations further down the algorithm only use the intra-point distances, they are calculated in the beginning. Several options are available, but our implementation uses Euclidean distance to determine how different two samples are.

Interaction happens between points which are defined as neighbors. There are several ways to define neighborhood, including Voronoi tessellation and k-nearest neighbors. We use the mutual K nearest neighbors, meaning two points v_i and v_j are neighbors if and only if v_j is one of v_i's K nearest neighbors and v_i is one of v_j's K nearest neighbors, according to Euclidean distance. This is the same definition as used by Blatt et al. [18].

The next step is to calculate interaction strength J_{ij} for every pair of points. This is defined as

$$J_{ij} = \begin{cases} \frac{1}{K} \exp(-\frac{d_{ij}^2}{2a^2}) & \text{if } v_i \text{ and } v_j \text{ are neighbors} \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

Here, $\hat{K}$ is the average number of neighbors for all points, and a is the average of all distances d_{ij} between all neighboring pairs of points v_i and v_j.

$$\hat{K} = \frac{2 \times \texttt{numNeighborEdges}}{\texttt{numPoints}} \tag{2.10}$$

The interaction strength J_{ij} between each pair of neighboring points (i, j) is then the only quantity used from the input samples. For larger datasets, these values may be stored for subsequent runs of the algorithm, saving calculations.

Locating the superparamagnetic regions The goal for this step, is to determine which of the temperatures give best results, so that one can perform clustering using these temperatures. This can be calculated analytically, as demonstrated by Blatt et al. [18]. This means that for a selected set of temperatures, each possible configuration of the system is checked, and the thermal average of the magnetization is used to determine susceptibility, χ, for the temperature. With each N point having one of Q possible spins, this gives N^Q configurations to check for each temperature. Even for small datasets, this gives an impractically high number of configurations, resulting in high complexity.

To solve this problem, we instead use Monte Carlo simulations, specifically the Swendsen-Wang (SW) method [65, 18], as described in Algorithm 2.2 on the next page.

Equation (2.11) shows the probability of two neighboring points with the same spin state, being added to the same SW-cluster.

$$p_{ij} = 1 - exp\left(-\frac{J_{ij}}{T}\right) \tag{2.11}$$

The quantity that we are interested in finding is the susceptibility, χ. This is proportional to the variance in magnetization between the Monte Carlo iterations. The magnetization is defined in Equation (2.12), where N_{max} is the maximum number of points in all spin states. N is the total number of points in the dataset.

$$m = \frac{qN_{max} - N}{(q-1)N} \tag{2.12}$$

$$\chi = \overline{m^2} - \overline{m}^2 \tag{2.13}$$

Algorithm 2.2 Swendsen-Wang: Susceptibility

Require: points $[1 \dots n]$
Require: $J[1 \dots n, 1 \dots n]$: interactionStrengths
Require: temperature $\in \mathbb{R}^+$
Require: iterationCount $\in \mathbb{N}$
for all (i, j) in J **do**
  calculate probability $P[i, j]$ {see Equation (2.11)}
end for
spins$[1 \dots n]$:= random spin-values $[1 \dots q]$
for iteration := 1 **to** iterationCount **do**
  g := graph, vertices$[1 \dots n]$, edges[]
  for all (i, j) in P **do**
    if spins$[i]$ = spins$[j]$ **and** $P[i, j] >$ uniformRandom$(0.0, 1.0)$ **then**
      add (i, j) to g.edges
    end if
  end for
  for all connected subgraphs g_c in g **do**
    newSpin = random spin-value $[1 \dots q]$
    for all i in g_c **do**
      spins$[i]$ = newSpin
    end for
  end for
  N_{max} = number of points in largest spinState
  mag$[i]$ = calculateMagnetization(N_{max}) {see Equation (2.12)}
  magSq$[i]$ = mag$[i]^2$
end for
avgMag = average(mag)
avgSqMag = average(magSq)
susceptibility = N / temperature * (avgSqMag - avgMag^2)
return susceptibility

We calculate the susceptibility, χ, for a range of temperatures, using the definition in Equation (2.13) on page 35. This will give a graph which has peaks at the temperatures where one should perform clustering in the final step of the algorithm. As described by Blatt et al. [18], the average in the physical model is a *thermodynamic* average, which should be calculated over all possible spin configurations. This would also include improbable spin configurations, which are then given lower weight in the thermodynamic average. When using Monte Carlo iterations like the SW method, however, after a few iterations, the system will change between probable configurations, reducing the average to an *arithmetic* average, as described in Equation (2.14).

$$\overline{x} = \frac{\sum_i x_i}{\text{numIterations}} \tag{2.14}$$

If calculated analytically, the plot will be smooth, with defined peaks visible only in transition phases. The plots shown in Blatt et al. [18] and Blatt et al. [17] are smooth, indicating that they have been calculated analytically. With Monte Carlo iterations, however, getting a smooth plot requires many iterations. With fewer iterations, the plot will be noisy, but show the same structure. Therefore, it is possible to reduce the number of iterations in a first run, and then increase the number of iterations in a more focused area. For examples of the plots, see Section 4.4.2.

Perform clustering The final step of the algorithm is to perform clustering with a given temperature. The algorithm is very similar to calculating the susceptibility in Algorithm 2.2 on the facing page. It is shown in Algorithm 2.3 on the next page for completeness.

Points are added to the same cluster if they have the same spin for more than a given fraction, θ, of the iterations. Blatt et al. [18] demonstrate how this value is not important for the results, as long as it is "bigger than 1/q and less than 1-2/q". Therefore, we have set this to 0.5.

For correctness, the correlation C_{ij} should be used to calculate the correlation function G_{ij}. Instead, we have augmented θ to θ_m, so that the relation is correct, as derived in the following equations.

$$G_{ij} > \theta \tag{2.15}$$

$$G_{ij} = \frac{(q-1)C_{ij} + 1}{q} \tag{2.16}$$

Algorithm 2.3 Swendsen-Wang: Clustering

Require: points $[1 \dots n]$
Require: $J[1 \dots n, 1 \dots n]$: interactionStrengths
Require: temperature $\in \mathbb{R}^+$
Require: iterationCount $\in \mathbb{N}$
Require: θ : clustering threshold

for all (i, j) in J **do**
  calculate probability $P[i, j]$ {see Equation (2.11)}
end for
spins$[1 \dots n]$:= random spin-values $[1 \dots q]$
correlation := $[1 \dots n][1 \dots n] := 0$
for iteration := 1 **to** iterationCount **do**
  g := graph, vertices$[1 \dots n]$, edges[]
  for all (i, j) in P **do**
    if spins$[i]$ = spins$[j]$ **and** $P[i, j] >$ uniformRandom$(0.0, 1.0)$ **then**
      add (i, j) to g.edges
    end if
  end for
  for all connected subgraphs g_c in g **do**
    newSpin = random spin-value $[1 \dots q]$
    for all i in g_c **do**
      spins$[i]$ = newSpin
    end for
  end for
  for all (i, j) in P **do**
    if spins$[i]$ = spins$[j]$ **then**
      correlation[i][j] := correlation[i][j] + 1
    end if
  end for
end for
correlation := correlation / iterationCount
h := graph, vertices$[1 \dots n]$, edges[]
for all (i, j) in P **do**
  if correlation[i][j] ¿ θ **then**
    add (i, j) to h.edges
  end if
end for
clusters := connected subgraphs in h
return clusters

$$C_{ij} > \frac{q\theta - 1}{q - 1} = \theta_m \tag{2.17}$$

The output from the final part of the algorithm is a collection of sets containing point indices for each cluster.

2.4 Spike recording equipment

The datasets provided to us for this thesis have been recorded using the Axona DacqUSB [1] recording system. It is a compact multi-channel data acquisition system which is used with a PC. It is designed to record data from twisted quadruple electrodes called tetrodes. The tetrodes are implanted into living tissue, where it records electrophysiological signals. Spike events will trigger simultaneous recording on all four channels.

The amplifier modules support 16 channels each, and four modules can be set up in a group for a total of 64 channels. It performs peak detection using a specified threshold. Using the digital oscilloscope, the user can control settings such as filtering, gain and triggering. When the signal crosses the threshold, all the channels will record for the $200\mu s$ preceding and $800\mu s$ following the trigger event. The recording is captured in a fixed number of 50 samples per channel, or 200 in total for all four channels. Each sample is stored using 8 bits.

The Axona system supports a directional video tracker, which enables the recording of metrics such as position and orientation. In this thesis such information is not relevant and thus not used.

Tint, a cluster cutting and analysis software, is provided to manage the recorded data. It is reviewed in Section 2.7.6.

2.5 Computer hardware

Our program is developed on an Intel Core II Duo E6400, running GNU/Linux. For testing, we have used Kongull, a super computer at NTNU. Kongull consists of several nodes, which may be used for multi-node parallel programming. Our program uses the shared memory model, meaning that we have only implemented parallel utilization of the available processor cores in a single node.

The computer we have used to develop the application was provided by IDI Drift, and is out-dated, as the processor is from 2006. Because of its age, we also chose to run our application on a single node on Kongull, as its specifications represent state-of-the-art to a much higher extent.

GPU could also be an attractive platform, especially given that most calculations in this project are floating point operations, but requires a different way of programming. As mentioned in Section 1.2, we consider GPU to be beyond the scope of this thesis.

As our implementation will be programmed to utilize all available cores, our ideal platform is any high-performing shared memory computer.

Hardware specifications are summarized in Table 2.1.

Table 2.1: Hardware

	Desktop	**Kongull**
CPU	Intel Core II Duo E6400	2 × AMD Opteron 2431 (Istanbul)
Frequency	2.13 GHz	2.4 GHz
Cores	2	$2 \times 6 = 12$
L1 cache	64 kB per core	128 kB per core
L2 cache	4MB	512kB per core
L3 cache	N/A	6MB

2.6 External libraries

In this section, we will describe the different libraries we have used to implement our program.

To make sure the application and code has as few restrictions as possible, we have only used free, open-source libraries.

2.6.1 Intel Threading Building Blocks (TBB)

Intel TBB [39] is a library for developing multi-threaded C and C++ programs. It covers algorithms and data structures for efficient parallel programming and parallel execution. It is designed to abstract threads away from the user. The most basic feature is the `parallel_for` construct, which allows specifying that a `for`-loop may be executed in parallel, utilizing the available processors/cores. This construct is similar to what is available with OpenMP, described in the next section, but as TBB is a library, it does not use *pragma*

statements, and does not depend on the compiler having built-in support for the parallelization to work. However, the syntax is slightly different, and the effort needed is a bit higher in TBB. On a lower level, the `parallel_for` is split in several *tasks* consisting of subranges of the for-loop-iteration.

Intel TBB also allows explicit coding of such tasks, enabling better control with dependencies between tasks, as well as parallelizing code which is not structured as simple loops.

When programming in the task based paradigm, you specify which part of the code is parallelizable, and leave it up to the task scheduler to determine how to execute these logical tasks. The task scheduler runtime will try to create an optimal number of threads, and map these tasks to the threads as they become available. This has a number of benefits. Since the task scheduler will reuse threads, it allows for a faster startup and shutdown of parallel sections. Because it creates a dependency graph of the tasks, it load-balances the tasks automatically.

Forcing the programmer to think at a higher, task-based level may be beneficial, because it allows one to disconnect from the mindset that there is one logical thread per physical thread. The programmer specifies task dependencies, and leaves the scheduling up to TBB.

The scheduler is one of the most important aspects of the library. It is designed to evaluate a graph of tasks. One of the design goals of it is *not* to be fair, because it has additional information it could exploit to create a better overall performance.

In the graph, each task points to its successor, which is waiting for it to complete, or `NULL` if it is finished. The result is a directed graph, which the scheduler uses to determine which task to schedule next. For task selection, a combination of breadth-first and depth-first is used: breadth-first is excellent for parallel execution, while depth-first is better for serial execution.

Breadth-first is used to keep the maximum number of cores busy. It will not create more tasks than what is needed to keep the cores busy, as a breadth-first approach is very memory demanding.

Depth-first is used for serial execution because the deepest tasks are the most recently created tasks, and therefore is more likely to have relevant data in the caches (or as the manual calls this: “strike while the cache is hot”). In addition it minimizes memory requirements because only a linear number of tasks have to exist at the same time [40].

Figure 2.9 on page 43 provides a nice overview of the TBB runtime

schematic. Each logical thread has its own deque[1], containing tasks that are ready to run. A task is run when its `execute` method is called. Initially when a task is spawned, it is added to the bottom of the deque. The algorithm for selecting a task from the deque, explained in detail in the TBB tutorial [40], performs the following steps:

1. Unless `execute` returns `NULL`, grab the task returned.
2. Unless its own deque is empty, remove a task from the bottom of the deque. This is its own most recent task.
3. Finally, if it still lacks a task, steal a task from the top of the deque of another thread. If the deque is empty, continue trying until successful.

The manual calls this strategy "breadth-first theft, depth-first work". If the logical thread cannot work on tasks in its own deque, it will steal the oldest one from one of the other threads. The oldest one is picked because it is the least likely to disrupt the cache flow.

Notice that there is no single "master thread" managing which task should go where. This means that the work assignment itself is distributed.

Finally, TBB supports several ways to control how the task graph is created, such as *recursive chain reaction*, which creates a tree quicker, and *continuation passing*, which allows the bulk of a parent task to continue in a *continuation task* while it is waiting for its children. For a detailed explanation, refer to the tutorial [40].

An example of a task graph is shown in Figure 2.10 on page 44. Here, tasks A, B and C have spawned tasks they are waiting upon. Task D is a running task which has not spawned any children, so its *refcount* is unset. Tasks E, F and G are spawned, but not running yet.

The TBB version used is open source software, licensed under the GNU GPLv2 license with the *runtime exception*. This means that the library may be used without requiring the developed software to be released as open source software, as long as TBB itself is not modified.

2.6.2 OpenMP

OpenMP [15] is an API for performing shared-memory multiprocessor programming, supporting C, C++ and Fortran. By using compiler directives,

[1] Deque - a double ended queue

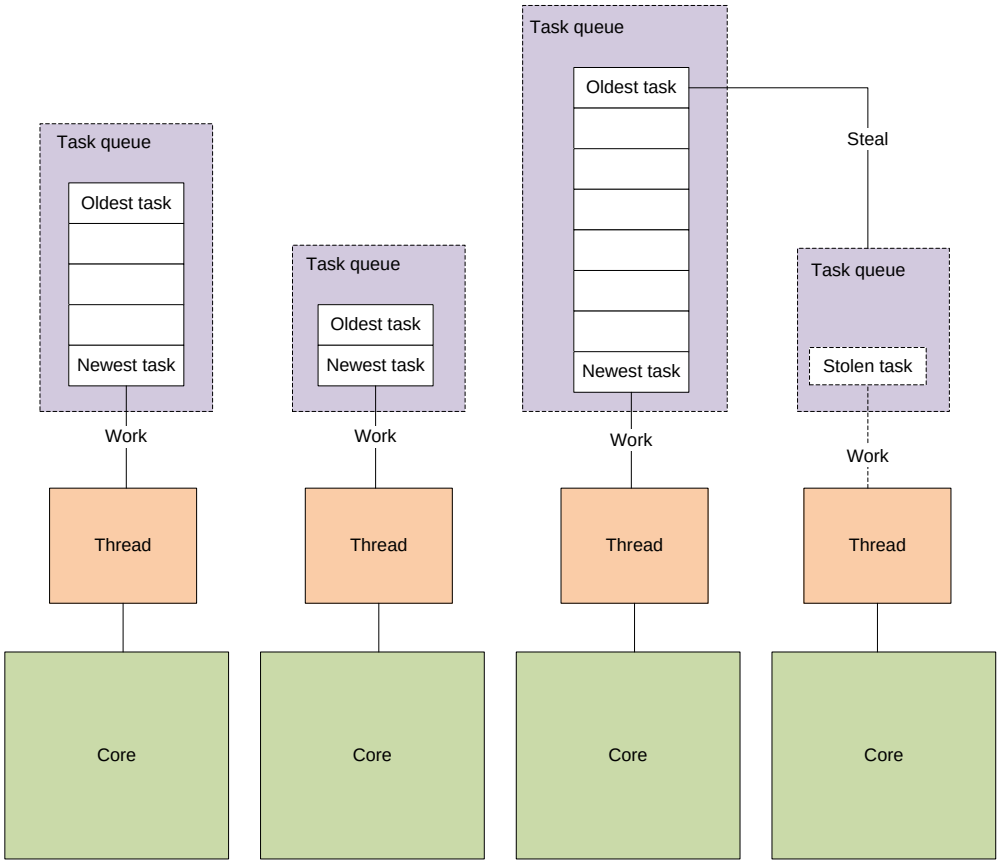


Figure 2.9: An illustration of the TBB runtime, courtesy of Hemmen [34].

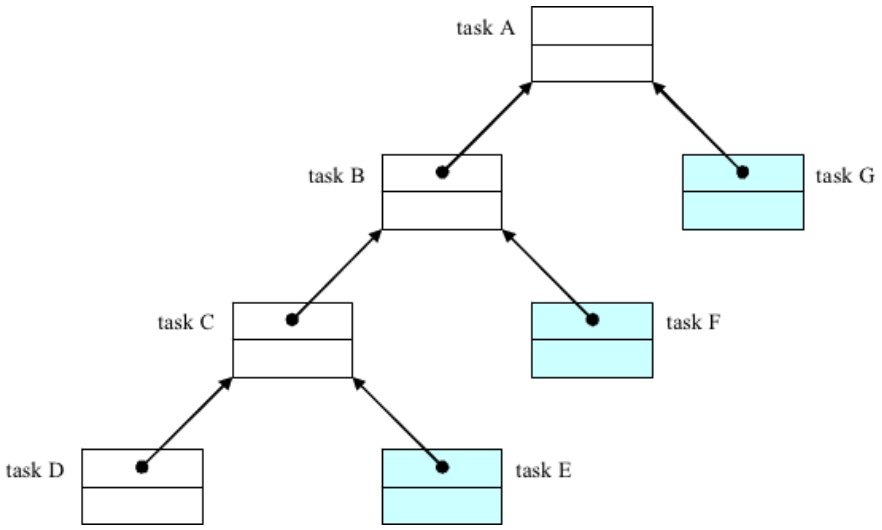


Figure 2.10: An example of a task graph in TBB, taken from the Fibonacci example in [40]

sections of a program can easily be parallelized by simply annotating the code. These directives will be ignored if the compiler does not understand them.

OpenMP supports many platforms, including Linux, Unix, Windows and OSX, which fits our multi-platform requirement.

We used OpenMP a few places where a simple loop-parallelization was all that was needed. After reading the TBB manual [42], we found that mixing these two libraries would not affect performance, as long as they were not nested.

2.6.3 Boost

Boost [8] is a wide collection of separate C++ libraries, providing functionality for different common tasks, including mathematical functions and string tokenizers. We use Boost's *random* library, which provides series of random number generators. The random generators do not keep a global state, and therefore several instances of the random generators may be used in parallel. We also make sure to seed the generators differently, so that we do not create identical random number sequences. In addition, we use Boost's tokenizer and string utilities, for reading and parsing the data sets.

Boost is licensed under the *Boost Software License*, which is similar to

GPL, but allows proprietary modifications to source code.

2.6.4 Qt

Qt [54] is a cross-platform application framework, originally developed by Norwegian Trolltech, now acquired by Nokia. It gives cross-platform abstractions for common tasks like network access, file management and thread management, as well as graphical user interface management. Qt is written in C++, and has bindings for many languages, including C++, Java and Python. Being cross-platform, the programs may be run on many different platforms, including Windows, Linux and Mac OS X. We use Qt for the graphical presentation, as well as managing the control flow between the different parts of the application.

For 2D graph visualization, we use Qwt – Qt Widgets for Technical Applications [10].

Qt and Qwt are both licensed under GNU LGPL.

2.6.5 GNU Scientific Library (GSL)

The GNU Scientific Library [9] is a numerical library for C / C++. It provides a large collection of numerical functions, including statistics, linear algebra and random number generation. We use GSL for its wavelet functions, as well as a selection of BLAS routines. The library is thread safe [9], and can therefore be used in parallel.

GSL is licensed under the GNU GPL.

2.6.6 Approximate Nearest Neighbor library (libANN)

libANN [53] is a C++ library providing algorithms and data structures for finding nearest neighbors in arbitrarily dimensional spaces. It provides both exact and approximate nearest neighbor searches, as well as different distance functions.

This library is not thread safe, as the algorithms retain global state. libANN is licensed under the GNU LGPL.

2.6.7 STANN

STANN (Simple, Thread-safe Approximate Nearest Neighbor) [11] is a C++ library designed to perform nearest neighbor searches on point clouds. It is parallelized with OpenMP, which means it will run in parallel if built using a compiler with support for OpenMP pragmas. If the compiler does not support OpenMP, the library will still compile, but parallel statements will be ignored, and the code will be executed sequentially.

We use STANN's functionality for building k-dimensional minimum spanning trees.

STANN is freely modifiable and redistributable for personal and academic use, as long as its copyright notice is kept.

2.6.8 Google Performance Tools

To see which parts of the application needed attention, we used Google Performance Tools [28] to profile our code. Google Performance Tools provides us with a call graph which may be built for the whole, or parts of the application. By inserting calls to the profiler within the source code, we may tell the profiler which parts to focus on, such as the algorithm itself, even when running the whole program. For an example of the output provided, see Figure B.7 on page B-9. We only profiled the command line interface, as we are mostly interested in the actual clustering algorithms.

2.7 Related work

In this section, we present existing applications in the domain of spike sorting. The section ends with a discussion of how our application differs from the existing applications, and what our contribution to the field is.

2.7.1 Wave_Clus

Wave_Clus [56] is a MATLAB program which implements SPC and wavelets as discussed in the article by Quiroga et al. [55]. Its main view is shown in Figure 2.11.

The program implements a number of interesting features. For feature extraction, PCA and wavelets are supported. These features can be visualized in a projection matrix, where each feature pair are combined for a

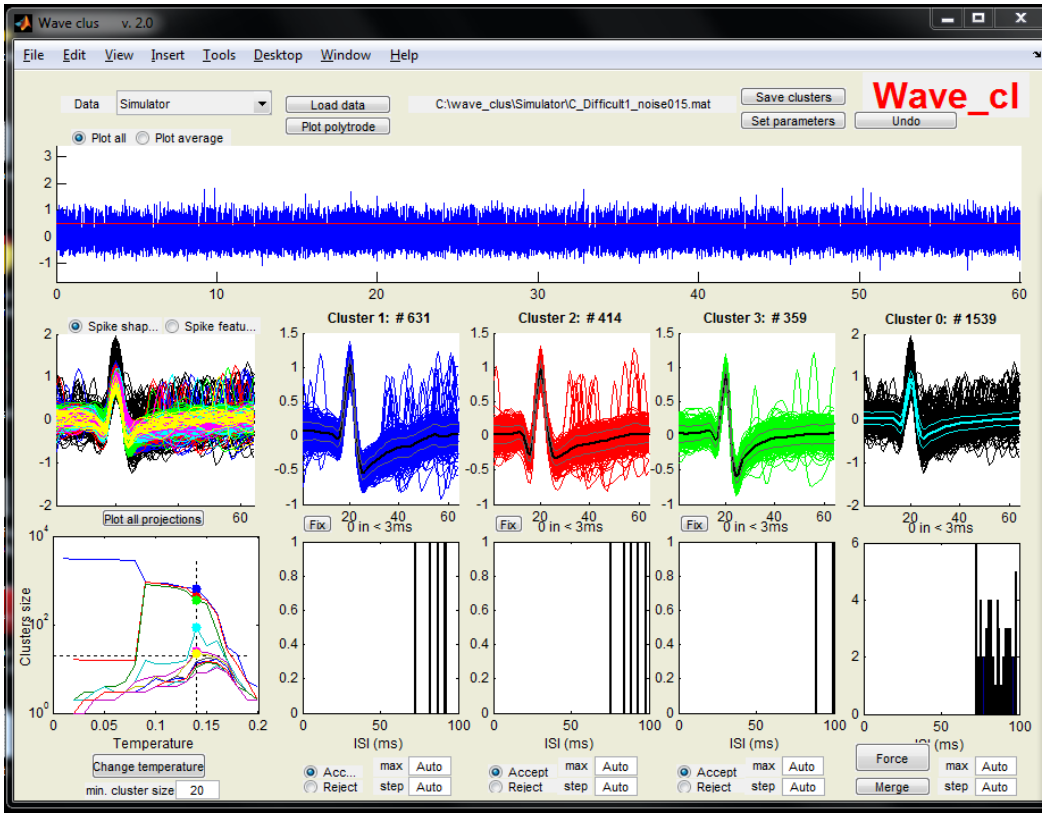


Figure 2.11: Wave_Clus, displaying a signal that has had the spikes identified and classified from a continuous dataset.

two dimensional plot. For a four dimensional dataset, this will create six 2D plots, and can be used to manually inspect the clustering. Clustering is performed by the super-paramagnetic clustering algorithm. To see at which temperature it might be sensible to perform the clustering, a temperature plot is provided, with graphs showing the different cluster sizes. To see a cluster quality indication, an inter-spike interval (ISI) plot is provided per cluster.

In addition to feature extraction and clustering, Wave_clus performs both spike detection and signal filtering. This means the program supports a continuous stream of data, and it can extract spikes automatically.

It is possible to merge two clusters, but not part of clusters, or perform operations on single points. This limits the possibilities of the manual aspect of cluster cutting.

The source code does not contain the SPC algorithm, as it is only attached as a binary executable. After conferring with Quiroga via email, we learned that they got the SPC implementation from Eytan Domany's group at the Weizmann Institute. Domany is co-author of Blatt et al. [18], in which SPC is presented. This is a serial SPC implementation.

The user interface is slow, and the last release is more than two years old (January 2009).

2.7.2 OSort

OSort [59] is a MATLAB implementation of an online spike sorting algorithm. The latest version is 2.1, released fall 2007. Rutishauser et al. [60] describe the details on the implementation. The fact that it is online, means that it could be used for real-time spike sorting, where spikes are sorted as they are detected, based only on previously detected spikes.

OSort can receive data directly from a network stream or read from a file. The program reads raw recorded data, and performs bandpass-filtering, spike detection, feature extraction and clustering in real-time. Spikes are detected using threshold crossings of a local energy measurement of the bandpass filtered signal.

The spike is then upsampled four times using interpolation to smoothen the waveform. This is done by first applying the Fourier transform, and then applying the inverse transform, back to a longer sample. The first spike is assigned to a new cluster. Subsequent spikes are compared to existing clusters, and either added to an existing one, or becomes the first spike in

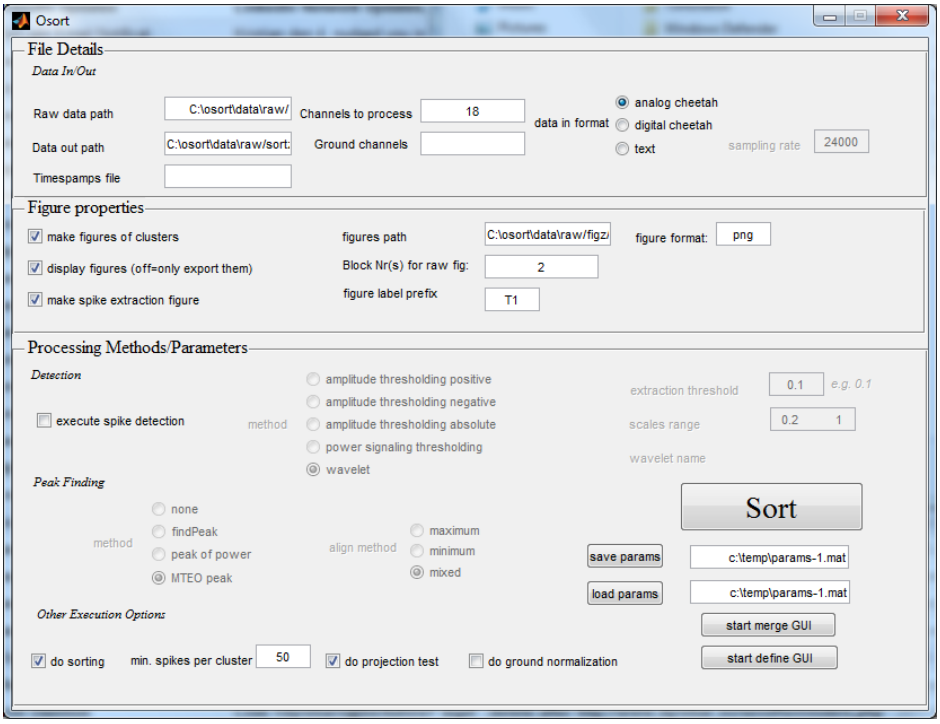


Figure 2.12: The OSort main window.

a new cluster. Similarity is calculated using a threshold, which is calculated locally (sliding window) from the noise properties of the signal. For each added spike, the mean waveform for the cluster is recalculated based on the N last assigned spikes. This may make mean waveforms of two clusters more similar over time. They will then be merged, if the distance between the waveforms is lower than another locally calculated threshold. When the execution is complete, clusters containing few spikes, or too low average firing frequency, are discarded.

The article compares OSort with KlustaKwik and Wave_Clus, and achieves similar results using their test data. With increased noise, OSort gets better results than the other two algorithms.

OSort only supports single-channel data, but the authors claim that it should be straight-forward to extend the algorithm to use multi-channel data [60]. The authors also state that the algorithm would perform much better if written in a compilable language such as C++, and that "optimizing this implementation will allow the realtime processing of many hundreds of channels" [60].

The application has evolved since the article was published, so the newer

version could have changes to the algorithms used. However, we did not inspect the source code to see if this is the case.

Figure 2.12 is a screenshot from the user interface. We tested OSort on the dataset which is available from the program website, which is a continuous single-channel raw signal lasting for approximately 8 minutes, with a sample frequency of 25`kHz`. After running the program, it fails in the last step, where results should be displayed, because of a MATLAB package we did not have access to. We also tried different versions of MATLAB, but were unable to use the whole program. Inspection while executing the application shows that the program only utilizes one of the available cores, i.e. it is not programmed for parallel execution. The process of detecting spikes and performing clustering took about an hour on our desktop workstation, with details as described in Section 2.5. For a recording of 8 minutes, this is close to their reported performance in their article, where they say that OSort can process a single channel in ten times the acquisition time. They claim that the implementation is not optimized for speed, and that with optimization, the algorithm should be able to sort "many hundreds of channels in realtime" [59]. We think this sounds extremely optimistic, given that this means a speedup of more than 1000 times compared to the performance of the current version.

OSort is not released on a specific license, but is stated to be copyright of the authors.

2.7.3 KlustaKwik

KlustaKwik [7] is a command line program which uses a Classification Expectation-Maximation (CEM) clustering algorithm to classify spikes. CEM is a "winner-take-all" version of the Expectation-Maximation algorithm, where no fractional assignments of clusters are allowed [49, 43]. Although this will handle non-Gaussian distributions better than k-means clustering, the benefits appear to be limited. First, its rate of convergence can be very slow. Second, the numer of conditional probabilities associated with each observation is equal to the number of components in the mixture, so it might not be practical for models with many components. Finally, the model breaks down when the covariance matrix corresponding to one or more components becomes ill-conditioned (singular or nearly singular) (Fraley and Raftery [24]). Our datasets can be very big, and the covariance matrix can also be singular.

Because of the limited benefits of adapting CEM (SPC already handles non-Gaussian distributions), and the limited user interface, this program is not explored further. A quick inspection of the source code reveals that

KlustaKwik is not programmed for parallel execution.

2.7.4 Klusters

Klusters (Hazan [32], Hazan et al. [33]) is an open-source cluster cutting program written for the KDE3 platform on GNU/Linux. The program is not significantly updated since 2004, and depends on old libraries to build – notably KDE3 and Qt3. Figure 2.13 shows the main window after a dataset has been sorted.

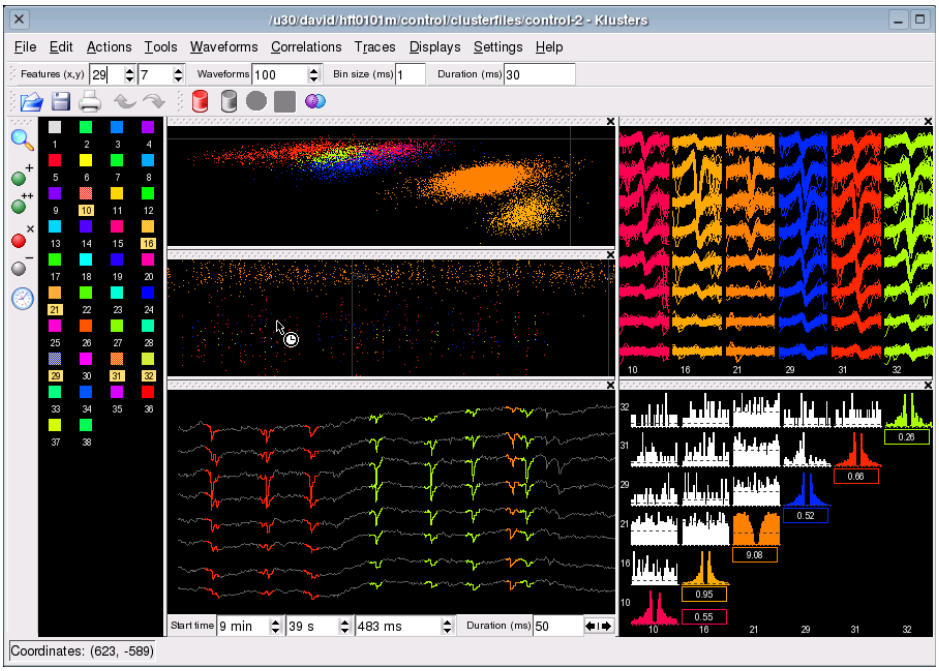


Figure 2.13: Klusters and its main window.

It has several views. Clusters can be viewed in a 2D projection using PCA to reduce the dataset. The spike waveforms for the selected cluster can be displayed as overlapping spikes (top-right in Figure 2.13). Spikes from two clusters can also be displayed either side by side or overlapped with different colors, to help see if they should be merged. It can also display the mean and standard deviation of the spikes. The correlation view displays auto- and cross-correlograms between the selected clusters This view is useful to distinguish medium to high firing neurons from neurons with a low firing rate in a group, as well as gauging how noisy a cluster is (what fraction of spikes belong to another cluster). Finally, a trace view, which displays the

identified spikes on the continuous signal. This can be a helpful visualization aid, but was unavailable to us because we lacked the necessary data. The program supports merging of clusters, as well as changing or removing cluster membership status on a point-by-point basis, but this did not work in our build – it simply ignored our commands. For clustering it uses KlustaKwik.

Its strengths appear to be its GUI and support for doing manual cluster cutting after the automatic suggestion, however as mentioned this part was only partially working. Although the program has not been updated lately, the authors state that it is used extensively in their research laboratory.

2.7.5 OpenElectrophy

OpenElectrophy [27] is an open-source program written to help with the whole toolchain of cluster cutting. It is written in Python, and uses a selection of scientific libraries for effective calculations. A part of the philosophy for the project is to provide a user friendly application in the open-source scene of neuroscience applications, which according to the authors is missing today. They have also spent time developing a neat-looking GUI, implemented using Qt4. An example of this may be seen in Figure 2.14.

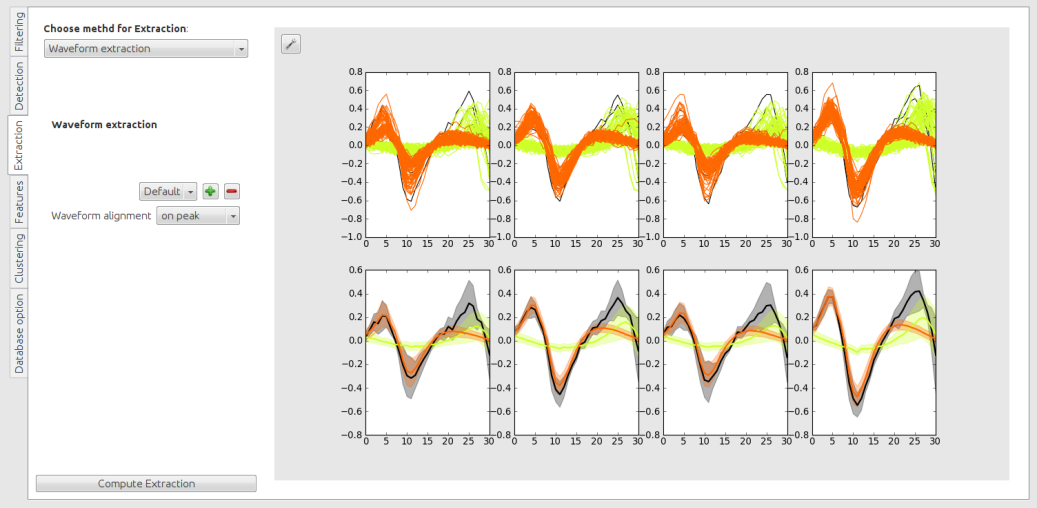


Figure 2.14: OpenElectrophy, displaying extracted waveforms colored by manually allocated clusters, on a test dataset supplied with the application.

OpenElectrophy handles importing different file formats and storing large datasets in different types of databases. There are a range of options for per-

forming all steps in cluster cutting: signal filtering, spike detection, feature extraction and clustering.

It supports k-means clustering, and has a version of SPC implemented. A closer inspection of the code shows that the algorithms could be optimized to a larger extent – one example is that SPC calculates all-to-all distances between all points, and then sorts the neighbors from closest to furthest, to find the K nearest neighbors. This results in a higher complexity than necessary (N^2), as one should be able to prune large parts of the search space. It also does not plot ranges of temperatures to help the user find the right temperature to use for clustering, and the source code seems to be experimental with many out-commented lines.

The program supports nice visualizations of the data in all steps of the toolchain, as well as manual editing of cluster results, such as manually classifying a spike as noise, or belonging to another cluster. It is not always intuitive to use, but it seems evident that persons who are trained in using the program can work very efficiently with help from the provided tools.

A close inspection of the system while performing clustering reveals that the program only uses one processor core. This suggests that the program is not optimized for parallel execution, perhaps because of limits with the python programming model. This would be a nice extension of the program.

Although OpenElectrophy supports importing many different data formats, including datasets of spikes (non-contiguous data), we did not find documentation for the applicable file formats. We therefore did not spend time converting our datasets to be usable for the application, though this would have been helpful to compare results, and for quick verification of functionality.

Garcia and Fourcaud-Trocme [27] are the main authors of the program, and two more contributors are mentioned on the project web page. The project is active, and several changes seem to be made in the code base every month.

2.7.6 Tint

Tint [13] is a cluster cutting and analysis software by Axona Ltd. It is specialized for the analysis of spatially-specific neural activity, and the tool used at the KI to perform manual spike sorting. It can display various spike parameters and enable cluster cutting and determination of locational variables such as position, heading and the speed of the rat.

To facilitate cluster cutting, a cut window is available with two modes of display. The first is a waveform view similar to Figure 2.2 on page 17. The second mode is a two-dimensional scatter plot which shows points in a two-dimensional section of a multi-dimensional space. The two-dimensional section is derived by comparing one electrode against the others, according to a selected metric. This creates six plots for a tetrode.

There are several options available for how to derive this two-dimensional data. The default option computes each point by taking the peak-through amplitude of one tetrode and matches it against another. Other options include voltage at time t, peak height, trough depth, time of trough and time of peak.

Cluster cutting can be done completely manually, where clusters are identified by eye and created by lassoing them together in the scatter plot. They can also be adjusted manually after the automatic clustering is finished.

To perform automatic cluster cutting, centroids have to be identified. These can either be specified manually, or Tint can guess them using the k-means algorithm. The clusters are then defined by the centroid and a rectangle or ellipse.

Tint appears to provide a decent interface for managing clusters manually, and the scatter plot can provide sensible visualizations. However, it does not support any advanced spike representations like wavelets, nor does it support any advanced clustering algorithms such as SPC.

2.7.7 Summary

Both Wave_Clus and OSort are MATLAB programs, with proprietary parts which cannot be inspected or modified. This applies to the SPC methods in Wave_Clus, which could not be used directly to verify our own implementation. OSort first failed because of a mismatch between our 64 bit MATLAB, and the provided 32-bit DLL files. We managed fixing this, but OSort still failed when it should display the results, meaning we had little use of the program.

Although MATLAB is excellent for doing mathematical operations, and has built-in support for many different algorithms, this requires a MATLAB license to use the program. This excludes users who do not use MATLAB, and increases the threshold for people to experiment with the program. Also, the two MATLAB applications seem to focus little on usability. Both require that you manually set up a working folder for the applications. Specifying input file paths has do be done manually – there is no “choose file” dia-

log. When the algorithms execute, you get no feedback on how much time remains. Because of the way the figures are displayed when the algorithm finishes, it keeps spawning windows which pop-up and stops you from doing anything usual while waiting.

Tint is a proprietary program, which is supplied together with the recording equipment, and is therefore not open for modification.

Klusters, KlustaKwik and OpenElectrophy are all open-source programs, and users may submit changes or additions to the projects through their code repositories. According to Tsao [69], KlustaKwik is the only program they have tried, but they do not use it, citing poor performance.

The program which seems to be updated most frequently is OpenElectrophy. This, and the fact that the code, including the used scientific libraries, is open-source, lowers the threshold for contributing to the project. It also worked out-of-the-box on our computers, unlike all the others we tested, and provides extra features for handling different databases. With an improved documentation of the supported data formats, this program would be really easy to use, as well as being easily extendable.

It is noteworthy that none of these programs appear to be parallel. We examined the available source code, as well as analyzed the run-time performance, and only one core was exploited. This means the programs are ill-prepared for the automatic benefits new hardware will bring.

We will not make a program to compete with these existing solutions, but will test our algorithms in a manner which is as efficient as possible. If our program had a broader scope, we would have tried to include our parts as modules in one of the existing solutions, instead of starting from scratch. To make sure our program may be useful for others in the future, we aim to use only open-source libraries, and to build the program as modular as possible. We also focus on making the computation-heavy parts of the program as efficient as possible, utilizing available hardware in parallel. Ultimately, we will focus on the application being user friendly: it should give error messages instead of crashing, and should provide the user with feedback such as time remaining when running long-lasting algorithms.

CHAPTER 3

Methodology

3.1 Datasets

In this section, we describe the datasets which will be used to test the algorithms in use.

Lower dimensional datasets, which are not directly connected to the problem, are summarized in Table 3.1. These are used to test functionality.

Table 3.1: Summary of lower dimensional datasets

Dataset	Classes	Instances	Features	Noise
Three islands	3	10	1	No
Iris	3	150	4	No
NTNU toy problem	6	26553	2	Yes

3.1.1 Iris

The Iris dataset [23] is a dataset containing 3 classes of 50 instances each. The features measured are sepal and petal lengths and widths for specimens of three different species of the Iris flower. One class is linearly separable from the two others, whereas the two latter are not.

3.1.2 NTNU toy problem

In order to easily see a visual result of our clustering efforts, we created a two dimensional dataset with a cluster distribution that is not linearly separable. It features the NTNU logo, as seen in Figure 3.1 on the next page.

The reason it is important to benchmark how we can handle datasets which are not linearly separable is because a linearly separable dataset which has gone though dimensional reduction could lose this property.

To make it more interesting, we applied a uniform noise distribution, inverting the color of each pixel with a probability of 10%. When clustered perfectly, the logo should consist of two clusters, while the text four, adding up to six clusters in total.

This dataset also expresses the ambiguity of clustering. It would be perfectly reasonable to claim that the logo should be one cluster, and the text another – or that they should comprise one big cluster, where the noise is excluded.

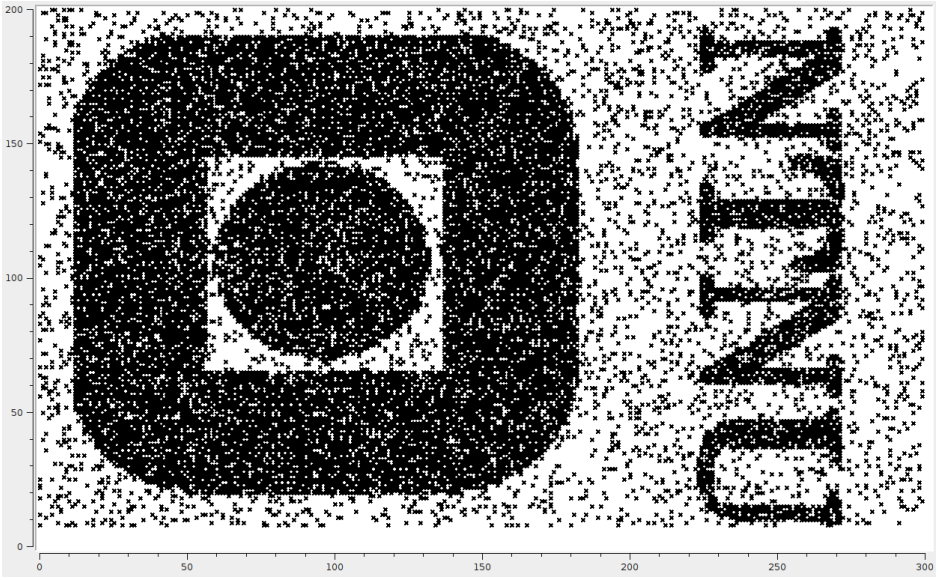


Figure 3.1: The unclustered NTNU toy problem. Notice that randomized noise is added to make cluster classification more difficult.

3.1.3 Three circles

We were given the dataset used by Blatt et al. [18] to compare our performance to theirs. It is similar to the NTNU toy problem in that it has noise added in addition to a non-Gaussian data distribution, as shown in Figure 3.2 on the facing page.

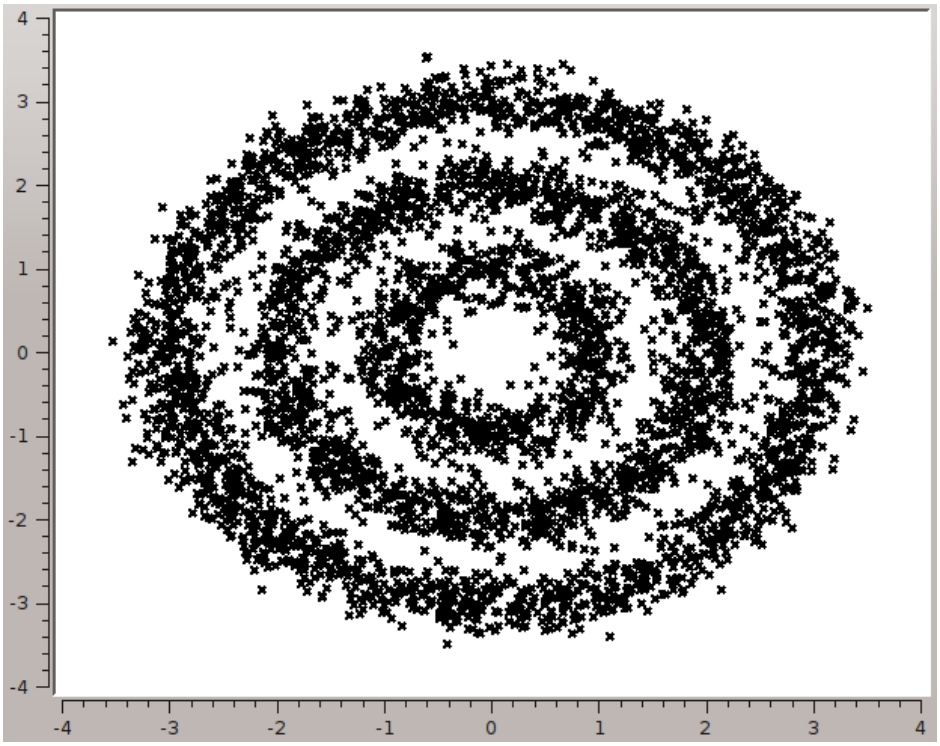


Figure 3.2: The unclustered three circles problem. Notice that the circles are not completely separated, to make classification more challenging.

3.1.4 Three islands

We created a dataset with three highly separated clusters, used to verify some of our algorithms. It consists of 10 points distributed in one dimension, with cluster points in close proximity and large gap between the clusters.

3.1.5 Datasets from the Kavli Institute

The datasets received from the Kavli Institute are recorded from live rats, with tetrodes implanted in the extracellular space of the brain. A recording lasts for ten minutes, where the rat is exploring a given environment. The recording equipment filters the signal, and detects and records spikes as they happen. Positional information is saved for each spike, and is therefore also available in the datasets. We ignore the positional information, as this is used after clustering, to see if firing neurons correspond to specific positions in the environment.

All the datasets provided also come with a *cut file*, which is a file that describes the results of manual classification as done by the KI scientists. Our automatic clustering efforts are compared to these. These clusters are also analyzed by our cluster quality algorithms. Just as with the automatic efforts, the manual clustering can contain classification errors.

As described in Section 2.4, each spike lasts for $1ms$ and is recorded in 4 channels, each with 50 samples of 1 byte each. The number of spikes in the datasets vary, as different recording sites have differing counts of active neurons. Also, the rat's activity level affects spike frequency.

Table 3.2 on the next page lists the different datasets we got access to, with information we found on classification and noise. The ones named Albert come from a single recording of 8 different tetrodes, and 180501 is from one tetrode in another recording.

3.2 Cluster quality measurements

Cluster quality measurements, or cluster evaluation, is important to quantify the reliability of clustering results. Some clustering algorithms, such as k-means, are dependent on an initial state. Cluster quality algorithms may be used to identify poor clustering results, so that the clustering algorithm may be run again to find better results. It is also useful to be able to say something about the quality of the clustering *across* different algorithms when

Table 3.2: Summary of the recordings

Dataset	Neurons	Instances	Noise	Difficulty[a]
Albert 1	3	931	88%	Easy
Albert 2	3	771	46%	Easy
Albert 3	3	685	31%	Easy
Albert 4	6	4 178	43%	Medium
Albert 5	1	7 385	96%	Easy
Albert 6	3	4 577	61%	Easy
Albert 7	2	5 992	54%	Medium
Albert 8	1	1 980	88%	Easy
180501	5	34 403	52%	

[a] According to Tsao [69]

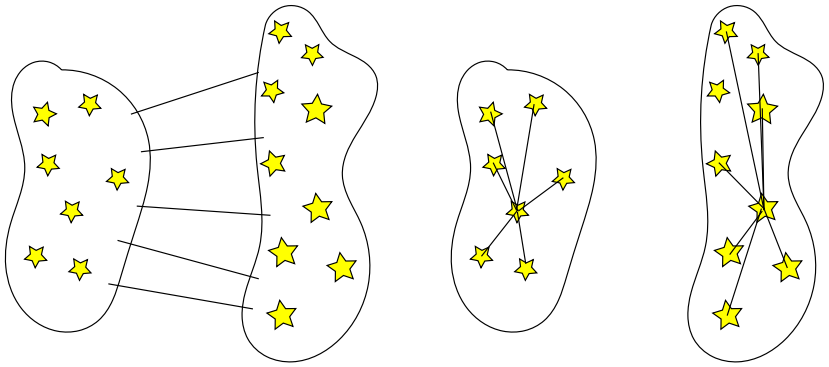
comparing their performance. Finally, it is interesting to run the algorithms on the manually cut set, to see which score they get compared to our efforts.

We may separate the algorithms into two categories: supervised and unsupervised cluster evaluation. The unsupervised algorithms are used to make general assumptions about cluster quality, based on the results alone. Supervised algorithms make use of actual knowledge, such as manual classification performed by experts.

Some measures, such as the Rand index, assume a degree-of-cluster membership. We can however only evaluate cluster quality based on absolute membership, so such measures are ignored.

Two important features for unsupervised cluster evaluation are *cohesion* and *separation*. High cohesion within a cluster means that the points in the cluster are very similar to each other. High separation between clusters, means that points in one cluster are not similar to points in other clusters. This is illustrated in Figure 3.3 on the following page.

As the focus for our thesis is to perform unsupervised clustering, we will implement unsupervised algorithms for cluster evaluation. However, as we have access to manually classified datasets, we will also use *F-measure*, which compares results to a known correct classification, as described in Section 3.2.5. The rest of this section describes the algorithms we use.



(a) Separation (b) Cohesion

Figure 3.3: Separation between clusters, and cohesion within clusters. Inspired by Tan et al. [66].

3.2.1 Cohesion and separation

Cohesion and separation are related to other algorithms, but may also be used directly. Tan et al. [66] demonstrate different approaches for calculating cluster validity based on cohesion and separation. They separate between graph based and prototype based algorithms. As K-means is prototype based, and SPC is graph based, we have both data structures available. However, as we are working in the Euclidean space, we may easily calculate prototypes for each cluster. We therefore focus on the prototype based algorithms.

The *distance* function used in the following equations is Euclidean distance.

Cohesion, $\mathcal{I}_2$ is calculated as the sum of distances of each data point to its cluster center c_i, as defined in Equation (3.1). Cohesion thus calculates the within-cluster sum of squares. Higher value means lower cohesion. To favor tightly packed clusters with many members, the result could be divided by the number of points, although this is not done in our references.

$$\mathcal{I}_2 = \sum_{i=1}^{K} \sum_{x \in C_i} distance(x, c_i) \tag{3.1}$$

Separation, $\mathcal{E}_1$ is calculated as the sum of distances from each cluster centroid c_i to the centroid for all data c in total. Separation thus calculates the between-cluster sum of squares. Each cluster is weighed by its number

of points m_i, as defined in Equation (3.2).

$$\mathcal{E}_1 = \sum_{i=1}^{K} m_i \times distance(c_i, c) \tag{3.2}$$

3.2.2 Silhouette coefficient

The silhouette coefficient (Rousseeuw [58]) is a way to visualize the cohesion and separation. The author describes cohesion as tightness, which is the same thing. The silhouette is based on a combination of separation and cohesion, and can help the user identify the quality of the different clusters when they are listed next to each other. An example plot is shown in Figure 3.4, which is the result of a manual classification. Note that cluster 0 is classified by Tsao [69] as a so called "noise cluster", and is not used for further analysis.

In our implementation, we base the silhouette on the dissimilarity between the points by using the Euclidean distance.

To define the cohesion, each point has its average distance $a(i)$ to all the other points in its cluster A computed, defined as:

$$a(i) = \frac{\sum_{x \in C_i} euclideanDistance(i, x)}{numObjects(A)} \tag{3.3}$$

Separation is defined by comparing each point in a cluster to the average distance of all the other clusters C – the closest cluster is defined to be its neighboring cluster with distance $b(i)$. For any cluster C, the distance is defined as:

$$d(i, C) = \frac{\sum_{x \in C} euclideanDistance(i, x)}{numObjects(C)} \tag{3.4}$$

We then define $b(i)$ as the minimum distance:

$$b(i) = \min_{A \neq C} d(i, C) \tag{3.5}$$

The closest cluster can be thought of as the next best classification choice for point i. $s(i)$ is used to draw the silhouette and is defined as:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{3.6}$$

$s(i)$ ranges from -1 to 1, where a high value implies that the intra-cluster dissimilarity is much smaller than the dissimilarity to its closest cluster. From this we can say that the point has been assigned to the correct cluster, as its closest neighbor cluster is still not nearly as close as the cluster it was assigned to. Conversely, a low value means that the point is actually much closer to its nearest neighboring cluster than the one it was assigned to, and thus has probably been misclassified. If $s(i)$ lies around zero, it suggests that the point lies between two clusters.

In addition to providing a visual feedback, it is possible to compute averages to automatically be able to say something about a clustering result. The average $s(i)$ in each cluster can indicate how tightly packed the cluster is – clusters with a high positive average s is a good cluster according to this method. It is also possible to take the average of these numbers again – resulting in a single real number for the entire clustering. This value can be used to indicate the overall quality of the clustering result.

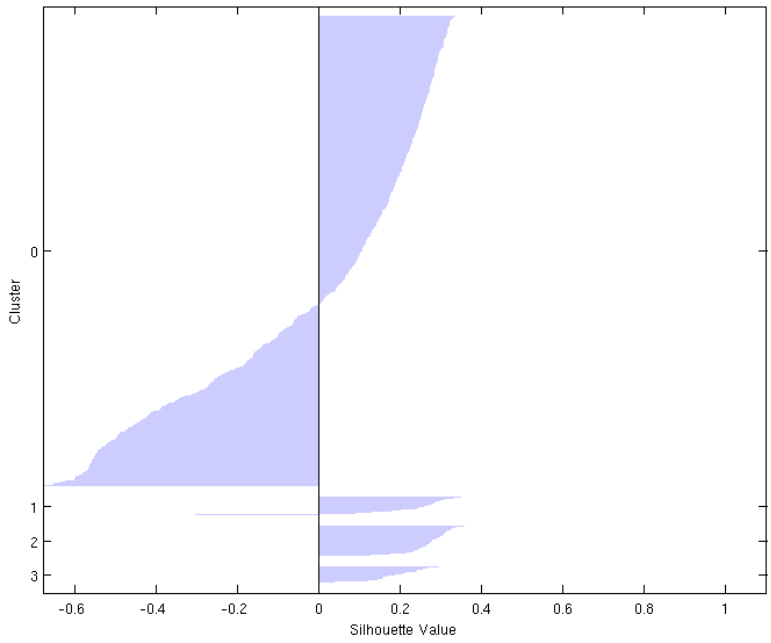


Figure 3.4: A silhouette plot of four clusters as performed by Tsao [69]. The Y axis denote points, the X axis silhouette value. Cluster 0 is by far the largest cluster. In this recording, it is a so called *noise cluster*, which contains data classified as noise.

3.2.3 L_{ratio}

L_{ratio} is based on the assumption that the distribution of cluster spikes is multivariate normal [61]. It uses *Mahalanobis distance*, D^2, a distance function here used between a spike and a cluster center. If the spike distribution is multivariate normal, then D^2 will be distributed as χ^2 with as many degrees of freedom as the number of dimensions in the samples.

The Mahalanobis distance is defined as:

$$D^2_{i,C} = (x_i - \mu_C)^T \Sigma_C^{-1} (x_i - \mu_C) \tag{3.7}$$

i is a spike, C is a cluster. x_i denotes the feature vector for a spike, and μ_C is the mean feature vector for the cluster. Σ_C is the covariance matrix for the spikes in the cluster.

$L_{\texttt{ratio}}$ for the cluster C is defined as:

$$L_{\texttt{ratio}}(C) = \frac{\sum_{i \notin C} 1 - CDF_{\chi^2_{df}}(D^2_{i,C})}{n_C} \tag{3.8}$$

Here, $CDF_{\chi^2_{df}}$ is the cumulative distribution function of χ^2 with *df* degrees of freedom. n_C is the number of spikes in cluster C.

We calculate distances for each spike outside the cluster, which from the cluster point of view may be considered "noise spikes". Noise spikes close to the cluster center will contribute strongly to the sum, while spikes far away will contribute only little. A high $L_{\texttt{ratio}}$ value thus indicates that the cluster is not well separated from the rest of the spikes. This means that there is a higher probability that the cluster includes noise spikes, and excludes spikes which should belong to it.

Ververidis and Kotropoulos [71] state that the Mahalanobis distance is very dependent on having enough samples in each cluster, compared to the number of dimensions. They use the fraction $\frac{N_{D_c}}{D}$ as a measure for validity of the Mahalanobis distance, where N_{D_c} is the number of samples (spikes) in a cluster, and D is the number of dimensions. They cite one experiment which found that the fraction should be at least 3, and another which demonstrates that it should be at least 10. When the fraction approaches 1, error estimate is said to approach that of random selection. This means that we cannot use the measurements based on the Mahalanobis distance without reduction of the datasets. For 200 dimensions, we would need 2000 spikes in each cluster. As can be seen in Section 3.1, this is not the case for any of the datasets. Therefore, we use different feature extractions to be able to calculate these cluster quality measurements.

3.2.4 Isolation distance

Isolation distance is also based on calculating the Mahalanobis distance. A cluster containing n_C spikes has an isolation distance defined as the Mahalanobis distance to the n_C'th closest noise spike. Hence, isolation distance is a measure of separation. It represents the radius of the smallest ellipsoid from the cluster center, containing all of the cluster spikes and an equal number of noise spikes [61]. Figure 3.5 shows a plot of how the isolation distance is calculated.

Schmitzer-Torbert et al. [61] successfully applied L_{ratio} and *isolation distance* as cluster quality measures for data sets from rodent hippocampus, and suggest that other researchers report their values of these quantities. We will follow their suggestions for how to apply these quantities to our data sets.

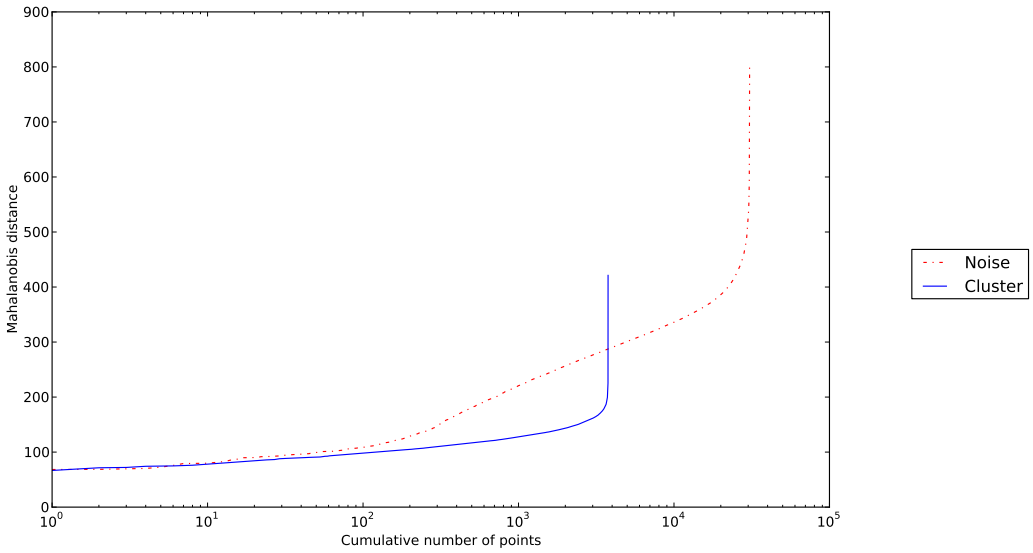


Figure 3.5: Isolation distance plot for the 180501 dataset. The second cluster of the cut file is shown. Isolation distance is 287.

3.2.5 F-measure

F-measure is calculated for a cluster with respect to a correctly classified cluster, which in our data sets means manually cut clusters. It is a combination of two quantities: *precision* and *recall*. Precision is defined as the fraction of samples in a cluster C_i which belongs to the correctly classified

cluster Q_j. Recall is the fraction of samples in the correctly classified cluster Q_j, which was clustered in cluster C_i. After calculating these, F-measure is defined as:

$$F(i,j) = \frac{2 \times precision(i,j) \times recall(i,j)}{precision(i,j) + recall(i,j)} \tag{3.9}$$

CHAPTER 4

Implementation

This chapter will describe the implementation of our application, *Paraspikes*. The application is developed separately from the recording system, and all spike recordings are read from files delivered by the KI. For an explanation of the recording system, see Section 2.4.

Paraspikes includes a graphical user interface, which gives access to loading data files, selecting clustering algorithms, altering clustering parameters, and performing clustering. There are also several options for visualizing datasets and clustering results. In the following sections, we will describe the system in the order it is accessed by a user.

The implementation has a main focus on ease of use, speed and utilizing all the available processor cores in a system. It is programmed in C++, and is coded to be highly portable. This has also affected which external libraries we use, as they also need to be portable. All external libraries which have been used are described in Section 2.6. Although only tested on GNU/Linux, we only use standardized APIs, so compiling the application for other architectures such as Windows or Mac OSX should be straight-forward.

To dynamically utilize all the available cores while keeping the overhead low, the task based programming paradigm is used, here realized with Intel TBB, as described in Section 2.6.1. For more trivial tasks, OpenMP described in Section 2.6.2 is used.

This rest of this chapter will explain our implementation in more detail. Section 4.1 gives a quick overview over the GUI and CLI. Section 4.2 details the two supported dataformats. The different ways the program can represent spikes are explained in Section 4.3. Our k-means and SPC implementation follows in Section 4.4. Section 4.5 describe how we implemented the algorithms that can give an indication of how well the clustering is. We talk about our optimization efforts in Section 4.6. Finally, we make a short

comment on our project experience in Section 4.7.

4.1 User interface

The initial plan for Paraspikes was to provide only a graphical user interface (GUI). After some experimenting, we found it necessary to include a command line interface (CLI) as well, to enable batch processing on cluster systems, and to support more efficient testing. In this section we will describe the two interfaces.

4.1.1 Graphical user interface

The graphical user interface is realized using Qt4, which provides abstractions for graphical presentation. We chose to split the interface in tabs ordered as the process of spike sorting.

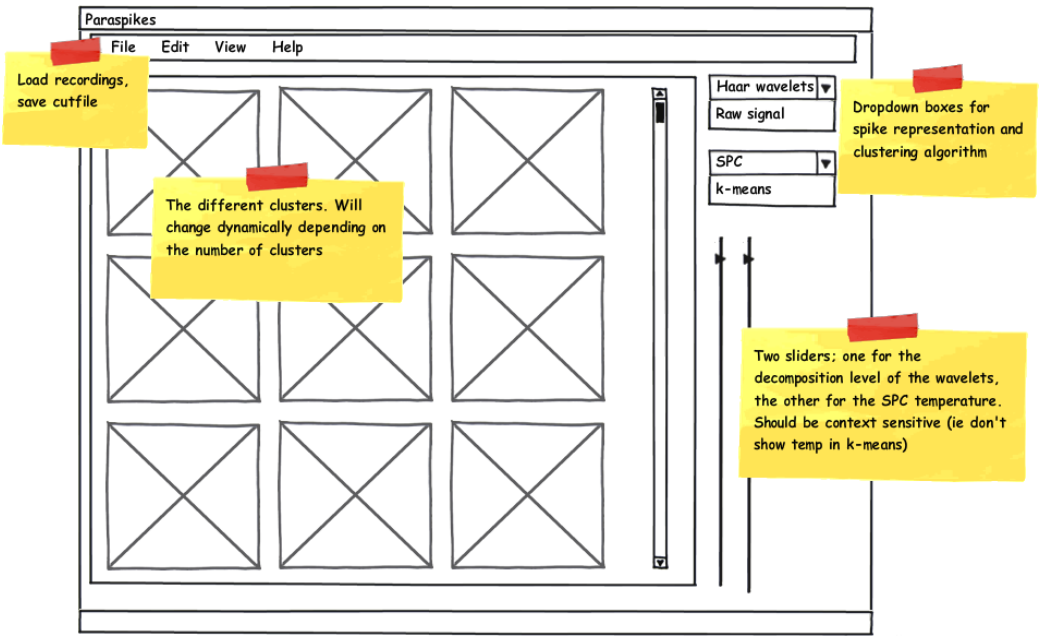


Figure 4.1: The initial wireframe of the GUI.

Figure 4.1 was the initial design. After adding features, a tab-based view was adopted as shown in Figure C.1 in Appendix C, the final result. The first tab covers file input, feature extraction, choice of clustering algorithm

and parameters. One of the functions in the first tab, is to compare the original and the reduced datasets. This allows for a visual verification of the reduction. In Figure C.2, we see the original spike signal, compared with the reverse of a wavelet transform configuration.

The second tab is specific to SPC, and allows the user to select which temperature range to plot, and displays susceptibility and cluster sizes after clustering. Here, the user chooses which temperature to use for clustering. Figure C.3 displays this tab.

The third tab, as shown in Figure C.4, displays the waveforms of the different clusters, including the average for the cluster, as well as coloring the standard deviation. The user may select whether to plot the spikes, and if the original spike, or the extracted features should be plotted. This also includes plotting the reverse of the reduced dataset. In the same tab, there are buttons for calculating cluster quality, loading cut files for matching, and exporting the clustering results as a cut file which is readable for Axona Tint.

The final tab displays a 2D scatter plot of the clustering. This may be done directly by plotting the two first features of each spike, or the user could specify to use the PCA reduction down to two dimensions. There is also an option to export a three dimensional PCA representation, and plot the dataset using a python script. This tab is demonstrated in Figure C.5, and Figure 5.14 on page 114 was generated using the 3D plot.

For interactions between the different GUI components, and for coupling between the GUI layer and the logic layer, we use the pattern of signals and slots, provided by Qt. In short, this means that you can connect components, so that when an action is performed on one component, a method in another component receives information about this, without the need of a separate method to handle the routing of interaction. This is, for example, used when comparing original and reduced dataset, as demonstrated in Figure C.2. When the slider in one window is moved, the slider in the other window follows, and the program makes sure that the same spike is shown in the two representations, at any given time.

A class diagram of the GUI is shown in Figure D.1.

4.1.2 Command line interface

The command line interface enables the user to run the different clustering algorithms without the need of the GUI libraries. This is mainly used to test run time on the Kongull compute cluster, as well as quickly evaluating the various clusterings using our cluster quality algorithms. By scheduling execu-

tion of several instances with different configurations, the different instances may be run in parallel on different nodes, enabling multi-node execution without explicitly coding for parallel execution. Listing C.1 shows the usage help screen for the CLI.

4.2 File parsing

Paraspikes supports two different file formats: an ASCII format and the Axona Dacq file format. The ASCII format starts with two integer values N and D. N describes the number of spikes, and D describes the number of dimensions, i.e. how many samples each spike has. Then follow N lines with D comma separated floating point numbers. This is the format which is used for all data not originating from the Axona Dacq system.

The Axona Dacq format is an ASCII/binary hybrid format which contains some meta data about a recording, as well as a collection of spikes. It stores the samples as a series of `char`s (bytes). These are converted to floating point numbers for easier calculations.

4.3 Feature extraction

Because every spike is processed independently, the data decomposition pattern [41] fits very well. In data decomposition, the same algorithm is applied to each data item. In this case, that means that a serial feature extraction algorithm is executed in parallel on several spikes.

To support easy extension of the application, we have used polymorphism to define clustering algorithms and feature extraction (dimensional reduction) algorithms. Figure 4.2 on the facing page shows how the `ReductionService` keeps a list of available reduction schemes, and delivers a list of its names. Based on indexes in this list, one can retrieve the different algorithms without hard coding them in where they are used. This makes it easy to provide new reduction schemes, without having to change neither the GUI nor the command line implementations at all.

When reporting complexity in this section, N refers to the number of spikes, and D refers to number of dimensions in the unreduced dataset.

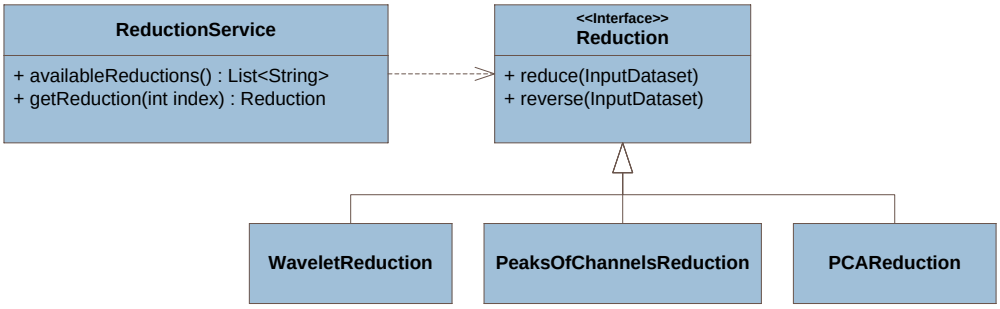


Figure 4.2: Class diagram for three of the `Reductions` and the `ReductionService`. `ReductionService` provides access to the available reductions, without need for changing the code where they are used.

4.3.1 Unreduced

When using the unreduced dataset, the original input data is simply copied into a new data structure, and is ready for clustering. With the KI datasets, this means retaining all 200 dimensions. The operation is performed by copying the dataset, and therefore has a complexity of $\mathcal{O}(ND)$.

4.3.2 Peaks of channels

Our peaks-of-channels implementation scans through the samples, and finds the highest value in each channel. This gives a reduction from 200 dimensions to 4 when used with tetrode data. As we scan through all samples, this has a complexity of $\mathcal{O}(ND)$.

4.3.3 Wavelet transform

The wavelet transform is implemented using GSL. The implementation is serial. However, this does not matter as the transform will be applied to thousands of independent spikes. This means that while the transform itself is serial, the process of converting all the spikes can be parallelized using TBB. Profiling revealed that this step only took a tiny fraction of the time, and the serial implementation was thus kept.

We have experimented with different wavelet representations, as well as implemented the Lilliefors modification of the Kolmogorov-Smirnov test

(LKS) for normality, as described in Section 2.3.3. Figure 2.7 on page 27 consists of screenshots from our application, and demonstrates how one may represent the signal using different levels of the Haar and Daubechies wavelets. For verification, we have also implemented the reverse of these, which is also shown in the figure.

Quiroga et al. [55] use 10 components from a Haar wavelet transform, chosen according to the LKS. In addition to this, we have tested different wavelet representations, such as keeping the first n components of the transformed data in the concatenated signal, or when performing a wavelet transform of each channel of the signal separately.

To make sure we use interesting wavelet components, we also tried calculating the LKS based on the Albert datasets, where the spikes belonging to cluster 0 (noise) are removed. This reduction is called *Wavelet KS (supervised)*.

Calculating the wavelet transforms is measured to scale with $\mathcal{O}(ND)$, and on the desktop computer takes 0.2 seconds for 200 dimensions and 20 000 spikes. With the LKS, it still scales with $\mathcal{O}(ND)$, but has a higher constant factor, and takes about 7 times as long.

The source code for the LKS wavelet transform and Waveleft first transform is included in Listing D.4 on page D-28 and Listing D.5 on page D-32, respectively. The other ones are not included because of code similarities, but are available in the code archive submitted with the report.

4.3.4 Principal Component Analysis

Principal component analysis (PCA) is realized with the help of GSL. First, the mean for each dimension is computed. We then compute the covariance matrix, which is used to retrieve the eigenvectors and eigenvalues. The eigenvalues and eigenvectors are then sorted descending by the eigenvalues. This is the step which ensures that the first component contains more information than the next one. The eigenvectors are then stored in an eigenmatrix, which is multiplied with the mean adjusted feature matrix, with the help of GSL BLAS. Using the first n columns of the eigenmatrix gives a reduction to n dimensions. The result is the PCA matrix, which is copied to the reduced dataset.

Each of these steps are parallelized using OpenMP. On the dual-core workstation, this resulted in a doubling of performance. Calculating PCA was measured to scale with $\mathcal{O}(ND^2)$.

Schmitzer-Torbert et al. [61] propose using a feature extraction consisting of the first PCA component and the energy, per channel, giving a reduction to 8 dimensions when working with tetrode data. We also implemented this for comparison, and it is called *Energy PCA* in the application. Energy is calculated in $\mathcal{O}(ND)$, and complexity of this reduction is therefore the same as for PCA.

The source code for the PCA reduction is included in Listing D.6 on page D-34.

4.3.5 Peak alignment

Peak alignment centers all the spikes according to each spikes' highest peak. This is done because even though the Axona recording system will record $200\mu s$ before it detects a spike, it is not necessarily exactly at the highest spike. A high spike is a better discriminator, so aligning all according to this can make the comparison better.

The alignment can be used in conjunction with other reductions. For instance, peak alignment with a PCA transform performed afterwards. We will benchmark a few of these combinations in Chapter 5.

We first iterate through all spikes to find the average index of the highest value. Then, we shift all signals to left or right, so that the highest peak aligns with this index. The shift in each channel is the same, so that the timing between the channels is left unchanged. Any unfilled gap created as a result of the translation is simply replaced with 0. The complexity of this is $\mathcal{O}(ND)$.

4.4 Clustering

4.4.1 K-means clustering

The background for this algorithm is described in Section 2.3.4. Figure D.4 shows a class diagram for k-means.

A serial implementation was first created. This was a nice way to make sure the k-means algorithm was fully understood, and as a way to validate the parallel implementation. It was also used to measure the parallel speedup. The code which does the actual cluster membership classification is shared among the two implementations. While k-means results depend on the initial centroid distribution, it should produce the same results every time given that

the initial centroid placements are equal. This holds for both the serial and parallel implementation – because the initial step (cluster initialization) is still done serially in our parallel implementation (in fact it is the same code). The actual classification is "embarrisingly parallel", because the points are simply divided among the tasks and processed in isolation. We verified that the parallel version produced the same results for a varying number of cores: the results should be exactly the same as for the serial version, in the same amount of iterations.

For the parallel implementation in our program, a recursive division scheme is used, inspired by the Fibonacci example in the TBB tutorial [40]. We used this approach because it makes it very scalable, without making any assumptions about the number of available cores. The choice as to whether a subproblem should be split or not is distributed among the nodes, avoiding the use of a master node which could be a potential bottleneck. The resulting task graph will be a binary tree, where only the leaves do any actual classification, and the internal nodes subdivide the problem set. This is similar to Figure 2.10 on page 44. Algorithm 4.2 contains a high-level description of the algorithm. Note that after the points are split, they are simply added to the TBB runtime. Whether this should be run in parallel or not is entirely up to TBB. Most of the relevant source code is attached in Listing D.2 on page D-8.

This approach scales really well both with regards to the size of the data, and with the number of cores. The centroids are shared among the tasks, and updated serially at the end of the task. However, this represents an insignificant portion of the algorithm, so k-means has near-linear scalability. This is demonstrated in our results in Section 5.3.1. Note that according to Tsao [69] the number of clusters, and thus the number of centroids, will rarely exceed 20. In the datasets we use, they never exceed 7.

Our implementation depends on three inputs: the dataset, the number of centroids, and which initial centroid distribution method is to be used. The centroids are then initialized, either randomly, or according to the algorithm described in Section 2.3.4, which maximizes the distance between the points. This is a serial task. Next, the classification iterations can begin.

The parallel version is realized using Intel TBB tasks. First, memory is reserved, and a task is created (here as `KMeansTask`). This will be the root-node of our task tree. It is spawned and executed, and the process is repeated.

Inside the execute method, we first check if the number of N points in the dataset is less than a threshold T – if it is, we stop splitting the dataset and

Algorithm 4.1 Serial k-means

Require: points $[1 \dots n]$
Require: centroids $[1 \dots k]$
Require: numDimensions, maxIterations
Require: threshold

```
1:  numPointsUpdated := 0
2:  repeat
3:    for all point in points do
4:      closestCentroidDistance := ∞
5:      closestCentroid := −1
6:      for all centroid in centroids do
7:        tempDistance := 0
8:        for all d in numDimensions do
9:          tempDistance[d]+ = (point[d] − centroid[d])^2
10:       end for
11:       if tempDistance < closestCentroidDistance then
12:         closestCentroid = centroid
13:         tempDistance = closestCentroidDistance
14:       end if
15:     end for
16:     if cluster membership changed(point) then
17:       update cluster membership(point)
18:       numPointsUpdated+ = 1
19:     end if
20:     for all d in numDimensions do
21:       newCentroids[closestCentroid][d]+ = point[d]
22:     end for
23:     newClusterSize[closestCentroid]+ = 1
24:   end for
25:   for all c in centroids do
26:     for all d in numDimensions do
27:       c[d] = newCentroids[c][d]/newClusterSize[c]
28:     end for
29:   end for
30: until numPointsUpdated/numPoints < threshold OR
    numIterations > maxIterations
```

Algorithm 4.2 High-level parallel k-means

Require: points $[1 \dots n]$
Require: centroids $[1 \dots m]$
1: Distribute centroids according to a heuristic (serial).
2: **repeat**
3: instantiate TBB task
4: execute task with current dataset and a stop criteria
5: subdivide()
6: updateCentroids(points)
7: **until** stop criteria (number of iterations, membership changes, ...)
8:
9: **procedure subdivide**
10: **if** $numberOfPoints > threshold$ **then**
11: (a, b) = subdivide(points)
12: create two tasks with a and b. Add to TBB scheduler.
13: **else**
14: classifyPoints(points)
15: **end if**

perform classification. If it is not, the current dataset is split in two, and two new tasks are created and executed. At the end of the method, the number of points updated are calculated.

This means that the task tree will have a depth of $\lceil \log_2 \frac{N}{T} \rceil$. In a D dimensional space with C centroids, the leaf-nodes will need to do $T \cdot C \cdot D$ distance calculation operations to perform classification. The total amount of work for all the nodes in each iteration are then the following:

$$\mathcal{O}(\frac{N}{T} \cdot (T \cdot C \cdot D)) = \mathcal{O}(N \cdot C \cdot D) \tag{4.1}$$

We then wait for the root node to finish – which means that the entire classification for this iteration is finished – and update the centroids, in addition to recording how many points were updated. This is our main convergence criteria – if the fraction of points changed in an iteration is below a certain delta, we stop. In case convergence is too slow, we abort after a maximum number of iterations. This very rarely happens. The number of iterations are highly dependent on how the dataset is distributed, the number of K centroids, and where they are placed. The analysis for the number of iterations required is therefore very hard to give a reasonable estimate of, other than what was mentioned in Section 2.3.4.

4.4.2 Superparamagnetic clustering

The background for the algorithm is described in Section 2.3.4. Our implementation of the three listed steps of the algorithm will be described in this section. A class diagram is included in Figure D.3, and an excerpt of the source code is listed in Listing D.3 on page D-14.

Calculating the values depending only on input

The first step is to define which points are neighbors. A naive implementation would be to calculate distances between all pairs of points, and then sort these. This approach does not scale well, as it introduces a complexity of $\mathcal{O}(N^2)$. As we are only interested in a small subset of the neighbors of each point, we can use properties of the Euclidean space to limit which points we have to check. We use a data structure called a kd-tree [51], a special case of binary space partitioning trees. Building the tree with the N points has a complexity of $\mathcal{O}(N \log^2 N)$, and searching for the K nearest neighbors of a point has a complexity of $\mathcal{O}(K \log N)$. This means that the total complexity for finding the K nearest neighbors for all N points is $\mathcal{O}(KN \log N + N \log^2 N)$. However, this complexity does not apply to higher dimensional datasets, as observed in Section 5.3.2, and it seems to approach $\mathcal{O}(N^2)$. The library used to implement the neighborhood search, libANN, is described in Section 2.6.

Ideally, we could search the kd-tree for neighbors in parallel, but the library uses global variables when searching, i.e. it is not thread safe. This means we would have had to modify the library to parallelize the neighborhood search. We had a plan of doing this, but as there were other challenges to be addressed, we did not get the time to finish this task.

When K neighbor edges have been found, we remove edges which are not mutual. To remove the possibility of a cyclic graph, and as an optimization step, we only keep edges going from lower to higher indices, so that each neighbor edge is only used once in following calculations.

In some datasets, where a subset of samples are very similar, this may result in the neighborhood graph not being connected, making it impossible to cluster these subsets together with more distant samples. To remove this effect, we have also added the option to superimpose a minimum spanning tree (MST) in the neighborhood graph. We found out this scales poorly with higher dimensions, meaning that this is only practical for datasets of few dimensions. The MST is calculated by STANN (see Section 2.6), which requires that there are no overlapping points. We enforced this by calculating

a hash value for each spike, and then adding each spike to a hashmap. If the hash value already exists in the hashmap, there is an identical spike, and we move the latest spike a small step in a random dimension, before performing the test again. The probability for overlapping points is very low, and the augmented spike is very close to its original position. This step therefore does not affect clustering negatively. The MST building method in STANN is programmed with OpenMP, and is therefore already parallelized.

At the end of this section, we now have two two-dimensional vectors, containing each point's set of neighboring indices, and the corresponding interaction strengths.

Locating the superparamagnetic regions and performing clustering

When the distances have been found, we follow the pseudo code in Algorithm 2.2 on page 36. As the extra work necessary for clustering, described in Algorithm 2.3, does not increase complexity, we have combined the two parts of the algorithm. Hence, we always find cluster sizes and susceptibility at the same time. We also added warm-up iterations, where susceptibility is not calculated, but spin state is kept between each iteration. This should reduce the amount of noise in the graph, as it is expected that the first iterations describe a random state, before the spins of the different clusters start aligning.

This part of the algorithm is trivially parallelizable, as long as one calculates clustering and susceptibility for a higher number of temperatures than available cores. We then use `parallel_for` in TBB, which makes tasks consisting of sub-ranges of temperatures. The use of TBB allows control of granularity (how many temperatures should be calculated in each task), as well as better control of local variables, such as spin states for each point. This again allows a task to re-use the spin-configuration between subsequent temperatures, reducing the need of warm-up iterations.

Each result is written to a class we called `XYPlot`, which encapsulates a concurrent hashmap, provided by TBB. This takes care of synchronization when writing the results, and makes sure that race conditions may not occur. There is always an extra cost when performing synchronization, but as this is one write operation per calculated temperature, the penalty is close to zero.

Random numbers are generated using the Boost library (Boost is described in Section 2.6.3). We initialize one instance of the random generator per task, using different seeds, so that there are no synchronization issues. When used like this, the Boost random generators are thread safe. Different

generators are available, but we ended up using the fastest one (rand48), as the range of random numbers to be generated is low.

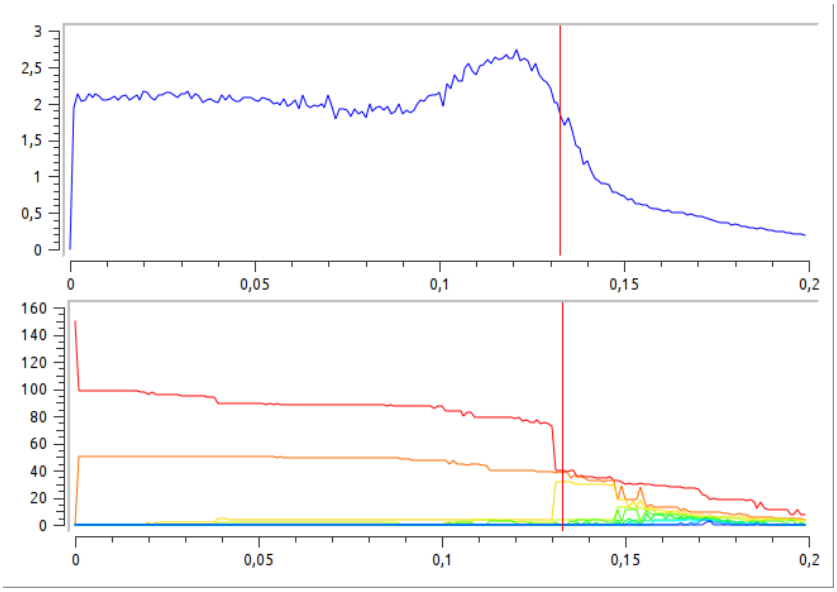
Random number generation takes up a big part of Monte Carlo algorithms. A possible further optimization of the algorithm, is to use processor vendor specific routines for random number generation, which are optimized for execution on given hardware. An example is Intel Math Kernel Library (MKL) [5], which provides a series of random generators optimized for SIMD execution, meaning it generates several random numbers in parallel. These are written to a buffer, from which they quickly may be retrieved when needed.

Verification and lessons learned

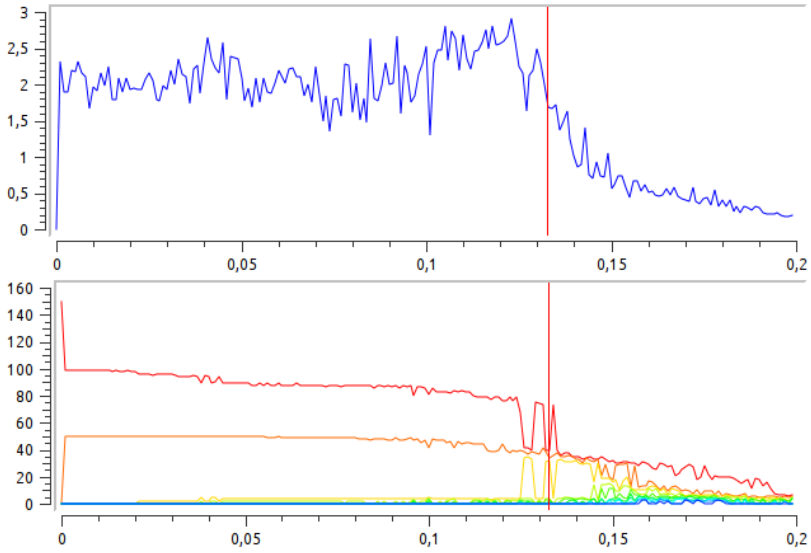
To test the implementation while developing, we mainly used the Iris set, as described on Section 3.1.1. Figure 4.3 on the next page shows the effect of reducing the number of iterations. With fewer iterations, the plot becomes more jagged, but the main structure is kept. The interesting range seems to lie between 0.12 and 0.16 in both plots, as this is where the susceptibility is decreasing rapidly from the peak (superparamagnetic phase). This indicates that it is possible to calculate susceptibility for larger ranges of temperatures with few iterations first, and then focus on interesting ranges with an increased number of iterations. Also, step size may be adjusted, to give more details in interesting ranges, and save calculations in uninteresting ranges. Figure 4.4 on page 83 demonstrates a step size increased by a factor of 10, giving a less detailed plot, but in $\frac{1}{10}$ of the time.

Because SPC is a stochastic method, it is difficult to make a perfect verification. However, we eventually got the SPC program used by Quiroga et al. [55] working with our datasets, and were able to compare results. The clustering performance was very similar to ours. An example of the three circles dataset explained in Section 3.1.3, can be seen in Figure 5.8 on page 102. The graph is similar to the one reported by Blatt et al. [18]. Note that to get an equally smooth graph we had to run the algorithm with many Monte Carlo iterations, which makes the classification slower.

As the neighbor graph and MST are deterministic properties of each dataset (after reduction), it would be beneficial to store this information for future runs once it is calculated. The user could then schedule batch jobs to build these graphs for several datasets over-night, to save time when loading datasets.



(a) 10 000 iterations



(b) 500 iterations

Figure 4.3: Susceptibility and cluster sizes for different number of Monte Carlo iterations with SPC on the Iris set. Step-size is 0.01 in both plots.

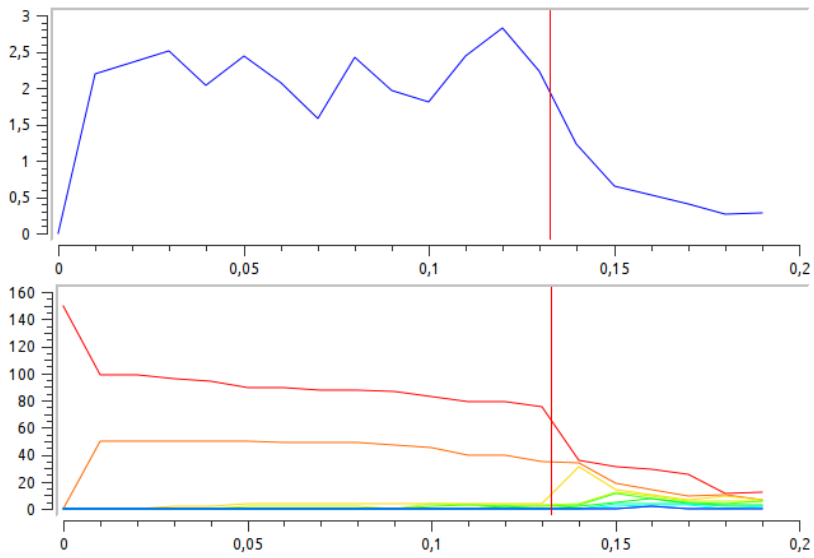


Figure 4.4: Increased step size in SPC on the Iris set. Step-size is 0.1, and iteration count is 500.

4.5 Cluster Quality

To assess cluster quality, we implemented the algorithms described in Section 3.2.

4.5.1 L_{ratio}

To find the L_{ratio} for a cluster C, we need to calculate the Mahalanobis distance between the cluster and all spikes not belonging in the cluster. To find the Mahalanobis distance, we first calculate the covariance matrix Σ_C using GSL. Second, we use LU-factorization (also provided by GSL) to find the inverse of the covariance matrix, Σ^{-1}. Finally, for each spike outside the cluster, we now multiply its vector x_i on both sides of the covariance matrix, and get the Mahalanobis distance: $D^2 = x_i^T \Sigma^{-1} x_i$. This multiplication is performed using BLAS in GSL.

With the Mahalanobis distance calculated, we then check the probability p for getting a value less than this in the cumulative χ^2 distribution function with as many degrees of freedom as we have dimensions. To verify our implementation of the Mahalanobis distance, we compared the results with MATLAB's Mahalanobis functions, and obtained equal results.

For a cluster, $L(c) = \sum 1 - p$ for each spike not belonging to the cluster. $L_{\texttt{ratio}}(C)$ is then calculated as $\frac{L(C)}{n_C}$, where n_C is the number of spikes in the cluster.

4.5.2 Isolation distance

When calculating the $L_{\texttt{ratio}}$, we save the calculated Mahalanobis distances in a list. This list is then sorted, and the isolation distance for each cluster is found as the n_C'th lowest Mahalanobis distance.

4.5.3 F-measure

Precision, recall and F-measure are calculated for each cluster C_i, with respect to clusters C_s in a manual ("correct") classification. This results in a matrix of $n_S \times n_C$ for each of these values, where n_C is the number of clusters, and n_S is the number of manually classified clusters. The calculation is straight-forward as explained in Section 3.2.5.

Note that for complete clusterings, each row in the *precision* matrix will sum to 1, as will each column in the *recall* matrix. This is because every point has belong to a cluster in both sets.

4.6 Optimization and parallelization

Exploiting available resources in multi-core CPUs was one of our main focus points when developing the application.

Ideally, we would have programmed every parallelizable for-loop using TBB, removing the OpenMP dependency. However, this results in a need for a slight change in design. Every `parallel_for` in TBB requires a separate class specifying how it should be initialized, and the method to be applied to each variable in the range. The need for this is removed with the new standard of C++, named C++0x, which allows the use of lambda functions. However, we chose not to use the C++0x standard, as only the newest compilers support this syntax. The compilers available on Kongull were too old to support it. For the smaller parallelizable for-loops, we have used OpenMP, which requires only one added line per for-loop. The TBB reference manual [42] states that mixing OpenMP and TBB is supported, and performs well as long as one does not nest TBB and OpenMP parallelism within each other. We have made sure not to mix these.

We here describe our optimization of k-means and SPC, and then wrap up with general remarks.

4.6.1 K-means

K-means is a fast algorithm, and with the parallel component it can handle very large datasets. The code was profiled to see where time was spent. Because of the near-linear speedup, it was expected that most of the time was spent in `classifyPoints()`, where the calculations performed by each thread is done. We profiled several datasets. To get stable results, we modified Paraspikes to run the classification 100 times, each time with a different seed for the random generator. Indeed the profiling results showed that most of the time was spent in this function. The appendix shows a call graph in Figure B.6, which is hard to read on paper but attached for completeness. 98.4% of the time was spent in `classifyPoints()`, which was a very typical output for k-means. 0.6% was spent initializing the centroids. The rest is spent on general logic and TBB.

It is interesting to see that `classifyPoints()` takes so much of the time, and `KMeansTask::execute()` so little, because the latter dynamically allocates and frees memory every time the data is split. Clearly, this is a fast operation.

As `classifyPoints()` exclusively performs floating point operations on arrays, a natural next optimization step would be to use intrinsics to perform several calculations at the same time on hardware which supports this. K-means performed well enough in this thesis, so this was not explored further.

4.6.2 SPC

SPC proved to be a hard algorithm to optimize. The different steps of the algorithm may be parallelized, but some operations also have to be done serially, such as determining which spikes had the same spin with which probability, and then build clusters of these. However, a complete clustering based on a temperature is an independent operation, and clusterings based on multiple temperatures can be done in parallel. We profiled the code to identify hotspots in the code. The call graph for the inner part of the algorithm is shown in Figure B.7. It is mainly included as an example of the callgraph output, and most of it is not readable on paper. However, as the size of the nodes corresponds to the amount of work performed in the given function, we can see that random number generation is taking up most of

the time. We were aware of this, but the profiling results inspired us to find a more efficient random generator. We generate random spins, which are in the range $0 \ldots 19$, and random fractions $(0.0 \ldots 1.0)$, which are used for generating probabilities. Initially, we used the Mersenne Twister generator, but replaced it with Boost's `rand48`, which is considered the fastest one in Boost, and has 227% of the Mersenne Twister's performance [2]. This resulted in a speed-up of 18%.

4.6.3 General remarks

The most compute-intensive part of the application, which has not yet been parallelized, is the neighbor search in SPC. This would require modifying libANN, which in its current state is not thread-safe. We examined the source code, and think this is an achievable task, as it seems that the algorithm keeps global state for convenience only – to avoid having to pass parameters between different functions in the library. As libANN is an open-source project, anyone could take on this task, which would be helpful to all future users of libANN who want to perform searches in parallel. Other parts of the algorithm might also be explicitly parallelized, increasing benefit even further, without changing the API.

Looking back at Amdahl's and Gustafson's laws (Section 2.2.2), we see that our application contains serial parts which limit the parallelization. Parsing input files is a serial task, but as it takes virtually no time, it does not contribute to the serial fraction. The main limit is the neighborhood search just mentioned. This also contributes to a large fraction of the runtime, as demonstrated in Section 5.3.2. It seems to take about 5%-10% of the execution time when the whole program uses one thread and we perform clustering for ten temperatures with 1 000 iterations. This means that before it is parallelized, parallel speed-up for the algorithm in total cannot exceed a factor of 10-20.

However, it seems clear that the size of datasets will increase in the future. Improved recording equipment may increase the number of spikes to be sorted. If recording sessions last longer, there will also be more spikes. We may also increase the number of Monte Carlo iterations, to give more consequent results, or reduce the interval between temperatures to provide better resolution. All these steps will decrease the serial part of the algorithm, in the spirit of Gustafson's law. Therefore, it seems clear that the parallelization of the algorithm is beneficial, and that further developments in parallel hardware will benefit the application.

4.7 Development comments

Programming the application and the algorithms has been very time-consuming, which is reflected by the amount of code produced. Since there is so much code, we chose to only include the most relevant parts of it in Appendix D. The complete source code, delivered with this report, contains around 10 000 lines of code. In addition, numerous scripts were created to facilitate benchmarking and graph creation. These were mostly written in Python or Bash.

To keep a steady flow during this thesis, we used a time-based iterative approach with two-week iterations and end-of-iteration goals. Each iteration had a number of tickets describing the work associated. At the end of the iteration, any incomplete tickets would be moved to later iterations. This way, we could more easily see how much work was left. The iterations and tasks were quite helpful near the end of the thesis, where we quickly saw that we had to start working longer days to finish our planned work.

We often used pair programming when development was non-trivial. This is a great way not only to solve the hard problems, but also to share knowledge.

A few interesting development graphs are placed in Appendix A. These count the total number of lines in the entire project, which include the datasets, so it is not as useful as it could be. However, it seems clear that the project has been active throughout the period, and that the work appears to be evenly distributed. The big spikes at the end are the additional datasets we imported for additional benchmarking.

CHAPTER 5

Results and evaluation

In this section we will report the results from our implemented algorithms. We have chosen not to analyze each spike reduction independently, as it is hard to say anything about the discriminating qualities without trying to cluster them. They are therefore evaluated in tandem with the clustering algorithms.

We first evaluate the cluster quality algorithms in Section 5.1. It is important to know how these perform, before relying on their results in Section 5.2, where we benchmark the clustering capabilities. We then talk about how the two clustering algorithms perform and scale in Section 5.3. Finally, we talk about the challenges during development in Section 5.4.

5.1 Cluster quality

5.1.1 Silhouette coefficient

The silhouette coefficient was computed for all of our KI datasets, with and without the cut files, in addition to our synthetic datasets.

K-means was benchmarked using a k from 2 to 20. Each test configuration was run 1000 times and then averaged to help ensure a more stable result. We set $k = 2$ as a minimum because otherwise no clustering is necessary. $k = 20$ was chosen as Tsao [69] told us this was the maximum number to expect from the KI recordings.

The result of one of the recordings is shown in Figure 5.1. Here, the silhouette is computed by comparing the points in the reduced dataset. The peaks of channels and PCA-based reduction schemes perform best for selecting a k based on a peak in the plot. For additional plots from the Albert set,

refer to Appendix B.

However, these also reduce the dataset greatly – down to between two and four dimensions. We also tried a few more configurations than the ones reported. We ran the PCA reductions with more components – 4 and 16. We also changed the Wavelet KS reduction to use the Haar wavelet, which is what Quiroga et al. [55] used. These modifications did not change the classification results noteworthy, and were thus left out.

We wanted to see how the silhouette looks when comparing the different representations but with the same unreduced dataset. That is, we perform the clustering in the reduced dataspace, but then perform the silhouette calculations in the original unreduced space. The idea is that it should be a more fair evaluation across different representations, since they are compared in the same dimensional space. Figure 5.2 on page 92 shows the result on the first dataset. This actually made the silhouette coefficient less useful as a discriminant, and we have not attached the results for the rest of the datasets.

Note that although Rousseeuw [58] reports that this metric should be usable to automatically choose a k, it does not give any meaningful hints on our datasets. In fact, according to this metric, the number of clusters in the dataset should often be 2, since the coefficient is usually just decreasing.

However, as we learned that cluster 0 in all the KI recordings is a so-called *noise cluster*, we tried running this again, but with the noise cluster removed. All the datasets were profiled again, with all the reduction schemes. Sometimes it did improve the results in that the silhouette coefficient would exhibit a peak, as observed in Figure 5.3 on page 93. A few more plots have been attached in Section B.1. Notice how PCA often has a peak, indicating the best number of clusters in this set, often matching the cut files. However, for the noisy datasets this method proved too unreliable, and we were unable to choose a k based on this metric.

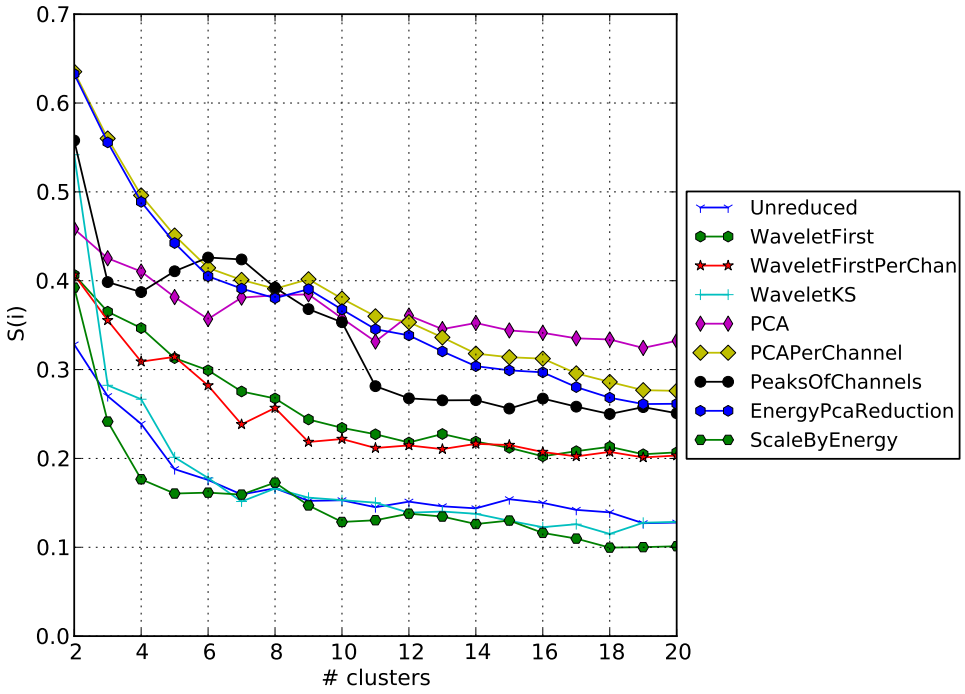


Figure 5.1: The silhouette coefficient plot of k-means clustering the Albert 1 dataset. The coefficient is calculated in the reduced space. According to the cut files, there should be 3 neurons and a noise cluster in this set.

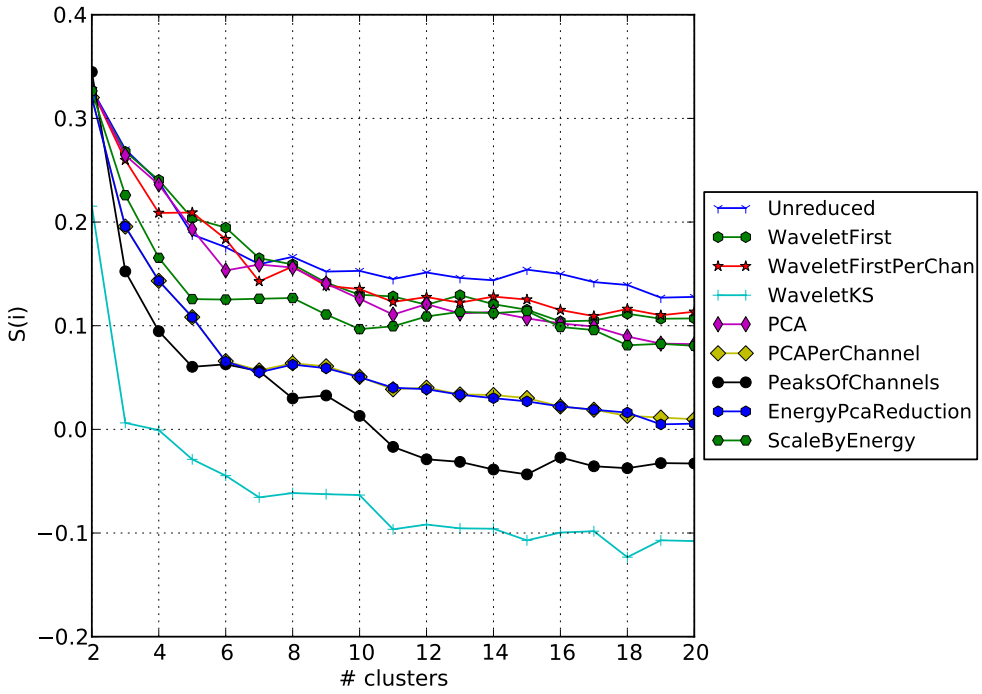


Figure 5.2: The silhouette coefficient plot of k-means clustering the Albert 1 dataset. The dataset was reduced and clustered. Then the cluster classification was used on the original unreduced set, and the silhouette was computed. According to the cut files, there should be 3 neurons and a noise cluster in this set.

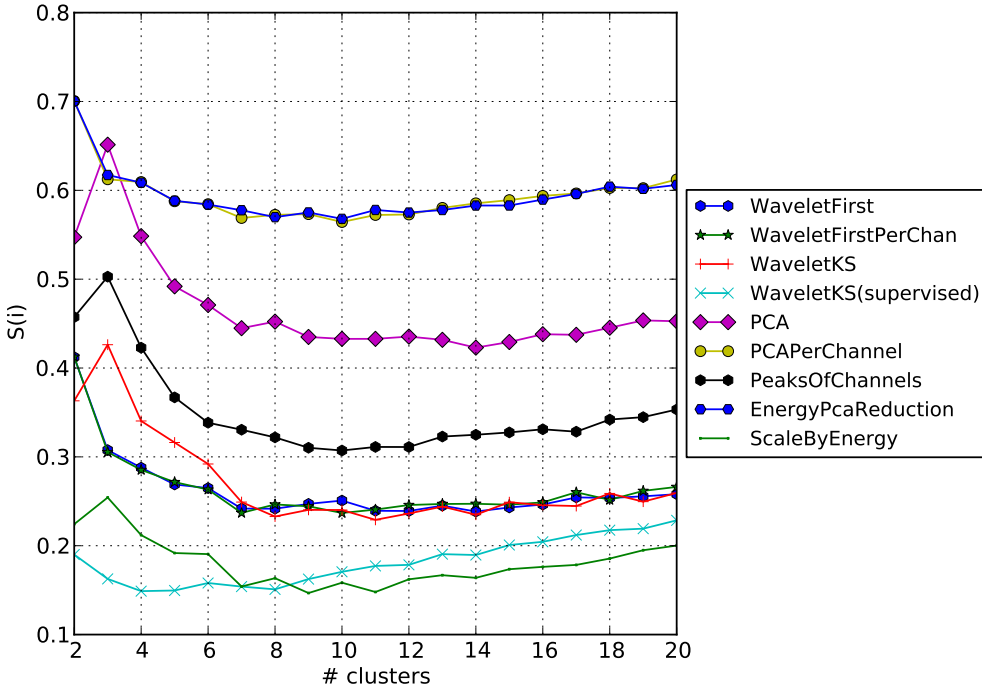


Figure 5.3: The silhouette coefficient on the Albert 1 dataset, with the noise cluster removed. This metric suggests $K = 3$ clusters. According to the cut files, this is correct.

To verify that this relates to the noisy dataset and not the algorithm, we ran it on the synthetic dataset "Three islands", which is a well-separated dataset with three clusters. The result is shown in Figure 5.4 on the following page. Here it it possible to say that k=3 is in fact best; k=10 is actually better, but that is because the dataset only contains 10 points. This indicates that the KI datasets are not separated enough to rely on the silhouette coefficient. Also, it requires the clusters to be linearly separable. To demonstrate, we evaluated the three circles set after being correctly classified by SPC. The silhouette coefficient claims that this is a bad clustering, because it is unable to classify the discrete circles when one is contained within another, giving negative averages to the outer circles.

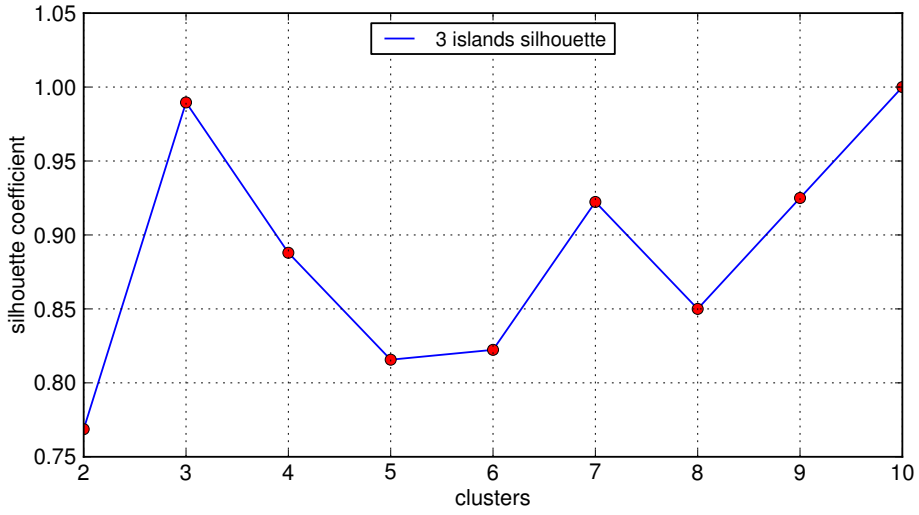


Figure 5.4: The silhouette of the k-means clustered three islands dataset, consisting of 10 points. The plot suggests $k = 3$ is the best.

5.1.2 L_{ratio}

We have had problems using L_{ratio} to assess quality in our datasets. This is probably related to the way clustering is performed. K-means clustering will always produce a clustering where all points are closer to its own center than other clusters. This means that there will always be a convex hull which may be wrapped around the cluster, and there will be no noise spikes within this border. For most experiments, this seems to be the case for SPC as well. The reported L_{ratio}, as well as L_C are then close to zero, meaning that there is little noise within the border of the cluster. Hence, we were not able to use L_{ratio} for assessing our clustering results.

As mentioned in Section 3.2.3, there needs to be ten times as many spikes in a cluster as the number of dimensions. This also means that it cannot be used for high-dimensional feature extraction, especially the unreduced set.

We tried measuring this quantity on data from the cut files. This gave a value of approximately 0 for the actual clusters, but a high value for the noise cluster (cluster 0). With the dataset Albert3, $L_C = 281.608$, and $L_{ratio} = 1.309$, when using the *Energy PCA* reduction.

Also, when used on the three circles dataset, clustered with SPC, the clusters are not linearly separable. Here, the inner circle, which does not contain noise, has $L_C = 0.0$, which is good. The others are reported to have higher numbers, 800 and 1 300, indicating bad clustering. The outer circles

then get L_{ratio} values of approximately 0.5.

Because of this, we have not reported L_{ratio} in other experiments.

5.1.3 Isolation distance

We evaluated the isolation distance for the different clustering algorithms, with different feature extraction algorithms. In addition, we did the same for the cut files. The results we get are inconclusive, as we see no clear trend in the isolation distance.

We believe that this and the L_{ratio} problems are caused by a low degree of separability in the datasets, as the Mahalanobis distance is based on the inner distribution of datasets.

5.2 Clustering results

5.2.1 K-means

Figure 5.5 on the next page shows the result of running k-means on the NTNU toy problem. It is obvious that k-means is unable to capture the non-linear properties of the dataset.

We also tried comparing the clustering results directly to the cut files. To do this, after performing the clustering we created a mean spike in each cluster, both for our automatic clustering and their cut files, and compared them using the sum of squares. The basic algorithm is:

- For each point in cluster A
- For each point in cluster B
- Sum the differences between the two clusters, in all dimensions

This will give a float value to every combination. A low score reflects a high similarity, called a similarity score. Each combination through the matrix creates a unique path. The path which has the lowest global similarity score represents the combination of the best *overall* matching clusters between the two clusterings.

Because you can not assume that the first cluster in the manually cut dataset is the one that matches best with the first cluster in the unsupervised

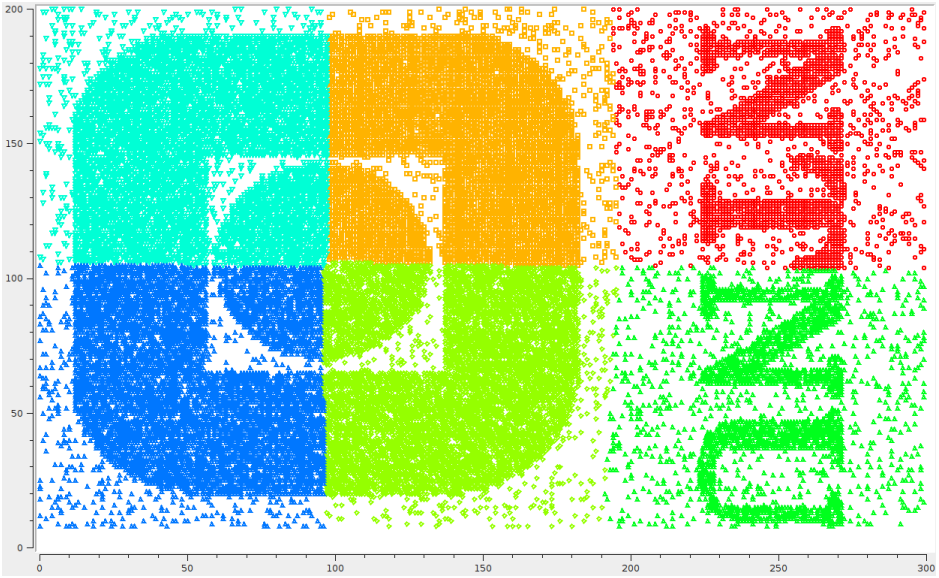


Figure 5.5: The NTNU toy problem clustered with k-means. Colors denote cluster membership.

solution, every permutation must be compared. This creates a 2D similarity matrix between the two types of clusterings, as shown in Table 5.1(a) on page 98, where the best global match is highlighted. Notice that a global match can mean that the best matching clusters are not always paired. In this example, **B3** is matched by **A4**, with a score of 45.84. The best *local* match however is **B3** to **A0**, which gives the lower value 44.87. But if **B3** is matched to **A0**, then **B4** must be matched against another (worse matching) cluster, and so on. This is once again one of the intricacies of clustering. Which pairing is the best? To try and answer this, we compare these results using precision, recall and finally the F-measure.

The precision tells us how many of the relevant spikes are in the cluster. Looking at Table 5.1(b) it seems that **A0** should indeed have been clustered with **B3**. Looking at the recall in Table 5.1(c), which measures how much of the correct data was actually put in the correct cluster, it is now clear that **A4** actually captures most of **B3**. This indicates that **A0** contains more than one cluster, compared to the B set.

The F-measure sums these up in Table 5.1(d), and actually tells us that the automatic clustering did fairly well. The worst pairing is **B3** and **B4**. It could be that these two should be merged.

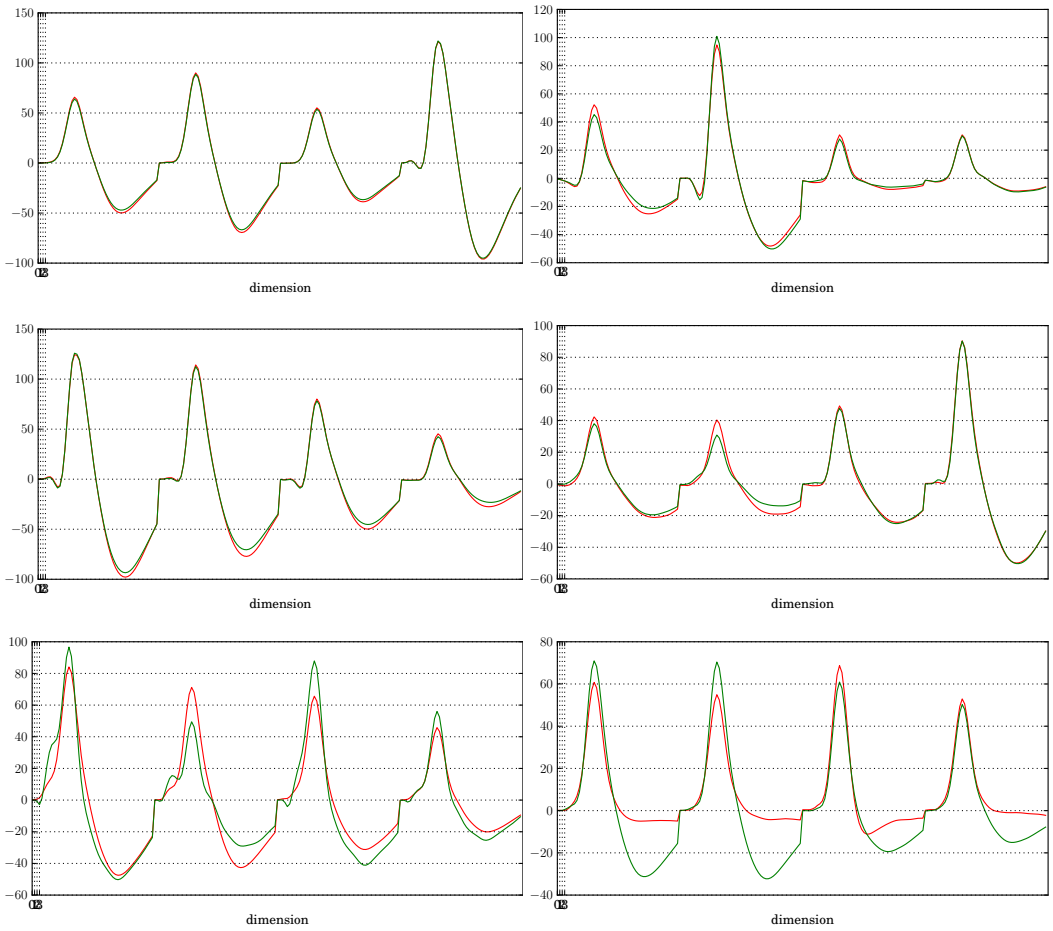


Figure 5.6: A similarity plot between the unsupervised k-means solution (red) and the manually cut set (green). All 200 samples were used as a feature vector, and had the peaks aligned. Each plot represents a cluster. Courtesy of Bergheim [16].

For a visualization of this, we plotted the best matching mean spike from our k-means result to the cut files. A plot with the peaks aligned can be seen in Figure 5.6. For completeness, we a plot without the peaks aligned are attached in Section B.1.1. You can see that some of the spikes match very well. It is likely that noise is part of the reason for this, because we know that a lot of the dataset is marked as noise in the manual cut files.

The code which generates the similarity matrix has been attached in Listing D.1 on page D-5.

It is interesting to see how the different spike representations perform

Table 5.1: Comparison of similarity matrix and F-measure for a clustering performed on the 180105 dataset. Bold denotes the global best match according to the similarity matrix in (a).

(a) Similarity matrix

	B0	B1	B2	B3	B4	B5
A0:	108.79	122.75	116.82	44.87	**39.35**	97.50
A1:	156.97	**16.52**	242.46	116.59	138.57	135.34
A2:	147.98	115.73	184.55	139.18	109.28	**14.22**
A3:	159.22	244.78	**14.07**	132.30	118.06	175.53
A4:	188.14	148.76	101.33	**45.84**	110.66	165.41
A5:	**4.67**	143.42	162.28	144.85	77.96	154.89

(b) Precision matrix

	B0	B1	B2	B3	B4	B5
A0:	0.17	0.41	0.33	0.86	**0.92**	0.18
A1:	0.00	**0.43**	0.00	0.00	0.00	0.00
A2:	0.00	0.17	0.00	0.00	0.00	**0.81**
A3:	0.00	0.00	**0.65**	0.00	0.03	0.00
A4:	0.00	0.00	0.02	**0.14**	0.00	0.00
A5:	**0.83**	0.00	0.00	0.00	0.05	0.00

(c) Recall matrix

	B0	B1	B2	B3	B4	B5
A0:	0.07	0.06	0.10	0.35	**0.37**	0.04
A1:	0.00	**0.98**	0.00	0.01	0.00	0.01
A2:	0.00	0.12	0.00	0.00	0.00	**0.88**
A3:	0.00	0.00	**0.95**	0.00	0.05	0.00
A4:	0.00	0.00	0.11	**0.88**	0.01	0.00
A5:	**0.94**	0.00	0.00	0.00	0.06	0.00

(d) F-measure matrix

	B0	B1	B2	B3	B4	B5
A0:	0.10	0.11	0.16	0.50	**0.53**	0.07
A1:	0.00	**0.59**	0.00	0.00	0.00	0.01
A2:	0.00	0.14	0.00	0.00	0.00	**0.84**
A3:	0.00	0.00	**0.77**	0.00	0.03	0.00
A4:	0.00	0.00	0.04	**0.24**	0.00	0.00
A5:	**0.88**	0.00	0.00	0.00	0.05	0.00

compared to the cut files and the F-measure. Table 5.2 on the next page lists the result for all our spike representations. The parenthesized numbers represent the best matching clusters, which were not chosen because we are seeking a global best match. Still, we opted to show them here, as they could indicate that clusters should be merged.

The top performer is the peaks of channels. It not only beats every other representation by a fair margin, it is the only reduction which did not have a match collision. That is, we were able to always chose the best matching clusters. It is no surprise that this reduction performs best – we are comparing our results to cutfiles created by humans based on a *peaks of channel reduction*. Supervised wavelet KS is the only other representation which is able to beat the unreduced representation. We tried running PCA with a number of different components, but the results did not change noteworthy.

5.2.2 SPC

Figure 5.7 demonstrates SPC running on the same dataset. Here it is clear that it is a capable algorithm when it comes to handling both non-Gaussian distributions as well as noise. With the constraint that a cluster must have 8 or more members to be concidered a cluster, it performs very well. Notice that the N, T and U are actually made up of two clusters each. This happened because noise is not only added to the image, but random bits of the original data was also removed. A comparison to Figure 3.1 on page 58 reveals that the clusters broke up where the letter was at its thinnest. Without any noise added, the dataset was clustered perfectly with six clusters.

The result of the clustering on the three clusters dataset was compared to the result reported by Blatt et al. [18]. The plot in Figure 5.8 on page 102 shows similar results.

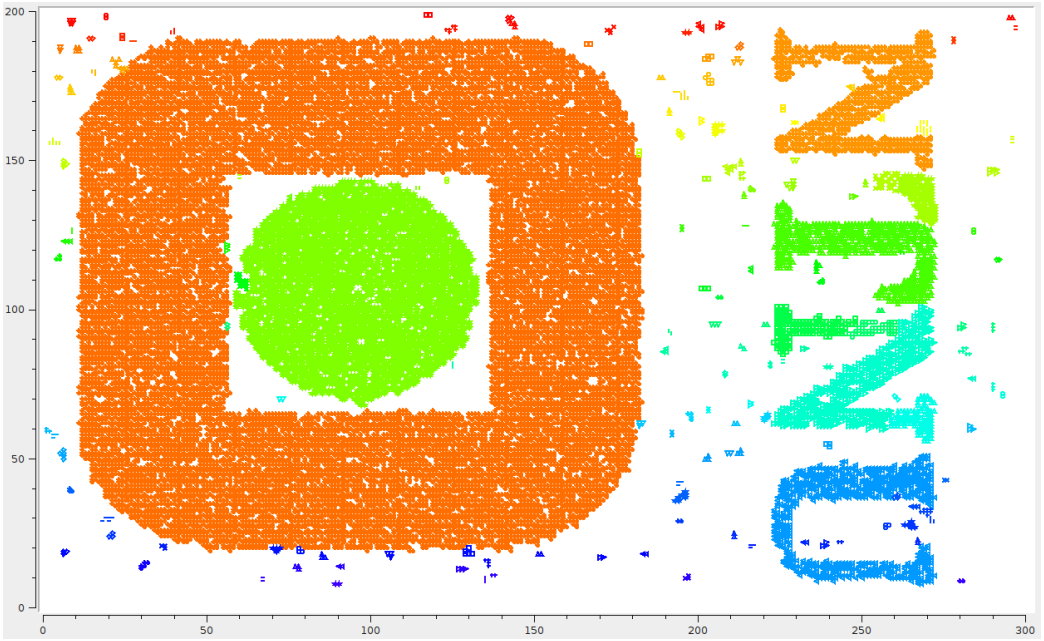
A graphical silhouette plot is not directly usable for the SPC algorithm because the number of clusters found depends on the temperature and a random element, instead of a predefined number of clusters.

We did however calculate the silhouette coefficient for a few sample clusterings. In general, SPC performed worse than k-means when compared solely to the cut files. When using good temperatures, the global silhouette value would be anywhere from 0.4 to 0.8. The reason for the low silhouette value is likely caused by the fact that there are a lot of points that are never clustered because of noise – these are still used when calculating the silhouette coefficient however.

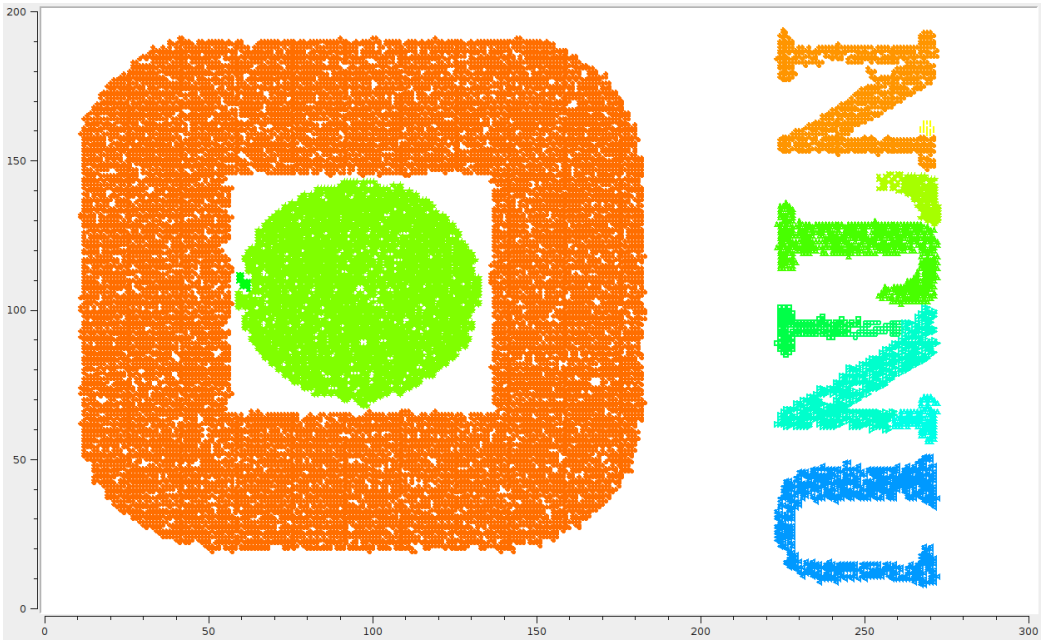
It is interesting to see how the clustering performs with regards to preci-

Table 5.2: A summary of k-means compared against the cutfile using the F-measure metric on the Albert3 dataset. S(i) is the silhouette coefficient. Peaks of channels performs best.

Reduction	**Cluster 0**	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Average**	**S(i)**
Unreduced	0.69	0.96	0.63	0.25 (0.56)	0.63 (0.71)	0.19
Wavelet First:	0.65	0.87	0.66	0.20 (0.52)	0.60 (0.68)	0.27
Wavelet First per chan:	0.65	0.94	0.67	0.20 (0.51)	0.62 (0.69)	0.26
Wavelet KS:	0.61	0.15 (0.60)	0.60	0.47	0.46 (0.57)	0.26
Wavelet KS Haar:	0.66	0.96	0.83	0.0 (0.18)	0.62 (0.66)	0.38
Wavelet KS (supervised):	0.66	0.98	0.83	0.22 (0.35)	0.68 (0.71)	0.31
PCA:	0.69	0.94	0.65	0.28 (0.54)	0.64 (0.71)	0.38
PCA 16 components:	0.69	0.96	0.65	0.24 (0.54)	0.64 (0.71)	0.38
PCA per channel:	0.68	0.19 (0.31)	0.41	0.25 (0.59)	0.39 (0.50)	0.37
Peaks of channels:	**0.68**	**0.84**	**0.91**	**0.77**	**0.80**	**0.52**
Energy PCA reduction:	0.68	0.19 (0.31)	0.40	0.25 (0.60)	0.38 (0.50)	0.38
Scale by energy:	0.72	0.18 (0.55)	0.64	0.57	0.53 (0.62)	0.21

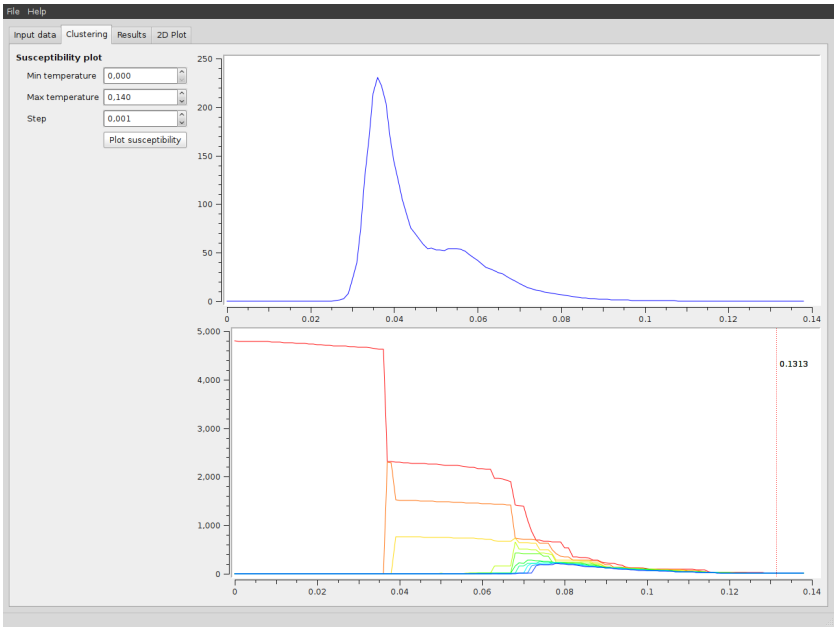


(a) No small clusters are removed. The small clusters are formed because of the noise distribution.

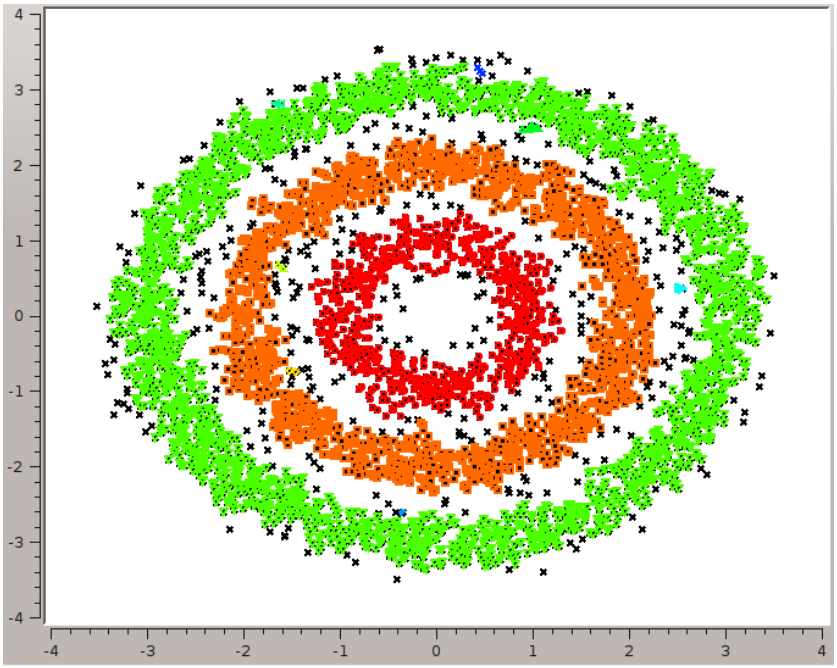


(b) Clusters with 8 or fewer members are removed.

Figure 5.7: The results from using super-paramagnetic clustering. As seen, the algorithm fares much better than k-means on this dataset. Colors denote cluster membership.



(a) Top: The susceptibility plot. Bottom: The cluster sizes



(b) An example classification performed by SPC

Figure 5.8: The plot for the three circles problem as used by Blatt et al. [18]. The unclustered dataset is shown in Figure 3.2 on page 59. Notice how the big cluster breaks down into three smaller clusters – the three circles.

Table 5.3: Cluster analysis of the Albert3 set, compared to the cutfile.

(a) Precision

	B0	B1	B2
A0:	**0.97**	0.00	0.03
A1:	0.03	**1.00**	0.00
A2:	0.00	0.00	**0.97**

(b) Recall

	B0	B1	B2
A0:	**0.68**	0.00	0.03
A1:	0.02	**0.77**	0.00
A2:	0.00	0.00	**0.96**

(c) F-measure

	B0	B1	B2
A0:	**0.79**	0.00	0.03
A1:	0.03	**0.87**	0.00
A2:	0.00	0.00	**0.96**

sion and recall. Table 5.3 shows that SPC has a very good precision. This means that the clusters it found were parts of the clusters that the manual operator found. Looking at the precision table, it is clear that where SPC fails, if the manual cut clustering is to be believed, is that it does not capture enough of some of the clusters. The F-measure sums this up – one clustering is almost perfect (96%), while the other two are good, but not perfect.

As with k-means, we ran all of the reductions on the Albert 3 set, which is summarized in Table 5.4 on the next page. Note that SPC will generate an arbitrary number of clusters, but the three most similar ones were kept. There are unclassified points in the set, which will affect the silhouette coefficient, S(i). Once again, the peaks of channels performed best. The silhouette coefficient is fragile as a single bad cluster can ruin the global index. As an example, the energy PCA reduction with a -0.08 average F-measure actually has 7 clusters with an F-measure of more than 0.8. However, the first cluster is the biggest, and it has an index of -0.18. If we discard this as a noise cluster, then S(i) increases to 0.88.

As observed, a good average F-measure does not guarantee that S(i) will be good. This could indicate that the cut files do not contain optimal clusters.

Table 5.4: A summary of SPC compared against the cut file using the f-measure metric on the Albert3 dataset. S(i) is the silhouette coefficient. Peaks of channels performs best.

	F-measure cluster #				
Reduction method	**0**	**1**	**2**	**Average**	**S(i)**
Unreduced	0.82	1.00	0.45	0.76	0.38
Wavelet First	0.83	0.05	0.47	0.45	0.49
Wavelet First per chan	0.82	0.41	0.08	0.44	0.50
Wavelet KS	0.82	0.49	0.03	0.45	0.54
Wavelet KS Haar	0.85	0.49	0.98	0.77	0.06
Wavelet KS (supervised)	0.83	0.91	0.43	0.72	0.44
PCA 2 components	0.85	0.55	0.86	0.75	0.48
PCA 16 components	0.85	0.98	0.50	0.78	0.48
PCA (2) per channel	0.83	0.58	0.03	0.48	0.00
Peaks of channels	**0.86**	**1.00**	**0.55**	**0.80**	**0.59**
Energy PCA reduction	0.83	0.03	0.59	0.48	-0.08
Scale by energy	0.44	0.76	0.92	0.71	0.12
Peaks of channels	0.76	0.88	0.37	0.67	0.32
Aligned Wavelet KS Haar	0.84	1.00	0.53	0.79	0.47
Aligned PCA	0.84	0.55	0.94	0.78	0.53

5.3 Performance

5.3.1 Speedup

To calculate how the algorithms scale with the number of available processor cores, we use the parallel speedup and parallel efficiency metrics, as explained in Section 2.2.1.

K-means

The k-means algorithm is parallelized by subdividing the dataset until a threshold value is reached. Once the size of the dataset is below this threshold, the actual calculations are performed in serial. Specifying such a threshold as a fixed value is not ideal, as it is bound to be influenced by the hardware used at the time the program was built. But it is not easy to make this scale with the increasing capabilities of a single core either – simply relying on the CPU clock frequency, for instance, is brittle, as many things influence how fast a processor is. As the processing power in a single core is increased, this fixed number should be raised to fully exploit the available resources.

However, the Intel TBB runtime will dynamically try to keep the maximum number of cores busy by managing many more tasks than physical threads. This means that if a core is not fully utilized because the serial execution is not computationally demanding enough, it will simply run more tasks on the same core. To see how this threshold could affect the performance, we ran a series of benchmarks against the serial version on the 180501 dataset. Each configuration was run 100 times, and the minimum value was kept. The benchmark was run on a Kongull node with 12 cores, where the ideal speedup should be 12. The result was plotted on a graph using a logarithmic scale in Figure 5.9 on the following page. Values between 10 and 1000 represent a nice speedup, and demonstrates TBB's ability to handle tasks of varying computational complexity. Even with a threshold of only 2 (the first red dot), the speedup is 8.24, or a parallel efficiency of around 69%. This is remarkably efficient. With a threshold of only 2, the 34 403 data points are split 16 times, demanding $2^{16} = 65\,536$ tasks. This beats a threshold of 4 000, which only creates 32 tasks. This is happening because TBB has such a low overhead, and is able to utilize all the cores down to a fine-grained level. The last point on the graph represents a threshold of 40 000, and performs exactly like the serial version. This is as expected, as the dataset is smaller than this, and no subdivision will occur. The same benchmark was run on a

synthetic dataset, and similar results were produced.

The k-means algorithm exhibits a nice linear parallel speedup, as the graph in Figure 5.10 on the next page shows. The efficiency is always close to one, as it has to be in order to exhibit a linear speedup.

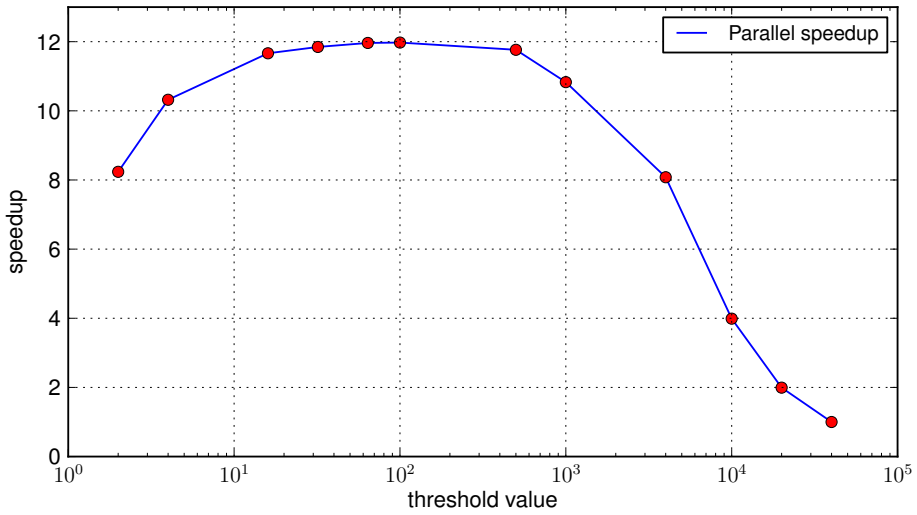
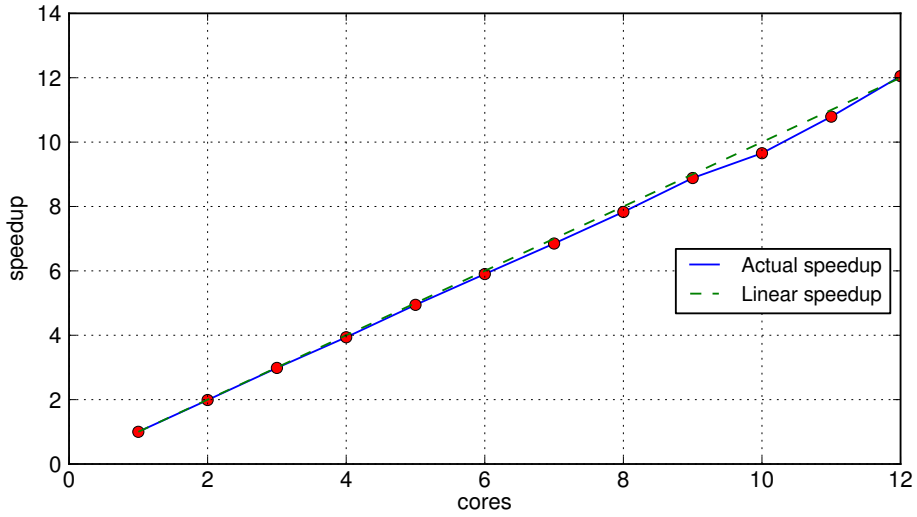


Figure 5.9: K-means parallel speedup using Intel TBB and several threshold values. The x-axis has a logarithmic scale.

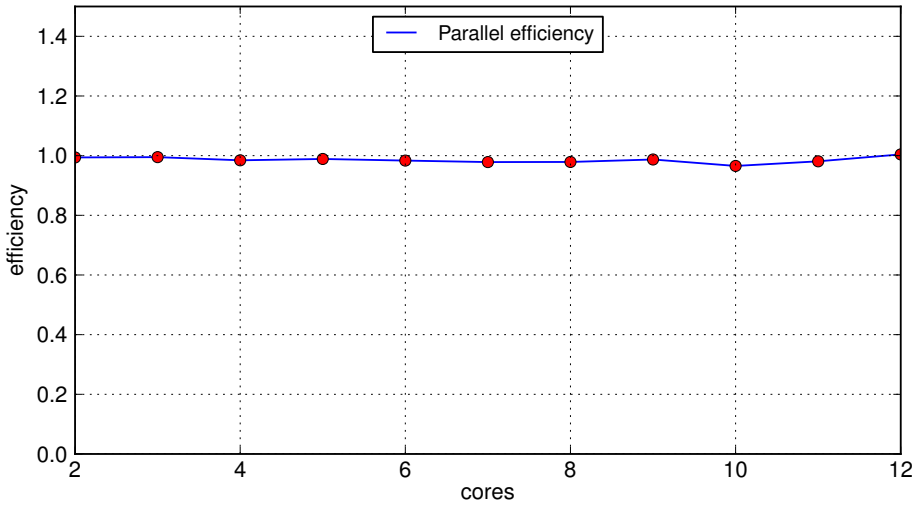
SPC

We measured speedup by limiting the number of threads in use by TBB on Kongull. Figure 5.11 on page 109 depicts the speedup when running SPC for 24 temperatures two datasets of different sizes. Note that the MST was not included in this part of the measurements. To better see how TBB subdivides the range of temperatures, we also print out each time a task is spawned for a temperature subrange. We do not adjust the size of a task ourselves, but rely on the default settings for granularity in TBB. With increasing number of threads, the subranges become smaller, until a task consists of plotting just one temperature. This is positive for better load balancing, as the work in each temperature may differ slightly.

We clearly see that the best performance is achieved when TBB is allowed to decide the number of threads itself. It is not far from linear, as the factor is between $\frac{9.5}{12}$ and $\frac{10}{12}$. This means a parallel efficiency measured to be approximately 80%. With an increased number of temperatures, the efficiency is expected to be higher. The reason for the sub-linear speedup



(a) Parallel speedup



(b) Parallel efficiency.

Figure 5.10: K-means scalability from 1 to 12 cores, showing a linear speedup. Each configuration (red dot) represents the minimum value from 100 runs each.

is explained in the next section, and is related to the serial fraction of the algorithm.

5.3.2 Scalability

We consider scalability as the algorithm's capability of handling increased amounts of data. This is tightly related to complexity, as linear scalability implies a complexity of $\mathcal{O}(n)$. With large datasets scalability is important. The datasets used here are often much bigger than those used in the published articles, where the number of members rarely exceed 4000. In comparison, the 180501 dataset contains 34 403 spikes, and larger recordings exist. The feature extraction scales linearly with the number of spikes, and does not take a significant amount of time, so these are not detailed here. An analysis of k-means and SPC follows.

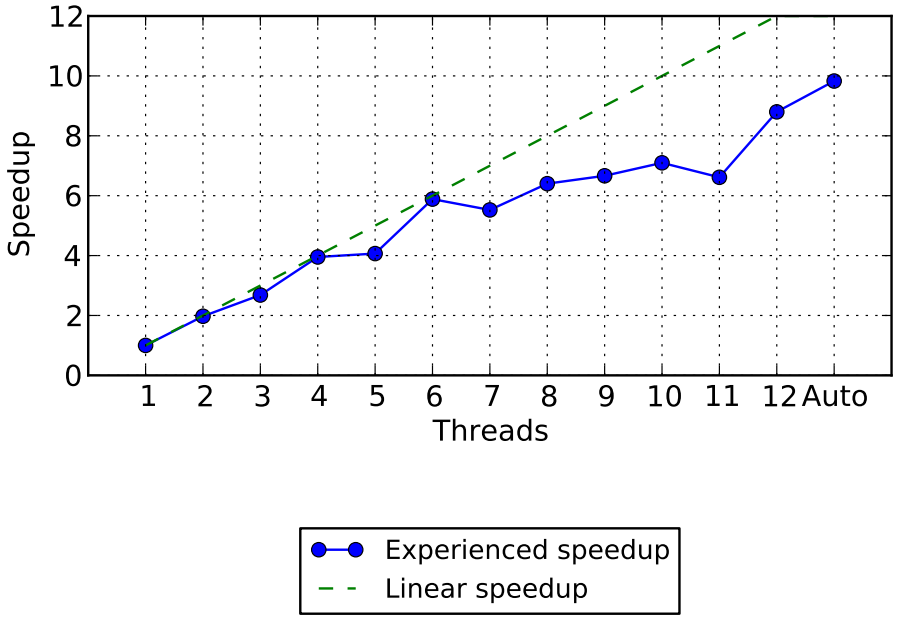
K-means

To see how the k-means algorithm scales in practice, it was run on the KI datasets. First, we modified the datasets so that the only stop-criteria was the number of iterations performed. This was done because a higher dimensional dataset might still converge in less iterations with the normal stop criterion. We benchmarked the 180501 dataset on 50, 100 and 200 dimensions. From our earlier analysis we know that the iteration itself should scale linearly, and this was confirmed with an almost perfect speedup; each increase added 1.96 runtime. The small overhead comes from the centroid initialization, which is serial.

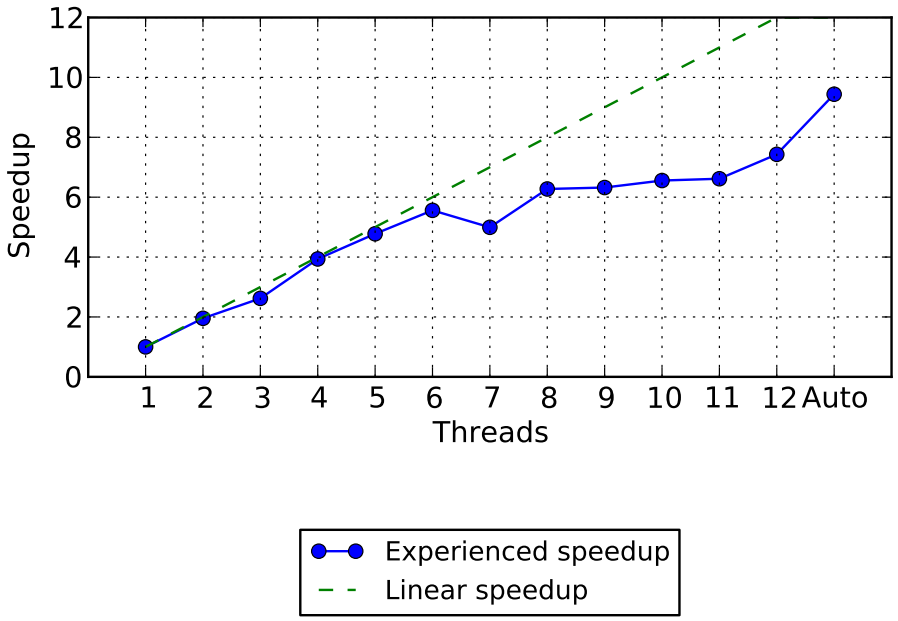
To see how convergence time is affected by the number of iterations, we ran the algorithm on all of our datasets with 10, 50, 100 and 200 dimensions. Each configuration was run 100 times, and then averaged. The sublinear scalability is reported in Table 5.5. We here use the term *slowdown*, which intuitively is the opposite of speedup.

Table 5.5: K-means dimensional scalability

Dimensions	**Slowdown**	**Stddev**
10	1	0
50	3.8	0.73
100	6.5	1.32
200	10.9	3.49



(a) Albert 5: 7 385 spikes.



(b) 180501: 34 403 spikes.

Figure 5.11: Measured Speedup for SPC on Kongull for different problem sizes. 10 components of Haar Wavelet KS reduction was used, and MST was not calculated.

Note that even with 200 dimensions and our biggest dataset, the classification is still done within seconds.

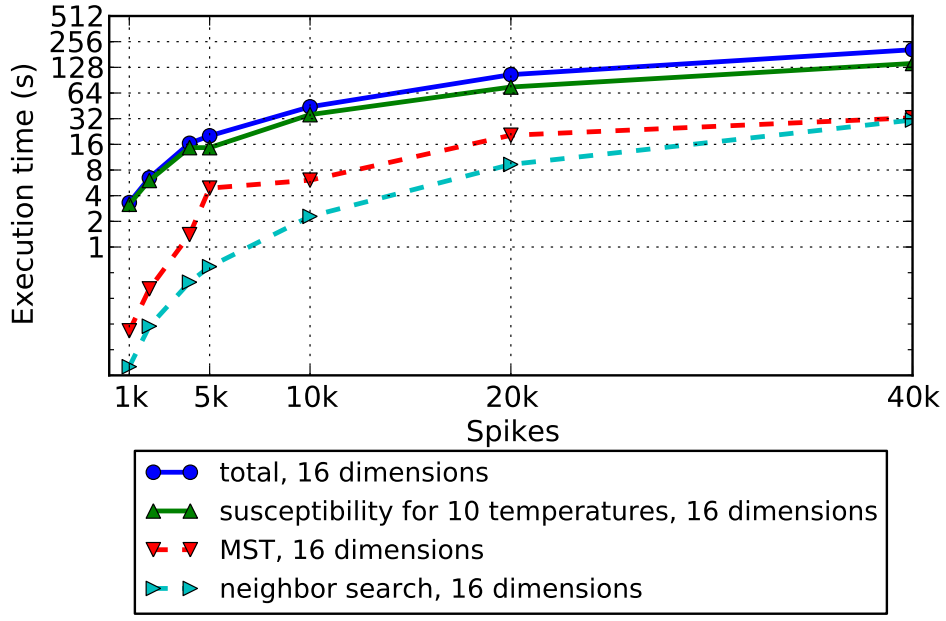
SPC

Figure 5.12 and Figure 5.13 illustrate performance for different input sizes to the SPC algorithm, when run on the desktop computer. Neighbor search and MST belong to the initial part of the algorithm, and should give deterministic results. These are costly calculations, and are the limiting factors.

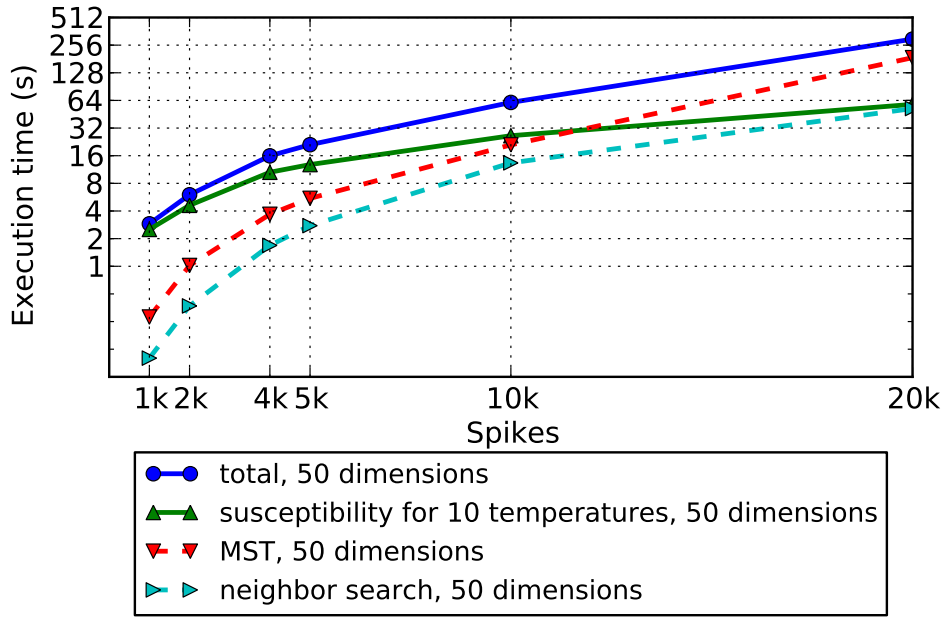
Finding nearest neighbors in high-dimensional Euclidean space is demanding. The performance seems to scale as badly as, or worse than, calculating the distance between all pairs of points, and then sorting these lists of distances. This gives a complexity of $\mathcal{O}(n^2)$. With 200 dimensions, the time for performing neighbor search was measured to increase with a factor of 4.08 when increasing from 10 000 to 20 000 spikes. MST increased with a factor of 7.47 in the same interval, giving an unacceptable scaling of close to $\mathcal{O}(n^3)$. For 50 dimensions, the growth was measured to 4.0 and 8.9 for neighbor search and MST respectively. For 16 dimensions, we also measured for a dataset of 40 000 spikes, and experienced a growth of 3.33 and 1.60 for neighbor search and MST when moving from 20 000 to 40 000 spikes. Note that total time in Figure 5.12(a) includes the wavelet transform from 200 to 16 dimensions, which is not included as a separate curve. Here, MST seems to grow slower than the neighbor search, which could indicate that neighbor searches could be sped up by using MST as basis. It should, however, be noted that the MST library (STANN) is written as parallel code, while the library used for neighbor searching (libANN) is not.

As the MST is an optional part of the algorithm, to ensure all points may be clustered together, it is clear that it cannot be used for larger inputs, or for high dimensions. However, for 16 dimensions, it is not the dominating part, and is included.

The susceptibility plot is not dependent on the number of dimensions, which is also evident from the figures, and scales better with larger inputs. In these runs, we perform clustering for ten temperatures. Calculated with the values from the run with 200 dimensions, it grows with a factor of 2.37 when increasing the number of spikes from 10 000 to 20 000, which is nearly linear.

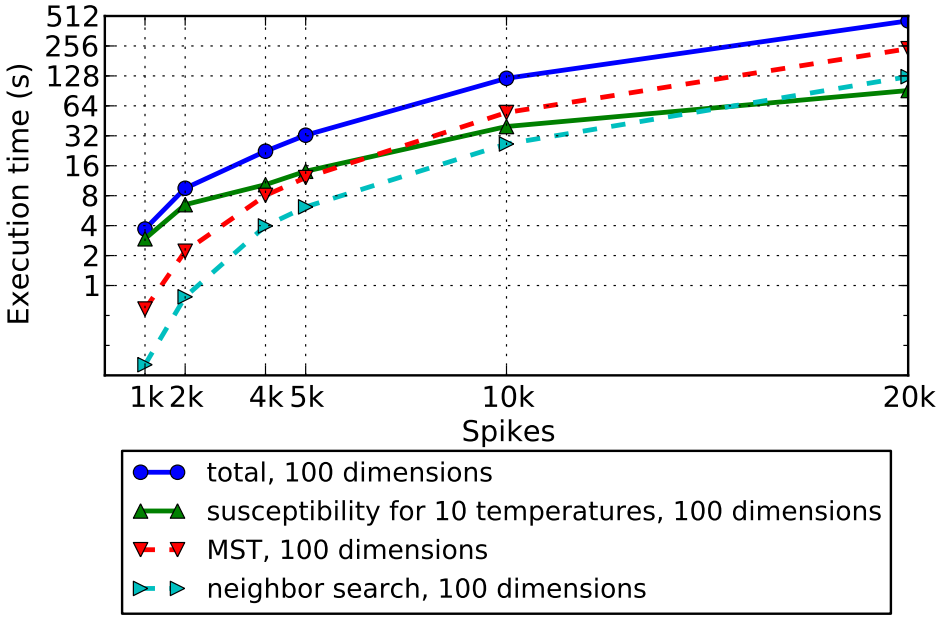


(a) 16 dimensions

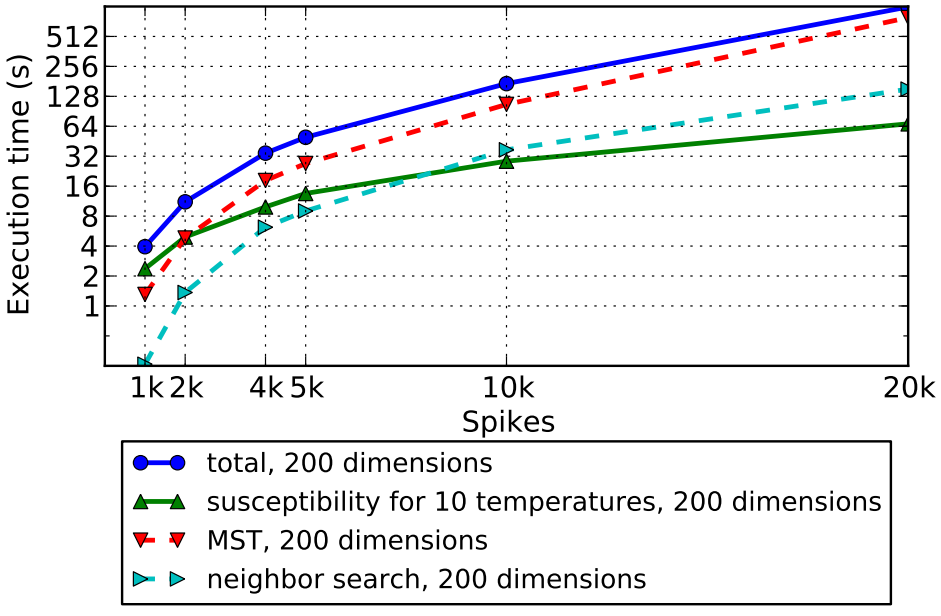


(b) 50 dimensions

Figure 5.12: Scalability of SPC for different dimensions and spike counts



(a) 100 dimensions



(b) 200 dimensions

Figure 5.13: Scalability of SPC for different dimensions and spike counts

5.4 Implementation challenges

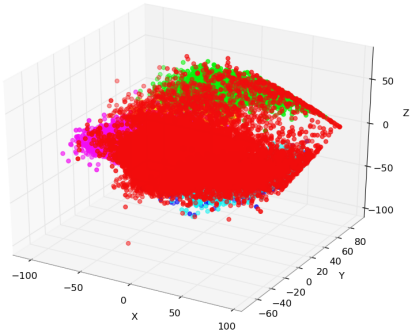
Originally we planned on including an additional advanced clustering algorithm in the program. However, the implementation of the super-paramagnetic clustering (SPC) algorithm posed problems and ended up taking a lot of time for our project. This was partly due to ambiguities in the articles describing the algorithm, but also due to the sheer effort needed to make it run fast and parallel. These ambiguities were attributed to definitions of concepts such as "average neighbor distance" without specifying whether it is average for each point's neighbors, or the global average. The required number of SW-iterations was not specified, and we got the impression that as few as 200 would be sufficient to produce a smooth susceptibility plot. Only when using as many as 10 000 did we get a fairly smooth graph, which looked similar to the ones in the article. If the authors had included a listing of pseudo code, instead of mixing the description of the algorithm with text explaining the background, there would likely be less ambiguities.

5.4.1 Noise

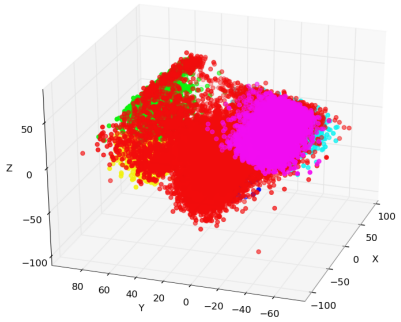
We first learned about the high noise levels in the datasets at a late stage of the project. After a meeting with Tsao [69], we realized that cluster zero from the manual clusterings were always a so called noise cluster – spikes to be discarded. This was a big surprise to us, and meant that most of the dataset was in fact noise, if the manual clustering is to be trusted.

However, a close inspection of the spikes in this cluster tells us that many spikes here are spikes that probably should not be classified as noise – their characteristic signature is similar to many classified spikes. It is likely that the easy and interesting cells have been isolated, and the rest ignored. This makes it harder to use unsupervised algorithms, as defining what is noise, and what is an interesting signal, is based on subjective measures.

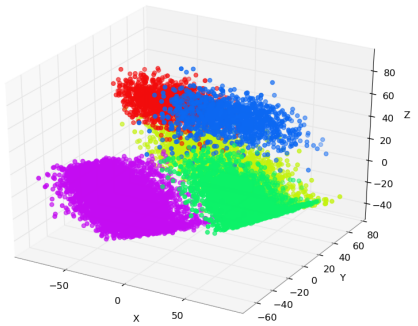
The cut files used to assess what is a correct clustering have been manually classified, with the help of the peaks-of-channels reduction. With four channels, this reduces the dataset to four dimensions, which cannot be directly displayed. To be able to visualize this, we used a PCA reduction from four to three dimensions, and plotted the 3D representation. Figure 5.14 on the following page demonstrates how noise in one dataset makes it harder to discern between clusters, as the "moats" between clusters become filled with noise.



(a) Cutfile classification (red is noise)



(b) Same as (a), rotated



(c) Cutfile classification, with noise removed

Figure 5.14: Visualization of how the noise level in datasets makes clustering harder. Here, 180501 (see Section 3.1), in which 52% of the spikes are considered noise by the cut file. The visualizations were made by extracting peaks of channels (four dimensions) reduced to three dimensions using PCA.

CHAPTER 6

Conclusions and future work

In this chapter, we conclude our findings throughout the project. We first describe how we have met our goals stated in Section 1.2, and then describe what can be done in the future, to improve on our work.

6.1 Goals

Here, we describe how we have accomplished our different goals.

G1: Graphical application

With the help of Qt, we have developed a graphical front-end to our application, *Paraspikes*, which lets the user experiment with the different feature extractions and clustering algorithms. Much effort has been put into making it portable, stable, and user friendly. A series of graphical representations of results are available, and we export results in a format which may be read by Tint, the cluster cutting software used at the KI. This application will be helpful for domain experts who want to assess the performance and quality of the algorithm, and may easily be extended with other algorithms.

G2, G3: Introduction and background

We have introduced the problem domain, both with regards to computer scientists unfamiliar with neuroscience, and for neuroscientists unfamiliar with computer science. A major part of the report is also spent describing the algorithms and other available applications in the field of spike sorting.

G4: Parallel implementation of selected algorithms

We have taken care to parallelize compute-heavy parts of the application as much as possible. This has been realized with the use of both Intel Threading Building Blocks and OpenMP. Measurements of parallel speedup have been performed, and we see a speed-up for both SPC and k-means when increasing the number of processors. K-means achieves a linear speedup, while SPC has a sub-linear speedup, with a parallel efficiency of around 80%. The steps needed to improve the efficiency are mentioned in Section 6.2. We are not aware of any other parallel implementations of SPC.

The results from the algorithms have also been evaluated, and we see that in general we can not rely on the clusterings performed by humans as a "gold standard". Some of these appear to have only a few easily identifiable neurons isolated, with the rest being labeled as noise. It is not only hard, but also not worthwhile to try and replicate these results. The peaks of channel reduction reduces 200 dimensions down to a simple few, and we believe automated tools which can perform clustering in a higher dimensional space can do better.

We only late in the project found out that the Axona recording system probably does not perform a good noise removal. In general, SPC appears to make solid clusters, but it could benefit from noise reduction. K-means is very fast, but since it does not handle noise at all, it should be paired with a good noise reduction scheme.

G5, G6: Time constraints and result quality

We have evaluated the execution time for the possible sequences of algorithms. For most datasets, the algorithms use less than the arbitrary limit we set to one minute. The most compute heavy feature extraction, PCA, spends 20 seconds to reduce the biggest dataset on our old workstation computer, well below the limit. K-means always spends less than 5 seconds, meaning that we with this algorithm easily fulfill the time constraint.

SPC requires much more work, and its execution time depends on different parameters, such as the number of Monte Carlo iterations, and how many temperatures to evaluate. Depending on which reduction is used, as well as the size of the datasets, the algorithm might take on the order of seconds to many minutes to complete. With the time measurements performed in Section 5.3, we see that the algorithm scales well with a low number of dimensions. However, for an input size of 40 000, even with few dimensions, the algorithm takes several minutes, and does not meet our time constraint.

The quality of the clustering results has been hard to determine, as the cut file results we have used for comparison, can not be viewed as a perfect clustering. This was also confirmed by Lisa Giocomo at the end of the thesis, who said that the neurons which were hard to isolate are usually just labeled as noise. To visualize the signals for an operator, a significant dimensional reduction must be performed. Not only does this introduce human bias, but this reduction could also discard important information required to properly separate the spikes. Automatic clustering algorithms may perform the classification in a high dimensional space, and may therefore use more information to discriminate between the different clusters.

6.2 Future work

To make sure SPC can scale better when increasing the number of available processor cores, it is necessary to parallelize the final part of the application. This is the neighbor search in the first part of the algorithm, which utilizes the `libANN` library. `libANN` is not thread safe, and can therefore not be used by multiple threads at the same time. As the library is open-source, anyone may take on this task, which will be helpful for all future users of the library. Our inspection of the source code suggests that it should be possible, as the global state which is retained seems to be caused by convenience in programming, rather than the algorithm itself. With this in place, SPC should scale nearly linearly when increasing available processor cores.

Random number generation is what takes the most time in SPC. An important optimization would therefore be to increase the performance of random number generation. We have used the quickest algorithm available in the open-source Boost Random library. More effort could be put into comparing its performance with vendor libraries such as Intel Math Kernel Library or AMD Core Math Library. These also provide functions for generating vectors of random numbers, which take advantage of vector parallelism on the processor.

Calculating the neighborhood graph and MST for SPC can consume a significant part of the total running time. The result of this could be stored the first time it is run, and simply loaded for subsequent uses. For our own purposes, this would have been a useful feature to have, since we analyzed the same datasets many times.

A next logical step should be a closer look at noise removal, although this does not necessarily have to be integrated into the program itself, and can

be done separately, before importing it into Paraspikes.

It would be interesting to see how large the noise fraction really is, by obtaining intracellular data. However, by inserting the probes necessary to record this data, the extracellular recordings can become affected, so it is difficult to determine exactly how much really is noise. We have argued that the manually cut files are biased because of the spike representation. Therefore, it would be helpful to compare the clustering algorithms to cut files based on intracellular recordings. More ways to evaluate the clusterings and reductions should also help to identify noise.

As for the GUI, being able to manually perform operations on the datasets such as merging and creating clusters could be a desirable feature. However, there are other applications which already support this. OpenElectrophy [27] seems have the most promising implementation, providing a user friendly interface for manually managing clusters.

We earlier mentioned that if our scope had been broader, we would have put effort into integrating our algorithms as modules in existing open-source applications, rather than developing our own application from scratch. It could probably also mean less work, as APIs are defined up-front, and basic application components are already available. Integrating the algorithms as part of existing software means that one could quickly get many users, who can provide valuable feedback, as well as contributing to improving the algorithm. For someone taking on a similar task in the future, we would recommend this approach.

Finally, comparing our algorithms to other clustering algorithms should be very interesting. SPC is a very capable algorithm, but is not very quick, and is still affected by noise.

References

[1] Axona – hardware and software for biomedical research. URL http://www.axona.com/. Cited on 14. April 2011.

[2] Boost random reference. URL http://www.boost.org/doc/libs/1_46_1/doc/html/boost_random/reference.html. Cited on 1. June 2011.

[3] Cell broadband engine architecture and its first implementation. URL http://www.ibm.com/developerworks/power/library/pa-cellperf/. Cited on 3. May 2011.

[4] NVIDIA Fermi. URL http://www.nvidia.com/object/fermi_architecture.html. Cited on 3. May 2011.

[5] Intel Math Kernel Library. URL http://software.intel.com/en-us/articles/intel-mkl/.

[6] Wavelet transform. URL http://www.jpeg.org/.demo/FAQJpeg2k/wavelet-transform.htm. Cited on 7. February 2010.

[7] KlustaKwik. URL http://klustakwik.sourceforge.net/. Cited on 10. February 2011.

[8] Boost, . URL http://www.boost.org/. Cited on 28. March 2011.

[9] The GNU Scientific Library, . URL http://www.gnu.org/software/gsl. Cited on 7. February 2010.

[10] Qwt – Qt Widgets for Technical Applications, . URL http://qwt.sourceforge.net/. Cited on 14. April 2011.

[11] STANN, . URL https://sites.google.com/a/compgeom.com/stann/Home. Cited on 14. April 2011.

[12] Tilera: Tile-Gx processor family. URL http://www.tilera.com/products/processors/TILE-Gx_Family. Cited on 3. May 2011.

[13] Tint: cluster-cutting and place field analysis. URL http://www.axona.com/html/tint.html. Cited on 2. May 2011.

[14] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. URL http://doi.acm.org/10.1145/1465482.1465560.

[15] OpenMP ARB. OpenMP. URL http://openmp.org/wp/. Cited on 3. June 2011.

[16] Thomas Bergheim. Parallel Algorithms for Neuron Spike Sorting. TDT4590 - Complex Computer Systems, Specialization Project, December 2010.

[17] Marcelo Blatt, Shai Wiseman, and Eytan Domany. Superparamagnetic clustering of data. *Phys. Rev. Lett.*, 76(18):3251–3254, Apr 1996. URL http://dx.crossref.org/10.1103%2FPhysRevLett.76.3251.

[18] Marcelo Blatt, Shai Wiseman, and Eytan Domany. Data clustering using a model granular magnet. *Neural Computation*, 9(8):1805–1842, 1997. URL http://dx.doi.org/10.1162/neco.1997.9.8.1805.

[19] R.J. Brychta, S. Tuntrakool, M. Appalsamy, N.R. Keller, D. Robertson, R.G. Shiavi, and A. Diedrich. Wavelet methods for spike detection in mouse renal sympathetic nerve activity. *Biomedical Engineering, IEEE Transactions on*, 54(1):82–93, 2006. ISSN 0018-9294.

[20] György Buzsáki. Large-scale recording of neuronal ensembles. *Nature neuroscience*, 7(5):446–451, May 2004. ISSN 1097-6256. URL http://dx.doi.org/10.1038/nn1233.

[21] Mircea I. Chelaru and Mandar S. Jog. Spike source localization with tetrodes. *Journal of Neuroscience Methods*, 142(2):305–315, 2005. ISSN 0165-0270. URL http://dx.doi.org/10.1016/j.jneumeth.2004.09.004.

[22] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, and Nacho Navarro. Network of Excellence on High Performance and Embedded Architecture and Compilation THE HIPEAC VISION, 2010.

[23] R. A. Fischer. Iris data set, 1936. URL http://archive.ics.uci.edu/ml/datasets/Iris. Cited on 28. March 2011.

[24] C. Fraley and A.E. Raftery. How many clusters? Which clustering method? Answers via model-based cluster analysis. *The Computer Journal*, 41(8):578, 1998. ISSN 0010-4620.

[25] Rob de Ruyter van Steveninck Fred Rieke, David Warland and William Bialek. *Spikes: Exploring the Neural Code*. The MIT Presss, 1999. ISBN 978-0-262-68108-7.

[26] S.H. Fuller and L.I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31 –38, jan. 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[27] Samuel Garcia and Nicolas Fourcaud-Trocme. Openelectrophy: an electrophysiological data- and analysis-sharing framework. *Frontiers in Neuroinformatics*, 3(0), 2009. URL http://www.frontiersin.org/Journal/Abstract.aspx?s=752&name=neuroinformatics&ART_DOI=10.3389/neuro.11.014.2009.

[28] Google. Google Performance Tools. URL http://code.google.com/p/google-perftools/. Cited on 1. June 2011.

[29] A. Graps. An introduction to wavelets, May 2004. URL http://www.amara.com/IEEEwave/IEEEwavelet.html. Cited on 24. May 2011.

[30] Kenneth D. Harris, Darrell A. Henze, Jozsef Csicsvari, Hajime Hirase, and Gyorgy Buzsaki. Accuracy of Tetrode Spike Separation as Determined by Simultaneous Intracellular and Extracellular Measurements. *J Neurophysiol*, 84(1):401–414, July 2000. ISSN 0022-3077. URL http://jn.physiology.org/cgi/content/abstract/84/1/401.

[31] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):pp. 100–108, 1979. ISSN 00359254. URL http://www.jstor.org/stable/2346830.

[32] Lynn Hazan. Klusters. URL http://klusters.sourceforge.net/. Cited on 10. February 2011.

[33] Lynn Hazan, Michaël Zugaro, and György Buzsáki. Klusters, neuroscope, ndmanager: A free software suite for neurophysiological data processing and visualization. *Journal of Neuroscience Methods*, 155(2):

207–216, 2006. ISSN 0165-0270. URL http://dx.doi.org/10.1016/j.jneumeth.2006.01.017.

[34] Peter Hemmen. Task-based Programming on a 64-core Tilera CPU. TDT4590 - Complex Computer Systems, Specialization Project, December 2010.

[35] P.M. Horton, A.U. Nicol, K.M. Kendrick, and J.F Feng. Spike sorting based upon machine learning algorithms (SOMA). *Journal of Neuroscience Methods*, 160:52–68, 2006.

[36] E. Hulata, R. Segev, Y. Shapira, M. Benveniste, and E. Ben-Jacob. Detection and sorting of neural spikes using wavelet packets. *Physical Review Letters*, 85(21):4637–4640, 2000. ISSN 1079-7114.

[37] E. Hulata, R. Segev, and E. Ben-Jacob. A method for spike sorting and detection based on wavelet packets and Shannon's mutual information. *Journal of Neuroscience Methods*, 117(1):1–12, 2002. ISSN 0165-0270.

[38] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering: (extended abstract). pages 332–339, 1994. doi: http://doi.acm.org/10.1145/177424.178042. URL http://doi.acm.org/10.1145/177424.178042.

[39] Intel. Intel threading building blocks for open source. URL http://threadingbuildingblocks.org/. Cited on 1. February 2011.

[40] Intel. Intel threading building blocks tutorial, 2010. URL http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial.pdf. Retrieved 18. November 2010.

[41] Intel. Data decomposition: Sharing the love and the data, 2011. URL http://software.intel.com/en-us/articles/data-decomposition-sharing-the-love-and-the-data/. Retrieved 18. April 2011.

[42] Intel. Intel Threading Building Blocks Reference Manual, 2011. URL http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf. Retrieved 2. June 2011.

[43] S. Jan, F. Urs, J. David, and H. Andreas. Independent-component-analysis-based spike sorting algorithm for high-density microelectrode array data processing. In *Sensors, 2009 IEEE*, pages 384–386. IEEE, 2010.

[44] K.H. Kim and S.J. Kim. A wavelet-based method for action potential detection from extracellular neural signal recording with low signal-to-noise ratio. *Biomedical Engineering, IEEE Transactions on*, 50(8):999–1011, 2003. ISSN 0018-9294.

[45] Christof Koch and Gilles Laurent. Complexity and the nervous system. *Science*, 284(5411):96–98, 1999. doi: 10.1126/science.284.5411.96. URL `http://www.sciencemag.org/content/284/5411/96.abstract`.

[46] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9): 1464–1480, September 1990. ISSN 0018-9219. URL `http://dx.doi.org/10.1109/5.58325`.

[47] Juan Carlos Letelier and Pamela P. Weber. Spike sorting based on discrete wavelet transform coefficients. *Journal of Neuroscience Methods*, 101(2):93 – 106, 2000. ISSN 0165-0270. URL `http://www.sciencedirect.com/science/article/pii/S0165027000002508`.

[48] C. Meenderinck and B. Juurlink. (When) Will CMPs Hit the Power Wall? In *Euro-Par 2008 Workshops-Parallel Processing*, pages 184–193. Springer, 2009.

[49] Marina Meila and David Heckerman. An experimental comparison of model-based clustering methods. *Machine Learning*, 42(1/2):9–29, 2001.

[50] Sturla Molden. Quantatitive analyses of single units recorded from the hippocampus and entorhinal cortex of behaving rats. Paper IV, 2005.

[51] Andrew W. Moore. An intoductory tutorial on kd-trees. 1991.

[52] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. URL `http://dx.doi.org/10.1109/JPROC.1998.658762`.

[53] David M. Mount and Sunil Arya. Approximate nearest neighbors library, Jan 2010. URL `http://www.cs.umd.edu/~mount/ANN/`. Cited on 14. April 2011.

[54] Nokia. Qt 4. URL `http://qt.nokia.com/`. Cited on 14. April 2011.

[55] R. Quian Quiroga, Z. Nadasdy, and Y. Ben-Shaul. Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. *Neural Computation*, 16:1661–1687, August 2004. ISSN 0899-7667. URL http://dx.doi.org/10.1162/089976604774201631.

[56] Rodrigo Quian Quiroga. Wave_Clus – Unsupervised detection and sorting. URL http://www2.le.ac.uk/departments/engineering/research/bioengineering/neuroengineering-lab/spike-sorting. Cited on 10. February 2011.

[57] A. Roberts. How does a nervous system produce behaviour? A case study in neurobiology. *Science progress*, 74(293 Pt 1):31, 1990. ISSN 0036-8504.

[58] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987. ISSN 0377-0427. doi: DOI:10.1016/0377-0427(87)90125-7. URL http://www.sciencedirect.com/science/article/B6TYH-45GN65V-6/2/18986ecfab1157f7f05da39a3b08ea73.

[59] Ueli Rutishauser. OSort – an online spike sorting algorithm. URL http://www.urut.ch/new/serendipity/index.php?/pages/osort.html. Cited on 10. February 2011.

[60] Ueli Rutishauser, Erin M. Schuman, and Adam N. Mamelak. Online detection and sorting of extracellularly recorded action potentials in human medial temporal lobe recordings, in vivo. *Journal of Neuroscience Methods*, 154(1-2):204–224, 2006. ISSN 0165-0270. URL http://dx.doi.org/10.1016/j.jneumeth.2005.12.033.

[61] N. Schmitzer-Torbert, J. Jackson, D. Henze, K. Harris, and A.D. Redish. Quantitative measures of cluster quality for use in extracellular recordings. *Neuroscience*, 131(1):1 – 11, 2005. ISSN 0306-4522. URL http://www.sciencedirect.com/science/article/B6T0F-4F6SSGV-4/2/5782dbaefa4f0f146a187dd6be5355ee.

[62] B.M. Shahshahani and D.A. Landgrebe. The effect of unlabeled samples in reducing the small sample size problem and mitigating the Hughes phenomenon. *IEEE Transactions on Geoscience and Remote Sensing*, 32(5):1087–1095, 1994. ISSN 0196-2892.

[63] Dan Siroker and Steve Miller. Drawbacks of k-means. 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.3216.

[64] Lindsay I Smith. A tutorial on principal components analysis, February 2002. URL http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf. Cited on 24. May 2011.

[65] Robert Swendsen, Jian-Sheng Wang, and Alan Ferrenberg. New monte carlo methods for improved efficiency of computer simulations in statistical mechanics. In Kurt Binder, editor, *The Monte Carlo Method in Condensed Matter Physics*, volume 71 of *Topics in Applied Physics*, pages 75–91. Springer Berlin / Heidelberg, 1995.

[66] P.N. Tan, M. Steinbach, and V. Kumar. Cluster analysis: basic concepts and algorithms. *Introduction to Data Mining, Addison-Wensley*, 2006.

[67] Ariel Tankus, Yehezkel Yeshurun, and Itzhak Fried. An automatic measure for classifying clusters of suspected spikes into single cells versus multiunits. *Journal of Neural Engineering*, 6(5):056001, 2009. URL http://stacks.iop.org/1741-2552/6/i=5/a=056001.

[68] L. Traver, C. Tarin, P. Marti, and N. Cardona. Adaptive-threshold neural spike detection by noise-envelope tracking. *Electronics Letters*, 43(24):1333–1335, 22 2007. ISSN 0013-5194. URL http://dx.doi.org/10.1049/el:20071631.

[69] Albert Tsao. PhD candidate, Kavli Institute for Systems Neuroscience and Centre for the Biology of Memory. Personal communication.

[70] C. Valens. A really friendly guide to wavelets. URL http://perso.wanadoo.fr/polyvalens/clemens/wavelets/wavelets.html. Cited on 1. February 2011.

[71] Dimitrios Ververidis and Constantine Kotropoulos. Information loss of the mahalanobis distance in high dimensions: Application to feature selection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:2275–2281, 2009. ISSN 0162-8828. URL http://doi.ieeecomputersociety.org/10.1109/TPAMI.2009.84.

[72] Filip Wasilewski. Wavelet browser. URL http://wavelets.pybytes.com/. Cited on 24. May 2011.

[73] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0131405632.

[74] F. Y. Wu. The potts model. *Rev. Mod. Phys.*, 54(1):235–268, Jan 1982. doi: 10.1103/RevModPhys.54.235.

Appendices

APPENDIX A

Implementation

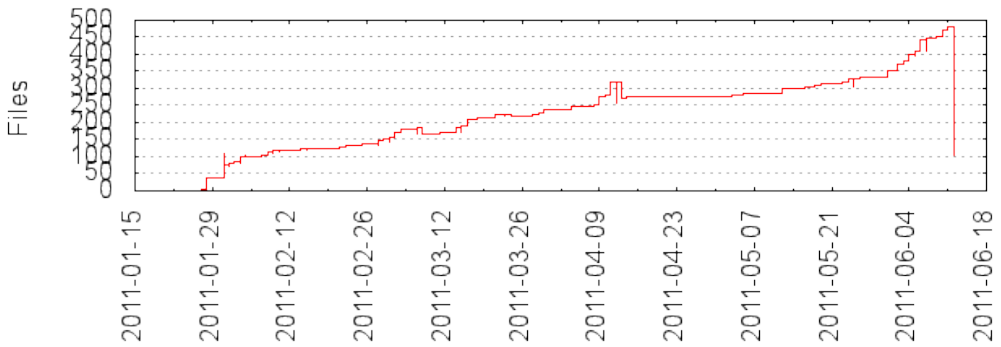


Figure A.1: Number of files by date. This includes the datasets.

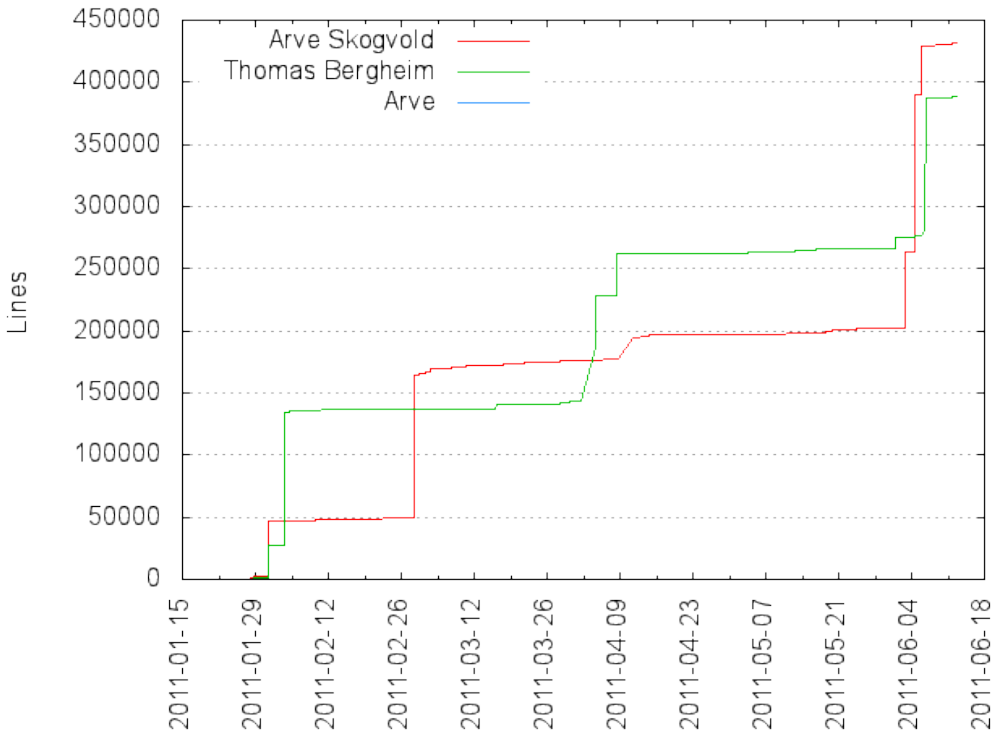


Figure A.2: Number of lines of code by author. This includes the datasets.

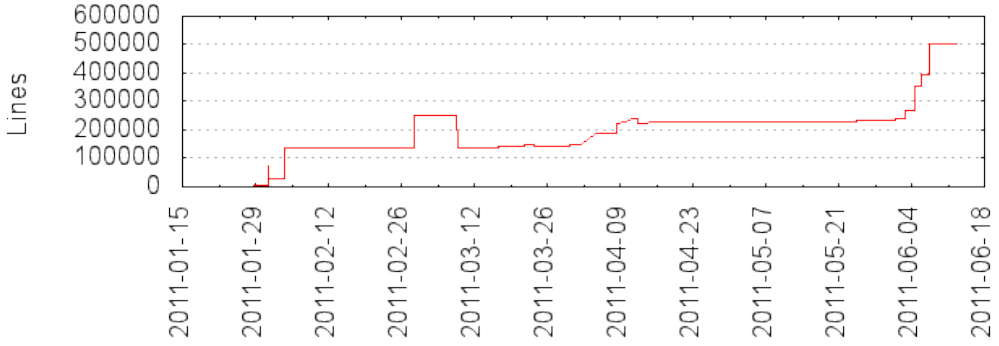


Figure A.3: Number of lines of code. This includes the datasets.

APPENDIX B

Results

B.1 Cluster quality

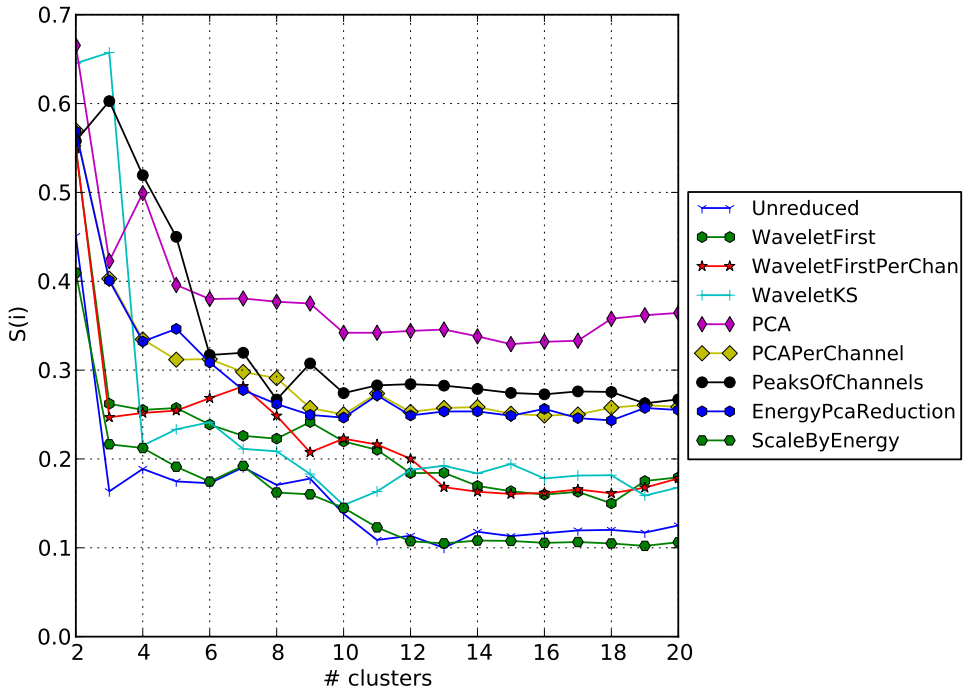


Figure B.1: The silhouette coefficient for all the reductions on the Albert 3 dataset. Notice that there is no clear trend among the reductions. However, PCA suggests 4 clusters, while the peaks of channels and unreduced suggest 3. Tsao [69] identified 3 neurons in this set, in addition to the noise cluster

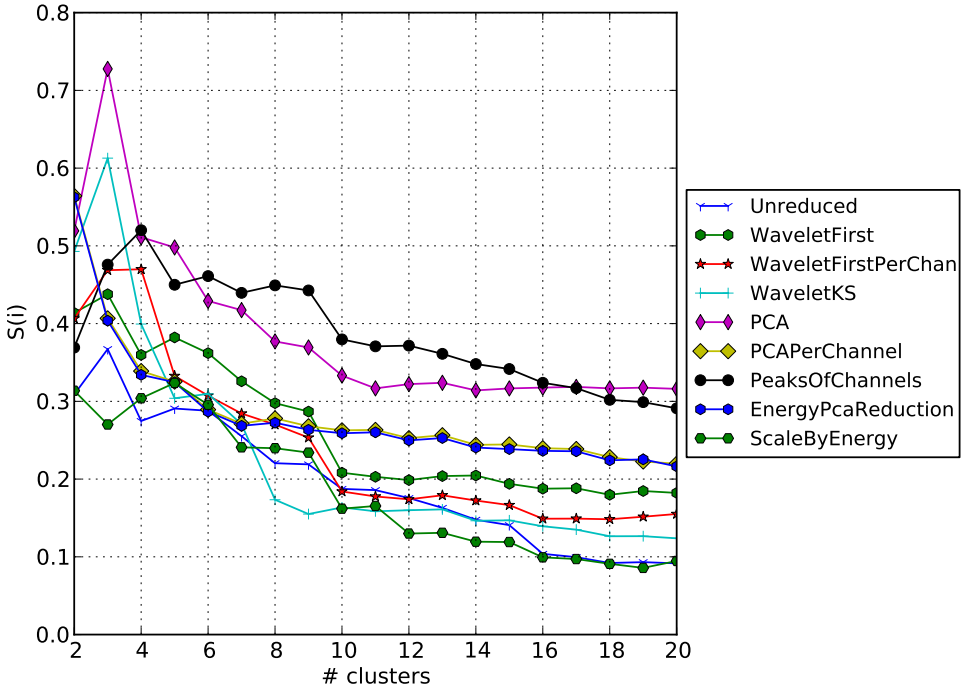


Figure B.2: The silhouette coefficient for all the reductions on the Albert 4 dataset. Here, some reductions, notably PCA and Wavelet KS suggests 3 clusters. The peaks of channels here suggests 4 clusters. Tsao [69] identified 6 neurons in this set, in addition to the noise cluster

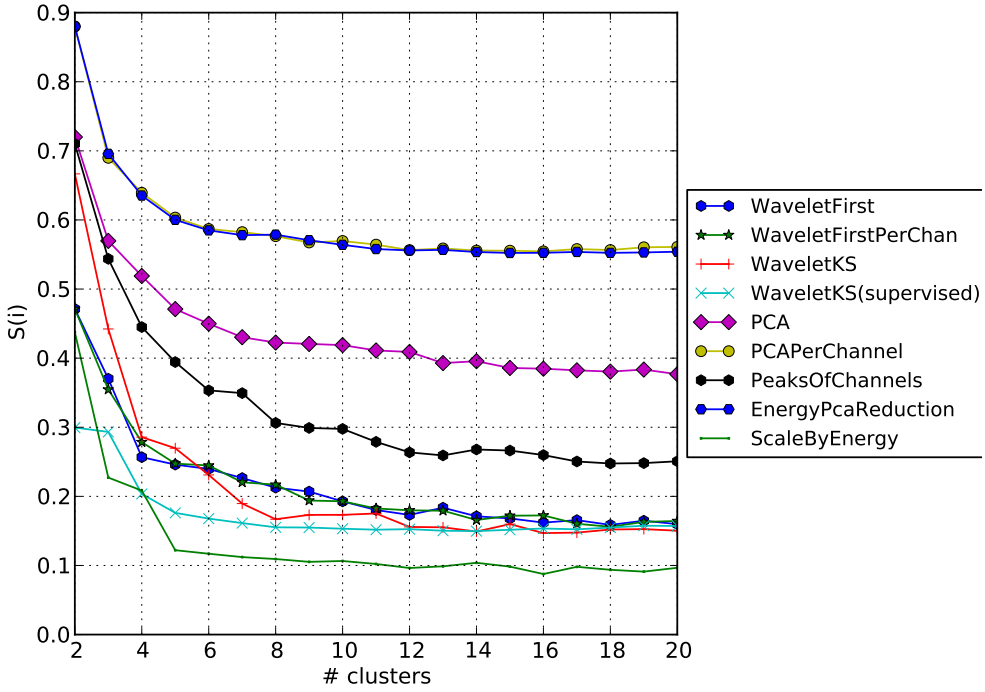


Figure B.3: The silhouette coefficient on the Albert 2 dataset, with the noise cluster removed. There is no clear indication of the number of clusters. According to the cut files, there should be 3 clusters.

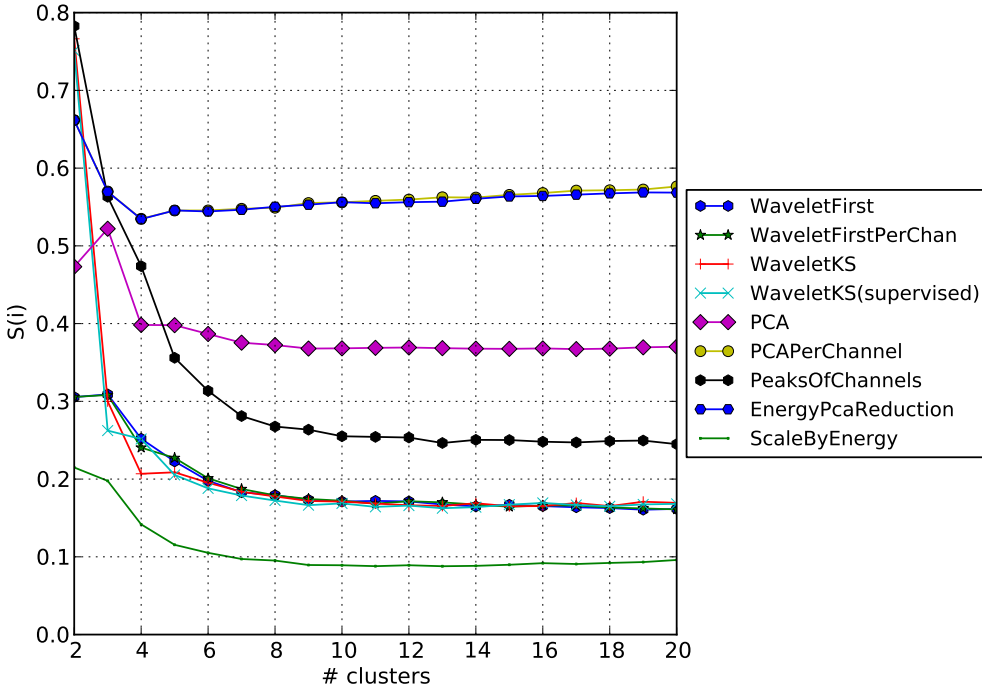


Figure B.4: The silhouette coefficient on the Albert 3 set, with the noise cluster removed. It is not clear how many clusters are best. PCA suggests $k = 3$, which according to the cut file is correct.

B.1.1 Sum of squares

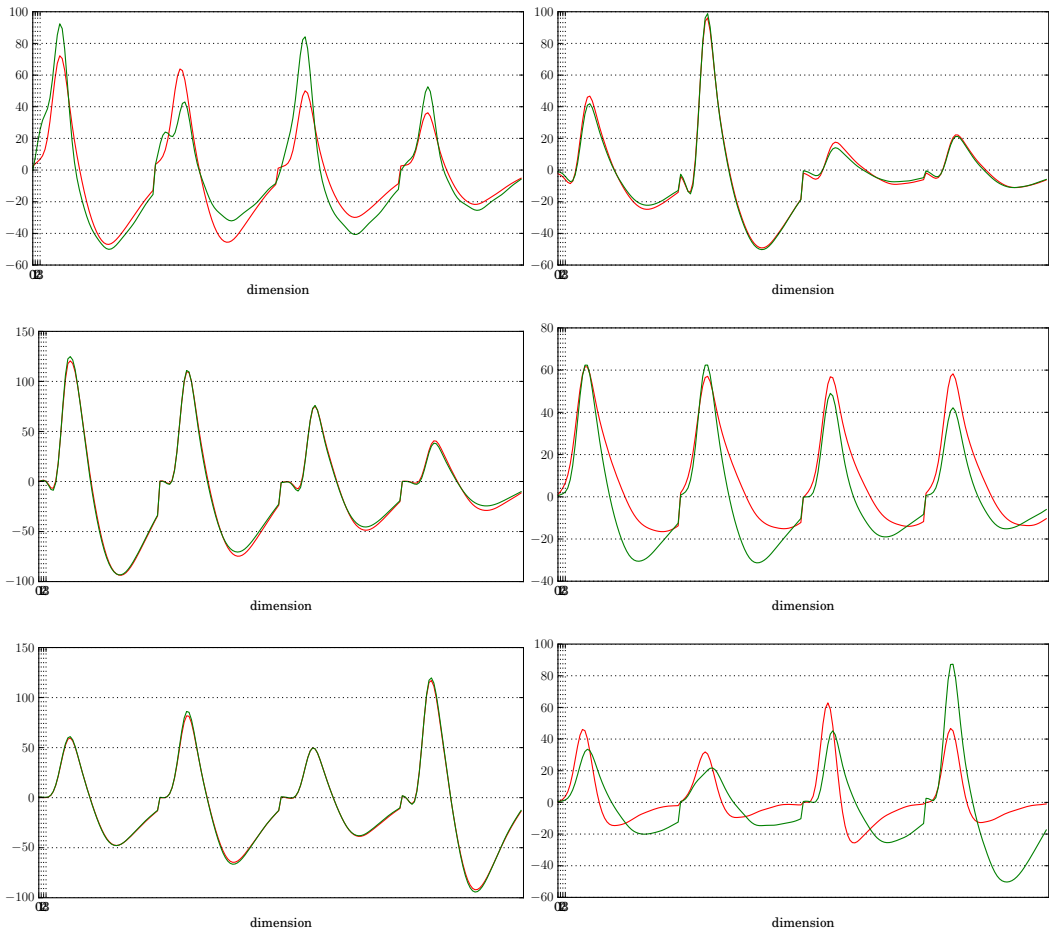


Figure B.5: A similarity plot between the unsupervised k-means solution (red) and the manually cut set (green). All 200 samples were used as a feature vector. Each plot represents a cluster.

B.1.2 Profiling results

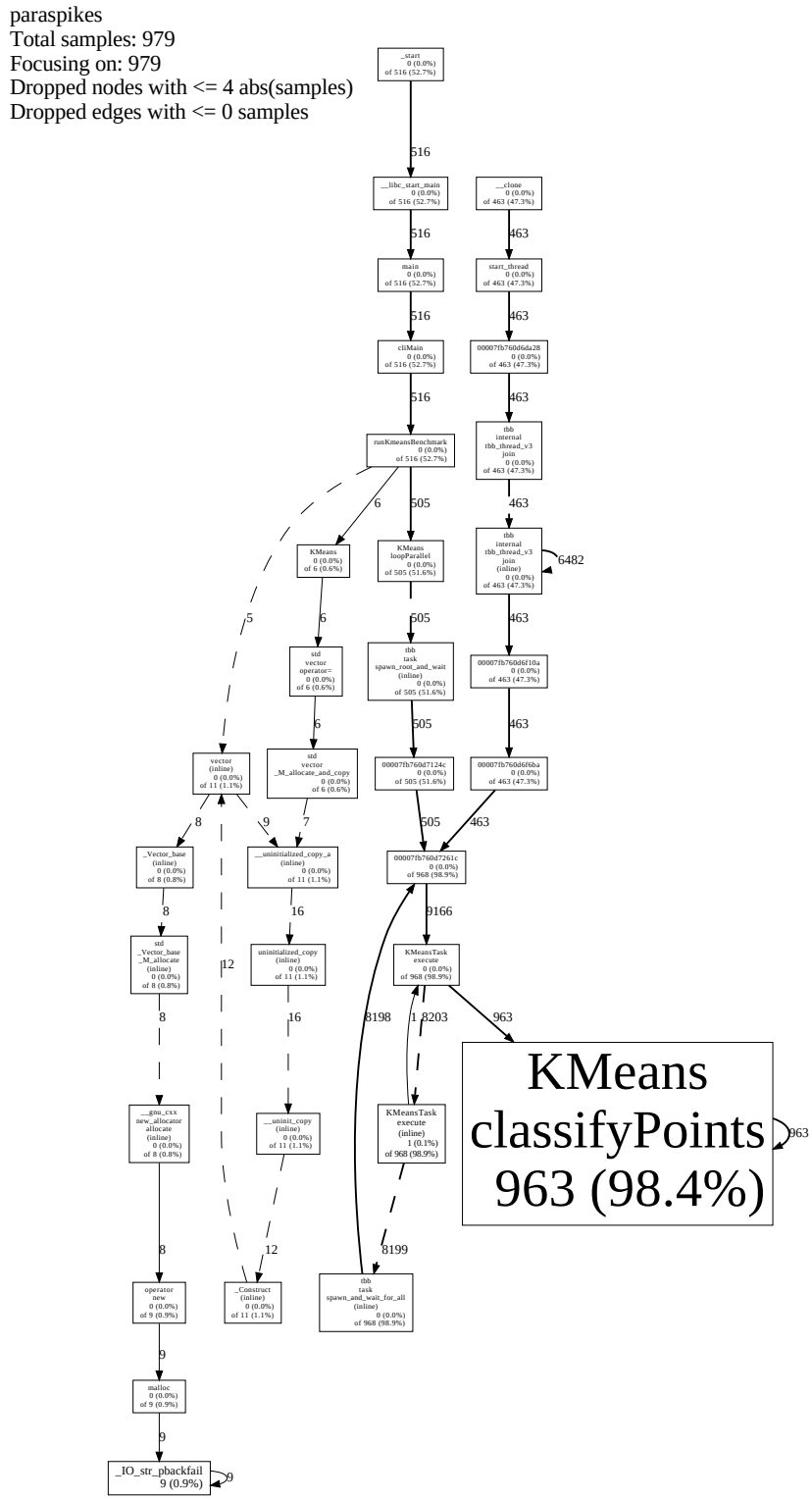


Figure B.6: K-means callgraph

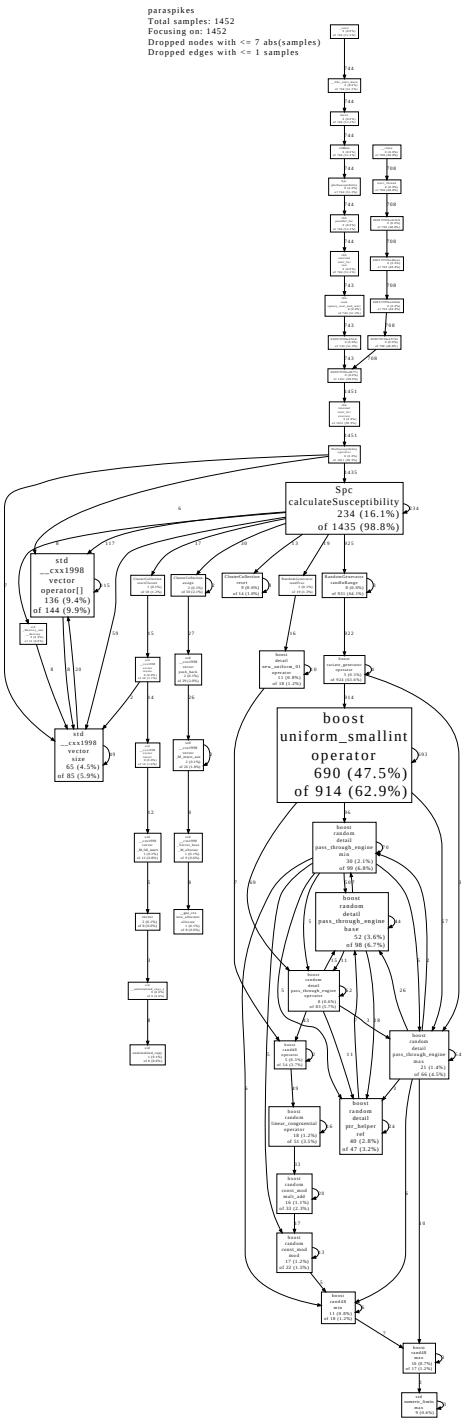


Figure B.7: SPC callgraph – inner part of the algorithm

APPENDIX C

Screenshots

This appendix contains screenshots from the different parts of the GUI application, as well as a listing of output from the command line interface.

Listing C.1: Command line interface: help screen

```
$ ./paraspikes -h
Paraspikes, built: Jun 10 2011 17:56:14
==========
Usage: ./paraspikes [options]
No options ==> GUI

-f [filename]  File to load
-g [filename]  Cutfile
-r [reducIdx]  Which reduction to use (see trailing list)
-c [clustAlgo] Which clustering algo to use
   SPC (s)
    -[ float   Left limit
    -] float   Right limit
    -_ float   Step
   K means (k)
    -n int     Centroids
    -P int     Cutoff
-a             ASCII input (default: Axona)
-b             Run benchmark suite
  -I           Num iterations
-S             Srand(0)
-C             Num cores enabled
-h             You just did

Reductions
0: No reduction
1: Wavelet transform (first)
2: Wavelet transform (first per channel)
3: Wavelet (KS)
4: Wavelet (KS supervised)
5: Wavelet transform (highest)
6: PCA
7: PCA per channel
8: Peaks of channels
9: Energy PCA
10: Scale by energy (no reduction)
```

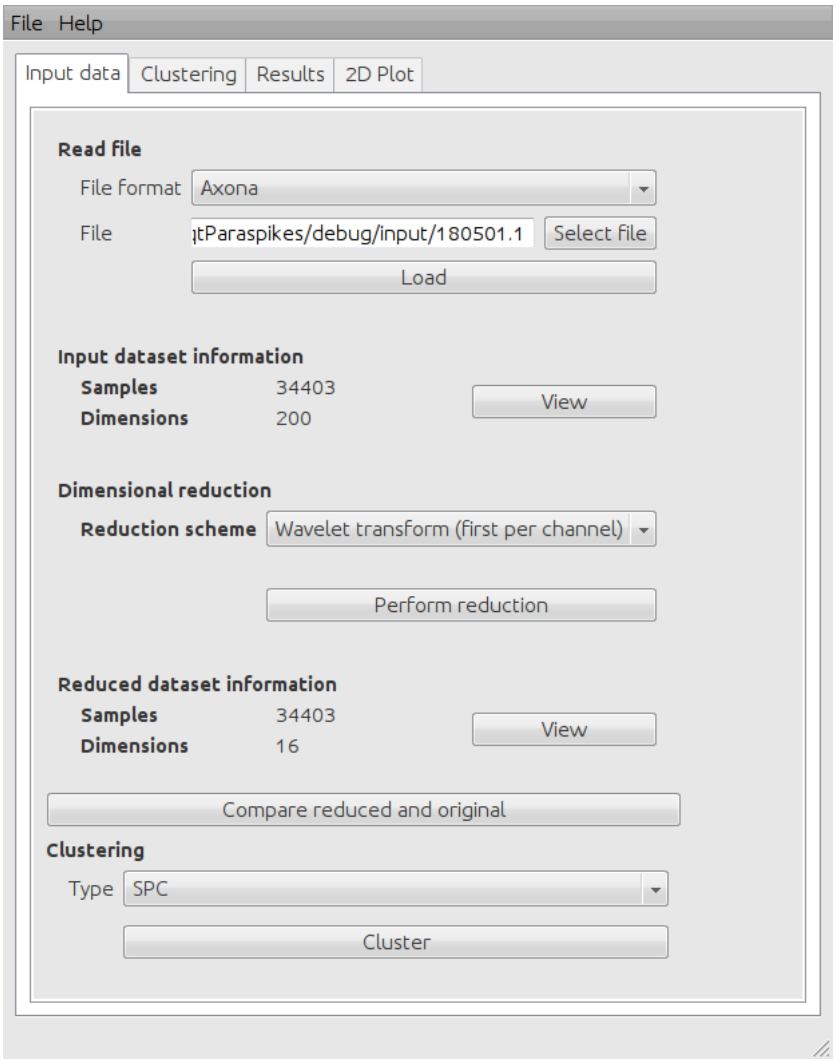


Figure C.1: GUI: Main window.

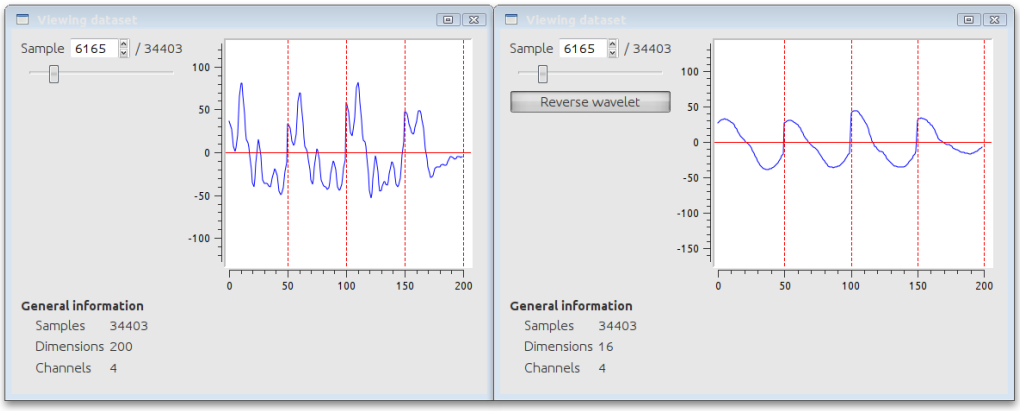


Figure C.2: GUI: Comparing original signal and reverse of wavelet transform.

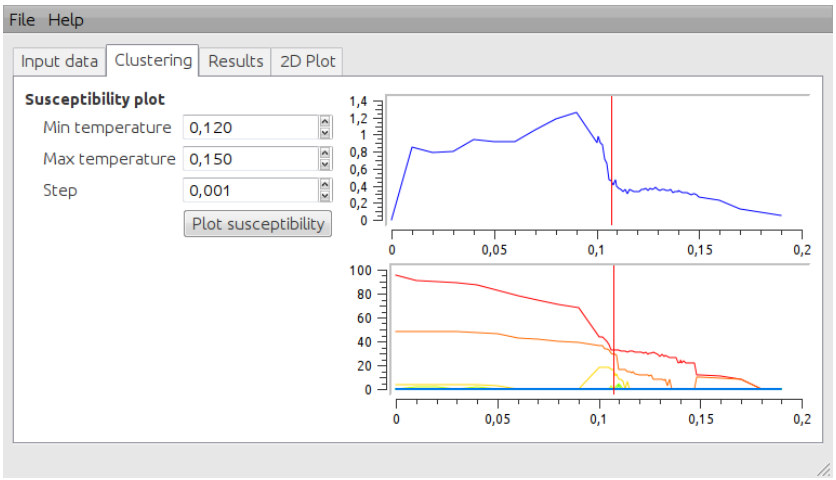


Figure C.3: GUI: SPC tab when run with Iris dataset.

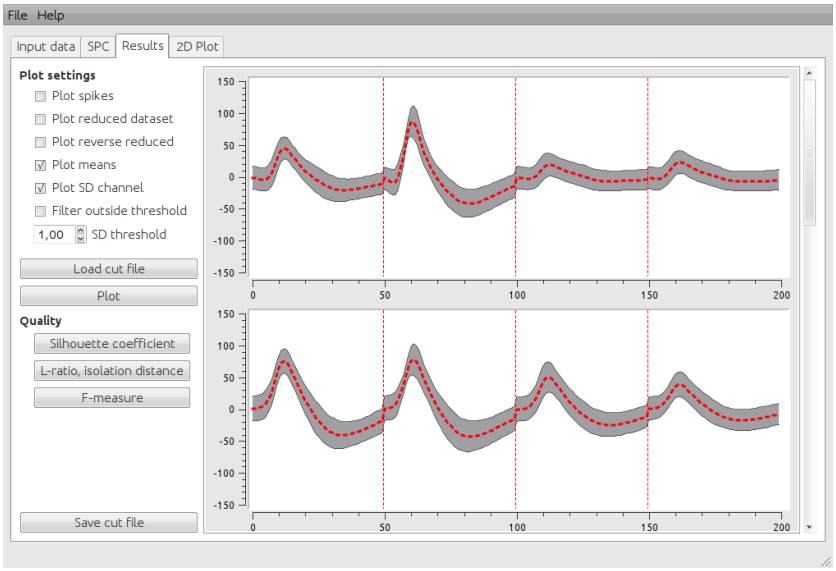


Figure C.4: GUI: Clustering results tab, displaying results for a clustering of the 180501 dataset.

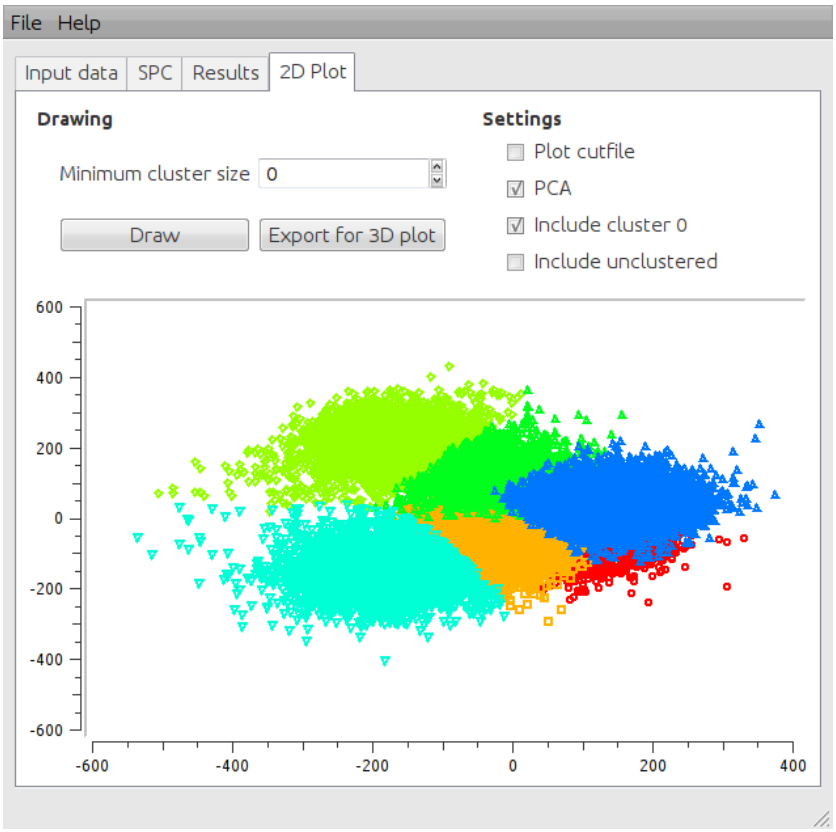


Figure C.5: GUI: 2D plot of a clustering performed on the 180501 dataset.

APPENDIX D

Source code

This appendix describes the structure of the source code which makes up our application. We have chosen only to include some excerpts from the code, as we think reading source code on paper provides little benefit.

Instead, we give a short guide on how to compile and execute our source code, so that the readers may try the application themselves. We also include a few class diagrams to show dependencies between classes in the code base.

The code is delivered in an archive together with the report, and is made available through DAIM[1].

D.1 Introduction to the code base

The libraries needed to execute our code have been chosen because of portability and being open-source software. Hence, one should be able to compile and run the program on both GNU/Linux, Windows and Mac OSX. We have, however, only tested the application on Linux.

The libraries which usually are not installed by default, are supplied in the archive submitted with the report. In addition, you may need to install Qt4, depending on whether your OS distribution already has this installed.

Compilation may be done using any newer C++ compiler, but we have only tested with GNU g++ 4.4.* and 4.5.*.

To build the application, extract the archive and go to the subfolder `libs`. Here, type `make` to compile the necessary libraries. Then, go to `qtParaspikes/qtParaspikes`, and type `qmake; make`. Depending on your Qt4 settings, you will find the executable in either `qtParaspikes/qtParaspikes/debug`

[1] `http://daim.idi.ntnu.no`

or `qtParaspikes/qtParaspikes-build-desktop/debug`. When in this folder, type `LD_LIBRARY_PATH=. ./paraspikes` to start the application. To use the command line interface, add `-h` as a parameter, to get information about usage.

D.2 Class diagrams

The figures in this section are class diagrams describing interesting parts of our application. We only include class methods, as including member variables would make very verbose figures. See the next section for listing of some source code, or see the previous section for information on how to obtain and use the complete source code.

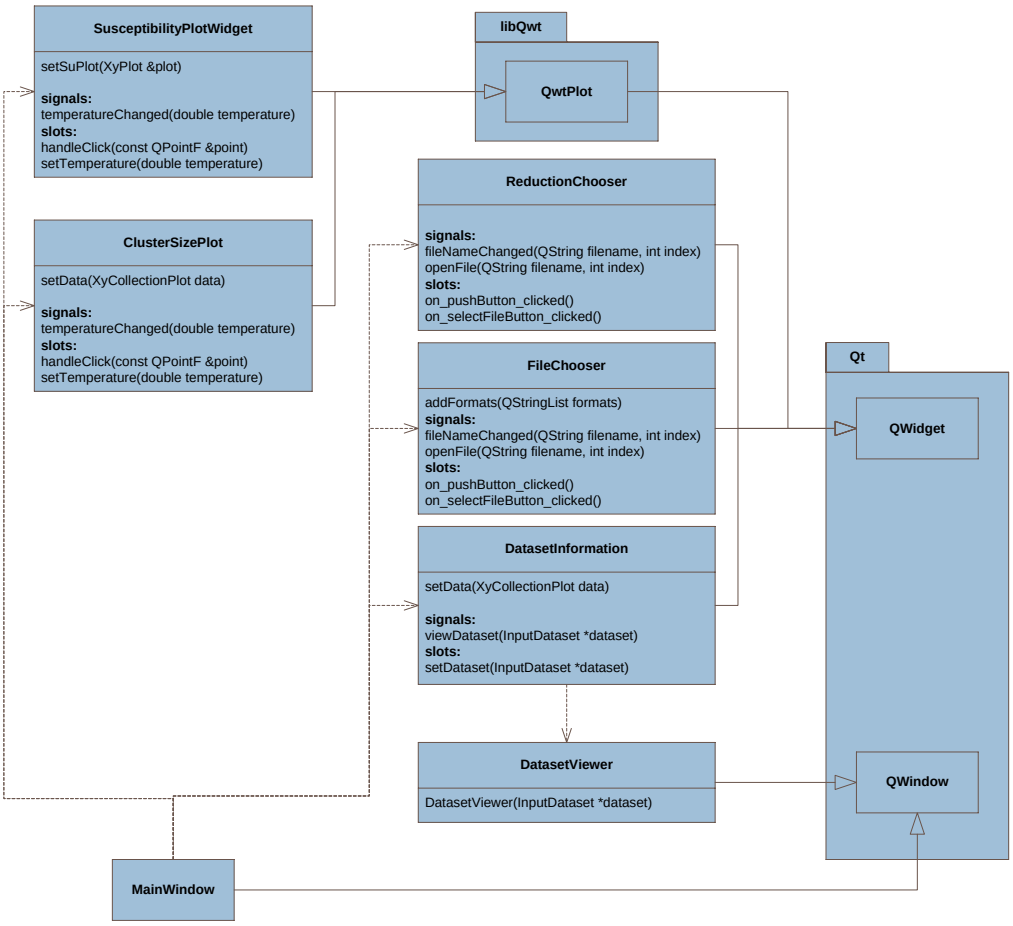


Figure D.1: Class diagram of GUI components

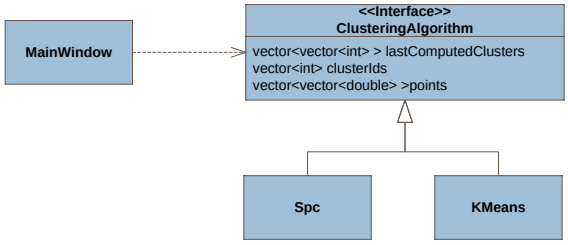


Figure D.2: Class diagram of clustering algorithms (overview)

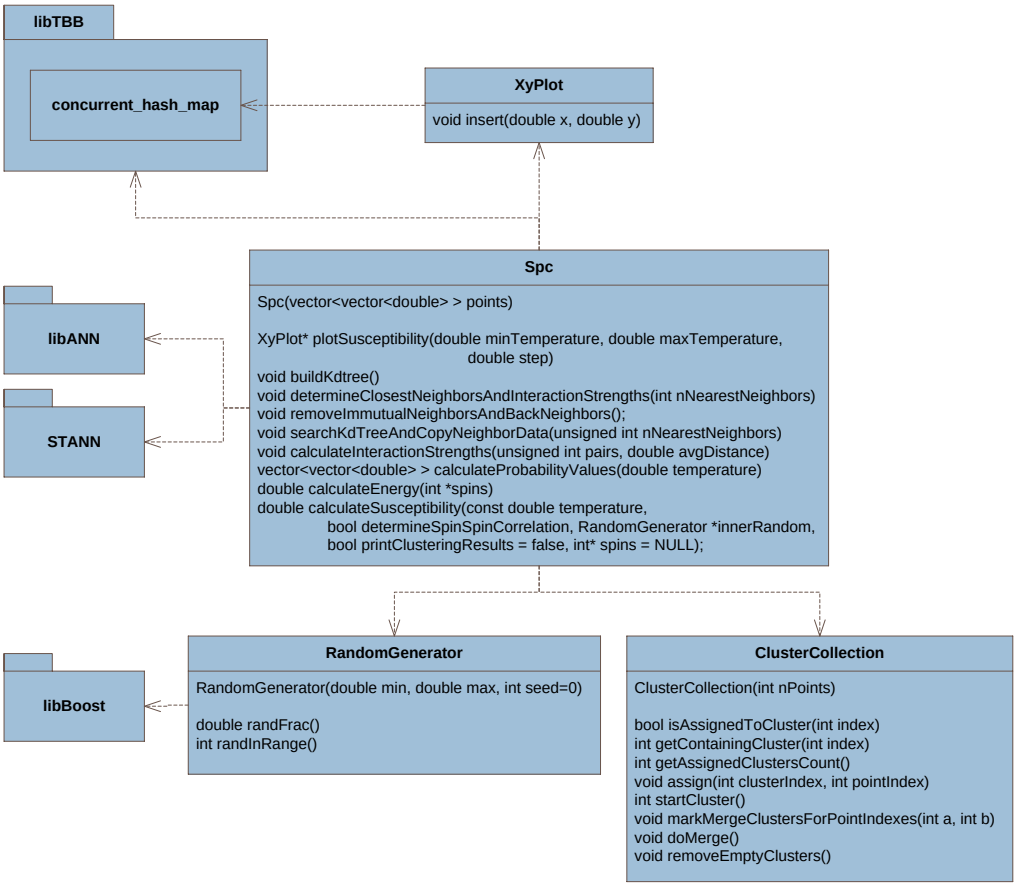


Figure D.3: Class diagram of SPC

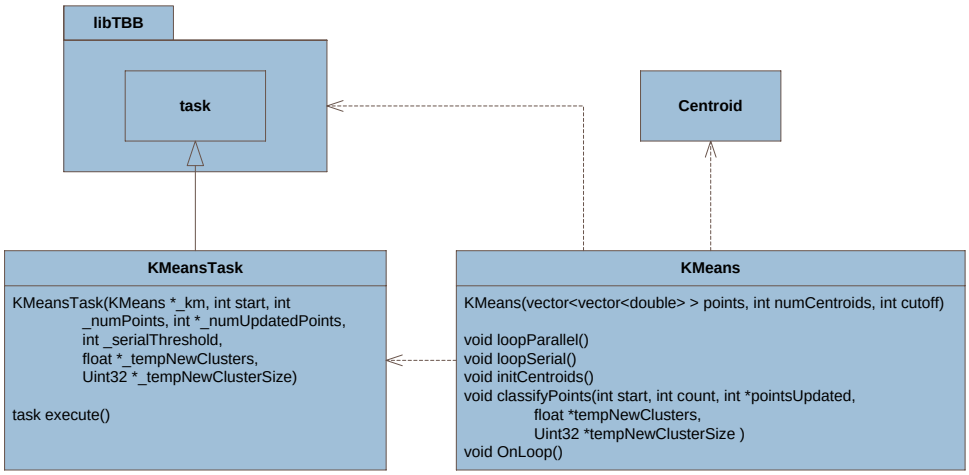


Figure D.4: Class diagram of k-means

D.3 Source code listings

Here follows a small subset of our source code, describing the most important parts of the application.

Listing D.1: The code that matches two clusterings to each other

```
1  bool unique_path(int *path, int v, int level)
2  {
3    for(int i = 0; i < level; ++i)
4      if(path[i] == v)
5        return false;
6
7    return true;
8  }
9
10 void compute_shortest_path_rec(float *t, int *path, float
       length, int *shortestPath, float *shortestLength, int level,
       int manualCutMaxLevel, int autoCutMaxLevel)
11 {
12   float localLength;
13
14   for(int a = 0; a < autoCutMaxLevel; ++a)  {
15     if(!unique_path(path, a, level))
16       continue;
```

```
17
18      int *localPath = new int[manualCutMaxLevel];
19      memcpy(localPath, path, sizeof(int) * manualCutMaxLevel);
20
21      localPath[level] = a;
22      localLength = length + t[localPath[level] +
            autoCutMaxLevel*level];
23
24      if(localLength > *shortestLength)
25        continue;
26
27      if(level == (manualCutMaxLevel-1)) {
28        if(localLength < *shortestLength) {
29          *shortestLength = localLength;
30          for(int i = 0; i < manualCutMaxLevel; ++i)
31            shortestPath[i] = localPath[i] + autoCutMaxLevel*i;
32        }
33      } else {
34        compute_shortest_path_rec(t, localPath, localLength,
              shortestPath, shortestLength, level+1,
              manualCutMaxLevel, autoCutMaxLevel);
35      }
36
37      delete localPath;
38    }
39  }
40
41  // prints manual X auto (row X col)
42  float similaritySumOfDifferences(double *manualCutData,
        unsigned int numManualCutData, double *autoCutData, unsigned
        int numAutoCutData, int numDimensions, std::vector<int> &out)
43  {
44    int numAutoCutDataZeroPadded = numAutoCutData;
45    if(numAutoCutData < numManualCutData)
46      numAutoCutDataZeroPadded = numManualCutData;
47
48    float *similarityMatrix = new float[numManualCutData *
          numAutoCutDataZeroPadded];
49    memset(similarityMatrix, 0, sizeof(float) * numManualCutData
          * numAutoCutDataZeroPadded);
50
51    std::cout << "Sum of absolute differences (cutter x kmeans in
          " << numManualCutData << " x " << numAutoCutData << "):"
          << std::endl;
52    std::cout << "\t";
53    for(int i = 0; i < numAutoCutDataZeroPadded; ++i)
54      std::cout << "    " << std::setw(10) <<
            std::setiosflags(std::ios::fixed) <<
            std::setprecision(2) << i;
```

```
55  std::cout << std::endl;
56
57  for(unsigned int a = 0; a < numManualCutData; ++a) {
58    std::cout << "\t" << a << ": ";
59    for(unsigned int b = 0; b < numAutoCutData; ++b) {
60      float diff = 0;
61      for(int dimension = 0; dimension < numDimensions;
            ++dimension) {
62        diff += fabs(manualCutData[a * numDimensions +
              dimension] - autoCutData[b * numDimensions +
              dimension]);
63      }
64      std::cout << "    " << std::setw(10) <<
            std::setiosflags(std::ios::fixed) <<
            std::setprecision(2) << (diff);
65      similarityMatrix[a * numAutoCutDataZeroPadded + b] = diff;
66    }
67    std::cout << std::endl;
68  }
69
70  int pathLength = numManualCutData;
71
72  std::cout << "pathlength: " << pathLength << std::endl;
73
74  int *path = new int[pathLength];
75  memset(path, -1, sizeof(int) * pathLength);
76  float length = FLT_MAX;
77
78  int *localPath = new int[pathLength];
79  memset(localPath, -1, sizeof(int) * pathLength);
80  float localLength = 0;
81  compute_shortest_path_rec(similarityMatrix, localPath,
        localLength, path, &length, 0, pathLength,
        numAutoCutDataZeroPadded);
82
83  std::cout << "Shortest path:\n";
84  float total_length = 0;
85
86  out.clear();
87  out.resize(pathLength);
88  for(int i = 0; i < pathLength; ++i) {
89    out[i] = path[i] % numAutoCutData;
90    std::cout << "(" << i << ", " << path[i] %
          numAutoCutDataZeroPadded << "): " <<
          similarityMatrix[path[i]] << std::endl;
91    total_length += similarityMatrix[path[i]];
92  }
93
94  std::cout << "Total length: " << total_length << std::endl;
```

```
95
96    delete[] similarityMatrix;
97    delete[] localPath;
98    delete[] path;
99
100   return total_length;
101 }
```

Listing D.2: The main parts of the k-means source code

```
1   int KMeans::loopSerial() {
2     int numUpdatedPoints = 0;
3
4     float *tempNewClusters = new float[numCentroids *
          nDimensions];
5     Uint32 *tempNewClusterSize = new Uint32[numCentroids];
6     memset(tempNewClusters, 0, sizeof(float) * numCentroids *
          nDimensions);
7     memset(tempNewClusterSize, 0, sizeof(Uint32) * numCentroids);
8
9     classifyPoints(0, numPoints, &numUpdatedPoints,
          tempNewClusters, tempNewClusterSize);
10
11
12    for(int centroid = 0; centroid < numCentroids; ++centroid)
13      for(int dimension = 0; dimension < nDimensions; ++dimension)
14      centroids[centroid].pos[dimension] =
            tempNewClusters[centroid*nDimensions + dimension] /
            tempNewClusterSize[centroid];
15
16    convergenceStatus = ((float)numUpdatedPoints/(float)numPoints
          < CONVERGENCE_DELTA);
17
18    delete[] tempNewClusters;
19    delete[] tempNewClusterSize;
20
21    return ++numIterationsRun;
22  }
23
24  int KMeans::loopParallel() {
25    int numUpdatedPoints = 0;
26
27    float *tempNewClusters = new float[numCentroids *
          nDimensions];
28    Uint32 *tempNewClusterSize = new Uint32[numCentroids];
29    memset(tempNewClusters, 0, sizeof(float) * numCentroids *
          nDimensions);
30    memset(tempNewClusterSize, 0, sizeof(Uint32) * numCentroids);
31
```

```
32    KMeansTask &a = *new (task::allocate_root()) KMeansTask(this,
          0, numPoints, &numUpdatedPoints, this->cutoff,
          tempNewClusters, tempNewClusterSize);
33    KMeansTask::spawn_root_and_wait(a);
34
35    convergenceStatus = ((float)numUpdatedPoints/(float)numPoints
          < CONVERGENCE_DELTA);
36
37    for(int centroid = 0; centroid < numCentroids; ++centroid)
38      for(int dimension = 0; dimension < nDimensions; ++dimension)
39      centroids[centroid].pos[dimension] =
            tempNewClusters[centroid*nDimensions + dimension] /
            tempNewClusterSize[centroid];
40
41    delete[] tempNewClusters;
42    delete[] tempNewClusterSize;
43
44    return ++numIterationsRun;
45  }
46
47  void KMeans::OnLoop() {
48    static boost::timer time;
49
50    if(convergenceStatus == 2)
51      return;
52
53    if(!numIterationsRun)
54      time.restart();
55
56    if(convergenceStatus == 0) {
57      numIterationsRun = loopParallel();
58    } else if (convergenceStatus == 1) {
59      std::cout << "Converged in " << numIterationsRun << "
            iterations." << std::endl;
60      std::cout << "Time taken: " << time.elapsed() << " secs.\n"
            << std::endl;
61      convergenceStatus = 2;
62    }
63  }
64
65  void KMeans::classifyPoints(int start, int count, int
        *pointsUpdated, float *tempNewClusters, Uint32
        *tempNewClusterSize ) {
66    Uint32 point = start;
67    int dimension, closestCentroid, centroid;
68    float distance, tempDistance;
69
70    for(std::vector<std::vector<double> >::const_iterator it =
          points.begin()+start; it != points.begin()+start+count;
```

```
      ++it, ++point) {
71      distance = FLT_MAX;
72      closestCentroid = -1;
73      for (centroid = 0; centroid < numCentroids; ++centroid) {
74        tempDistance = 0;
75        for (dimension = 0; dimension < nDimensions; ++dimension)
76          tempDistance += ((*it)[dimension] -
                centroids[centroid].pos[dimension]) *
                ((*it)[dimension] -
                centroids[centroid].pos[dimension]);
77
78        if (tempDistance < distance) {
79          closestCentroid = centroid;
80          distance = tempDistance;
81        }
82      }
83      if(clusterIds[point] != closestCentroid) {
84        clusterIds[point] = closestCentroid;
85        ++(*pointsUpdated);
86      }
87      for (dimension = 0; dimension < nDimensions; ++dimension) {
88        tempNewClusters[closestCentroid * nDimensions +
              dimension] += (*it)[dimension];
89        //tempNewClusters[closestCentroid * nDimensions +
              dimension] += arrayPoints[point*nDimensions +
              dimension];
90      }
91      ++tempNewClusterSize[closestCentroid];
92
93      assert(clusterIds[point] > -1);
94    }
95  }
96
97  void KMeans::initCentroids() {
98    float *color;
99
100   if(rawCentroids)
101     delete[] rawCentroids;
102   rawCentroids = new float[numCentroids * nDimensions];
103
104   if(centroids)
105     delete[] centroids;
106   centroids = new Centroid[numCentroids];
107
108   for(int centroid = 0; centroid < numCentroids; ++centroid)
109     centroids[centroid].pos =
          &rawCentroids[centroid*nDimensions];
110
111   std::set<int> takenPoints;
```

```
112   for (int centroid = 0; centroid < numCentroids; ++centroid) {
113     //uncomment the following to get a random init
114     int index = rand() % numPoints;
115
116     //don't init >1 cluster to the same point
117     while (find(takenPoints.begin(), takenPoints.end(), index)
          != takenPoints.end())
118       index = rand() % numPoints;
119
120     takenPoints.insert(index);
121
122     for (int dimension = 0; dimension < nDimensions;
          ++dimension) {
123       // the first inits a random place, the second on a random
            point.
124       // the latter guarantees that every centroid will have at
            least one point assigned, GIVEN that each point is
            unique...
125       //centroids[centroid].pos[dimension] = ((rand() % 1000) /
            1000.0f) * (maxPoints[dimension] -
            minPoints[dimension]);
126       centroids[centroid].pos[dimension] =
            points[index][dimension];
127     }
128
129
130     color = centroids[centroid].color;
131     switch (centroid) {
132
133       case 0:
134         color[0] = 1.0f;
135         color[1] = 1.0f;
136         color[2] = 1.0f;
137         break;
138       case 1:
139         color[0] = 1.0f;
140         color[1] = 0.0f;
141         color[2] = 1.0f;
142         break;
143       case 2:
144         color[0] = 0.0f;
145         color[1] = 1.0f;
146         color[2] = 1.0f;
147         break;
148       case 3:
149         color[0] = 0.0f;
150         color[1] = 0.0f;
151         color[2] = 1.0f;
152         break;
```

```
153         case 4:
154           color[0] = 1.0f;
155           color[1] = 0.0f;
156           color[2] = 0.0f;
157           break;
158         case 5:
159           color[0] = 0.0f;
160           color[1] = 1.0f;
161           color[2] = 0.0f;
162           break;
163         default:
164           color[0] = (rand() % 100) / 100.0f;
165           color[1] = (rand() % 100) / 100.0f;
166           color[2] = (rand() % 100) / 100.0f;
167       }
168     }
169 }
170
171
172 KMeansTask::KMeansTask(KMeans *_km, int _start, int _numPoints,
        int *_numUpdatedPoints, int _serialThreshold, float
        *_tempNewClusters, Uint32 *_tempNewClusterSize) :
173     serialThreshold(_serialThreshold),
174     numPoints(_numPoints),
175     numUpdatedPoints(_numUpdatedPoints),
176     start(_start),
177     km(_km),
178     tempNewClusters(_tempNewClusters),
179     tempNewClusterSize(_tempNewClusterSize)
180 {
181 }
182
183 task *KMeansTask::execute() {
184 #ifdef PROFILEKM
185     ProfilerRegisterThread();
186 #endif
187     if(numPoints < serialThreshold) {
188       km->classifyPoints(start, numPoints, numUpdatedPoints,
            tempNewClusters, tempNewClusterSize);
189       return NULL;
190     }
191
192     int splitNumPoints = numPoints / 2;
193
194     int numUpdatedPointsTask1 = 0;
195     int numUpdatedPointsTask2 = 0;
196
197     int size = km->numCentroids * km->nDimensions;
198
```

```
199   float *tempNewClusters1 = new float[size];
200   Uint32 *tempNewClusterSize1 =  new Uint32[km->numCentroids];
201   memset(tempNewClusters1, 0, sizeof(float) * size);
202   memset(tempNewClusterSize1, 0, sizeof(Uint32) *
          km->numCentroids);
203
204   float *tempNewClusters2 = new float[size];
205   Uint32 *tempNewClusterSize2 =  new Uint32[km->numCentroids];
206   memset(tempNewClusters2, 0, sizeof(float) * size);
207   memset(tempNewClusterSize2, 0, sizeof(Uint32) *
          km->numCentroids);
208
209   KMeansTask &subTask1 = *new (allocate_child())
          KMeansTask(this->km, start, splitNumPoints,
          &numUpdatedPointsTask1, serialThreshold, tempNewClusters1,
          tempNewClusterSize1);
210
211   int offset = splitNumPoints;
212   if(numPoints % 2)
213     offset += 1;
214
215   KMeansTask &subTask2 = *new (allocate_child())
          KMeansTask(this->km, start+splitNumPoints, offset,
          &numUpdatedPointsTask2, serialThreshold, tempNewClusters2,
          tempNewClusterSize2);
216
217   set_ref_count(3);
218   spawn(subTask2);
219
220   spawn_and_wait_for_all(subTask1);
221
222   *numUpdatedPoints = numUpdatedPointsTask1 +
          numUpdatedPointsTask2;
223
224   for(int centroid = 0; centroid < km->numCentroids;
          ++centroid) {
225     for(int dimension = 0; dimension < km->nDimensions;
            ++dimension) {
226     int index = centroid * km->nDimensions + dimension;
227     tempNewClusters[index] = tempNewClusters1[index] +
            tempNewClusters2[index];
228     }
229     tempNewClusterSize[centroid] =
            tempNewClusterSize1[centroid] +
            tempNewClusterSize2[centroid];
230   }
231
232   delete[] tempNewClusters1;
233   delete[] tempNewClusters2;
```

```
234     delete[] tempNewClusterSize1;
235     delete[] tempNewClusterSize2;
236
237     return NULL;
238   }
```

Listing D.3: The main parts of the SPC source code

```
1   #define NDEBUG
2   #ifndef NDEBUG
3   #define D(a) std::cout << (a) << std::endl;
4   #else
5   #define D(a) ;
6   #endif
7
8   #ifndef MST_DIMENSION
9   #define MST_DIMENSION 16
10  #endif
11
12  tbb::mutex printMutex;
13
14  typedef struct beginend {
15    const int a, b;
16    int begin() const { return a; }
17    int end() const { return b; }
18  } beginend;
19
20  #define SERIAL_FOR_START(start, end ) beginend r = {start,
        end};
21  #define SERIAL_FOR_END()
22
23  Spc::Spc(std::vector<std::vector<double> > points) {
24    unsigned int nNearestNeighbors = 11;
25
26    if(points.size() <= nNearestNeighbors) {
27      nNearestNeighbors = points.size() - 1; // can max have
           nPoints - itself num of neighbors
28      std::cout << "Reducing number of neighbors to " <<
           nNearestNeighbors << " because the dataset is too
           small." << std::endl;
29    }
30
31    thetaForGij = (nSpinStates*theta-1)/(nSpinStates-1);
32    this->points = points;
33    nPoints = (unsigned int) points.size();
34    nDimensions = (unsigned int) points[0].size();
35
36    clusterIds.resize(nPoints);
37
```

```
38    susceptibilityPlot = new XyPlot();
39    correlationClusterSizesPlot = new XyCollectionPlot();
40    shouldCancelSusceptibilityPlot = false;
41
42    randomGen = new RandomGenerator(0, nSpinStates-1);
43
44    buildKdtree();
45    D("built tree");
46    determineClosestNeighborsAndInteractionStrengths(
          nNearestNeighbors);
47    D("found neighbors");
48
49    initialGuessTemperature = round(1000.0 * exp(-0.5) / (4*log(1
          + sqrt(nSpinStates)))) / 1000.0;
50    std::cout << "Guess temp: " << initialGuessTemperature <<
          std::endl;
51  }
52
53  void Spc::buildKdtree() {
54    kdPoints = annAllocPts(nPoints, nDimensions); // data points
55    for (int i = 0; i < nPoints; i++) {
56      for (int j = 0; j < nDimensions; j++) {
57        kdPoints[i][j] = points[i][j];
58      }
59    }
60
61    kdTree = new ANNkd_tree(kdPoints, nPoints, nDimensions);
62  }
63
64  void
        Spc::determineClosestNeighborsAndInteractionStrengths(unsigned
        int nNearestNeighbors) {
65    searchKdTreeAndCopyNeighborData(nNearestNeighbors);
66    removeImmutualNeighborsAndBackNeighbors();
67    addMstToNeighborGraph(); // comment to remove mst
68    checkConnectedNeighborGraph();
69
70    // Determine average neighbor-distance (a)
71    unsigned int pairs = 0;
72    double avgDistance = 0.0;
73    for (int i = 0; i < nPoints; i++) {
74      pairs += (unsigned int) (((neighborDistances[i].size())));
75      for (unsigned int j = 0; j < neighborDistances[i].size();
            j++) {
76        avgDistance += neighborDistances[i][j];
77      }
78    }
79
80    avgDistance /= pairs;
```

```
81    calculateInteractionStrengths(pairs, avgDistance);
82  }
83
84  void Spc::searchKdTreeAndCopyNeighborData(unsigned int
        nNearestNeighbors) {
85    tbb::tick_count start = tbb::tick_count::now();
86
87    unsigned int K1 = nNearestNeighbors + 1; // search for one
          more, and then remove self (in the rare case of
          overlapping points)
88
89    neighborIndexes.resize(nPoints);
90    neighborDistances.resize(nPoints);
91    interactionStrengths.resize(nPoints);
92
93
94    SERIAL_FOR_START(0, nPoints); // this is to make it easier to
          parallelize when thread safe implementation is used.
95    ANNidx *kdIndexes = new ANNidx[K1]; // WORK
96    ANNdist *kdSquareDistances = new ANNdist[K1]; // WORK
97    for (int i = r.begin(); i != r.end(); i++) {
98      ANNpoint queryPoint = kdPoints[i];
99      kdTree->annkSearch(queryPoint, K1, kdIndexes,
            kdSquareDistances, annErrorToleranceFraction);
100     neighborIndexes[i].resize(nNearestNeighbors);
101     neighborDistances[i].resize(nNearestNeighbors);
102     int skip = 0;
103     for (unsigned int j = 0; j < nNearestNeighbors; j++) {
104       if (kdIndexes[j] == i) {
105         skip = 1; // Used not to copy self reference
106       }
107       neighborIndexes[i][j] = kdIndexes[j + skip];
108       neighborDistances[i][j] = sqrt(kdSquareDistances[j +
              skip]); // ||x-y||^2 == sum((xi-yi)^2), if we skip
              this we dont have to square at the interaction str
109     }
110   }
111   delete[] kdIndexes;
112   delete[] kdSquareDistances;
113   SERIAL_FOR_END();
114
115   tbb::tick_count stop = tbb::tick_count::now();
116   std::cout << "time for Neighbor search: " << (stop -
          start).seconds() << std::endl;
117
118 }
119
120 void Spc::removeImmutualNeighborsAndBackNeighbors() {
121   tbb::tick_count start = tbb::tick_count::now();
```

```
122   for (int i = 0; i < nPoints; i++) {
123     for (unsigned int j = 0; j < neighborIndexes[i].size();
          j++) {
124       int otherIndex = neighborIndexes[i][j];
125       if (otherIndex < i) {
126         // Remove back-neighbors
127         neighborIndexes[i].erase(neighborIndexes[i].begin() +
              j);
128         neighborDistances[i].erase(neighborDistances[i].begin()
              + j);
129         j--;
130       } else {
131         // Remove immutual neighbors
132         bool mutual = false;
133         for (unsigned int k = 0; k <
              neighborIndexes[otherIndex].size(); k++) {
134           if (neighborIndexes[otherIndex][k] == i) {
135             mutual = true;
136             break;
137           }
138         }
139
140         if (!mutual) {
141           neighborIndexes[i].erase(neighborIndexes[i].begin() +
                j);
142           neighborDistances[i].erase(neighborDistances[i].begin()
                + j);
143           j--;
144         }
145       }
146     }
147   }
148   tbb::tick_count stop = tbb::tick_count::now();
149   std::cout << " time for removeImmutualAndBackNeighbors: " <<
        (stop - start).seconds() << std::endl;
150 }
151
152 void Spc::checkConnectedNeighborGraph() {
153
154   ClusterCollection clusters(nPoints);
155   for (int i = 0; i < nPoints; i++) {
156     for (unsigned int j = 0; j < neighborIndexes[i].size();
          j++) {
157       int neighborIndex = neighborIndexes[i][j];
158
159       if (!clusters.isAssignedToCluster(i) &&
            !clusters.isAssignedToCluster(neighborIndex)) {
160         int clusterIndex = clusters.startCluster();
161         clusters.assign(clusterIndex, i);
```

```
162           clusters.assign(clusterIndex, neighborIndex);
163         } else if (clusters.isAssignedToCluster(i) &&
                !clusters.isAssignedToCluster(neighborIndex)) {
164           int clusterIndex = clusters.getContainingCluster(i);
165           clusters.assign(clusterIndex, neighborIndex);
166         } else if (!clusters.isAssignedToCluster(i) &&
                clusters.isAssignedToCluster(neighborIndex)) {
167           int clusterIndex =
                  clusters.getContainingCluster(neighborIndex);
168           clusters.assign(clusterIndex, i);
169         } else if (clusters.getContainingCluster(i) !=
                clusters.getContainingCluster(neighborIndex)) {
170           clusters.markMergeClustersForPointIndexes(i,
                  neighborIndex);
171         }
172       }
173     }
174     clusters.doMerge();
175     clusters.removeEmptyClusters();
176
177     std::cout << "Enforced connectedness. All points are
          connected in " << clusters.clusters.size() << "
          subgraphs:" << std::endl;
178
179 }
180
181 unsigned long long int doubleHash(double d, unsigned int
        leftRotate) {
182     unsigned long long int hash = *(unsigned long long int*)(&d);
183     return (hash << leftRotate) | (hash >> (64-leftRotate));
184 }
185
186 void Spc::addMstToNeighborGraph() {
187
188     std::cout << "MST" << std::endl;
189     if(nDimensions != MST_DIMENSION) {
190       std::cout << "MST not added, as it is hard-coded for " <<
              MST_DIMENSION << " dimensions" << std::endl;
191       return;
192     }
193
194     tbb::tick_count startMst = tbb::tick_count::now();
195
196     typedef reviver::dpoint<double,MST_DIMENSION> StannPoint;
197     std::vector<StannPoint> stannPoints;
198     stannPoints.resize(nPoints);
199     std::set<unsigned long long int> uniqueXorSet; // Because
          STANN requires unique datapoints
200
```

```
201   std::pair<std::set<unsigned long long int>::iterator, bool>
          ret;
202
203   std::cout << "Copying into STANN datatype" << std::endl;
204   for(int i=0;i<nPoints;i++) {
205     unsigned long long int xorValue = 0;
206     for(int currentDimension=0; currentDimension<nDimensions;
            currentDimension++) {
207       xorValue ^= doubleHash(points[i][currentDimension],
              currentDimension);
208     }
209     ret = uniqueXorSet.insert(xorValue);
210
211     while(ret.second == false) {
212       unsigned int changeDim = nDimensions *
              randomGen->randFrac();
213       std::cout << "Point " << i << " moved in dimension " <<
              changeDim << std::endl;
214       double currentValue = points[i][changeDim];
215       xorValue ^= doubleHash(currentValue, changeDim); //
              revert old contribution
216       currentValue += currentValue / 1e6 + 1e-10;
217       xorValue ^= doubleHash(currentValue, changeDim); // add
              new contribution
218       points[i][changeDim] = currentValue;
219       ret = uniqueXorSet.insert(xorValue);
220     }
221
222     for(int j=0;j<nDimensions;j++) {
223       stannPoints[i][j] = points[i][j];
224     }
225
226   }
227
228   typedef std::vector<StannPoint>::size_type stype;
229
230   std::vector< std::pair<stype,stype> > outputmst;
231
232   std::cout << "Starting MST" << std::endl;
233   gmst(stannPoints, outputmst);
234   std::cout << "MST complete - adding edges" << std::endl;
235
236   for(unsigned int i=0;i<outputmst.size();i++) {
237     int from = outputmst[i].first;
238     int to =  outputmst[i].second;
239     if(from < to) {
240       connectPoints(from, to);
241     } else {
242       connectPoints(to, from);
```

```
243       }
244     }
245
246     tbb::tick_count endMst = tbb::tick_count::now();
247     std::cout << "time for MST: " << (endMst-startMst).seconds()
          << std::endl;
248   }
249
250   void Spc::connectPoints(int fromIndex, int toIndex) {
251     if (std::find(neighborIndexes[fromIndex].begin(),
          neighborIndexes[fromIndex].end(), toIndex) ==
          neighborIndexes[fromIndex].end()) {
252       neighborIndexes[fromIndex].push_back(toIndex);
253       neighborDistances[fromIndex].push_back(distance(fromIndex,
            toIndex));
254     }
255   }
256
257   double Spc::distance(int fromIndex, int toIndex) {
258     double distance = 0;
259     for(unsigned int i=0;i<points[fromIndex].size();i++) {
260       distance += pow(points[fromIndex][i] -
            points[toIndex][i],2.0);
261     }
262     return sqrt(distance);
263   }
264
265   void Spc::calculateInteractionStrengths(unsigned int pairs,
        double avgDistance) {
266     tbb::tick_count start = tbb::tick_count::now();
267     double kHat = (2.0 * pairs) / nPoints; // Blatt section 4.1.3
268
269     for (int i = 0; i < nPoints; i++) {
270       interactionStrengths[i].resize(neighborIndexes[i].size());
271       for (unsigned int j = 0; j < neighborDistances[i].size();
            j++) {
272         double Jij = (1.0 / kHat) * exp(-(neighborDistances[i][j]
              * neighborDistances[i][j]) / (2.0 * avgDistance *
              avgDistance)); // Blatt eq 4.1
273         interactionStrengths[i][j] = Jij;
274       }
275     }
276     tbb::tick_count stop = tbb::tick_count::now();
277     std::cout << "time for calculateInteractionStrengths: " <<
          (stop - start).seconds() << std::endl;
278   }
279
280   XyPlot* Spc::plotSusceptibility(double minTemperature, double
        maxTemperature, double step) {
```

```
281 #ifdef PROFILESPC
282     ProfilerStart("profile/plotSusceptibility");
283 #endif
284     tbb::tick_count start = tbb::tick_count::now();
285     D("Susceptibility");
286     int numResults = (int) ((maxTemperature - minTemperature) /
            step);
287
288     int *iterationsPerformed = new int(0);
289
290     //tbb::task_scheduler_init init(1);
291
292     tbb::parallel_for(tbb::blocked_range<size_t>(0,numResults),
            PlotSusceptibility(this, iterationsPerformed,
            minTemperature, step, numResults));
293
294     delete iterationsPerformed;
295
296     shouldCancelSusceptibilityPlot = false;
297 #ifdef PROFILESPC
298     ProfilerStop();
299 #endif
300     tbb::tick_count stop = tbb::tick_count::now();
301     std::cout << "time for plotSusceptibility: " << (stop -
            start).seconds() << std::endl;
302     return susceptibilityPlot;
303 }
304
305 double Spc::calculateSusceptibility(const double temperature,
        bool determineSpinSpinCorrelation, RandomGenerator
        *innerRandom, bool printClusteringResults, int *spins) {
306     if(innerRandom == NULL)
307       innerRandom = randomGen;
308
309     std::vector<std::vector<double> > probabilityValues =
            calculateProbabilityValues(temperature);
310
311     bool spinsWereSpecifiedExternally = true;
312
313     if(spins == NULL) { // Local spins if not reusing from outer
            scope
314       spins = new int[nPoints];
315       spinsWereSpecifiedExternally = false;
316       // Assign random initial spin
317       for (int i = 0; i < nPoints; i++) {
318         spins[i] = innerRandom->randInRange();
319       }
320     }
321
```

```
322   double magnetizationSum = 0.0;
323   double magnetizationSquareSum = 0.0;
324
325   ClusterCollection clusters = ClusterCollection(nPoints);
326   int spinCount[nSpinStates];
327
328   // Setup correlation sum
329   std::vector<std::vector<double> > correlationSum(nPoints); //
          can be int until averaging
330   for (unsigned int i = 0; i < correlationSum.size(); i++) {
331     correlationSum[i].resize(neighborIndexes[i].size(), 0.0);
332   }
333
334   for (int iteration = 0; iteration < nMonteCarloIterations +
          nWarmupIterations; iteration++) {
335     // Empty clusters
336     clusters.reset();
337
338     for (int i = 0; i < nPoints; i++) {
339       for (unsigned int j = 0; j < neighborIndexes[i].size();
              j++) {
340         int neighborIndex = neighborIndexes[i][j];
341
342         if (spins[neighborIndex] == spins[i] &&
              probabilityValues[i][j] > innerRandom->randFrac()) {
343           if (!clusters.isAssignedToCluster(i) &&
                !clusters.isAssignedToCluster(neighborIndex)) {
344             int clusterIndex = clusters.startCluster();
345             clusters.assign(clusterIndex, i);
346             clusters.assign(clusterIndex, neighborIndex);
347           } else if (clusters.isAssignedToCluster(i) &&
                !clusters.isAssignedToCluster(neighborIndex)) {
348             int clusterIndex = clusters.getContainingCluster(i);
349             clusters.assign(clusterIndex, neighborIndex);
350           } else if (!clusters.isAssignedToCluster(i) &&
                clusters.isAssignedToCluster(neighborIndex)) {
351             int clusterIndex =
                  clusters.getContainingCluster(neighborIndex);
352             clusters.assign(clusterIndex, i);
353           } else if (clusters.getContainingCluster(i) !=
                clusters.getContainingCluster(neighborIndex)) {
354             clusters.markMergeClustersForPointIndexes(i,
                  neighborIndex);
355           }
356         }
357       }
358     }
359
360     clusters.doMerge();
```

```
361
362     // Change spins
363     for (int i = 0; i < nPoints; i++) {
364       spins[i] = innerRandom->randInRange();
365     }
366     for (unsigned int i = 0; i < clusters.clusters.size(); i++)
          {
367       std::vector<int> *currentCluster = &clusters.clusters[i];
368       int newSpin = innerRandom->randInRange();
369       for (std::vector<int>::const_iterator it =
            currentCluster->begin(); it != currentCluster->end();
            ++it) {
370         spins[*it] = newSpin;
371       }
372     }
373
374     if (iteration >= nWarmupIterations) {
375
376       // Spin-spin correlation
377       if (determineSpinSpinCorrelation) {
378         for (int i = 0; i < nPoints; i++) {
379           for (unsigned int j = 0; j <
                correlationSum[i].size(); j++) {
380             if (spins[i] == spins[neighborIndexes[i][j]]) {
381               correlationSum[i][j] += 1.0;
382             }
383           }
384         }
385       }
386
387       // Count number of spin members
388       memset(spinCount, 0, nSpinStates * sizeof(int));
389       for (int i = 0; i < nPoints; i++) {
390         spinCount[spins[i]]++;
391       }
392
393       // Calculate magnetic properties
394       int Nmax = 1; // Number of points in "biggest" spinState
395       for (int i = 0; i < nSpinStates; i++) {
396         if (spinCount[i] > Nmax) {
397           Nmax = spinCount[i];
398         }
399       }
400       //double currentEnergy = calculateEnergy(spins);
401       double currentMagnetization = ((double) nSpinStates *
            Nmax - nPoints) / ((nSpinStates - 1.0) * nPoints); //
            Blatt eq 2.4
402       magnetizationSum += currentMagnetization;
```

```
403         magnetizationSquareSum += currentMagnetization *
                currentMagnetization;
404         //std::cout << "Energy in iteration " << iteration << ":
                " << currentEnergy << std::endl;
405       }
406     }
407     if(!spinsWereSpecifiedExternally) {
408       delete[] spins;
409     }
410     double averageMagnetization = magnetizationSum /
            nMonteCarloIterations;
411     double averageSquareMagnetization = magnetizationSquareSum /
            nMonteCarloIterations;
412
413     double susceptibility = (double) nPoints / temperature *
            (averageSquareMagnetization - averageMagnetization *
            averageMagnetization); // Blatt eq 2.6
414
415     if (determineSpinSpinCorrelation) {
416       for (int i = 0; i < nPoints; i++) {
417         for (unsigned int j = 0; j < correlationSum[i].size();
                j++) {
418           correlationSum[i][j] /= nMonteCarloIterations;
419         }
420       }
421       assert(correlationSum.size() == neighborIndexes.size());
422       // Perform clustering based on spin-spin correlation
423       ClusterCollection correlationClusters =
              ClusterCollection(nPoints);
424       for (int i = 0; i < nPoints; i++) {
425         for (unsigned int j = 0; j < neighborIndexes[i].size();
                j++) {
426           int neighborIndex = neighborIndexes[i][j];
427           if (correlationSum[i][j] > thetaForGij) {
428             if (!correlationClusters.isAssignedToCluster(i) &&
                    !correlationClusters.isAssignedToCluster(
                    neighborIndex)) {
429               int clusterIndex =
                      correlationClusters.startCluster();
430               correlationClusters.assign(clusterIndex, i);
431               correlationClusters.assign(clusterIndex,
                      neighborIndex);
432             } else if (correlationClusters.isAssignedToCluster(i)
                    && !correlationClusters.isAssignedToCluster(
                    neighborIndex)) {
433               int clusterIndex =
                      correlationClusters.getContainingCluster(i);
434               correlationClusters.assign(clusterIndex,
                      neighborIndex);
```

```
435             } else if
                    (!correlationClusters.isAssignedToCluster(i) &&
                    correlationClusters.isAssignedToCluster(
                    neighborIndex)) {
436               int clusterIndex =
                      correlationClusters.getContainingCluster(neighborIndex);
437               correlationClusters.assign(clusterIndex, i);
438             } else if
                    (correlationClusters.getContainingCluster(i) !=
                    correlationClusters.getContainingCluster(
                    neighborIndex)) {
439               correlationClusters.markMergeClustersForPointIndexes(i,
                      neighborIndex);
440             }
441           }
442         }
443       }
444       correlationClusters.doMerge();
445       correlationClusters.removeEmptyClusters();
446
447       std::vector<std::vector<int> > *clusters =
             &correlationClusters.clusters;
448
449
450       clusterIds.resize(nPoints, -1);
451       for (unsigned int cluster = 0; cluster <
             lastComputedClusters.size(); ++cluster) {
452         for(unsigned int point = 0; point <
                lastComputedClusters[cluster].size(); ++point) {
453           clusterIds[ lastComputedClusters[cluster][point] ] =
                  cluster;
454         }
455       }
456
457       if(printClusteringResults) {
458         std::cout << "Printing " << clusters->size() << "
                clusters.." << std::endl;
459         for(unsigned int cluster = 0; cluster < clusters->size();
                ++cluster) {
460           std::cout << "Cluster #" << cluster << " (" <<
                  (*clusters)[cluster].size() << "): ";
461           std::sort((*clusters)[cluster].begin(),
                  (*clusters)[cluster].end());
462           for(unsigned int point = 0; point <
                  (*clusters)[cluster].size(); ++point) {
463             std::cout << (*clusters)[cluster][point] << " ";
464           }
465           std::cout << std::endl;
466         }
```

```
467       }
468
469       std::vector<int> currentSizes;
470
471       for(unsigned int i=0;i<clusters->size();i++) {
472         currentSizes.push_back((*clusters)[i].size());
473       }
474       std::sort(currentSizes.begin(), currentSizes.end(),
              std::greater<int>());
475       correlationClusterSizesPlot->insert(temperature,
              currentSizes);
476
477       lastComputedClusters = *clusters;
478     }
479     if(temperature == 0.0) {
480       return 0.0;
481     }
482     return susceptibility;
483 }
484
485 std::vector<std::vector<double> >
      Spc::calculateProbabilityValues(const double & temperature) {
486     // Determine probabilities for temperature
487     std::vector<std::vector<double> > probabilityValues =
          std::vector<std::vector<double> >(nPoints);
488     for (int i = 0; i < nPoints; i++) {
489       probabilityValues[i].resize(interactionStrengths[i].size());
490       for (unsigned int j = 0; j <
              interactionStrengths[i].size(); j++) {
491         if(temperature == 0)
492           probabilityValues[i][j] = 1.0;
493         else
494           probabilityValues[i][j] = 1.0 -
                exp(-interactionStrengths[i][j] / temperature); //
                Blatt eq 3.2
495       }
496     }
497
498     return probabilityValues;
499 }
500
501 double Spc::calculateEnergy(int *spins) {
502     double energy = 0;
503     for (int currentIndex = 0; currentIndex < nPoints;
          currentIndex++) {
504       for (unsigned int j = 0; j <
              neighborIndexes[currentIndex].size(); j++) {
505         int neighborIndex = neighborIndexes[currentIndex][j];
506         if (spins[currentIndex] == spins[neighborIndex]) {
```

```
507           energy -= interactionStrengths[currentIndex][j];
508         }
509       }
510     }
511     return energy;
512 }
513
514 PlotSusceptibility::PlotSusceptibility(Spc *spc, int
      *iterationsPerformed, double minTemperature, double step,
      int numResults) {
515     this->spc = spc;
516     this->iterationsPerformed = iterationsPerformed;
517     this->minTemperature = minTemperature;
518     this->step = step;
519     this->numResults = numResults;
520 }
521
522 void PlotSusceptibility::operator()(tbb::blocked_range<size_t>
      &r) const {
523     {
524       printMutex.lock();
525       std::cout << "SPC task range: [" << r.begin() << "," <<
            r.end()-1 << "] (" << (r.end() - r.begin()) << ")" <<
            std::endl;
526       printMutex.unlock();
527     }
528 #ifdef PROFILESPC
529     ProfilerRegisterThread();
530 #endif
531     RandomGenerator *innerRandom = new
          RandomGenerator(spc->randomGen);
532     int *spins = new int[spc->nPoints];
533     // Assign random initial spin
534     for (int i = 0; i < spc->nPoints; i++) {
535       spins[i] = innerRandom->randInRange();
536     }
537
538     for (unsigned int i = r.begin(); i != r.end() &&
          !spc->shouldCancelSusceptibilityPlot; i++) {
539       double temperature = minTemperature + step * i;
540       double susceptibility =
            spc->calculateSusceptibility(temperature, true,
            innerRandom, false, spins);
541       spc->susceptibilityPlot->insert(temperature,
            susceptibility);
542       spc->finishness =
            (double)((*iterationsPerformed)++)/numResults;
543     }
544     delete[] spins;
```

```
545     delete innerRandom;
546 }
```

Listing D.4: Wavelet KS reduction

```
1  WaveletReductionKS::WaveletReductionKS()
2  {
3    outDimensions = 10;
4    waveletType = gsl_wavelet_haar;
5    k = 2;
6  }
7
8  WaveletReductionKS::~WaveletReductionKS()
9  {
10 }
11
12 InputDataset* WaveletReductionKS::reduce(const InputDataset
       *input) {
13   return reduce(input, false);
14 }
15
16 InputDataset* WaveletReductionKS::reduce(const InputDataset
       *input, bool manualIndexes)
17 {
18   int inputSampleSize = input->data[0].size();
19   n = 1<< (int)ceil(log2(inputSampleSize)); // pow2-padded
20
21   InputDataset *outData = new InputDataset(outDimensions,
         input->nSamples, 1);
22   outData->resizeVectorsToParameters();
23
24   gsl_wavelet *wavelet = gsl_wavelet_alloc(waveletType, k);
25   gsl_wavelet_workspace *work = gsl_wavelet_workspace_alloc(n);
26
27   assert(wavelet != NULL);
28   assert(work != NULL);
29
30   double *data = new double[n];
31
32   std::vector<std::vector<double> > transformedData(
         input->nSamples, std::vector<double>(n, 0));
33   std::vector<double> means(n);
34   std::vector<double> variances(n);
35   std::vector<double> distributionDifferences(n);
36
37   //TODO could be parallelized with some effort
38   for(unsigned int
         sampleIndex=0;sampleIndex<input->nSamples;sampleIndex++) {
39     memset(data, 0, n*sizeof(double));
```

```
40      std::copy(input->data[sampleIndex].begin(),
            input->data[sampleIndex].end(), data);
41
42      gsl_wavelet_transform_forward(wavelet, data, 1, n, work);
43      for(int i=0;i < n; i++) {
44        transformedData[sampleIndex][i] = data[i];
45      }
46    }
47
48    // means
49    for(int dim=0; dim < n; dim++) {
50      double mean = 0;
51      for(unsigned int sample=0; sample < input->nSamples;
            sample++) {
52        mean += transformedData[sample][dim];
53      }
54      mean /= input->nSamples;
55      means[dim] = mean;
56    }
57
58    // variances
59    for(int dim=0; dim < n; dim++) {
60      double variance = 0;
61      for(unsigned int sample=0; sample < input->nSamples;
            sample++) {
62        variance += pow(transformedData[sample][dim] -
              means[dim], 2);
63      }
64      variances[dim] = variance / input->nSamples;
65    }
66
67    std::vector<double> sortedSamples(input->nSamples);
68    std::vector<double> probabilities(input->nSamples);
69
70    // Empirical CDF and KS distance
71    for(int dim=0; dim < n; dim++) {
72
73      for(unsigned int sample=0; sample < input->nSamples;
            sample++) {
74        sortedSamples[sample] = transformedData[sample][dim];
75      }
76      std::sort(sortedSamples.begin(), sortedSamples.end());
77
78      for(unsigned int i=0; i< sortedSamples.size()-1;) {
79        int forwardCounter = 0;
80        while(sortedSamples[i] == sortedSamples[i +
              forwardCounter + 1] && (i + forwardCounter + 1) <
              sortedSamples.size()-1) {
81          forwardCounter++;
```

```
82        }
83        int maxIndex = i + forwardCounter +1;
84        for(int j=i;j < maxIndex; j++) {
85          probabilities[j] = ((double)maxIndex) /
              sortedSamples.size();
86        }
87        probabilities[sortedSamples.size()-1] = 1;
88        i = maxIndex;
89      }
90
91      double distributionDifference = 0;
92      if(means[dim] == 0 || variances[dim] < 0.0010) {
93        distributionDifferences[dim] = 0;
94      } else {
95        for(std::vector<double>::iterator it =
            sortedSamples.begin(); it != sortedSamples.end();
            it++) {
96          double value = *it;
97          double z = (value-means[dim])/sqrt(variances[dim]);
98          int probabilityIndex =
              std::lower_bound(sortedSamples.begin(),
              sortedSamples.end(), value) - sortedSamples.begin();
99          double localDistributionDifference =
              fabs(probabilities[probabilityIndex] -
              gsl_cdf_ugaussian_P(z));
100         if(localDistributionDifference >
              distributionDifference) {
101           distributionDifference = localDistributionDifference;
102         }
103       }
104
105       distributionDifferences[dim] = distributionDifference;
106     }
107   }
108
109   if(!manualIndexes) {
110     indexes = new int[n];
111     double* devData = new double[n];
112
113     std::copy(distributionDifferences.begin(),
            distributionDifferences.end(), devData);
114
115     for(int i=0; i < n; i++) {
116       indexes[i] = i;
117     }
118
119     //insertion sort, to obtain sorting permutation
120     for(int j = 1; j < n; j++){
121
```

```
122         double key = devData[j];
123         int keyIndex = indexes[j];
124         int i = j - 1;
125
126         while(i >= 0 && devData[i] > key){
127           devData[i + 1] = devData[i];
128           indexes[i + 1] = indexes[i];
129           i--;
130         }
131         devData[i + 1] = key;
132         indexes[i + 1] = keyIndex;
133       }
134
135     } else {
136       std::cout << "Used manual indexes: ";
137     }
138     // Print the indexes
139     std::cout << "KS wavelet\nindexes= {";
140     for(int i=0; i < n-1; i++) {
141       std::cout << indexes[i] << ", ";
142     }
143     std::cout << indexes[n-1] << "};" << std::endl;
144
145     // Now sorted
146     for(unsigned int dimIndex=0; dimIndex < outDimensions;
          dimIndex++) {
147       for(unsigned int spike=0; spike < input->nSamples; spike++)
            {
148         outData->data[spike][dimIndex] =
              transformedData[spike][indexes[n-1-dimIndex]];
149       }
150     }
151
152     gsl_wavelet_free (wavelet);
153     gsl_wavelet_workspace_free (work);
154     delete[] data;
155
156     outData->isReduced = true;
157     outData->reduceClass = this;
158     return outData;
159 }
160
161 InputDataset* WaveletReductionKS::reverse(const InputDataset
      *input) {
162     int outN = ((WaveletReductionKS*) input->reduceClass)->n;
163
164     InputDataset *outData = new InputDataset(outN,
          input->nSamples, input->nChannels);
165     outData->resizeVectorsToParameters();
```

```
166
167   gsl_wavelet *wavelet = gsl_wavelet_alloc(waveletType, k);
168   gsl_wavelet_workspace *work = gsl_wavelet_workspace_alloc(n);
169
170   assert(wavelet != NULL);
171   assert(work != NULL);
172
173   double *data = new double[outN];
174
175   //TODO could be parallelized with some effort
176   for(unsigned int
          sampleIndex=0;sampleIndex<input->nSamples;sampleIndex++) {
177     memset(data, 0, n*sizeof(double));
178     for(unsigned int i=0; i < outDimensions; i++) {
179       data[indexes[n-1-i]] = input->data[sampleIndex][i];
180     }
181
182     gsl_wavelet_transform_inverse(wavelet, data, 1, n, work);
183     for(int i=0;i < outN; i++) {
184       outData->data[sampleIndex][i] = data[i];
185     }
186   }
187
188   gsl_wavelet_free (wavelet);
189   gsl_wavelet_workspace_free (work);
190   delete[] data;
191   outData->reduceClass = this;
192   return outData;
193 }
```

Listing D.5: Wavelet first reduction

```
1  WaveletReductionFirst::WaveletReductionFirst()
2  {
3    outDimensions = 16;
4
5    waveletType = gsl_wavelet_daubechies;
6    k = 8;
7
8  }
9
10 InputDataset *WaveletReductionFirst::reduce(const InputDataset
       *input) {
11   int inputSampleSize = input->data[0].size();
12   int n = 1<< (int)ceil(log2(inputSampleSize)); // pow2-padded
13
14   InputDataset *outData = new InputDataset(outDimensions,
         input->nSamples, input->nChannels);
15   outData->resizeVectorsToParameters();
```

```
16
17    gsl_wavelet *wavelet = gsl_wavelet_alloc(waveletType, k);
          //which to choose - supported ones are 4,6,8,...,20. See
          http://wavelets.pybytes.com/wavelet/db8/
18    gsl_wavelet_workspace *work = gsl_wavelet_workspace_alloc(n);
19
20    assert(wavelet != NULL);
21    assert(work != NULL);
22
23    double *data = new double[n];
24
25    std::vector <std::vector <double > > transformedData;
26
27    transformedData.resize( input->nSamples, std::vector<double>(
          n, 0) );
28
29    for(unsigned int
          sampleIndex=0;sampleIndex<input->nSamples;sampleIndex++) {
30      memset(data, 0, n*sizeof(double));
31      std::copy(input->data[sampleIndex].begin(),
            input->data[sampleIndex].end(), data);
32
33      gsl_wavelet_transform_forward(wavelet, data, 1, n, work);
34      for(unsigned int i=0;i < outDimensions; i++) {
35        outData->data[sampleIndex][i] = data[i];
36      }
37    }
38
39    gsl_wavelet_free (wavelet);
40    gsl_wavelet_workspace_free (work);
41    delete[] data;
42
43    outData->isReduced = true;
44    outData->reduceClass = this;
45    return outData;
46  }
47
48  InputDataset *WaveletReductionFirst::reverse(const InputDataset
        *input) {
49    int n = 256; // pow2-padded
50    int outN = 200;
51
52    InputDataset *outData = new InputDataset(n, input->nSamples,
          input->nChannels);
53    outData->resizeVectorsToParameters();
54
55    gsl_wavelet *wavelet = gsl_wavelet_alloc(waveletType, k); //
          K/2 vanishing points
56    gsl_wavelet_workspace *work = gsl_wavelet_workspace_alloc(n);
```

```
57
58    assert(wavelet != NULL);
59    assert(work != NULL);
60
61    double *data = new double[n];
62
63    for(unsigned int
          sampleIndex=0;sampleIndex<input->nSamples;sampleIndex++) {
64      memset(data, 0, n*sizeof(double));
65      std::copy(input->data[sampleIndex].begin(),
            input->data[sampleIndex].end(), data);
66
67      gsl_wavelet_transform_inverse(wavelet, data, 1, n, work);
68      for(int i=0;i < outN; i++) {
69        outData->data[sampleIndex][i] = data[i];
70      }
71    }
72
73    gsl_wavelet_free (wavelet);
74    gsl_wavelet_workspace_free (work);
75    delete[] data;
76    outData->reduceClass = this;
77    return outData;
78  }
79
80  WaveletReductionFirst::~WaveletReductionFirst() {
81
82  }
```

Listing D.6: PCA reduction

```
1   PcaReduction::PcaReduction()
2   {
3   }
4
5   InputDataset *PcaReduction::reduce(const InputDataset *input) {
6     return reduce(input, -1);
7   }
8
9   InputDataset *PcaReduction::reduce(const InputDataset *input,
        int numComponents)
10  {
11    if(numComponents == -1) {
12      numComponents = 2;
13    }
14
15    int numSpikes = input->data.size();
16    int numFeatures = input->data[0].size();
17
```

```
18    gsl_matrix *mdev = gsl_matrix_alloc(numFeatures, numSpikes);
19    gsl_matrix *m = gsl_matrix_alloc(numFeatures, numSpikes);
20
21    double *mean = new double[numFeatures];
22    memset(mean, 0, sizeof(double)*numFeatures);
23
24  #pragma omp parallel for
25    for(int i = 0; i < numSpikes; ++i) {
26      for(int j = 0; j < numFeatures; ++j) {
27        mean[j] += input->data[i][j];
28      }
29    }
30
31  #pragma omp parallel for
32    for(int j = 0; j < numFeatures; ++j) {
33      mean[j] /= numSpikes;
34    }
35    std::cout << std::endl;
36
37    gsl_matrix *covarianceMatrix = gsl_matrix_alloc(numFeatures,
          numFeatures);
38    gsl_vector_view a, b;
39
40  #pragma omp parallel for
41    for(unsigned int spike = 0; spike < input->data.size();
          ++spike) {
42      for(int feature = 0; feature < numFeatures; ++feature) {
43        gsl_matrix_set(m, feature, spike,
              input->data[spike][feature]);
44        gsl_matrix_set(mdev, feature, spike,
              input->data[spike][feature] - mean[feature]);
45      }
46    }
47
48    // Calculate covariance matrix (lower triangle, then copy to
          other half)
49    for (int dim1 = 0; dim1 < numFeatures; dim1++) {
50  #pragma omp parallel for
51      for (int dim2 = dim1; dim2 < numFeatures; dim2++) {
52        double v;
53        a = gsl_matrix_row(mdev, dim1);
54        b = gsl_matrix_row(mdev, dim2);
55        v = gsl_stats_covariance(a.vector.data, a.vector.stride,
              b.vector.data, b.vector.stride, a.vector.size);
56        gsl_matrix_set(covarianceMatrix, dim1, dim2, v);
57      }
58    }
59
60  #pragma omp parallel for
```

```
61    for (int dim1 = 1; dim1 < numFeatures; dim1++) {
62      for (int dim2 = 0; dim2 < dim1; dim2++) {
63        gsl_matrix_set(covarianceMatrix, dim1, dim2,
              gsl_matrix_get(covarianceMatrix, dim2, dim1));
64      }
65    }
66
67    gsl_vector *eigenValues = gsl_vector_alloc (numFeatures);
68    gsl_matrix *eigenVectors = gsl_matrix_alloc (numFeatures,
          numFeatures);
69
70    gsl_eigen_symmv_workspace * work =
          gsl_eigen_symmv_alloc(numFeatures);
71    gsl_eigen_symmv (covarianceMatrix, eigenValues, eigenVectors,
          work);
72
73    gsl_eigen_symmv_free(work);
74
75    gsl_eigen_symmv_sort (eigenValues, eigenVectors,
          GSL_EIGEN_SORT_ABS_DESC);
76
77    gsl_matrix *subsetEigenVectorsMatrix =
          gsl_matrix_alloc(numFeatures, numComponents);
78
79  #pragma omp parallel for
80    for(int i=0; i<numFeatures; i++) {
81      for(int j=0; j<numComponents; j++) {
82        gsl_matrix_set(subsetEigenVectorsMatrix, i, j,
              gsl_matrix_get(eigenVectors, i, j)); // TODO quicker
              copying
83      }
84    }
85
86    gsl_matrix *pcaData = gsl_matrix_alloc(numComponents,
          numSpikes);
87    gsl_blas_dgemm(CblasTrans, CblasNoTrans, 1,
          subsetEigenVectorsMatrix, mdev, 0, pcaData);
88
89    InputDataset *outData = new InputDataset(numComponents,
          numSpikes, 1);
90
91  #pragma omp parallel for
92    for(int i=0; i < numComponents; i++) {
93      for(int j=0; j < numSpikes; j++) {
94        outData->data[j][i] = gsl_matrix_get(pcaData, i, j);
95      }
96    }
97
98    gsl_matrix_free(pcaData);
```

```
 99    gsl_matrix_free(covarianceMatrix);
100    gsl_matrix_free(subsetEigenVectorsMatrix);
101    gsl_matrix_free(eigenVectors);
102    gsl_vector_free(eigenValues);
103
104    delete[] mean;
105    return outData;
106  }
107
108  InputDataset* PcaReduction::reverse(const InputDataset *input) {
109    return NULL;
110  }
```