A Hierarchical Strategy for Computing Nonparametric Survival Functions with Interval-censored Data

Stephen M. Taylor

Supervisor: Dr Yong Wang

19 February 2009

A thesis submitted in partial fulfilment
of the requirements for the degree of Master of Science
in the Department of Statistics, The University of Auckland.

Abstract

Consider survival data where some or all of the event times are censored by arbitrary intervals in the positive real line. In such a case, finding the nonparametric maximum likelihood survival function is a task that requires an efficient iterative algorithm. Several algorithms are currently available but none is clearly best in all situations. The choice of the best algorithm depends on the size of the data set and the proportion of exact observations.

The aim of this research was to develop an algorithm that is the best in all situations. One of the existing algorithms, the Constrained Newton Method (CNM), was enhanced in this study to incorporate a divide and conquer approach. The enhancements resulted in the development of the Hierarchical Constrained Newton Method (HCNM).

At each iteration, the new HCNM algorithm partitions the support set into a number of blocks, solves the allocation of probability mass within each block and then performs a global reallocation of probability mass among the blocks. The reallocation among blocks is achieved by the HCNM algorithm calling itself recursively, making use of a 'mixture of mixtures' model. With this approach, the algorithm converges rapidly to the solution.

Results from simulation studies suggest that HCNM is the best choice for data sets of any size and with any proportion of exact observations.

Acknowledgements

Firstly a great big thank-you goes to my supervisor Dr Yong Wang for his infectious enthusiasm for my research topic, his great ideas and for his ongoing support during my studies. I have learnt a lot from him over the past year.

My project was made possible by the generous scholarship provided by the Marsden Fund, for which I am very grateful.

Interval-censored data from a clinical trial were kindly provided, thanks to Jull et al. (2008). This trial was very interesting; it assessed the effectiveness of manuka honey in speeding the healing of leg ulcers. This data set was used only for illustrations.

This thesis was created using GNU Emacs 22.1.1 and was typeset in LaTeX. Simulations and algorithm development were performed using the **R** statsitical programming language (R Development Core Team, 2008). Tables and figures were also created in **R** directly from the simulation results, using the Sweave package. Simulations were performed on a Sun N1 Grid Engine, a network of computers available for large computing projects such as this one. Thanks to the Department of Statistics for the use of computing resources. In particular, thanks to Stephen Cope for his assistance with the grid engine.

Proof-reading a thesis like this one is a considerable challenge. Thanks to Katharina Parry for reading through the thesis and checking it for clarity of explanations and definitions. Lastly, I have great pleasure in thanking my wife Carol Stewart for her proof-reading assistance, with a focus on grammar and layout. Carol also provided wise advice on the general structure of the thesis and support throughout my year of research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main aim of the research presented in this thesis was to develop a new, reliably fast algorithm to solve a particular problem in the area of survival analysis. This chapter introduces the main concepts that are important to this research. The new algorithm is called the Hierarchical Constrained Newton Method (HCNM).

The new algorithm is applicable to a specific type of survival data set, which is described in Section 1.1. Given such a data set, the computational goal of estimating the nonparametric survival function is introduced in Section 1.2. A brief overview of existing methods is provided in Section 1.3, followed by a description of the new algorithm in Section 1.4. Finally, an outline of the remainder of the thesis is provided in Section 1.5.

Several methods are currently available for achieving this computational goal, but none of them is the best in all situations. Part of the motivation for this research was to attempt to create an algorithm that is the best in all situations. Further motivation for a fast algorithm comes from applications that must solve the problem many times. Examples of this are bootstrap resampling methods and profile likelihood of semi-parametric models.

1.1 Description of data

Survival data is a common form of data requiring statistical analysis. In some settings, it is also known as failure time data or time to event data. In engineering, the term reliability is used.

In survival analysis, the time taken for a specified event to occur is measured for each subject of interest. This is usually identified by the transition from an 'initial' state to a 'failure' state. For example, the subjects may be people suffering from a specific medical condition and participating in a clinical trial (unwell$\rightarrow$recovered). Another example is the monitoring of machine parts for reliability (functioning$\rightarrow$broken). After collecting information about the event times, inference can then be made about how survival progresses with time.

Some types of event can be recorded exactly at the moment they occur. An example of this is in the analysis of mortality rates; when someone dies, the date of death is normally recorded.

However, there are many types of event that cannot be directly observed. Such events require periodic examination of the subjects in order to assess whether or not the event has occurred. The healing of ulcer under a dressing is an example of this type of event. In such a case, the investigator must periodically inspect the status of each subject and record whether or not it is in the specified failure state. When the failure state is first observed at one of these time points, all that is known is that the actual event occurred at some time since the previous inspection. At the previous inspection, the subject was still in the initial state and at the current inspection, the failure state was observed. This time interval between inspections is said to 'censor' (hide) the actual event time, which is then said to be interval-censored. Note also that the event may not yet have occurred by the final inspection, and may never occur for that subject. This situation is known as right censoring because the actual event time lies to the right of the final inspection time. Similarly, left censoring occurs when the failure state is observed on the first inspection, in which case the censoring interval starts at time zero. Event times from these two situations are described as right-censored and left-censored, respectively, and these situations are special cases of interval censoring.

To complicate matters further, some studies are interested in the combined risk of more than one type of event. In such a case, 'survival' may mean the time elapsed until the first occurrence of an event of any type. In their studies, the National Cancer Institute (United States) uses the following definition of disease-free survival (National Cancer Institute, 2009).

> **Disease-free survival:** The length of time after treatment for a specific disease during which a patient survives with no sign of the disease. Disease-free survival may be used in a clinical study or trial to help measure how well a new treatment works.

This is a measure of the time a patient remains alive and free from their original disease. There are two types of event that can terminate this survival period. Assessing the recurrence of a cancer is the type of inspection that can result in interval-censored observations. When a patient dies, the time of this event is usually known exactly. So this definition provides a mechanism that generates data containing both interval-censored observations (recurrence of the disease) and exact observations (deaths).

The study reported by Kumwenda et al. (2008) gives a real-life example. The study looked at infection-free survival times of more than 3000 uninfected infants born to HIV-1 infected mothers in Malawi. Transmission of the virus from mother to child was possible in the breast milk. Several treatments were studied to help prevent the transmission of the virus. The infection-free survival time could be terminated either by the discovery of the virus in the infant or by the death of the infant. A schedule of follow-up visits was implemented to test for viral infection. This means that some survival times were interval-censored and others were measured exactly.

The interval-censored observations arose from the need to perform a test for the presence of infection. When an infant died, from whatever cause, the date of death was recorded and this represents an exact observation of the survival time. This example is used again in Section 6.1 as the basis for a method of generating simulated data.

1.2 Computational goal

The goal of survival analysis is to describe how survival progresses over time, which involves describing the distribution of event times. To describe the survival curve with the type of data described in the previous section, the Nonparametric Maximum Likelihood Estimate (NPMLE) of the survival function is often used. The NPMLE is a monotonically decreasing step function. It is an empirical estimate of the survival function that does not constrain the shape of the function using a parametric family.

In a nutshell, the NPMLE locates a set of time points and time intervals, and allocates probability mass among them in such a way that is the most likely, based on the original data. Since there is no closed-form analytical solution for the NPMLE, it is necessary to use iterative numerical methods. The form of this solution has a mixture distribution, a fact that is utilised in the construction of the new algorithm.

Alternatives to the NPMLE

The Kaplan-Meier (KM) product limit estimator is a popular method of deriving a nonparametric survival function (Kaplan and Meier, 1958; Kalbfleisch and Prentice, 2002). The method is applicable to so-called case I censored data, where there are only exact, left-censored and right-censored observations. In this case, each censoring interval includes either zero or infinity, and an exact closed-form solution exists, which the KM method locates. However, the KM method cannot be used for case II (or, more generally, case-k and mixed-case) censored data, when arbitrary interval-censored observations are found. The reason why it cannot be used is that it relies on the ability to define the risk set at any given time, but this is made impossible by the interval-censored event times.

One common method of dealing with interval censoring is to use some form imputation (Sun, 2006, section 2.4). These methods make simplifying assumptions about the event times in order to progress the analysis. Each interval-censored observation may be replaced by a single number from within the interval (usually one of the end-points or the mid-point). However, this throws away some of the information in the data and can introduce bias. Moreover, it is not necessary because a method exists for deriving a nonparametric survival function (specifically, the NPMLE) without making those simplifying assumptions.

With general interval-censored survival data, the sizes of the censoring intervals can be comparable to the actual range of survival times. This can happen when there are relatively few opportunities to make the necessary inspections, or when there are large gaps between inspections. In such cases, it is important to allow for this censoring in order to obtain more accurate inference. Using the NPMLE approach instead of KM with imputation can also increase the power of statistical tests for detecting treatment effects.

1.3 Existing methods

Several algorithms exist for finding the NPMLE survival function in the presence of interval censoring. However, no single algorithm is best in all situations (Wang, 2008).

The Expectation-Maximisation (EM) algorithm (Turnbull, 1976; Dempster et al., 1977) is often used to deal with missing data and incomplete data problems, such as interval-censored observations. The performance of EM on the NPMLE problem is known to be very poor. Pilla and Lindsay (2001) enhanced the EM algorithm in a number of ways and found some significant improvements in performance.

The Iterative Convex Minorant (ICM) algorithm was introduced by Groeneboom (1991) and was studied by Groeneboom and Wellner (1992) and Jongbloed (1998). Wellner and Zhan (1997) enhanced the ICM algorithm by building a hybrid algorithm, which combined it with the EM algorithm. The hybrid ICM-EM algorithm benefited from the strengths of both algorithms and performed very well. Dümbgen et al. (2006) described a Subspace-based Newton method, which is also applicable to the NPMLE problem and generally performs well.

Wang (2008) introduced a technique to reduce the dimensionality of the NPMLE problem for highly censored datasets. The technique is applicable to any NPMLE-finding algorithm, since it works by identifying a specific subset of the problem to focus on. The same paper also tested the Constrained Newton Method (CNM) against other algorithms on the problem of finding the NPMLE survival function with interval-censored data. The CNM algorithm was compared to the ICM-EM and SBN algorithms, both with and without the dimension reduction technique. Results indicated that CNM worked well with data sets with a low proportion of exact observations, but that the other algorithms were faster in other situations.

More details about existing algorithms are given in Section 2.4 and Section 2.5.

1.4 New algorithm

A new algorithm, the Hierarchical Constrained Newton Method (HCNM), has been developed to solve the NPMLE survival function problem. HCNM is based on CNM and enhances it by using a divide and conquer approach that aims to make efficient use of the information in the data. By dividing a large problem into a number of smaller problems, and then pulling them together again, the HCNM algorithm makes some considerable improvements on its predecessor's performance. Results from simulation studies are promising and indicate that HCNM may well be the overall best choice of algorithm.

1.5 Thesis outline

The remainder of this thesis is structured as follows. Chapter 2 provides some background on the research problem and some reviews of existing methods. Chapters 3 to 6 describe the research that was conducted for this Masters thesis.

The theoretical basis and internal workings of the new HCNM algorithm are explained in Chapter 3, while Chapter 4 describes the process of fine-tuning the algorithm to achieve high performance. The new HCNM algorithm depends on an old algorithm for solving the least squares problem with non-negativity constraints, the so-called NNLS algorithm. Chapter 5 explores the time complexity of the NNLS, HCNM and CNM algorithms. This provides additional motivation for the main research.

Chapter 6 describes simulation studies, which were designed to compare the performance of the new HCNM algorithm with existing methods. Finally, Chapter 7 provides discussion, conclusions and suggestions for further research.

Appendix A provides source code that was used during this research. The Bibliography provides a list of references and the Glossary gives definitions for some key terms, abbreviations and mathematical symbols that are used in this thesis.

Chapter 2

Background

This chapter provides background on the topic of this thesis, including a definition of the research problem and a review of relevant existing literature. Section 2.1 describes the type of data being considered and defines the computational goal in detail. The computational goal is to derive the nonparametric maximum likelihood estimate (NPMLE) of the survival function, given a survival data set where some or all of the event times are censored by arbitrary intervals in the positive real line. Section 2.2 describes an approach to solving the NPMLE problem using a finite mixture model. Section 2.3 describes a technique for reducing the dimensionality of the problem, thereby improving computational performance.

Several methods are available for achieving this goal, but none is the best in all situations. Section 2.4 describes the recently developed Constrained Newton Method (Wang, 2007c) for solving the computational goal. Section 2.5 describes some other existing algorithms for solving the problem, giving some background and explanations of how they work. Section 2.6 looks at a technique to ensure convergence and at criteria that are used by algorithms to determine when to stop. Finally, Section 2.7 gives some concluding remarks and leads into Chapter 3.

2.1 The research problem

This section provides a definition of the research problem. Subsection 2.1.1 describes interval censoring in survival data. Subsection 2.1.2 introduces the idea of nonparametric survival functions. Subsection 2.1.3 defines what is meant by the 'nonparametric maximum likelihood estimate' (NPMLE) and describes how it can be constructed.

2.1.1 Survival analysis with interval censoring

In survival analysis, inference is made about event times. By examining the times when a specified 'failure' event occurs for a number of subjects, a picture of the progression of survival over time can be constructed. This can be achieved even when some or all of the event times are censored (hidden) by arbitrary time intervals.

Let T be a univariate random variable with cumulative probability distribution $F(t)$ giving rise to a set $\{t_i\}$ of n independent event times. Each t_i may not be directly observable, but instead may be censored by an interval of time between two values L_i and R_i. Although the actual value of t_i is not observed, the two censoring values L_i and R_i provide useful information about it, since the unknown value lies somewhere in the interval between them.

The left endpoint $L_i \in [0, \infty)$ is the last known time that subject i was in the 'initial' state. The right endpoint $R_i \in (0, \infty]$ represents the first time ($R_i \geqslant L_i$) at which subject i was observed as being in the 'failure' state, with $R_i = \infty$ indicating that this was never observed. When $L_i = R_i$ this represents an exact observation, whereas when $L_i < R_i$ the event time is interval-censored. The latter case includes both left censoring ($L_i = 0$) and right censoring ($R_i = \infty$) as special cases.

This representation provides a general way of presenting almost all types of censoring that are commonly encountered in univariate data in practice (Wellner, 1995; Schick and Yu, 2000). These include the case I censoring model (also known as 'current status'), in which every subject is inspected once and every observation is either left-censored or right-censored. An example of case I censored data is an experiment that looks for the incidence of tumors in mice. If the presence/absence of a tumour can only be identified by means of dissection, then there can be only one inspection per mouse.

Also included in this representation are case II and more generally case $k > 2$ interval censoring, and mixed-case interval censoring. The case parameter k represents the number of inspections per subject. In mixed-case interval censoring, k varies among the subjects, so this is the most general situation of those mentioned. In practical settings where multiple inspections are possible, it is unlikely that all subjects will be inspected exactly the same number of times, especially if the inspections cease at the first sight of failure. Mixed-case interval censoring is common in clinical trials.

Even though some subjects may have been inspected many times, for every subject just two of the inspection times are sufficient to give all the information available about the actual (hidden) event time. Those are the last recorded time that the subject was in the 'initial' state and the first recorded time of the 'failure' state. These are represented by L_i and R_i above, respectively.

Whether exact or interval-censored, it is convenient to denote the observed times by the set of intervals $\{O_i : i = 1, \ldots, n\}$ defined by:

$$O_i = \begin{cases} \{R_i\} & \text{if } L_i = R_i \\ (L_i, R_i] & \text{if } L_i < R_i. \end{cases} \tag{2.1}$$

In either case, the observation about the event time t_i is that $t_i \in O_i$.

The mechanism that generates the censoring may be random or fixed according to a predefined schedule of inspection times. Whatever the mechanism, it is assumed to be non-informative about the event times. That means that for any subject surviving to time t, the expected future survival time remains the same regardless of whether or not they are censored at that point. Because of this assumption, the censoring mechanism can be ignored when constructing the nonparametric likelihood function (Betensky, 2000; Oller Piqué, 2006).

An example of interval-censored data

Figure 2.1 shows an example data set that contains interval-censored survival data. The example data set contains seven subjects. Subject 1 (top) illustrates a left-censored event time, since its interval $O_1 = (0, 2]$ starts at zero. Subjects 2 and 6 illustrate exact observations, at $O_2 = \{1\}$ and $O_6 = \{5\}$ respectively. Subjects 4 and 7 are examples of right-censored event times, with $O_4 = (3, \infty]$ and $O_7 = (7, \infty]$. Subjects 3 and 5 are examples of general interval-censored observations, with $O_3 = (3, 4]$ and $O_5 = (4, 6]$.

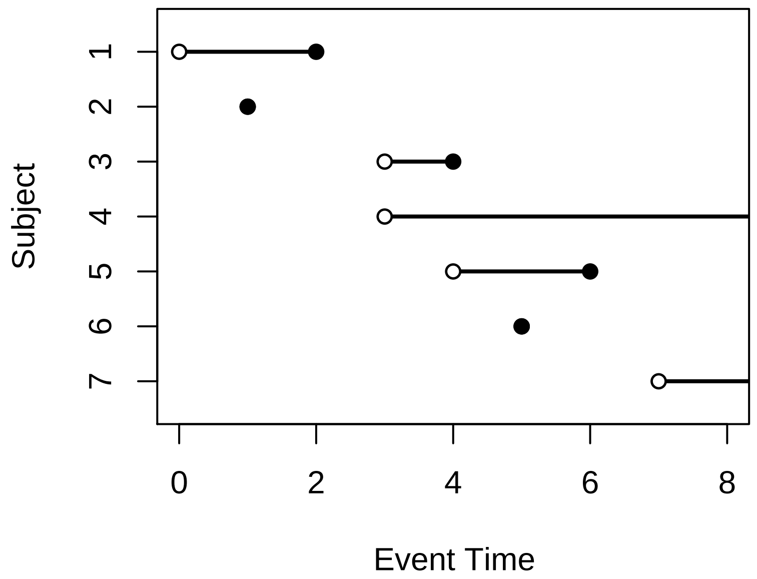


Figure 2.1: Illustration of exact, left-censored, right-censored and interval-censored event times.

2.1.2 Nonparametric survival function

Consider a random variable T representing the time taken until a specified event occurs, having cumulative probability distribution function $F(t)$ for times $t \geqslant 0$. A survival function $S(t)$ describes the probability of surviving to a particular time t. This is the probability that the event has not yet occurred at that point. So the survival function can be expressed in terms of $F(t)$ as:

$$S(t) = 1 - F(t). \tag{2.2}$$

Typically, the functions $F(t)$ and $S(t)$ are not known and need to be estimated from observed data.

For a nonparametric estimate of a survival function, no assumptions are made about the probability distribution of the event times (such as the fitting of a distribution from a parametric family). Instead, the nonparametric estimate is an empirical estimate of the survival function. It takes the form of a monotonically decreasing step function because the sample size is finite and the survival function can be estimated only at the time points that were observed.

Consider an event that has occurred at time t_i but which is not directly observed. Instead it is censored by an observed interval $O_i = (L_i, R_i]$. This gives information about the survival function at the two ends of that interval. Specifically, two observations have been made: that the event occurred at or before time R_i and that the event did *not* occur at or before time L_i. The probability of observing $T \leqslant R_i$ is $F(R_i) = 1 - S(R_i)$. Similarly, the probability of observing $T \leqslant L_i$ is $F(L_i) = 1 - S(L_i)$. Hence the probability of observing $T \in O_i$ is given by

$$\Pr[T \in O_i] = \Pr[T \leqslant R_i] - \Pr[T \leqslant L_i] = S(L_i) - S(R_i). \tag{2.3}$$

Now consider an exact observation $O_i = \{R_i\}$. From a nonparametric point of view, the probability of such an observation is defined by the probability mass at exactly that point in time, namely $f(R_i)$ as follows:

$$\Pr[T \in O_i] = \Pr[T = R_i] = f(R_i). \tag{2.4}$$

The downward steps in the nonparametric survival function are defined by the magnitudes of these probability masses.

Figure 2.2 provides an illustration of a nonparametric survival function that has been derived from a data set containing interval-censored observations. Data for this example come from a New Zealand study of manuka honey proposed as an aid to the healing of leg ulcers (Jull et al., 2008). In this study, leg ulcers were covered with dressings and inspected weekly for 12 weeks, which led to observations of healing times being interval-censored. The survival curve begins with a value of one at $t = 0$. By the end of the 12 weeks, the estimated survival probability (that the ulcer has not yet healed by this time) was just over 40%.

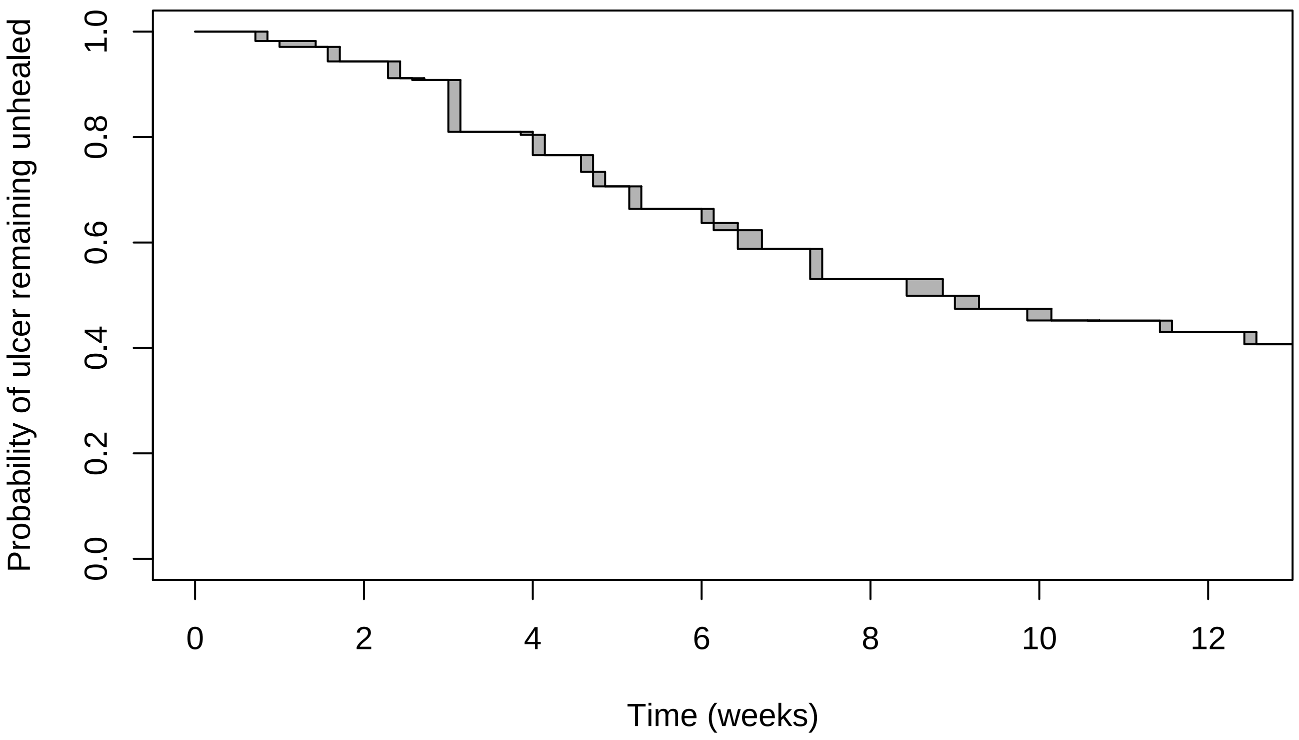


Figure 2.2: Example of a nonparametric survival function from interval-censored data. The plot shows how the probability of remaining unhealed varies with time, for leg ulcers dressed with manuka honey in a New Zealand clinical trial.

2.1.3 Nonparametric maximum likelihood estimate

The nonparametric maximum likelihood estimate (NPMLE) of a survival function is a nonparametric estimate that is derived using maximum likelihood theory. For this reason, the resulting estimate of the survival function is the one that is the most likely, given the observed data. The NPMLE survival function is a useful tool in exploratory data analysis, or as a necessary replacement for the popular Kaplan-Meier product limit estimator (Kaplan and Meier, 1958) when some of the data are interval-censored.

The NPMLE is constructed by deriving a finite set of mutually disjoint support intervals and allocating probability mass to each of them. The probability mass assigned to a support interval defines how much the survival function drops within that interval. The support intervals are constructed from the smallest intervals of time that can be distinguished from the data. A support interval can be a single point, resulting from an exact observation. Others are narrow intervals of time. When there is a positive gap between two consecutive support intervals, the NPMLE survival function is constant in the gap, since probability mass is only allocated to the support intervals.

An interesting feature of the NPMLE is illustrated by the grey boxes in Figure 2.2. These boxes are examples of areas where the survival function cannot be pinned down exactly. For support intervals of positive width and positive probability mass, there exists a rectangle within which the NPMLE gives no information about how the survival function progresses. From a completely nonparametric point of view, any (valid) path from the top-left corner to the bottom-right corner has the same likelihood. It is equally likely that the survival function runs along the top and then down the right of a box as running down the left side then along the bottom.

There are two parts to the process of deriving the NPMLE. The first part is the 'reduction' step, where the full set of possible support intervals is defined and then reduced in size by removing unnecessary ones. This leaves a set of candidate support intervals. The second part is the 'allocation' step, where unit probability mass is distributed among the candidate support intervals in order to maximise the likelihood function. During the second step, the size of the support set may be further reduced when a candidate support interval is assigned a zero probability mass.

Reduction step

The reduction step constructs a set of candidate support intervals from the set of observation intervals. The combination $\Omega = \bigcup_{i=1}^{n} O_i$ of all the observed time intervals gives the set of all times that are possible from the observed data. For the nonparametric estimate of the survival function, probability mass cannot be allocated outside this set.

Firstly, the set Ω is partitioned into a finite number of partition intervals, which will be denoted by $\mathfrak{I}_\mathfrak{j} : \mathfrak{j} = 1, \ldots, \mathfrak{m}$. Because they form a partition, no two of these intervals intersect with each other:

$$\forall \mathfrak{i}\ \forall \mathfrak{j} \neq \mathfrak{i} : \mathfrak{I}_\mathfrak{i} \cap \mathfrak{I}_\mathfrak{j} = \emptyset. \tag{2.5}$$

They combine to form the same set as the observations, so that

$$\bigcup_{\mathfrak{j}=1}^{\mathfrak{m}} \mathfrak{I}_\mathfrak{j} = \bigcup_{i=1}^{n} O_i. \tag{2.6}$$

The partition intervals are created as follows. Each element in the partition is a point or interval that is a non-empty intersection of some combination of the $\{O_i\}$ intervals. The combinations are chosen so that each partition interval is as small as possible without being empty. Let U be the set of unique values from $\{L_1, \ldots, L_n, R_1, \ldots, R_n\}$. The partition intervals are constructed using these points as the partitioning points.

When O_i is an exact observation, $O_i = \{R_i\}$, there will be one of the intervals $\mathfrak{I}_\mathfrak{j} = \{R_i\}$. There may also be an interval-censored observation that overlaps that exact observation, $O_j : O_i \in O_j$. In this case there will be an open partition interval $\mathfrak{I}_{\mathfrak{j}-1} = (t, R_i)$ where t is the greatest value in U that is less than R_i.

When there is no exact observation at one of the values R_i, the partition will include a semi-open interval $\mathfrak{I}_\mathfrak{j} = (t, R_i]$ where t is the greatest value in U that is less than R_i. Thus the set of partition intervals is constructed.

Secondly, the set of intervals $\{\mathfrak{I}_\mathfrak{j}\}$ is pruned to create a subset of $m \leqslant \mathfrak{m}$ candidate support intervals. According to Peto (1973) and Turnbull (1976), positive probability mass will be allocated to a subset of the partition intervals. Some of them can be immediately eliminated as they will certainly receive zero probability mass when the likelihood is maximised. The key to the reduction step is to identify and remove those elements of the partition.

The term 'clique' for a partition interval is used to describe the set of observations that contain that partition interval. This word is used in the sense of a subset from a population that all have something in common. So the clique of $\mathfrak{I}_\mathfrak{j}$ is the set of observation indices $\mathfrak{C}_\mathfrak{j} = \{i : \mathfrak{I}_\mathfrak{j} \subseteq O_i\}$.

A 'maximal clique' is defined as a clique that is not a subset of either of its immediate neighbour cliques. The clique $\mathfrak{C}_\mathfrak{j}$ is non-maximal if either $\mathfrak{C}_\mathfrak{j} \subset \mathfrak{C}_{\mathfrak{j}-1}$ or $\mathfrak{C}_\mathfrak{j} \subset \mathfrak{C}_{\mathfrak{j}+1}$, otherwise it is maximal. Note that only one of these can be tested when $\mathfrak{j} = 1$ or $\mathfrak{j} = \mathfrak{m}$.

Partition intervals that have non-maximal cliques are removed from the set. What remains is a set of m intervals, which will be denoted by $\{I_j : j = 1, \ldots, m\}$. The important characteristic of a partition interval with a maximal clique is that it can support at least one of the O_i observations that neither of its neighbours can support. These partition intervals qualify for the label 'candidate support intervals' because they provide probability support to the nonparametric survival function.

The term 'maximal intersection' has also been used to refer to a partition interval that has a maximal clique. Each I_j is a maximal intersection of a subset of the $\{O_i\}$ intervals (Wong and Yu, 1999; Gentleman and Vandal, 2001; Maathuis, 2005).

The $n \times m$ clique matrix $\mathbf{A}$ for a survival data set is the matrix that defines which of the candidate support intervals I_j intersects (and therefore can support) each of the observation intervals O_i. The element in the i^{th} row and j^{th} column of the matrix $\mathbf{A}$ is the indicator:

$$\delta_{ij} = \begin{cases} 1 & \text{if } I_j \subseteq O_i \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

An example of a large clique matrix is illustrated in Figure 6.1 (on page 73).

Before the reduction step can begin, the set of observations $\{O_i\}$ must be sorted. The time complexity (defined in the Glossary) for the sorting task is $O(n \log(n))$. For data already sorted, the time complexity of the reduction step has been shown to be linear for univariate data and $O(n^2)$ for bivariate data (Maathuis, 2005). Wong and Yu (1999), Gentleman and Vandal (2001) and Maathuis (2005) have considered the support set reduction process for bivariate and multivariate survival data sets. The time complexity of creating the clique matrix is at least $O(nm)$. Since the reduction step is part of any NPMLE-finding algorithm, the time complexity of such an algorithm must be at least that of the worst time complexity from the components of the reduction step. Chapter 5 explores the time complexity of algorithms in more detail.

Allocation step

The key to finding the NPMLE is to find the best way to allocate probability mass among the candidate support intervals. This is done in such a way as to maximise the value of the likelihood function, which is defined in Section 2.2 using a mixture model.

Unfortunately, there is no closed-form solution to this allocation problem, meaning that a mathematical expression cannot be derived that will immediately produce the solution. So the problem is a numerical optimisation problem and it must be solved using iterative computational methods. For all known algorithms for solving this problem, the time complexity per iteration of the allocation step is at least $O(nm)$ (the time complexity of the reduction step). So the allocation step determines the time complexity of algorithms overall.

2.2 Mixture model approach

This section describes an approach to solving the NPMLE allocation problem using mixture models. Subsection 2.2.1 defines finite mixture models and describes how the NPMLE allocation problem can be modeled using a finite mixture distribution. Subsection 2.2.2 defines the likelihood function.

2.2.1 Finite mixture models

A finite mixture model uses a probability distribution that is constructed from a number of other probability distributions, as a weighted combination of them. Typically, the component distributions are known (or assumed to be known) but the weights are not. The weights must be non-negative and add up to one. Sampling from this model can be imagined as a two-step process: select one of the components using the weights as probabilities, then sample from that component.

For a mixture of m components, the combined density can be expressed in the following form:

$$f(x) = \sum_{j=1}^{m} \pi_j f_j(x). \tag{2.8}$$

The j^{th} mixture component has density function $f_j(x)$ and weight π_j.

The vector of weights $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_m)^T$ is a point in the unit simplex, which is an $m-1$ dimensional constrained hyperplane within the space $\mathbb{R}^m$. The constraints at the boundaries of the unit simplex are that each component π_j must satisfy $0 \leqslant \pi_j \leqslant 1$ and that the total $\sum_{j=1}^{m} \pi_j = 1$.

Suppose the random variable X has a probability density defined by Eq. (2.8), from which a sample of n independent observations $x_1, \ldots, x_n$ is taken. The likelihood function expresses the likelihood of the observed sample as a function of a proposed vector of weights $\boldsymbol{\pi}$, given by

$$L(\boldsymbol{\pi}) = \prod_{i=1}^{n} f(x_i|\boldsymbol{\pi}) = \prod_{i=1}^{n} \sum_{j=1}^{m} \pi_j f_j(x_i). \tag{2.9}$$

This is an n^{th} degree polynomial on m-dimensional vectors. For convenience the log-likelihood is used, defined as

$$\ell(\boldsymbol{\pi}) = \log(L(\boldsymbol{\pi})) = \sum_{i=1}^{n} \log\left(\sum_{j=1}^{m} \pi_j f_j(x_i)\right). \tag{2.10}$$

The log-likelihood function is concave and has a unique finite maximum (Gentleman and Geyer, 1994; Lindsay, 1995; Böhning et al., 1996).

Locating the nonparametric maximum likelihood estimate of the weights is an optimisation problem that seeks a vector $\hat{\boldsymbol{\pi}}$ that maximises the log-likelihood (equivalent to maximising the likelihood) over the entire simplex. That means for all $\boldsymbol{\pi} \neq \hat{\boldsymbol{\pi}}$ the log-likelihood $\ell(\boldsymbol{\pi})$ is less than the maximum log-likelihood $\ell(\hat{\boldsymbol{\pi}})$.

Hill analogy

The log-likelihood function can be thought of as the surface of a big hill, with just one peak. We normally think of the surface of a hill as being two-dimensional. It is a measure of height, expressed as a function of two directions in which we can move, for example north-south and east-west. The log-likelihood hill may have many more dimensions, but the idea is the same. The shape of the surface is concave, which means that the function values lie on or above the line between any two points on the surface. Now, there may be a fence running along it somewhere and we may not be allowed to cross the fence. This is a constraint such as $\pi_j \geqslant 0$.

Mixture model for the research problem

The allocation step, which allocates probability mass to the candidate support intervals, has a mixture distribution. In this context, each f_j is a distribution on a finite sample space. Indeed, each one corresponds to a single candidate support interval. So $f_j(x_i)$ describes the conditional probability of observing the event represented by x_i given that the j^{th} mixture component has been observed. This is the conditional probability $\Pr[T \in O_i | T \in I_j]$. Because of the way the support intervals I_j have been constructed from the observed censoring intervals O_i, this conditional probability has the value of δ_{ij} which is either 1 or 0.

$$f_j(x_i) = \Pr[T \in O_i | T \in I_j] = \delta_{ij} \tag{2.11}$$

One interpretation of this is that each event time observation $t_i \in O_i$ is actually an observation from one of the mixture components from which it could have arisen (those with $\delta_{ij} = 1$), although we do not know which one.

2.2.2 Nonparametric likelihood function

This subsection continues from the 'allocation' step, at the end of Subsection 2.1.3, making use of the mixture distributions defined above. It defines the nonparametric likelihood function for a survival data set containing interval-censored observations.

Suppose that each candidate support interval I_j has an associated probability mass π_j, which together form a probability vector $\boldsymbol{\pi} = (\pi_1, \dots, \pi_m)^T$. The vector $\boldsymbol{\pi} \in \mathbb{R}^m$ is a point in the unit simplex, which means that it provides a valid set of probabilities.

For an individual observation O_i the probability of seeing an event in this time interval is denoted by $p_i = \Pr[T \in O_i]$. This value can be derived from the probability masses of candidate support intervals that intersect with O_i, which are those where $\delta_{ij} = 1$. That probability p_i also gives the likelihood $L_i(\boldsymbol{\pi})$ of the i^{th} observation, when expressed as a function of the proposed probability vector:

$$L_i(\boldsymbol{\pi}) = p_i = \Pr[T \in O_i | \boldsymbol{\pi}] = \sum_{j=1}^{m} \delta_{ij} \pi_j. \tag{2.12}$$

Since they are assumed to be independent, the likelihood of the full set of n observations is

$$L(\boldsymbol{\pi}) = \prod_{i=1}^{n} L_i(\boldsymbol{\pi}) \tag{2.13}$$

and the log-likelihood function $\ell(\boldsymbol{\pi})$ is defined by taking logs:

$$\ell(\boldsymbol{\pi}) = \log(L(\boldsymbol{\pi})) = \sum_{i=1}^{n} \log\big(L_i(\boldsymbol{\pi})\big) = \sum_{i=1}^{n} \log\left(\sum_{j=1}^{m} \delta_{ij} \pi_j\right). \tag{2.14}$$

Within the space of the unit simplex, the log-likelihood function defined by Eq. (2.14) is continuous, concave and twice differentiable. It has a unique finite global maximum (Gentleman and Geyer, 1994; Böhning et al., 1996). The goal of the allocation step is to locate the maximum likelihood, which is defined to be at the point $\hat{\boldsymbol{\pi}}$. This is the solution to the NPMLE allocation problem.

Vertex-directional gradient of $\ell(\boldsymbol{\pi})$

The j^{th} vertex of the unit simplex, denoted by $\mathbf{e}_j$, is the point with $\pi_j = 1$ and all the other components zero. The NPMLE can be characterised in terms of vertex-directional gradients, which are the gradients in the direction of each of the vertices of the unit simplex (Lindsay, 1995; Böhning et al., 1996; Wang, 2008).

The vector of vertex-directional gradients is denoted by $\mathbf{d} = (d_1, \ldots, d_m)^T$. Each component d_j is the vertex-directional gradient of $\ell(\boldsymbol{\pi})$ at the estimate $\boldsymbol{\pi}$ in the direction of the vertex $\mathbf{e}_j$. By defining a set of m column vectors $\mathbf{s}_j = (s_{1j}, \ldots, s_{nj})^T \in \mathbb{R}^n$ for $j = 1, \ldots, m$ by their elements

$$s_{ij} = \frac{\delta_{ij}}{p_i} \tag{2.15}$$

the j^{th} component of $\mathbf{d}$ is given by

$$d_j = \mathbf{s}_j^T \mathbf{1} - n \tag{2.16}$$

as explained in Wang (2008).

At the point $\hat{\boldsymbol{\pi}}$ the log-likelihood function is maximised, so the vertex-direction gradient in every direction must be zero unless the boundary has been hit in that direction:

$$\begin{aligned} d_j(\hat{\boldsymbol{\pi}}) &= 0 \quad \text{if } \hat{\pi}_j > 0 \\ d_j(\hat{\boldsymbol{\pi}}) &\leqslant 0 \quad \text{if } \hat{\pi}_j = 0 \end{aligned} \tag{2.17}$$

In the first situation, the co-ordinate $\hat{\pi}_j$ is inside the open set $(0, 1)$. The log-likelihood function is flat and curving downwards in either direction, towards and away from the vertex $\mathbf{e}_j$. In the second situation, the boundary of the simplex has been reached at $\hat{\pi}_j = 0$. The log-likelihood function slopes downwards from this point into the simplex in the direction towards the vertex $\mathbf{e}_j$.

At any non-optimal point $\boldsymbol{\pi} \neq \hat{\boldsymbol{\pi}}$ at least one of the vertex-directional gradient components must be positive. This vertex direction gives a clue to where $\hat{\boldsymbol{\pi}}$ can be found, a fact that is used by the dimension reduction technique.

2.3 Dimension reduction technique

Several algorithms are available for solving the problem of finding the NPMLE survival function. Wang (2008) introduced a dimension reduction technique that can potentially improve any NPMLE-finding algorithm, especially in the case of a highly censored data set (with a low proportion or absence of exact observations). In such a case, the dimensionality of the solution is typically much less than the size of the full set of candidate support intervals. The benefit of reduced dimensionality is that it can considerably speed up the search for the solution during the 'allocation' step mentioned above. In that paper, the performance of algorithms was investigated with and without the dimension reduction technique. The results demonstrated the improvement in computation times gained by this technique.

There are two key ideas to the dimension reduction technique: starting small and growing rapidly.

Starting small

The first idea is to start with as few support intervals as possible, while still ensuring that every one of the original observations receives positive probability mass. From this idea we get the name ‘dimension reduction’, since the search for a solution occurs within a space of reduced dimensions. Each of the original observations must receive support from at least one of the support intervals in the current support set. Ensuring this means that we can still calculate a finite log-likelihood value, which is necessary for the optimisation algorithm to work successfully.

To find the initial small support set, the technique works as follows. Firstly, all of the unique exact observations must be included. These are identified by finding rows in the clique matrix with a sum of one, meaning that these original observations receive support from only one of support intervals. Each support interval identified here must be in the final solution. However, this set may not yet support the full set of observations. Next, a process of adding further support intervals is repeated until every original observation is supported. This is done by examining the column sums of the clique matrix, looking only at rows that are yet to receive support from the current support set. The support interval that supports the greatest number of as yet unsupported observations (with the greatest column sum) is then added to the support set. When this repetitive process completes, the result is a (hopefully small) subset of the full set of support intervals, which gives a good starting point in the search for the solution.

Growing rapidly

The second key idea of the dimension reduction technique is to expand the support set rapidly from one iteration of the algorithm to the next. Between each pair of consecutive support intervals in the current set, a new support interval may be added. This potentially doubles the size of the support set at each iteration, allowing for exponential growth of the size of the support set. Previous algorithms have expanded the support set too slowly, such as by adding a single support interval at each iteration, as pointed out by Wang (2008). The key to this second idea is in the choice of which one to add. The interval that has the maximum vertex-directional gradient is selected. Choosing this one means the new support interval is expected to receive positive probability mass and thereby increase the value of the log-likelihood (to climb further up the hill).

Referring to the hill analogy, say we’re on the south side of the hill, on a reasonably flat one-dimensional path leading east-west. This newly added support interval gives a new dimension in which to move: we can now move in the north-south dimension, so we can climb higher.

Sometimes the support set can reduce in size because a support interval may receive zero probability mass during the algorithm’s quest to maximise the likelihood value. However, within the HCNM algorithm proposed in Chapter 3, this did not happen very often. This is because the dimension reduction technique is good at predicting which ones should be included.

2.4 Constrained Newton method

Wang (2007c) introduced the Constrained Newton Method (CNM) in the general context of mixture distributions. It uses the non-negative least squares (NNLS) algorithm provided by Lawson and Hanson (1974) to redistribute probability mass among the mixture components. CNM possesses a quadratic order of convergence, which has been described as the gold standard for the convergence of algorithms. This means that it converges to a solution rapidly in terms of the number of iterations required.

Wang (2008) applied CNM to the survival analysis problem of finding the NPMLE, and performed some simulations to compare it with the ICM-EM and SBN algorithms. The dimension reduction technique was included within the CNM algorithm. The performance of the other two algorithms was investigated with and without the dimension reduction technique. Results in that paper indicate that CNM performed well only when the proportion of exact observations was small. As this proportion increased, the performance of CNM deteriorated rapidly, when compared to the other two algorithms (Wang, 2008). This deterioration was seen in the time taken per iteration, and can be explained by the time complexity of the NNLS algorithm and the way that CNM uses NNLS (see Chapter 5).

How the CNM algorithm works

The CNM algorithm makes use of a quadratic approximation to the log-likelihood function, which is based on the Taylor series. The Taylor series approximation is derived from the first and second derivatives of the log-likelihood function.

It is convenient to express the first and second derivatives in terms of the column vectors $\mathbf{s}_j$ (defined in Eq. (2.15)). The $n \times m$ matrix $\mathbf{S}$ is defined by binding the $\mathbf{s}_j$ column vectors together:

$$\mathbf{S} = [\mathbf{s}_1, \ldots, \mathbf{s}_m]. \tag{2.18}$$

The ordinary gradient of $\ell(\boldsymbol{\pi})$ is the first derivative given by

$$\mathbf{g} = \nabla \ell(\boldsymbol{\pi}) = \mathbf{S}^T \mathbf{1} \tag{2.19}$$

and the second derivative is the Hessian matrix, defined as

$$\mathbf{H} = \nabla^2 \ell(\boldsymbol{\pi}) = -\mathbf{S}^T \mathbf{S}. \tag{2.20}$$

Typically, the Hessian matrix is computationally expensive to compute (Wang, 2008) but the approach used by CNM is designed to avoid this.

Denoting the difference between two estimates $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ by $\boldsymbol{\eta} = \boldsymbol{\pi}' - \boldsymbol{\pi}$, the Taylor series expansion of $\ell(\boldsymbol{\pi})$ is given by

$$\ell(\boldsymbol{\pi}') = \ell(\boldsymbol{\pi}) + \mathbf{g}^T\boldsymbol{\eta} + \frac{1}{2}\boldsymbol{\eta}^T\mathbf{H}\boldsymbol{\eta} + O(\|\boldsymbol{\eta}\|^3). \tag{2.21}$$

Making use of the matrix $\mathbf{S}$, the difference in log-likelihood values is given by:

$$\ell(\boldsymbol{\pi}') - \ell(\boldsymbol{\pi}) = \mathbf{1}^T\mathbf{S}\boldsymbol{\eta} - \frac{1}{2}\boldsymbol{\eta}^T\mathbf{S}^T\mathbf{S}\boldsymbol{\eta} + O(\|\boldsymbol{\eta}\|^3) \tag{2.22}$$

By dropping the cubic and higher order terms, and re-arranging, Wang (2008) expressed this by the following approximation:

$$\ell(\boldsymbol{\pi}') - \ell(\boldsymbol{\pi}) \approx -\frac{1}{2}\left\|\mathbf{S}\boldsymbol{\pi}' - \mathbf{2}\right\|^2 + \frac{n}{2} \tag{2.23}$$

where $\mathbf{2}$ is a vector of twos.

To use this approximation to maximise $\ell(\boldsymbol{\pi}')$ in the neighbourhood of $\boldsymbol{\pi}$, Wang (2007c, 2008) suggests applying a numerically stable method from Haskell and Hanson (1981), as follows. A new value $\boldsymbol{\pi}'$ can be found by solving the following least squares expression with non-negativity constraints:

$$\text{minimise} \quad \left|\mathbf{1}^T\boldsymbol{\pi}' - 1\right|^2 + \gamma\left\|\mathbf{S}\boldsymbol{\pi}' - \mathbf{2}\right\|^2 \quad \text{subject to } \boldsymbol{\pi}' \geqslant \mathbf{0}, \tag{2.24}$$

for some small value of γ such as 10^{-6}. By using an idea from Dax (1990) a better approach without the arbitrary constant γ is provided by:

$$\text{minimise} \quad \left|\mathbf{1}^T\boldsymbol{\pi}' - 1\right|^2 + \left\|\mathbf{Z}\boldsymbol{\pi}'\right\|^2 \quad \text{subject to } \boldsymbol{\pi}' \geqslant \mathbf{0}, \tag{2.25}$$

for a matrix $\mathbf{Z}$ that contains columns $\mathbf{z}_j$ defined by

$$\mathbf{z}_j = \mathbf{s}_j - (\mathbf{S}\boldsymbol{\pi} + \mathbf{1}). \tag{2.26}$$

Note that the matrix $\mathbf{S}$ is required but the Hessian matrix is not.

Lawson and Hanson (1974) provided a Fortran implementation of their non-negative least squares (NNLS) algorithm, which the CNM algorithm uses to solve the above problem at each iteration. The CNM algorithm also makes use of the stopping criterion and line search described in Section 2.6.

2.5 Other existing algorithms

This section describes some other existing methods for solving the NPMLE allocation problem within reasonable computation times.

2.5.1 Expectation Maximisation

Censored observations can be considered as a missing data problem. The popular Expectation-Maximisation (EM) algorithm (Turnbull, 1976; Dempster et al., 1977) is often used to deal with missing data. It consists of two steps, the E step (for expectation) and the M step (for maximisation).

Much has been written about the EM algorithm. Very briefly, it works as follows. During the Expectation step, missing data is replaced by its expected value based on the current parameter estimates. The Maximisation step locates a new parameter that maximises the likelihood using the complete data and expected values of the missing data. These two steps are repeated alternately in an iterative way until a stopping criterion is satisfied. A common stopping criterion is that the algorithm will stop when the movement of the solution from one iteration to the next stabilises to within a predefined tolerance threshold.

However, the performance of EM is known to be very poor when applied to interval-censored survival data (Pilla and Lindsay, 2001; Wang, 2007b).

Enhancements to EM

Pilla and Lindsay (2001) investigated the problem of maximising the likelihood of a nonparametric mixture with a finite number of components (as has been identified in Section 2.2.1 above). They reported that the convergence of the conventional EM algorithm was extremely slow in situations where the component densities are poorly separated and when the final solution requires some of the components to have zero probability mass. Both of these situations occur with interval-censored survival data. The reason given in the latter case was that the EM algorithm cannot reach the boundary point that would set the mass of a component to zero. Therefore, they focused their strategy on ways of improving the EM algorithm.

The first strategy they reported was the 'paired complete data EM algorithm' or 'paired EM' for short. This approach placed consecutive pairs of component densities together, thereby reducing by half the number of components that the EM algorithm must deal with. Within each pair, paired EM used a single step from the univariate Newton-Raphson method to reallocate probability between the two elements of the pair. To reallocate probability mass among these pairs, the conventional EM algorithm was used. For this strategy, an acceleration factor of between two and nine was reported.

The next strategies proposed (Pilla and Lindsay, 2001, sections 4·2 and 4·3) were based on cycles. Their 'rotation cycles' extended the concept of pairs in paired EM by regrouping the component densities into different pairs at each iteration. Specifically, at all odd iterations, components were paired as $\{(f_1, f_2), (f_3, f_4), \ldots, (f_{m-1}, f_m)\}$ and at even iterations the pairing was shifted to $\{(f_2, f_3), (f_4, f_5), \ldots, (f_m, f_1)\}$. The resulting 'rotated EM' algorithm was reported to be highly effective because it allowed probability mass to be efficiently transported throughout the set of component densities.

To extend the idea of rotation cycles further, the 'paired block EM' algorithm was developed. In this algorithm, the full support set was partitioned into an arbitrary (but even) number of blocks of consecutive components. Empirical results for paired block EM were not provided, as it was seen as a step towards the next improvement.

Finally, the 'hierarchical paired complete data EM algorithm' or simply 'hierarchical EM' was proposed (Pilla and Lindsay, 2001, section 4·4). It was based on a hierarchical implementation of paired EM. Pairs of component densities were themselves grouped into pairs, which were grouped into pairs repeatedly in a hierarchical fashion. The advantage of this approach was to reduce the workload assigned to the conventional EM algorithm and rely more on the Newton-Raphson solution with each pair. When enhanced with the rotation idea, hierarchical EM became 'composite EM'. Acceleration factors of up to 500 were reported for these two algorithms. The 'hierarchical EM' algorithm is similar to the new Hierarchical Constrained Newton Method (HCNM, introduced in Chapter 3) with a block size of 2. Results of simulations using HCNM with blocks of size 2 indicate that this is not a good choice of block size (Chapter 4).

Results reported by Pilla and Lindsay (2001) were analysed with a focus on the rate of convergence, given that the order of convergence of EM is linear. Their investigations looked only at relatively small data sets, with $m = 32$ and $m = 64$ mixture components. In terms of the order of convergence, the EM algorithm is inferior to the CNM algorithm, since CNM has quadratic convergence.

In practical applications of an algorithm, particularly when it must be used a large number of times, the mean computation time is of considerable interest. This takes into account both the order of convergence and the time complexity per iteration.

2.5.2 Iterative convex minorant method

The Iterative Convex Minorant (ICM) algorithm was introduced by Groeneboom (1991) and was studied by Groeneboom and Wellner (1992) and Jongbloed (1998).

For a set of points in the plane, the 'convex minorant' is a curve consisting of line segments running under all the points from the first to the last, such that all the points lie on or above the curve. The convex minorant can be visualised as an elastic string, tightly stretched under all the points.

Let $\mathbf{q} = (q_0, \ldots, q_m)^T$ be the vector of cumulative probabilities of $\boldsymbol{\pi}$, defined by $q_0 = 0$ and

$q_j = q_{j-1} + \pi_j$ for $j > 0$ (thus $q_m = 1$). The ICM algorithm uses a reparametrised log-likelihood function, expressed in terms of these cumulative probabilities:

$$\ell(\mathbf{q}) = \sum_{i=1}^{n} \log\left(\sum_{j=1}^{m} \delta_{ij}(q_j - q_{j-1})\right) \tag{2.27}$$

By using a quadratic approximation of this function in the neighbourhood of an estimate $\mathbf{q}$, the ICM algorithm locates a new estimate $\mathbf{q}'$. The new estimate is subjected to the monotonicity constraint, so that $q'_j \geqslant q'_{j-1}$ for $j = 1, \ldots, m$. When a diagonal Hessian approximation is used, this approach turns the problem into an isotonic regression problem, for which several algorithms are available. An example is the pool-adjacent violators algorithm (Ayer et al., 1955; Grotzinger and Witzgall, 1984)

For each new estimate $\mathbf{q}'$, a line search is performed, to ensure that the log-likelihood actually increases. The algorithm is therefore guaranteed to converge (Jongbloed, 1998). When a new estimate has been found, a new quadratic approximation is formed about it. The algorithm repeats this process iteratively until stopping criteria are satisfied.

Hybrid of ICM and EM

Wellner and Zhan (1997) created a hybrid algorithm, combining the strengths of the ICM algorithm with the consistency of the EM algorithm. They noted that the EM algorithm may converge extremely slowly in cases of more heavily censored data. Their new hybrid ICM-EM algorithm alternates between an iteration of ICM and then an iteration of EM. It performed very well, out-performing either of the individual algorithms remarkably. The book by Sun (2006, Section 3.4) provides an excellent description of these three algorithms; EM, ICM and the ICM-EM hybrid.

Wang (2008) described each of these algorithms, and then went on to further enhance the ICM-EM hybrid algorithm by building in the dimension reduction technique into the ICM step. The resulting enhanced ICM-EM hybrid algorithm was labeled ICMDR-EM. Results from that paper indicate that ICMDR-EM was the best choice of algorithm in situations of large data sets ($n = 1600$) and large proportions of exact observations ($r > 0.5$). In such situations the performance of ICM-EM was comparable to ICMDR-EM. With so many exact observations, the dimension reduction technique was unable to make a discernable difference. The investigations in Chapter 6 include the enhanced hybrid ICMDR-EM algorithm.

2.5.3 Subspace-based Newton method

Dümbgen et al. (2006) investigated the problem of estimating the survival function from 'mixed-case' interval-censored data. They investigated three different types of nonparametric estimator. As well as examining the unrestricted NPMLE (as described in Section 2.1.3) they also looked at two constrained approaches. One was to constrain the distribution of event times to being concave and the other was to assume the distribution was unimodal. For comparison to the research presented in this thesis, only the unconstrained NPMLE is considered.

In this paper (Dümbgen et al., 2006) the authors proposed a Subspace-based Newton (SBN) method for solving the NPMLE allocation problem. To eliminate the equality constraint, the authors proposed a modified objective function that included a Lagrangian term in the log-likelihood function. This Lagrangian term automatically ensured that $\sum_j \pi_j = 1$ when the objective function was optimised. Their algorithm is based on a quadratic approximation to the objective function, or more precisely, to the difference from one iteration to the next.

Wang (2008) summarised their method and investigated the performance of the SBN algorithm in more detail. He implemented the SBN algorithm and enhanced it to include the dimension reduction technique. This new algorithm was labeled SBNDR. Results of this enhanced algorithm were promising, with performance being generally better than ICM-EM and CNM in some situations and worse in others. The investigations in Chapter 6 include the SBNDR algorithm.

2.6 Line search and stopping criteria

This section introduces two related topics that apply generally to several algorithms. The iterative algorithms discussed in this chapter create a series of estimates of the unknown vector $\hat{\boldsymbol{\pi}}$, which maximises the log-likelihood function $\ell(\boldsymbol{\pi})$ over the unit simplex. The first topic is a line search, which provides a guarantee that an algorithm will converge. The second topic looks at criteria that determine when an algorithm should terminate.

Line search

The line search is a technique that is used to ensure that the value of the log-likelihood function increases with each successive iteration. This provides a guarantee that an algorithm will convergence to the solution (Jongbloed, 1998).

At each iteration, an algorithm derives a new estimate $\boldsymbol{\pi}'$ of the probability masses for the candidate support intervals. This estimate should be closer to the solution $\hat{\boldsymbol{\pi}}$ than the previous estimate $\boldsymbol{\pi}$ was, in the sense that it should provide a higher log-likelihood value. Typically, when an algorithm is working well, the log-likelihood value increases rapidly. But sometimes, especially when the estimates are getting close to the solution, the log-likelihood may not increase. This can happen because of approximations in the algorithm, such as using a quadratic approximation. Different algorithms derive $\boldsymbol{\pi}'$ in different ways that generally do not guarantee that $\ell(\boldsymbol{\pi}') > \ell(\boldsymbol{\pi})$. For example, the new estimate may overshoot the true maximal point $\hat{\boldsymbol{\pi}}$, which hints at there being a better estimate somewhere in between the two points.

When $\ell(\boldsymbol{\pi}') \leqslant \ell(\boldsymbol{\pi})$ a line search is required. The EM algorithm does not require a line search, because of its built-in consistency. However, all the other algorithms require this step.

The line search involves searching the line between the points $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ to find a point where the log-likelihood value is greater than $\ell(\boldsymbol{\pi})$. Denoting the difference between the estimates by $\boldsymbol{\eta} = \boldsymbol{\pi}' - \boldsymbol{\pi}$, the line between them is defined by

$$\boldsymbol{\pi} + \sigma\boldsymbol{\eta}, \quad \text{for } 0 \leqslant \sigma \leqslant 1. \tag{2.28}$$

Since $\ell(\boldsymbol{\pi})$ is concave and the vector $\boldsymbol{\eta}$ provides an ascent direction away from $\boldsymbol{\pi}$, there must be a point somewhere on the line that provides a higher log-likelihood value. The common step-halving technique is used to search this line. Values of $\sigma = 2^{-k}$ for each $k = 1, \ldots, 30$ are attempted. The first to satisfy the condition

$$\ell(\boldsymbol{\pi} + \sigma\boldsymbol{\eta}) > \ell(\boldsymbol{\pi}) + \tfrac{1}{3}\sigma\nabla\ell(\boldsymbol{\pi})^T\boldsymbol{\eta} \tag{2.29}$$

provides a replacement value for $\boldsymbol{\pi}' = \boldsymbol{\pi} + \sigma\boldsymbol{\eta}$.

From a practical perspective, the limited precision that computers use to store real numbers has an impact on the line search. Sometimes the line search can fail to find a suitable new point.

This happens when the estimated log-likelihood value is very close to the true value, within the precision that is possible on the computer. If none is found, the algorithm terminates, although this situation was seldom observed during this research.

Stopping criteria

The accuracy of an estimate $\boldsymbol{\pi}$ of the optimum point $\hat{\boldsymbol{\pi}}$ is measured by the difference between the function values at the solution and at the estimate, $\ell(\hat{\boldsymbol{\pi}}) - \ell(\boldsymbol{\pi})$. The primary criterion that is used to decide when an algorithm should stop is based on this difference.

Since $\ell(\hat{\boldsymbol{\pi}})$ is not known, a known upper bound on this difference is used. The vertex-directional gradient vector at an estimate $\boldsymbol{\pi}$, denoted by $\mathbf{d}$, was defined above in Eq. (2.16). The maximum component of $\mathbf{d}$, denoted by $d_{max} = \max(d_1, \ldots, d_m)$, provides the upper bound (Wang, 2008), thus:

$$\ell(\hat{\boldsymbol{\pi}}) - \ell(\boldsymbol{\pi}) \leqslant d_{max}\ . \tag{2.30}$$

This upper bound indicates how accurate an estimate $\boldsymbol{\pi}$ is, regardless of which algorithm was used to produce the estimate.

At each successive iteration of an algorithm, the value of d_{max} decreases towards zero as the estimates approach the true solution. For all the algorithms investigated in this research, the primary stopping criterion is a tolerance threshold τ applied to d_{max} suitably scaled to allow for the magnitude of $\ell(\boldsymbol{\pi})$ (Wang, 2007c). Specifically, the algorithms terminate when the following condition is satisfied:

$$\frac{d_{max}}{\left|\ell(\boldsymbol{\pi})\right|} \leqslant \tau. \tag{2.31}$$

The value of τ can be specified by the user, but a default value of 10^{-6} seemed to work well in simulations.

Another more prosaic stopping criterion is also built into algorithms. It is a simple upper limit imposed by the user on the number of iterations. When an algorithm has reached this limit, it terminates even though it may not have found the solution yet.

2.7 Summary

This chapter defined the specific research problem that has been addressed in this thesis and examined the background to the problem. The aim of the research presented in this thesis was to develop a new algorithm to solve a particular problem in the area of survival analysis. The computational goal of the new algorithm is to derive the nonparametric maximum likelihood estimate (NPMLE) of the survival function, given a survival data set where some or all of the event times are censored by arbitrary intervals in the positive real line.

Existing algorithms for solving the NPMLE allocation problem have been discussed in this chapter. Wang (2008) investigated the algorithms ICM, ICMDR, ICM-EM, ICMDR-EM, SBN, SBNDR and CNM for data sets with parameters $n \in \{400, 1600\}$ and $r \in \{0, 0.1, 0.3, 0.5, 0.7, 0.9\}$. The parameter n represented the number of observations and the parameter r represented the proportion of observations that were exactly measured (thus a proportion $1-r$ were interval-censored).

Table 2.1 summarises some of the results presented in Wang (2008), showing the mean computation times over 10 replications. For each situation, the algorithm that produced the fastest mean time is indicated in the right-most column.

Table 2.1: Mean computation time (seconds) of five existing algorithms.

n	r	ICM-EM	ICMDR-EM	SBN	SBNDR	CNM	Best algorithm
400	0.0	0.62	0.24	0.45	0.26	0.08	CNM
	0.1	0.91	0.45	0.37	0.18	0.10	CNM
	0.3	1.34	0.91	0.29	0.24	0.24	SBNDR/CNM
	0.5	1.35	1.18	0.23	0.28	0.45	SBN
	0.7	1.07	1.07	0.36	0.45	0.68	SBN
	0.9	0.77	0.84	0.55	0.58	0.84	SBN
1600	0.0	6.80	2.97	15.08	3.30	1.20	CNM
	0.1	16.99	8.33	7.92	3.41	2.17	CNM
	0.3	17.53	13.01	8.17	5.13	9.47	SBNDR
	0.5	13.84	12.16	13.12	12.56	21.85	ICMDR-EM
	0.7	10.71	10.75	17.51	18.27	36.65	ICM-EM
	0.9	8.04	8.73	24.09	24.89	45.94	ICM-EM

In some of these situations, the number of replications was insufficient to definitively say which algorithm is best. In particular, when r is large, those versions with dimension reduction may be substantially the same as those without it.

However, it is clear that no single algorithm is the fastest in all situations. This is the motivation for the research presented in this thesis. Chapter 3 introduces the new algorithm, which was developed to address this. The strategy that was chosen was to enhance the CNM algorithm to create the hierarchical constrained Newton method.

Chapter 3

Hierarchical Constrained Newton Method

This chapter describes a new algorithm called the Hierarchical Constrained Newton Method (HCNM). The algorithm is applicable to the problem described in Section 2.1. Specifically, the algorithm finds the nonparametric maximum likelihood estimate (NPMLE) of the survival function, from data containing both exact observations and general interval-censored observations in any proportion.

The new HCNM algorithm is based on an algorithm called the Constrained Newton Method (CNM). The term *hierarchical* refers to the way the new algorithm divides the problem up into pieces in order to solve it. It also refers to the way the HCNM algorithm uses itself recursively in a hierarchical (tree-like) fashion.

Section 3.1 gives some background to the new algorithm, including a brief description of its purpose and the algorithm that it was based on. Section 3.2 discusses aspects of the problem that motivate the research and the new algorithm. Section 3.3 explains how the new algorithm differs from its predecessor. Section 3.4 provides the new algorithm in pseudo-code.

3.1 Background

The CNM algorithm was originally published by Wang (2007c) in the general context of mixture distributions. Then Wang (2008) applied CNM to the survival analysis problem under consideration here, with the result that it performed well in some situations and poorly in others. Consequently, the aim of the research presented in this thesis was to enhance the CNM algorithm to create an algorithm that performed well in all situations.

Purpose

Both the CNM and HCNM algorithms are intended to find the NPMLE survival function from interval-censored data. The task of finding the NPMLE solution involves distributing probability mass among a potentially large set of candidate support intervals. This is the 'allocation' step of finding the NPMLE, which was described in Subsection 2.1.3.

The original observations (exact or interval-censored) from a survival data set are represented by intervals O_i for $i = 1, \ldots, n$. For constructing the NPMLE, a set of m candidate support intervals I_j are derived from those intervals. The intervals in $\{I_j\}$ are mutually disjoint.

Unit probability mass is distributed among these candidate support intervals. Each candidate support interval I_j is assigned a probability mass π_j. The survival function $S(t)$ is derived from these, as follows. It starts at $S(0) = 1$ and ends at $S(\infty) = 0$ with a step down of π_j at each candidate support interval I_j. Some, possibly many, of the candidate support intervals may receive zero probability mass.

The $n \times m$ clique matrix $\mathbf{A}$ is constructed to identify which candidate support intervals intersect (and therefore can support) each of the original observations. The value in the i^{th} row and j^{th} column is δ_{ij} with the value 1 if $I_j \subseteq O_i$ or zero if $I_j \cap O_i = \emptyset$ (these are the only two possibilities, because of the way the intervals I_j are constructed). The clique matrix is used by both algorithms in creating the initial dimension-reduced subset of $\{I_j\}$ (Wang, 2008) and in calculating the log-likelihood.

Both CNM and HCNM use the non-negative least squares (NNLS) algorithm provided by Lawson and Hanson (1974) to adjust the values of $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_m)$ at each iteration. Section 2.4 discusses the CNM algorithm, including its use of the NNLS algorithm.

Stopping criteria for algorithms were discussed in Section 2.6. The HCNM algorithm uses the same stopping criteria as the CNM algorithm. The main stopping criterion is based on the maximum vertex-directional gradient, as defined in Eq. (2.31). The algorithms also terminate if the number of iterations has reached the limit set by the user, or the limit of precision possible on the computer has been reached.

Finite mixture models

Finite mixture models were discussed in Section 2.2. They were applied to the problem of finding the nonparametric maximum likelihood estimate (NPMLE) of the survival function from interval-censored data.

This NPMLE problem can be expressed as a finite mixture model, as follows. The component densities of the mixture are defined by the conditional probabilities $\Pr[T \in O_i | T \in I_j] = \delta_{ij}$. The mixing proportions are given by the probability masses in $\boldsymbol{\pi}$. An observation of an event in the time interval O_i is interpreted as having come from one of the mixture components (candidate support intervals) for which that is possible, namely those with $\delta_{ij} = 1$. So the probability p_i of an event occurring in the time interval O_i can be expressed as a mixture density:

$$p_i = \Pr[T \in O_i] = \sum_{j=1}^{m} \pi_j \Pr[T \in O_i | T \in I_j] = \sum_{j=1}^{m} \pi_j \delta_{ij} \tag{3.1}$$

Thus the vector of observation probabilities $\mathbf{p} = (p_1, \ldots, p_n)^T$ is given by $\mathbf{p} = \mathbf{A}\boldsymbol{\pi}$. This is used in the log-likelihood function, by both the CNM and HCNM algorithms.

Dimension reduction technique

The dimension reduction technique (Wang, 2008) was described in Section 2.3. It provides a means of reducing the size of the set $\{I_j\}$ by defining a subset of the best candidates. The subset varies with each iteration of an algorithm. At the start, it is defined to be very small, but big enough to provide support to all the observations. Additional candidate support intervals are added at each iteration, allowing the subset under consideration to grow rapidly if necessary.

This technique is an essential component of the HCNM algorithm, as it is for the CNM algorithm. It provides a significant improvement in computation times over an algorithm that does not use it. These performance gains are seen particularly for data with a low proportion of exact observations (Wang, 2008). On the other hand, the worst performance of the CNM algorithm was with data containing a high proportion of exact observations. This latter situation is the focus of the enhancements provided by the new HCNM algorithm.

The use of the dimension-reduction technique within HCNM is the same as in CNM, and it does not materially affect the theory behind the enhancements that HCNM adds to CNM. To simplify the notation and discussion in this chapter, no distinction has been made between the full set $\{I_j : j = 1, \ldots, m\}$ of candidate support intervals, and the dimension-reduced subsets thereof. The same applies to the associated probability masses $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_m)^T$ and the columns of the matrices $\mathbf{A}$ and $\mathbf{S}$. This is because all candidate support intervals that are not in the dimension-reduced subset have zero probability mass, so they do not affect the log-likelihood calculations.

3.2 Research motivation

This section discusses aspects of the problem that motivate the research and the new algorithm.

3.2.1 Quadratic approximation

Section 2.4 described the CNM algorithm in some detail. Key to the way it works is the quadratic approximation, which is derived from the Taylor series of the log-likelihood function.

The quadratic approximation is derived from the first two derivatives of the function $\ell(\boldsymbol{\pi})$. The first derivative of the log-likelihood function $\ell(\boldsymbol{\pi})$ with respect to $\boldsymbol{\pi}$ is $\mathbf{g} = \nabla\ell(\boldsymbol{\pi})$, which gives the gradient of the function. The vector $\mathbf{g}$ has the same dimension as the vector $\boldsymbol{\pi}$, namely $\mathbf{g} \in \mathbb{R}^m$. The Hessian matrix $\mathbf{H} = \nabla^2\ell(\boldsymbol{\pi})$ is the second derivative and it indicates the curvature of the function. The Hessian matrix can be an enormous object, since $\mathbf{H} \in \mathbb{R}^m \times \mathbb{R}^m$. Eq. (2.20) defines $\mathbf{H}$ in terms of the matrix $\mathbf{S}$, as $\mathbf{H} = -\mathbf{S}^T\mathbf{S}$. Since the matrix $\mathbf{S}$ has full column rank, the Hessian matrix is negative-definite, and so the log-likelihood function is concave. It has a unique finite global maximum over the unit simplex (Gentleman and Geyer, 1994; Lindsay, 1995; Böhning et al., 1996). The Hessian contains only negative and zero values.

The quadratic approximation is given by

$$\ell(\boldsymbol{\pi}') - \ell(\boldsymbol{\pi}) \approx \mathbf{g}^T\boldsymbol{\eta} + \frac{1}{2}\boldsymbol{\eta}^T\mathbf{H}\boldsymbol{\eta} \tag{3.2}$$

where $\boldsymbol{\eta} = \boldsymbol{\pi}' - \boldsymbol{\pi}$ provides an ascent direction away from $\boldsymbol{\pi}$. This approximation enables the rapid location of the global maximum at $\hat{\boldsymbol{\pi}}$. Ideally, the full Hessian matrix should be calculated at every iteration in order to provide the best quadratic approximation. However, it is computationally expensive to compute the full Hessian matrix (Wang, 2008) and indeed this is not necessary. This can be seen by observing that Eq. (2.23) uses the matrix $\mathbf{S}$ but not $\mathbf{S}^T$ or $\mathbf{H}$.

Figure 3.1 provides an illustration of a typical Hessian matrix (of size 72×72) that might be encountered during an algorithm that solves the NPMLE problem with interval-censored data. Heat-map colours are used to indicate the values of the elements of the matrix, with dark colours indicating values close to zero and bright colours indicating values close to the minimum (largest

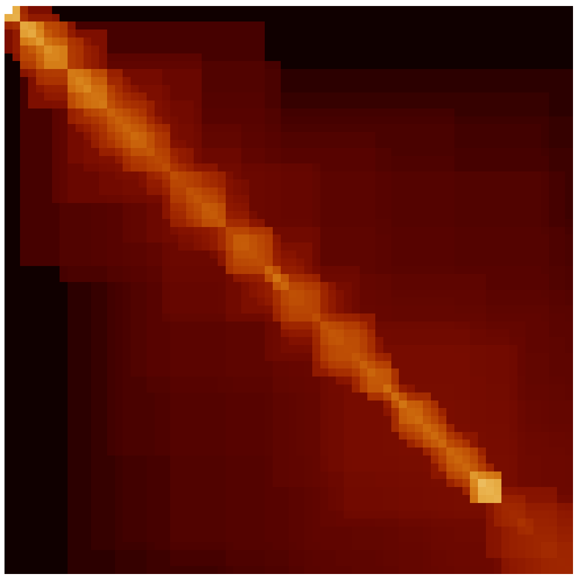


Figure 3.1: Illustration of a Hessian matrix, using heat-map colours.

negative) value in the matrix. Data for this illustration come from a New Zealand study of manuka honey proposed as an aid to the healing of leg ulcers (Jull et al., 2008), which was also used in Section 2.1.2.

Most of the curvature information is located on or near the diagonal. Because of this observation, an approximation to the Hessian can be used that does not require all the values to be computed. The Hessian can be approximated by calculating just the values in a diagonal band close to the main diagonal, while assuming that those further from the diagonal are zero. One way of doing this is to partition the set $\{I_j\}$ into a number of separate blocks, which is what the HCNM algorithm does. Several different ways of creating these blocks are shown in Figure 4.3 (on page 47). The figure shows which parts of the Hessian matrix are used by the different block structures. More details on this are provided in Chapter 4.

3.2.2 Mixture of mixtures model

A finite mixture model uses a probability distribution that is constructed from a number of other probability distributions, as a weighted combination of them. Each component can itself be a finite mixture. So, a finite mixture model can be expressed recursively as a finite mixture of finite mixture models.

The density function of a finite mixture of m components is given by

$$f(x) = \sum_{j=1}^{m} \pi_j f_j(x) \tag{3.3}$$

as in Eq. (2.8). The 'mixture of mixtures' model is constructed as follows.

Suppose the set of m mixture components is partitioned into κ subsets, called blocks. This can be achieved, for example, by breaking the set at $\kappa - 1$ points along it. Let the $m \times \kappa$ matrix $\mathbf{B}$ define which block contains each component. The element in the j^{th} row and b^{th} column of $\mathbf{B}$ is denoted by β_{jb} and has the value 1 if the j^{th} component is in the b^{th} block, or zero otherwise. Each component belongs to exactly one block (and no block is empty).

Note that a more general mixture of mixtures model is possible by allowing the j^{th} component to belong in proportions to more than one block. The values in the block membership matrix $\mathbf{B}$ can be any $\beta_{jb} \in [0, 1]$ as long as the rows sums are $\mathbf{1}^T\mathbf{B} = \mathbf{1}$. Since these blocks overlap with each other, they are called *fuzzy* blocks.

The mixture of mixtures model is composed of a mixture of κ components, one for each block. That is the 'outer' layer of the mixture. Each of these components is a mixture, composed of the individual components that make up that block. Those are the 'inner' layer of the model. Let ω_b denote the combined probability mass for the b^{th} block, so that $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_\kappa)^T = \boldsymbol{\pi}^T\mathbf{B}$.

Thus Eq. (3.3) can be expressed as:

$$f(x) = \sum_{b=1}^{\kappa} \sum_{j=1}^{m} \beta_{jb}\pi_j f_j(x) = \sum_{b=1}^{\kappa} \omega_b \sum_{j=1}^{m} \frac{\beta_{jb}\pi_j}{\omega_b} f_j(x) = \sum_{b=1}^{\kappa} \omega_b f_b'(x). \tag{3.4}$$

This defines the outer layer of the model as a mixture of κ components with component densities $f_b'(x)$ and mixing proportions ω_b.

The component density $f_b'(x)$ is the conditional probability of x given the b^{th} block, given by

$$f_b'(x) = \sum_{j=1}^{m} \frac{\beta_{jb}\pi_j}{\omega_b} f_j(x) = \sum_{j=1}^{m} \phi_{jb} f_j(x) \tag{3.5}$$

where $\phi_{jb} = \frac{\beta_{jb}\pi_j}{\omega_b}$. This is the inner layer of the model for the b^{th} block. Note that it has fewer than m components, since only components with $\beta_{jb} > 0$ are needed.

This mixture of mixtures model provides the basis of the divide and conquer approach used by the HCNM algorithm. Note that this model can be extended hierarchically and recursively to give a mixture of a mixture of a mixture of a mixture of a mixture of . . .

3.3 Enhancements to CNM

The HCNM algorithm is an extension of the CNM algorithm. There are two ideas that HCNM adds to its predecessor CNM. The first idea is to split a large set of candidate support intervals into a number of smaller subsets, called blocks. The second idea is the use of recursion in the algorithm.

Pilla and Lindsay (2001) enhanced the EM algorithm using blocks and hierarchical cycles (see Subsection 2.5.1). They explained how this approach allows probability mass to be efficiently transported among the support intervals. The HCNM algorithm extends their ideas by applying them within the context of the CNM algorithm.

The HCNM algorithm partitions the dimension-reduced set of candidate support intervals into a number of separate blocks. It considers each block in turn. Within each block, HCNM redistributes probability mass among the support intervals using the NNLS algorithm (Lawson and Hanson, 1974). The CNM algorithm also uses the NNLS algorithm to redistribute probability mass. However, it does this with a single call to NNLS at each iteration, using all the intervals that are under consideration at that point. In the context of the CNM and HCNM algorithms, the time complexity of the NNLS algorithm is such that the number of probabilities that need to be estimated has a particularly detrimental effect on the computation time. Therefore, the divide and conquer approach used in HCNM saves considerable time by keeping this number small.

When the redistribution of probability *within* all blocks has been completed, the HCNM algorithm then reallocates probability mass *among* the blocks, treating each block as a whole. It does this by calling itself recursively, making use of the mixture of mixtures model.

This two-step process of solving the within-block and then among-blocks problems is repeated until one of the stopping criteria is reached. The number of blocks and their sizes are determined dynamically, using rules based on simulations (see Chapter 4). The block size is chosen so that both parts of the iteration contribute efficiently to the rapid growth of the log-likelihood estimate.

3.3.1 Blocks

Suppose the set $\{I_j\}$ of candidate support intervals is partitioned into κ blocks. The blocks are created by dividing the set of m intervals at $\kappa - 1$ approximately evenly spaced positions. This automatically places neighbouring intervals together, which is important because these neighbours are (in a sense) competing with each other for the right to support some of the interval-censored observations.

Let the $m \times \kappa$ matrix $\mathbf{B}$ represent the membership of candidate support intervals in blocks. Each row represents a candidate support interval and each column represents a block. The element in the j^{th} row and b^{th} column of $\mathbf{B}$ is denoted by β_{jb}. In the simplest case, when every candidate support interval belongs to exactly one block, the values in $\mathbf{B}$ are $\beta_{jb} = 1$ if I_j belongs to the b^{th} block or zero otherwise. As mentioned in Subsection 3.2.2, more complicated situations involving *fuzzy* blocks are possible, which are explored in Section 4.4. For the purpose of description in this section, the simplest case is assumed.

Let $m_b = \sum_{j=1}^{m} \mathbf{I}(\beta_{jb} > 0)$ denote the number of support intervals in the b^{th} block. Note that $\sum_{b=1}^{\kappa} m_b = m$ in the simplest case, whereas for fuzzy blocks the sum can be greater since some intervals belong to more than one block.

Allocation within a block

The b^{th} block consists of a union of its m_b candidate support intervals and is denoted by U_b as follows:

$$U_b = \bigcup_{j:\beta_{jb}>0} I_j \tag{3.6}$$

The probability of an event time in U_b gives the combined weight of that block. This is denoted by ω_b and is given by the sum of the weights of its candidate support intervals:

$$\omega_b = \Pr[T \in U_b] = \sum_{j=1}^{m} \pi_j \beta_{jb} \tag{3.7}$$

which is the b^{th} element of the vector $\boldsymbol{\pi}^T\mathbf{B}$. Within a block, the conditional probability of I_j given U_b is:

$$\Pr[T \in I_j | T \in U_b] = \frac{\Pr[T \in I_j \text{ and } T \in U_b]}{\Pr[T \in U_b]} = \frac{\beta_{jb}\pi_j}{\omega_b} \tag{3.8}$$

This allows HCNM to focus on a single block, finding a locally better allocation of probability mass for the support intervals within the block. Let $\boldsymbol{\pi}_b$ denote the vector (of length m_b) of probability masses of candidate support intervals in U_b, given by $\boldsymbol{\pi}_b = (\pi_j : \beta_{jb} > 0)$. The column vectors $\mathbf{s}_j$ were defined by their elements in Eq. (2.15). Let the $n \times m_b$ matrix $\mathbf{S}_b$ be constructed by binding together the vectors $\{\mathbf{s}_j : \beta_{jb} > 0\}$. This is potentially a much narrower matrix than $\mathbf{S}$.

The HCNM algorithm adjusts the probability masses within the block U_b in the same way that CNM does for the entire (dimension-reduced) set. The total probability mass in the block is ω_b, as it must remain when a new allocation of probability masses has been completed within this block.

Let the vector $\mathbf{z}_j$ be defined by

$$\mathbf{z}_j = \mathbf{s}_j - \frac{1}{\omega_b}(\mathbf{S}_b\boldsymbol{\pi}_b + \mathbf{1}) \tag{3.9}$$

and the $n \times m_b$ matrix $\mathbf{Z}_b$ be defined by binding together those vectors $\{\mathbf{z}_j : \beta_{jb} > 0\}$. Then the HCNM algorithm solves the following non-negative least squares problem, using an idea from Dax (1990) as in Section 2.4, for the current block U_b:

$$\text{Find } \mathbf{x} \in \mathbb{R}^{m_b} \text{ to minimise } \quad \left|\mathbf{1}^T\mathbf{x} - \omega_b\right|^2 + \left\|\mathbf{Z}_b\mathbf{x}\right\|^2 \quad \text{subject to } \mathbf{x} \geqslant \mathbf{0}. \tag{3.10}$$

A new allocation $\boldsymbol{\pi}'_b$ within the block can then be derived from the solution $\mathbf{x}$ to that problem. For the simple non-overlapping block situation, $\frac{\omega_b}{\mathbf{1}^T\mathbf{x}}\mathbf{x}$ gives the new $\boldsymbol{\pi}'_b$ directly. In the case of fuzzy blocks, the new values are given by

$$\boldsymbol{\pi}'_b \leftarrow \boldsymbol{\pi}_b + \mathbf{B}_b(\frac{\omega_b}{\mathbf{1}^T\mathbf{x}}\mathbf{x} - \boldsymbol{\pi}_b) \tag{3.11}$$

where the b^{th} column of $\mathbf{B}$ is denoted by $\mathbf{B}_b$, which is multiplied element-by-element with the vector following it.

Thus the problem of adjusting the weights of the m candidate support intervals is replaced by a smaller problem of adjusting the weights of m_b intervals, subject to the constraint that the total must remain ω_b. Although there are now κ such problems to solve (rather than one), this approach saves considerable computation time because of the nature of the NNLS algorithm. In the estimated time complexity $O(nm^2)$ of NNLS (see Chapter 5) the factor m^2 is replaced by $\sum_{b=1}^{\kappa} m_b^2$ which is typically smaller and potentially much smaller.

Allocation among blocks

When the redistribution of probabilities within each block has been completed, the mixing components ω_b of the blocks can then be adjusted. This is achieved by considering each block as a single component of a mixture model. Optimising the values of $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_\kappa)^T$ also occurs within a much smaller dimension, since typically $\kappa \ll m$.

Let q_{ib} denote the conditional probability of an event time in O_i given that the event time is in the block U_b. The value of q_{ib} is given by:

$$q_{ib} = \Pr[T \in O_i | T \in U_b] = \frac{\Pr[T \in O_i \text{ and } T \in U_b]}{\Pr[T \in U_b]} = \frac{\sum_{j=1}^{m} \delta_{ij}\beta_{jb}\pi_j}{\omega_b} \tag{3.12}$$

The probability p_i of an event in O_i in Eq. (3.1) can be expressed as a mixture model, with one component for each block:

$$p_i = \Pr[T \in O_i] = \sum_{b=1}^{\kappa} \omega_b \Pr[T \in O_i | T \in U_b] = \sum_{b=1}^{\kappa} \omega_b q_{ib} \tag{3.13}$$

The values q_{ib} form an $n \times \kappa$ matrix $\mathbf{Q}$, which contains the conditional probabilities of the mixture component densities, one for each block. Using this matrix, a new set of block weights can be found by finding the NPMLE of the mixing components in $\boldsymbol{\omega}$. The HCNM algorithm is able to solve this problem. This leads to the next topic: recursion.

3.3.2 Recursion

The HCNM algorithm can be used to solve the more general problem of finding the NPMLE for any finite mixture model, given a sample of data. For this purpose, it requires a matrix of the component densities of the mixture. Each row represents one of the observations and each column represents one of the mixture components. The component densities are given as conditional probabilities (the clique matrix is a simple example, containing only zeros and ones). The element in the i^{th} row and j^{th} column gives the conditional probability of observing the i^{th} observation, given the j^{th} mixture component. With this matrix, the HCNM algorithm finds the nonparametric maximum likelihood estimate of the mixing proportions.

The problem of reallocating probability mass among the blocks has a mixture distribution, much like the original problem of allocating probability mass to the whole support set. Because of this similarity, the HCNM algorithm calls itself recursively to perform the reallocation among blocks.

3.4 HCNM in pseudo-code

Algorithm 1 Hierarchical Constrained Newton Method (HCNM)

```
Require: either vectors $\mathbf{L}$ and $\mathbf{R}$, or the matrix $\mathbf{A}$ of mixture probabilities with optional first
    guess of $\boldsymbol{\pi}$
Ensure: vector $\boldsymbol{\pi}$ maximises the nonparametric log-likelihood function.
 1: Create $\mathbf{A}$ if not supplied. {Derive clique matrix from $\mathbf{L}$ and $\mathbf{R}$}
 2: Derive initial weights $\boldsymbol{\pi}$ if not supplied. {Using the dimension reduction technique}
 3: for $i$ from 1 to $i_{max}$ do {Begin the main iterative loop}
 4:   $\mathbf{p} \leftarrow \mathbf{A}\boldsymbol{\pi}$ {Probabilities of observations}
 5:   $\ell \leftarrow \sum \log \mathbf{p}$ {Log-likelihood}
 6:   $\mathbf{d} \leftarrow$ vertex-directional gradient vector
 7:   if $d_{max}/|\ell(\boldsymbol{\pi})| \leqslant \tau$ then {The primary stopping condition}
 8:     break {from for loop}
 9:   end if
10:   $\kappa \leftarrow$ Decide how many blocks there should be.
11:   Create the matrix $\mathbf{B}$ of block memberships.
12:   $\boldsymbol{\omega} \leftarrow \boldsymbol{\pi}^T\mathbf{B}$ {Vector of block weights}
13:   for block $b$ from 1 to $\kappa$ do
14:     Create matrix and vector arguments for NNLS for block $b$.
15:     Call NNLS to allocate probability mass within block $b$.
16:     Adjust the values in $\boldsymbol{\pi}$ within block $b$.
17:   end for {Now there is a new estimate $\boldsymbol{\pi}'$}
18:   if $\ell(\boldsymbol{\pi}') \leqslant \ell(\boldsymbol{\pi})$ then
19:     Perform a line search, redefining $\boldsymbol{\pi}'$ so that $\ell(\boldsymbol{\pi}') > \ell(\boldsymbol{\pi})$.
20:     if line search failed then {Due to limited computer precision}
21:       break {from for loop}
22:     end if
23:     $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi}'$ {Update $\boldsymbol{\pi}$}
24:   end if
25:   Derive a matrix $\mathbf{Q}$ with elements $q_{ib} = \Pr[T \in O_i | T \in U_b]$.
26:   Reallocate mass among the blocks, by a recursive call to HCNM using $\mathbf{Q}$ and $\boldsymbol{\omega}$.
27:   Adjust values in $\boldsymbol{\pi}$ according to new block weights.
28: end for
29: return $\boldsymbol{\pi}$ {and info about the $I_j$ intervals}
```

Chapter 4

Tuning the HCNM Algorithm

This chapter details the process of development and fine-tuning that resulted in the Hierarchical Constrained Newton Method (HCNM). The new method is applicable to the problem of finding the nonparametric maximum likelihood estimate of a survival function, from survival data containing both exact observations and general interval-censored observations in any proportion.

The main idea that HCNM adds to its predecessor CNM is to split a large set of candidate support intervals into a number of smaller subsets, which we have called blocks. This approach extends idea of the rotating cycles and hierarchical cycles used by Pilla and Lindsay (2001) by applying it to the CNM algorithm. They explained how their techniques allowed probability mass to be efficiently transported from one block to the next.

This chapter presents numerical simulations that were conducted with the aim of optimising the performance of the HCNM algorithm. Section 4.1 describes the method that was used to generate simulated data for the purpose of running simulations. Section 4.2 mentions practical issues in running simulations on computers. Section 4.3 addresses the question of how many blocks there should be, and consequently what the optimal block size is. Section 4.4 explores some methods of creating the blocks, looking at the question of what form the blocks should take in terms of their sizes, positions and the way they might overlap. Finally, Section 4.5 explores an aspect of the use of recursion within HCNM.

4.1 Simulated data

The research presented in this chapter was based on simulated data. Wang (2008) simulated survival data containing both exact observations and interval-censored observations by using pseudo-random numbers from the exponential distribution. For the simulations presented in this chapter, the same method of generating data was used. This section gives a brief description of that method.

A survival data set consists of n observations of event times, denoted by t_i for $i = 1, \dots, n$

and assumed to be drawn independently from the same probability distribution. Initially, each t_i was drawn from an exponential distribution with a mean of one. For the exact observations, these were the values that were used. However, to create an interval-censored observation, the value t_i was replaced by an interval $(L_i, R_i]$ surrounding it. The true value is therefore hidden for the interval-censored observations.

The proportion of observations that are exactly measured is represented by the parameter r. When $r = 0$, all of the observations are censored. More generally, when $0 < r < 1$, the data set consists of both exact and interval-censored observation times. The special case $r = 1$ was ignored because it is a degenerate situation, for which an exact solution is obvious: put a mass of $\frac{1}{n}$ at each observation.

To simulate a general mechanism for interval-censored observations, a parameter k was introduced to represent case k censoring, for $k \geqslant 2$. This number represents the number of inspections. For an interval-censored observation, the times of the k inspections were also drawn independently from an exponential distribution with mean 1. These k times divide up the range $(0, \infty]$ into $k + 1$ intervals. The interval that contains the actual event time t_i is the one that defines the censoring interval $(L_i, R_i]$ that replaces t_i. Typically, a proportion $\frac{1}{k+1}$ of the censored observations are left-censored (having $L_i = 0$) and similarly for right-censored observations (having $R_i = \infty$).

For convenience, all observations were represented by two values L_i and R_i. Exact observations were therefore identified by $L_i = R_i$.

4.2 Computers

This section briefly describes some practical issues in conducting numerical optimisation with computers. The simulations performed for the research presented in this thesis were conducted on linux machines using the **R** statistical software.

Computers represent real numbers in an approximate way, using a finite number of significant figures (typically about 16). For this reason, algorithms that are converging to an exact unique solution must terminate at or before the point when this precision is reached.

Computations take time and make use the computer's resources such as memory and CPU processing power. When timing the result of a computation, the **R** function `proc.time` was used. It returns the CPU time used by the **R** process, accurate to three decimal places. For very brief calculations, this does not provide sufficient accuracy. However, for the purpose of assessing the speed of an algorithm, larger inputs can be provided that take longer to complete.

Simulations in this thesis were run on a Sun N1 Grid Engine. This is a network of computers, to which users can submit jobs. The grid engine places the jobs in queues and allocates them to individual computers to run, as computers become available.

4.3 Number and size of blocks

This section describes the experiment that was undertaken to find the optimum block size. The optimum block size is the size of blocks within HCNM that typically produces the fastest computation times. Block size will be denoted by b.

Under development, the HCNM algorithm accepted a parameter to specify the block size and thus to set the number of blocks in a given situation. For the final version of the algorithm, the goal was to derive the block size by examining the data, rather than requiring the user to provide it.

Methods

At each iteration, the HCNM algorithm considers a subset of the total set $\{I_j\}$ of m candidate support intervals. The dimension reduction technique of Wang (2008), described in Section 2.3, defines the initial subset and how this is expanded from one iteration to the next.

Suppose that there are m_s intervals under consideration during iteration s. When $m_s \geqslant 3b/2$ this set will be divided into $\kappa = m_s/b$ blocks (rounded to the nearest integer) each of approximate size b. For example, if there are $m_s = 400$ support intervals and a block size of $b = 30$ has been specified, then there will be 13 blocks, each containing either 30 or 31 support intervals. On the other hand, when $m_s < 3b/2$ there will be just one block, of size m_s, containing all of the support intervals.

To test various block sizes, a variety of simulated survival data sets was randomly generated. A total of 1280 simulated data sets were created, with parameters as follows:

- Number of observations, $n \in \{200, 400, 800, 1600, 3200, 6400, 12800, 25600\}$
- Proportion of exact observations, $r \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$
- Case-k censoring, $k \in \{2, 10\}$

The differing proportions of exact observations lead to differing sizes of the resulting NPMLE solution. Two different values of k were used to include both wide censoring intervals ($k = 2$) and narrow censoring intervals ($k = 10$).

The number of replications was typically ten, but was reduced to two for the largest two values of n due to the extreme computational times and computing resources required. Multiple replications were implemented by running the same set of simulations repeatedly, with different random seed values.

For each data set, the HCNM algorithm was used to compute the NPMLE solution a number of times. Each time a different block size parameter was used to specify the block size b that should be used. Each block size $b \in \{0, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144\}$ was tested, where zero means that HCNM did not create multiple blocks. For each calculation, the user-imposed iteration limit was set at 20, since the algorithm, when performing well, was expected to need at most ten iterations. Results from each calculation were recorded, including the computation time taken, the number of iterations required, the maximum recursion depth, the final log-likelihood value and the final maximum vertex-directional gradient component.

Source code for these simulations is provided in Appendix A (page 90).

Results

Figure 4.1 shows how the mean computation time varied by block size, for blocks of size two up to 144 (i.e. excluding $b = 0$). A separate plot is given for each value of n. For the largest two values of n, block sizes below 13 were not attempted due to the poor performance of small blocks that had already been seen at smaller values of n. Results for different values of r are shown in varying colours and line styles. Results for $k = 2$ and $k = 10$ were very similar, apart from computations for $k = 10$ taking slightly longer than $k = 2$. So the results for all replications and for both values of k have been aggregated by taking the mean computation time and mean number of iterations.

The important point to note in these plots is the general U shape. The base of this U shape gives an indication of what the best block size might be for each situation. In many cases, the base is quite flat, indicating that there may be a relatively wide range of block sizes that would provide acceptable computational times.

Throughout Figure 4.1 we can see approximately where the best block size lies, i.e. the size that minimises the computation time. These best block sizes are shown in Table 4.1, with zero indicating that the CNM algorithm without blocks was the best. Table 4.2, placed directly below Table 4.1 for comparison, shows the default block sizes used by HCNM (discussed below).

Another interesting feature is the extremely poor performance of very small blocks, particularly when both n and r are large. This produces an asymmetrical U shape, which is particularly steep on its left side (a good example is the plot for $n = 12800$). The main reason for this poor performance is that these situations converged too slowly, meaning that a greater number of iterations were required to reach the main stopping criterion. Another possible explanation is that such small blocks require much deeper levels of recursion, and each of these levels comes with a burden of computational overheads. The maximum level of recursion encountered in the simulations was nine levels deep. Because of the asymmetrical U shape, it would be safer to overestimate the necessary block size, rather than to underestimate it.

Figure 4.2 shows the mean number of iterations that were needed, and how this varied by the block size b for each combination of n and r. This shows the cause of the extremely poor performance of small block sizes, particularly when $n \geqslant 6400$ and from about $r \geqslant 0.2$. For larger block sizes, the number of iterations tends to stabilise, to values shown in Table 4.3.

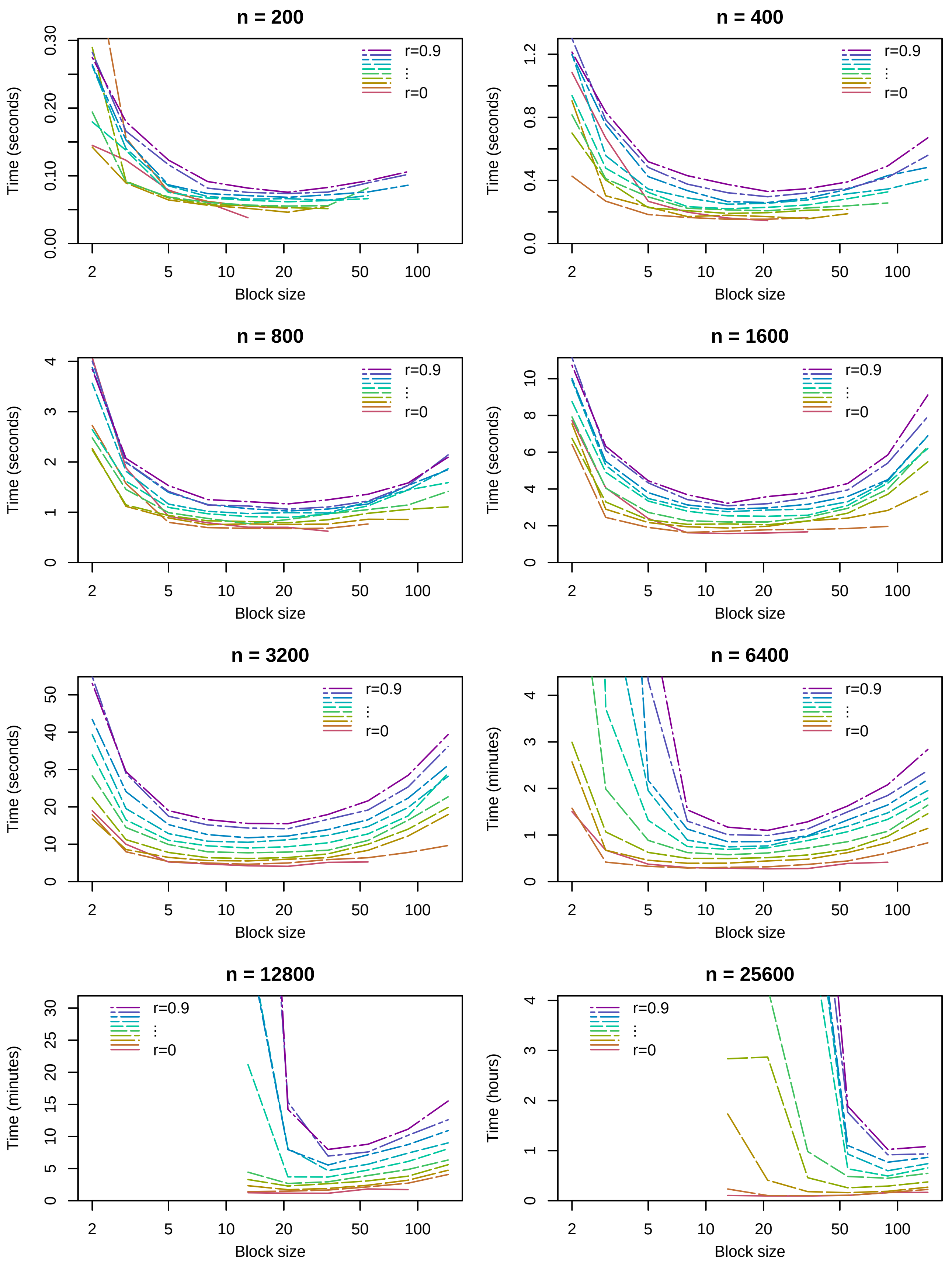


Figure 4.1: Mean HCNM computation time varying by block size, for various n and r values.

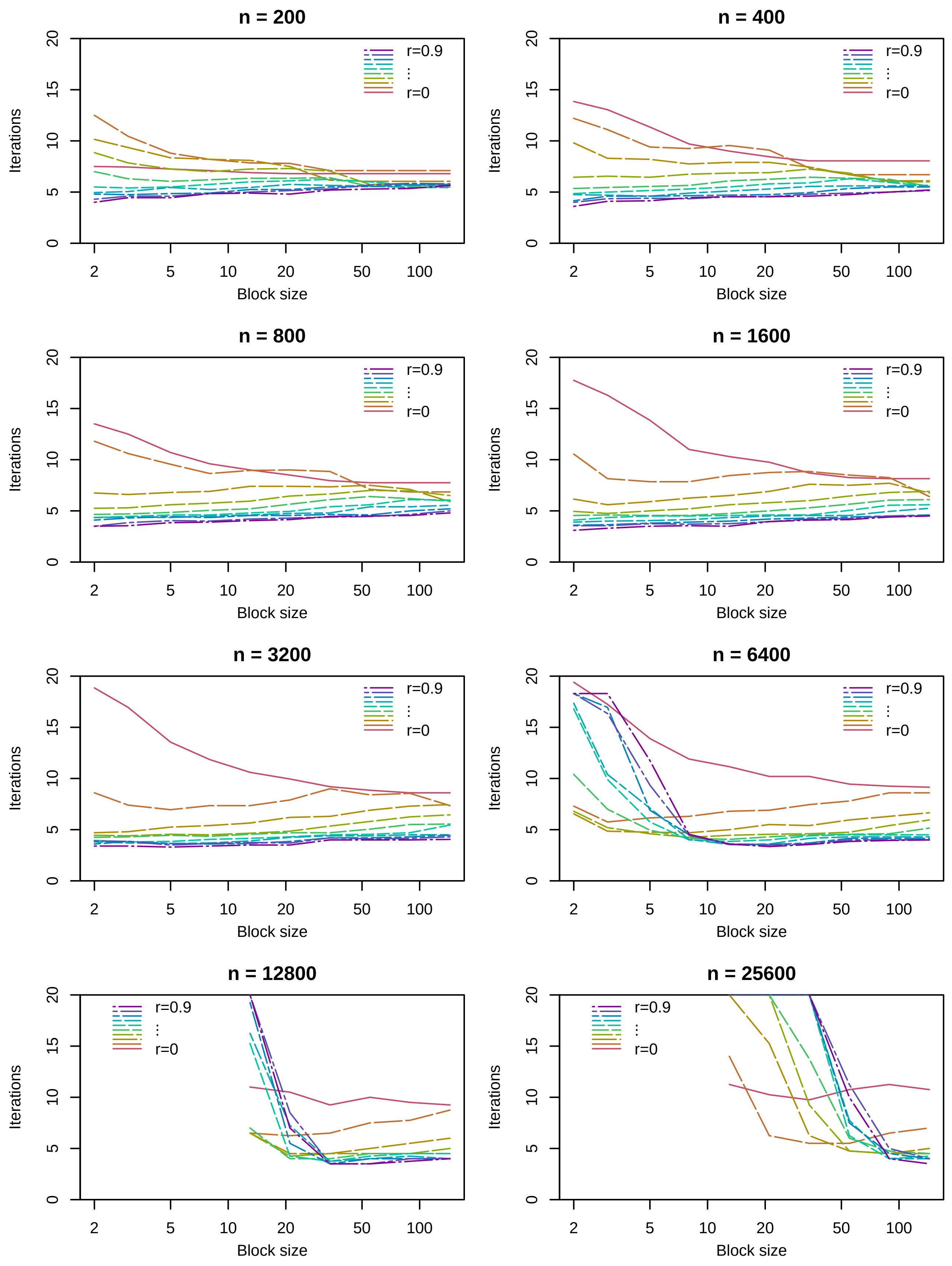


Figure 4.2: Mean number of HCNM iterations varying by block size, for various n and r values.

Table 4.1: The block size b that resulted in the best mean computation time.

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	0	0	0	0	0	0	34	21	21	21
400	0	0	0	13	21	13	13	21	21	21
800	0	0	8	21	13	21	13	21	21	21
1600	0	8	13	21	13	21	13	13	13	13
3200	0	13	13	13	13	13	13	13	21	21
6400	0	8	8	13	13	13	13	13	21	21
12800	0	13	21	21	21	34	34	34	34	34
25600	34	34	55	55	89	89	89	89	89	89

Table 4.2: Block sizes chosen by HCNM's default rule.

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	0	0	0	0	0	0	0	0	20	20
400	0	0	0	0	20	20	20	20	20	20
800	0	0	20	20	20	20	23	25	27	29
1600	0	20	20	23	27	30	33	35	37	39
3200	0	20	27	33	37	41	43	46	48	49
6400	0	27	37	43	48	51	54	56	58	60
12800	0	37	48	54	58	61	64	67	69	70
25600	0	48	58	64	69	72	75	77	79	81

Table 4.3: Mean number of HCNM iterations required, for the largest block size.

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	6.8	7.1	6.0	5.8	5.5	5.7	5.5	5.7	5.8	5.7
400	8.1	6.7	6.1	6.0	5.5	5.5	5.7	5.5	5.2	5.2
800	7.8	6.8	5.9	6.5	6.0	6.0	5.5	5.2	5.0	4.8
1600	8.2	6.4	6.8	6.9	6.1	5.6	5.2	4.6	4.5	4.5
3200	8.6	7.3	7.5	6.5	5.5	5.5	4.5	4.5	4.3	4.0
6400	9.2	8.6	6.7	6.0	5.2	4.5	4.3	4.2	4.0	4.0
12800	9.2	8.8	6.0	5.0	4.5	4.5	4.0	4.0	4.0	4.0
25600	10.8	7.0	5.0	4.5	4.5	4.2	4.0	4.0	4.0	3.5

Table 4.3 shows the mean number of iterations required by the HCNM algorithm for the largest block size. This table summarises the number of iterations shown on the right-most edge of each plot in Figure 4.2. For sufficiently large block sizes (including the CNM case with a single block) the algorithm converges efficiently and the number of iterations stabilises. Therefore the problem with the increasing computation times for larger blocks (the right side of the U shapes in Figure 4.1) is caused by the time complexity, per iteration, of the algorithm.

Results were also analysed by focusing on the best block size for each individual data set. The best block size varied, even among data sets that were created from the same set of parameters. For each data set, the value of b that produced the fastest computational time was labeled as the best for that data set. The way this best block size varied by the original parameters n and r was then analysed.

Table 4.4 shows the mean of block size used in the fastest block size computations. For comparison, Table 4.5 gives the mean support set size for each combination of n and r. This is the final size $\hat{m}$ of the set that HCNM must break up into blocks. Note that the number of exact observations is given by $n \times r$ and that each of these must be included in the final support set.

Table 4.6 shows the median number of blocks used by those computations that were the fastest. When the median number of blocks is one, the CNM algorithm (without blocks) was the best for at least half the simulated data sets.

Default Block Size Rule

From these results, the conclusion is that there is a minimum number of exact observations ($n \times r$) required before the multiple blocks of HCNM become advantageous. This limit is identified by the transition (in Table 4.6) from a single block being best to multiple blocks being best. Comparing this with the support set sizes in Table 4.5 indicates that the limit is at about 120–160 exact observations.

Therefore, the best rule for determining the block size was determined as follows:

- When $n \times r < 150$, do not create blocks (same as CNM).
- Otherwise, set the block size to the greater of 20 and $(15 \log(m_s) - 70)$ where m_s is the size of the dimension-reduced support set at iteration s.

As has already been mentioned, Table 4.2 shows the block size derived by the above rule, for each of the combinations of n and r that were considered. Chapter 6 puts the new HCNM algorithm through its paces, with this default rule in place, and compares it with existing algorithms.

Table 4.4: Mean block size used by fastest simulations

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	19	29	44	62	79	86	40	20	23	22
400	28	46	70	38	20	16	18	19	25	24
800	35	71	38	16	13	18	21	23	21	21
1600	44	15	13	13	15	20	17	17	16	18
3200	46	12	13	15	16	14	13	15	17	16
6400	48	11	11	13	14	14	14	15	16	17
12800	86	15	21	21	21	28	34	34	39	39
25600	41	36	50	55	80	72	89	103	116	65

Table 4.5: Mean size $\hat{m}$ of support set of the NPMLE solution.

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	19	29	44	62	81	101	120	140	160	180
400	28	47	83	121	160	200	240	280	320	360
800	37	87	161	240	320	400	480	560	640	720
1600	52	164	321	480	640	800	960	1120	1280	1440
3200	70	322	640	960	1280	1600	1920	2240	2560	2880
6400	94	641	1280	1920	2560	3200	3840	4480	5120	5760
12800	127	1280	2560	3840	5120	6400	7680	8960	10240	11520
25600	168	2561	5120	7680	10240	12800	15360	17920	20480	23040

Table 4.6: Median number of blocks used by fastest simulations

n	r=0	r=0.1	r=0.2	r=0.3	r=0.4	r=0.5	r=0.6	r=0.7	r=0.8	r=0.9
200	1	1	1	1	1	1	5	7	8	9
400	1	1	1	6	8	12	15	13	15	17
800	1	1	8	18	25	25	23	27	30	34
1600	1	13	25	37	49	38	74	53	80	111
3200	2	25	49	74	98	123	148	172	122	222
6400	2	80	98	148	197	246	295	345	394	274
12800	1	98	122	183	244	246	226	264	301	339
25600	8	75	93	140	115	188	173	201	186	1016

4.4 Block structure

In this section, we now turn to the question of how the blocks should be structured, given that the number of blocks (or, equivalently, the block size) has been decided. In particular, there are various ways of placing the blocks, varying their sizes and overlapping them.

Methods

The following seven methods for creating the blocks were examined. Examples of these are illustrated in Figure 4.3, with reference to the Hessian matrix shown in Figure 3.1.

1. Fixed sizes, alternating positions, i.e. shifted by a half-width on even iterations
2. Fixed positions and sizes; no overlapping or shifting of blocks
3. Varying sizes based on diag($\mathbf{S}^T\mathbf{S}$) and alternating positions
4. Fuzzy, with ramped overlaps from 0 to 1 then back to 0
5. Fuzzy, with flat 50% overlaps (no ramp)
6. Fuzzy, with ramps but with twice as many blocks
7. Fuzzy, with flat 50% overlaps, but with twice as many blocks

Methods 1 and 2 divide the current support set into the specified number of blocks so that all the blocks are (approximately) the same size. On even iterations, the first method shifts the boundaries by half a block-width (meaning there is one extra block, and the first and last are half the usual size). The reason for doing this is to avoid having fixed block boundaries and to efficiently transport probability mass through the support set, as suggested by the idea of rotating cycles proposed by Pilla and Lindsay (2001).

Method 3 attempts to locate parts of the support set where there is a large gradient. It then creates smaller blocks in such parts and larger blocks elsewhere. There is an additional computation cost in computing diag($\mathbf{S}^T\mathbf{S}$) and the hope was that this would be offset by a more rapid convergence to the solution.

With the fuzzy methods (numbers 4–7), consecutive blocks overlap each other, with weights defining how much the local NNLS solution affects the global probability vector. Each block overlaps its immediate neighbours by half its width. The sum of the block weights for an individual support interval is always one. The ramp is a smooth transition of weight from zero to one and back to zero again over the width of the block, whereas the step method uses a flat 50%/50% equal weighting.

Since the blocks overlap with each other in the fuzzy methods, they are generally twice the size (for the same number of blocks) compared to the non-overlapping methods. The last two

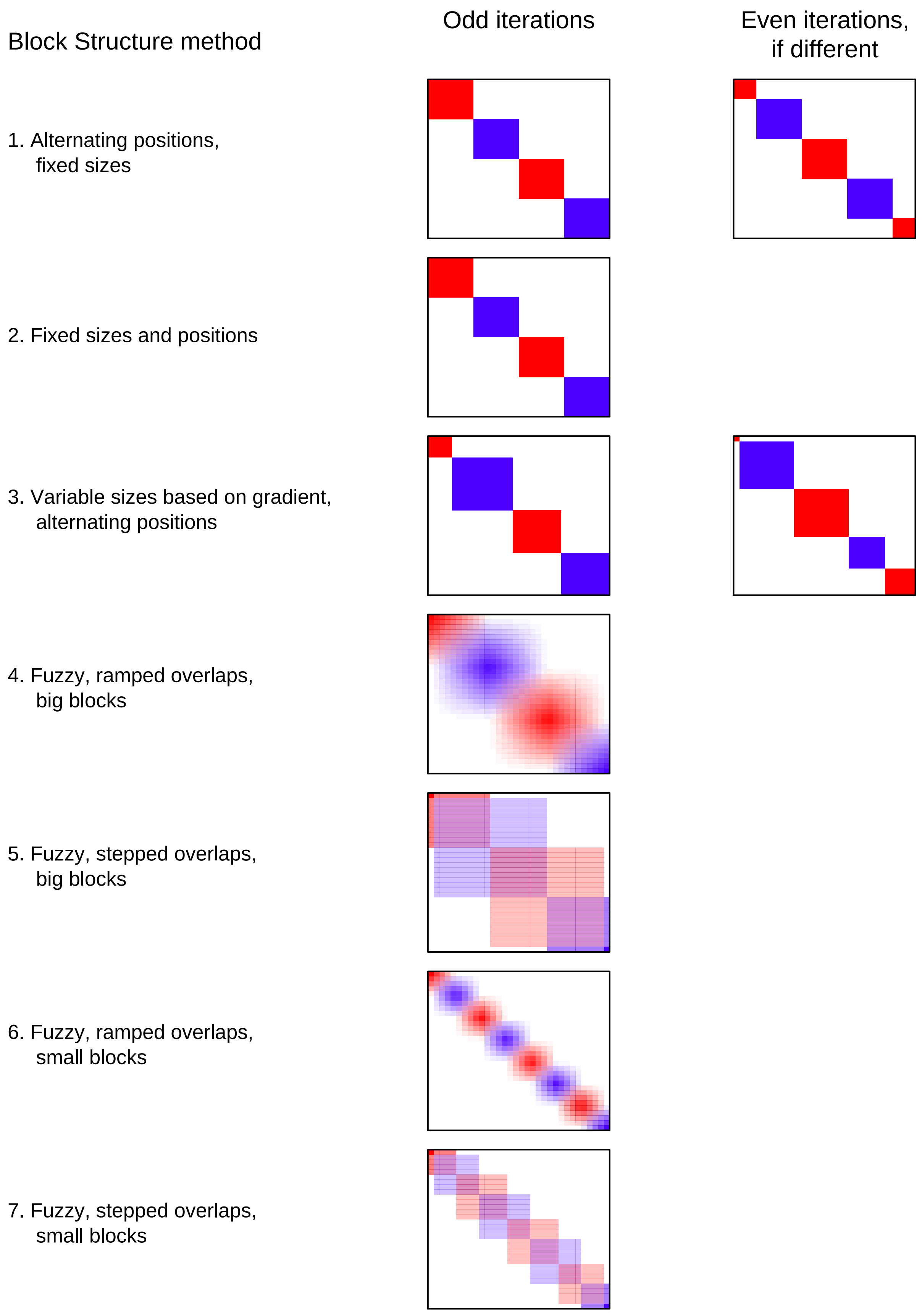

Block Structure method
Odd iterations
Even iterations, if different
1. Alternating positions, fixed sizes
2. Fixed sizes and positions
3. Variable sizes based on gradient, alternating positions
4. Fuzzy, ramped overlaps, big blocks
5. Fuzzy, stepped overlaps, big blocks
6. Fuzzy, ramped overlaps, small blocks
7. Fuzzy, stepped overlaps, small blocks

Figure 4.3: An illustration of seven block structure methods, for odd and even iterations. Odd numbered blocks are shown in red and even numbered blocks are blue.

methods were added to assess the impact of this effect; instead of using the same number of blocks twice the size, these two methods use blocks of the same size as methods 1–3 but with twice as many.

Results

To assess the performance of these seven methods, a new set of simulations similar to those of Section 4.3 was performed. The same values of n, r and c were used, although with fewer replications. Instead of fixing the block size, the simulations allowed HCNM to use its default formula. Each of the seven methods was applied to each simulated data set and the computation time and other results were recorded.

To see the effect of each method, a log-linear model for the computational time t was derived as follows:

$$\log(t) = \alpha \log(n) + \beta \log(\hat{m}) + \sum_{j=2}^{7} \gamma_j + \delta$$

where α and β are coefficients for $\log(n)$ and $\log(\hat{m})$ respectively, $\{\gamma_j\}$ is a set of terms for the last six block structure methods (i.e. excluding method 1) and δ is the intercept term. The three parameters α, β and δ allow for a known model of computation time. So it is the values of $\{\gamma_j\}$ that are of interest. The parameter γ_j for $j \geqslant 2$ measures the relative effect of method j compared to method 1. A significant negative value for γ_j would indicate that method j outperformed method 1, by producing significantly faster computations.

Table 4.7 shows the estimated coefficients of the model, based on the results of the simulations. Method 2, with its negative coefficient of $\gamma_2 = -0.0023$, was slightly faster than method 1, but the difference was not significant. Method 3 and all the fuzzy methods were significantly slower than method 1. Therefore, method 1 was chosen as the default and was used in all simulations outside this section.

Table 4.7: Estimated coefficients of log-linear model for Block Structure methods.

	Estimate	Std. Error	t value	Pr(>|t|)
α	1.3221	0.0056	234.0616	0.0000
β	0.6622	0.0056	118.8830	0.0000
γ_2 (Fixed)	−0.0023	0.0123	−0.1885	0.8505
γ_3 (Variable)	0.1969	0.0123	15.9722	0.0000
γ_4 (Fuzzy Ramped Big)	0.4820	0.0123	39.0923	0.0000
γ_5 (Fuzzy Stepped Big)	0.6461	0.0123	52.4092	0.0000
γ_6 (Fuzzy Ramped Small)	0.2380	0.0123	19.3013	0.0000
γ_7 (Fuzzy Stepped Small)	0.3618	0.0123	29.3421	0.0000
δ	−12.9101	0.0253	−509.5753	0.0000

4.5 Recursion within HCNM

The HCNM algorithm calls itself recursively to reallocate probability mass among the blocks at each iteration. These calls do not need to completely solve the reallocation problem because the completed reallocation solution will not normally be the correct one for the final NPMLE solution. Thus, to save some time on unfruitful computation, a restriction on the number of iterations was imposed at second and subsequent levels of recursion within HCNM. This experiment aims to find the optimum number of those iterations, which will be denoted by s.

In Section 5.1 the time component of recursion is shown to be a very small proportion of the time spent by the algorithm. It is therefore concluded that there is little cost in performing additional iterations at the recursive levels. The benefit of additional iterations at deeper levels of recursion is to achieve a faster rate of convergence, meaning fewer iterations are needed in the main loop.

Methods

A total of 384 simulated data sets were created, with 16 replications of each combination of parameters as follows:

- Number of observations, $n \in \{800, 1600, 3200, 6400\}$
- Proportion of exact observations, $r \in \{0.3, 0.6, 0.9\}$
- Case-k censoring, $k \in \{2, 10\}$

This parameter space is more restrictive than that of previous experiments so that situations where HCNM does not create blocks were avoided (since there would be no recursion).

For each simulated data set, the experiment tried the values of $s \in \{1, 2, 5, 20\}$ as the iteration limit for recursive calls to HCNM. These iteration limits may not necessarily be reached, as the recursive calls to HCNM may find a solution in fewer iterations based on the main stopping criterion.

Results

Table 4.8 shows the mean number of main loop iterations required to find the NPMLE, varying by n and r for various values of s. This shows the impact that different values of s have on the main loop's convergence. Similarly, Table 4.9 shows the mean computational time, although variations of time are primarily due to variations in the number of iterations, shown in Table 4.8.

Based on these results, the conclusion is that $s = 2$ is the best number of iterations for the recursive calls to HCNM. The reason for this conclusion is that two iterations (rather than just one) generally seems to reduce the number of iterations required at the top level, but more than that does not seem to help any further.

Table 4.8: Mean number of HCNM iterations, varying by recursion iteration limit s

n	r	$s = 1$	$s = 2$	$s = 5$	$s = 20$
800	0.3	6.7	6.6	6.7	6.7
	0.6	5.4	4.6	4.6	4.6
	0.9	5.2	4.4	4.3	4.3
1600	0.3	6.2	5.8	5.9	5.9
	0.6	5.4	4.5	4.5	4.5
	0.9	4.8	4.1	4.1	4.1
3200	0.3	6.0	5.1	5.4	5.4
	0.6	5.2	4.2	4.2	4.2
	0.9	4.9	3.7	3.7	3.7
6400	0.3	6.1	4.9	4.9	4.9
	0.6	5.3	4.1	4.1	4.1
	0.9	5.0	3.5	3.5	3.5

Table 4.9: Mean HCNM computational time (seconds) varying by recursion iteration limit s

n	r	$s = 1$	$s = 2$	$s = 5$	$s = 20$
800	0.3	0.8	0.8	0.8	0.8
	0.6	0.9	0.8	0.8	0.8
	0.9	1.3	1.1	1.1	1.1
1600	0.3	2.5	2.4	2.4	2.4
	0.6	3.7	3.2	3.3	3.4
	0.9	5.1	4.6	4.8	4.8
3200	0.3	10.1	8.5	9.2	9.2
	0.6	17.8	14.3	14.5	14.4
	0.9	26.2	19.5	19.5	19.3
6400	0.3	49.4	39.5	39.3	39.3
	0.6	88.0	67.0	68.1	67.7
	0.9	129.4	90.2	90.7	89.8

4.6 Summary

Experiments to fine-tune the performance of the HCNM algorithm were presented in this chapter. The main results from this chapter are as follows.

A formula for the default block size was derived from simulations. A threshold of 150 exact observations was derived, below which the HCNM uses a single block, as in the CNM algorithm. Above that, the rule determines the block size using the size of the set of support intervals that are under consideration.

Various block structures were examined. The best choice of block structure was to use equally sized, non-overlapping blocks. For recursive calls to HCNM, a limit of 2 iterations was imposed on those calls.

Chapter 5

Time Complexity

This chapter explores the time complexity of algorithms. Time complexity is a description of how the time taken to complete an algorithm varies according to the size of the inputs (and sometimes the outputs). In theoretical terms, time complexity is expressed in the limit as the size of inputs grows without bound. Complexity is also known as computational order, and is often expressed using big O notation. For example, the time complexity of an algorithm may be expressed as $O(n^p)$, which means that as n grows larger the computation time grows in proportion to n^p.

In this chapter the practical time complexity of algorithms is estimated. This is done using empirical methods on simulated data. Simulated data sets were created in varying sizes, with the size of the largest being constrained by the capacity limits of the available computers. Some situations were also constrained by the long computation times involved. This approach was taken with the aim of using the practical time complexity to closely estimate the theoretical time complexity.

Section 5.1 explores the time that is spent by the HCNM and CNM algorithms, breaking the time down into several component tasks. The non-negative least squares (NNLS) algorithm is a key component of the newly developed HCNM algorithm and its predecessor CNM. The time complexity of NNLS is examined in Section 5.2 (in a general context) and Section 5.3 (within the context of HCNM). Section 5.4 investigates the time complexity of the HCNM and CNM algorithms. Finally, Section 5.5 provides concluding remarks on time complexity.

5.1 Proportional breakdown of time spent

This section looks deeper into the computation time that is spent by the HCNM and CNM algorithms, breaking it down into several main component tasks. These are studied by looking at the proportion of computation time that was spent on each specific task. The aims of this part of the research were to identify which tasks contribute the greatest proportion of time, and to understand how these proportions varied for different types and sizes of data set. The component task that contributes the greatest proportion is the one that ultimately determines the time complexity of an algorithm, especially if that proportion approaches 100% for large inputs.

The performance of the HCNM algorithm was investigated using the new default block size rule that was derived in Section 4.3. The CNM algorithm was examined by using the HCNM algorithm with the number of blocks fixed at one. Source code for the HCNM algorithm (in the **R** language) is provided in Appendix A (page 85).

To analyse an algorithm and record the time taken by portions of it, a set of `timing` functions were developed (Appendix A, page 94). These functions can be used to record the CPU processing time spent on different tasks within an algorithm. They are based on the `proc.time` function (available in **R**), which returns the total CPU time used by the **R** process (R Development Core Team, 2008). By taking the difference between the current value of total CPU time and the previously recorded value, the CPU time spent on each small portion of the algorithm can be recorded and labelled. One can imagine this as a stopwatch with a lap-counter function, marking the moments when specific marked points are reached and recording the time differences between them.

Markers were strategically placed at points in the HCNM algorithm, typically immediately before and after a section with a specific purpose. These markers divided the algorithm up into the following six component tasks:

1. **Set-up.** This component included everything that must be done before the main iterative loop begins. The set-up task is only performed once and is identical for both the CNM and HCNM algorithms.

2. **Miscellaneous.** This component was used to summarise several smaller components within the algorithm. It included the time taken for evaluation of the log-likelihood, gradient components and the matrix S. It also included the time spent on implementing the dimension-reduction technique.

3. **Block creation.** When HCNM has determined the necessary number (and size) of blocks, it creates a matrix of block membership weights. The time taken for this task was recorded in the this component.

4. **NNLS.** The HCNM algorithm calls the NNLS algorithm once per block per iteration. It does this to adjust the probability masses within a block, which is a key aspect of finding the solution. The total time spent during NNLS calculations was recorded by the fourth time component.

5. **Line search.** When the probability masses have been adjusted within each block, a

line search (Section 2.6) is conducted to ensure that the log-likelihood value has strictly increased. The fifth component recorded the time taken for this task.

6. **Recursion.** The last component recorded the time spent within recursive calls to HCNM. This task reallocates probability mass among the blocks, considering each block as a unit. All the time spent at levels of recursion deeper than one was recorded in this component, i.e. without breaking that time down into the other components. Since the CNM algorithm uses a single block, this task was not performed within CNM.

Apart from the set-up task, the other five components are tasks that are repeated within the iteration loop. For each of these five components, the total computation time over all iterations was recorded.

Methods

A total of 3440 simulated data sets were created, using the method described in Section 4.1 with parameters as follows:

- Number of observations, $n \in \{200, 400, 800, 1600, 3200, 6400, 12800, 25600\}$
- Proportion of exact observations, $r \in \{0, 0.3, 0.6, 0.9\}$
- Case-k censoring, $k \in \{2, 10\}$.

These parameter values were chosen to provide a wide variety of simulated data sets, varying in size from small to large. The differing proportions of exact observations lead to differing sizes of the resulting NPMLE solution. Two different values of k were used to include both wide censoring intervals ($k = 2$) and narrow censoring intervals ($k = 10$).

For smaller data sets, with values of $n \leqslant 3200$, both the CNM and HCNM algorithms were investigated. A total of 40 replications of each combination of the parameters n, r and k were performed for each algorithm. For larger data sets ($n \geqslant 6400$) only the HCNM algorithm was investigated, due to the extremely long computation times of CNM. There were ten replications of each combination of n, r and k in this case.

With each data set, either the CNM or the HCNM algorithm was used to find the NPMLE survival function. Equal numbers of data sets were allocated to both algorithms. Within each algorithm, the times taken to compute each of the six component tasks were recorded.

Results

Table 5.1 summarises the results by showing the mean computation times of the CNM and HCNM algorithms, for each combination of n and r. Results for $k = 2$ and $k = 10$ were found to be very similar, so they have been combined.

Table 5.1: Mean computation time (seconds) by n and r for the CNM and HCNM algorithms.

Algorithm	r	n=200	n=400	n=800	n=1600	n=3200	n=6400	n=12800	n=25600
CNM	0.0	0.034	0.081	0.27	1.2	4.1			
	0.3	0.052	0.189	1.23	10.9	87.9			
	0.6	0.094	0.520	4.16	37.7	297.0			
	0.9	0.161	1.103	9.29	79.7	617.9			
HCNM	0.0	0.032	0.082	0.27	1.1	4.5	17	76	372
	0.3	0.051	0.201	0.64	2.3	9.5	43	183	953
	0.6	0.096	0.263	0.78	3.4	16.6	72	328	2674
	0.9	0.129	0.331	1.06	4.9	22.1	97	510	5027

When all simulations were completed, totals of the time taken for each of the six component tasks were derived by taking the sum over the replications and values of k. This was done at each combination of the parameters n and r, for each algorithm. In each case, these six component totals were converted into proportions by dividing by their combined sum.

The component task with the greatest proportion for the CNM algorithm was NNLS, with 98.03% at $n = 3200$ and $r = 0.9$. For the HCNM algorithm, the component with the greatest proportion was Set-up, with 63.9% at $n = 25600$ and $r = 0$. The greatest proportion of time taken by NNLS within HCNM (excluding cases where HCNM did not create multiple blocks) was 51.5% at $n = 6400$ and $r = 0.9$.

Figure 5.1 shows how the proportions of the six components varied for different values of n and r. The CNM algorithm is shown in the top part of the figure, for values of n from 200 up to 3200. This figure dramatically illustrates the dependence on the NNLS algorithm, showing how it fills up CNM's time in situations when $r > 0$ and n grows large. This suggests that it is NNLS that determines the time complexity of the CNM algorithm. More precisely, NNLS determines the time complexity, per iteration, of the CNM algorithm (see Section 5.4).

The HCNM algorithm is shown in the lower part of the figure. In comparison to CNM, the strategy used by the HCNM algorithm keeps the time spent on NNLS to a smaller (possibly constant) proportion of the total. As a result, HCNM is much faster than CNM in some situations, as can be seen in Table 5.1. Because the times are presented as proportions of the total,

Another interesting feature is that the proportion of time used by recursion reduces as n grows larger. Note that for some combinations of n and r the default rule in HCNM makes it the same as CNM because it does not create blocks. These situations can be identified by a dashed line for the sixth component, since there is no recursion in such a case.

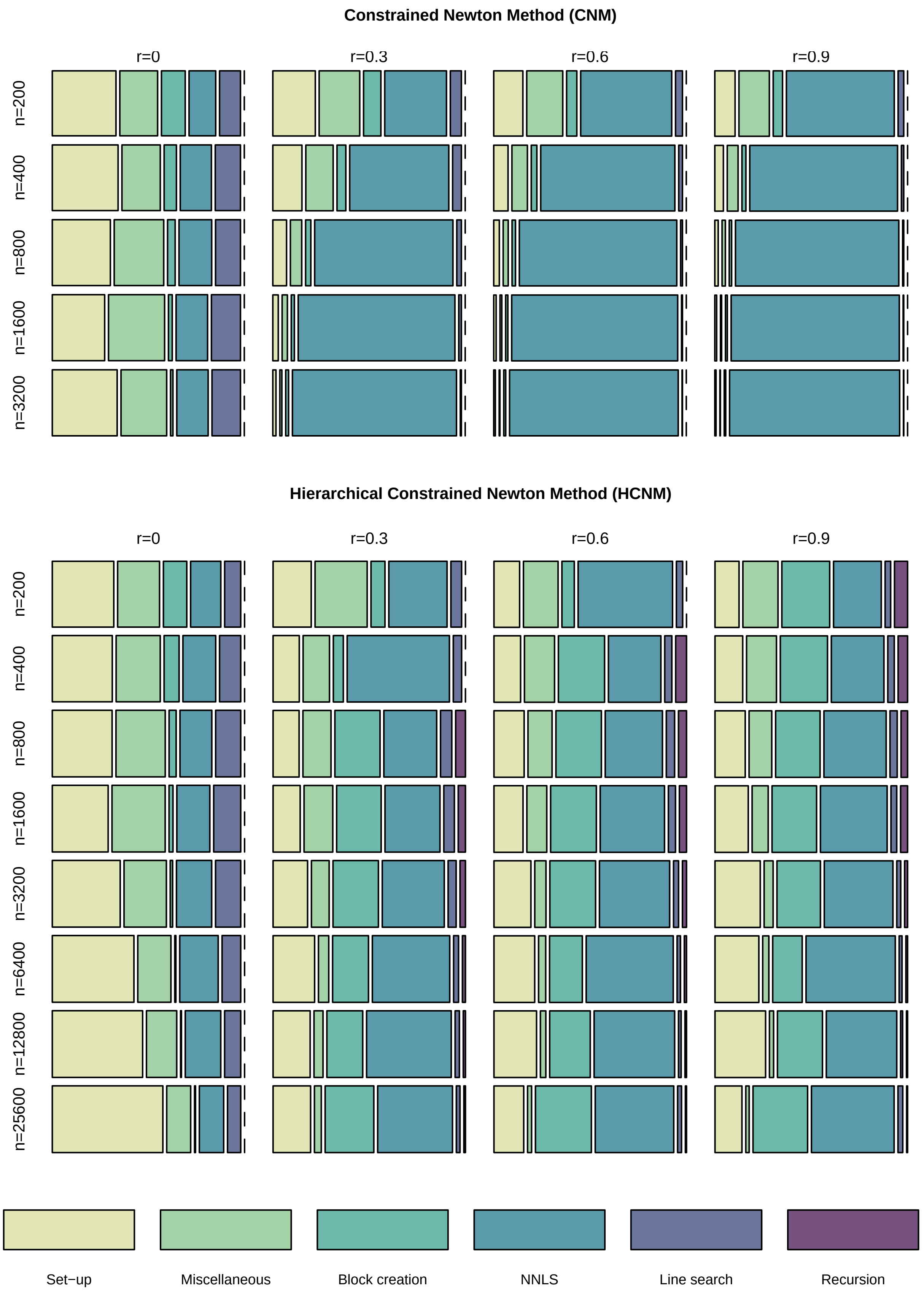


Figure 5.1: Proportional breakdown of computation time into six component tasks for the CNM and HCNM algorithms.

5.2 General time complexity of NNLS

The previous section identified NNLS as the task that contributed the most to the computation time taken by the CNM algorithm. This section describes an experiment to estimate the practical time complexity of the NNLS algorithm, based on random simulated data. Since both CNM and HCNM make heavy use of the NNLS algorithm, its performance was investigated as part of the research into developing HCNM.

The method of least squares is a fundamental concept in the subject of statistics. It describes a method of fitting a model so that the sum of squared differences between observations and expected values is minimised. A familiar application is fitting a regression line, with a slope and intercept, to a set of observations each based on a single covariate, for example height as a function of age. In some situations, there is a need to constrain a fitted parameter. An example is that the slope of a fitted regression line cannot be negative (for some physical reason, perhaps). More generally there may be many covariates, so such an inequality constraint may then be applied to a specified linear combination of the covariates.

Lawson and Hanson (1974, chapter 23) considered the problem of solving least squares with inequality constraints. They provided a Fortran implementation of their Non-Negative Least Squares (NNLS) algorithm. Their algorithm is the de facto standard for solving this particular problem. Given an $n \times m$ matrix $\mathbf{A}$ and a vector $\mathbf{b} \in \mathbb{R}^n$, the NNLS algorithm finds $\mathbf{x} \in \mathbb{R}^m$ to minimise the sum of squares $\|\mathbf{Ax} - \mathbf{b}\|^2$ subject to $\mathbf{x} \geqslant \mathbf{0}$.

The NNLS algorithm consists of an outer loop and an inner loop (see the pseudo-code on page 96 in Appendix A). Lawson and Hanson (1974) proved that both loops, and therefore the NNLS algorithm as a whole, must terminate in a finite number of iterations. Furthermore, they observed that the outer loop typically required $\frac{m}{2}$ iterations in a small number of test cases. In their Fortran code, a limit of $3m$ iterations is imposed as the maximum number of times the inner loop can be used. If the limit is exceeded, their program returns an error code. This error code was never observed throughout the research conducted for this thesis. Note that their notation differs from that used here in that the parameter names n and m are reversed.

The parameters n and m express the size of the inputs that are given to the NNLS algorithm. Another parameter that can be measured from the output of NNLS is the number of positive components of $\mathbf{x}$, which is denoted by $\hat{m}$. This is expected to have an effect on computation times. These three parameters can be used in the expression of the time complexity. For example, the practical time complexity of NNLS could be expressed in the form $O(n^\alpha m^\beta \hat{m}^\gamma)$, where the coefficients α, β and γ need to be estimated from the results of simulations.

Methods

A number of NNLS simulations were performed, for various sizes and shapes of the matrix $\mathbf{A}$. To get an accurate estimate of the time complexity of NNLS, data sets with a wide variety of sizes were used, up to those possible within the constraints of the available computers.

For the number of rows in the matrix, values of n ranging from 64 up to 262144 (2^{18}) were used. For the number of columns in the matrix, values of m ranging from two up to 2580 were used. The total number of unique values used was 37 for n and 31 for m. In both cases, values were chosen so that they formed a regular progression between those extremes on a log scale. Even though NNLS can handle the case where $m > n$, the simulations were limited to combinations with $m \leqslant n$ because that is always the case when HCNM or CNM makes use of the NNLS algorithm. Furthermore, some combinations of these n and m values were excluded, either because the computation would be too quick (less than the precision of time measurement) or because it would take too long. Specifically, only combinations where the product $n \times m$ was between 3000 and 10^7 were included.

Each individual simulation involved creating a matrix $\mathbf{A}$ and a vector $\mathbf{b}$. Both were filled with pseudo-random values from a Normal distribution. Values in $\mathbf{A}$ were taken from $\mathrm{N}(\mu, 1)$ for values of $\mu \in \{$-1.25, -1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1, 1.25$\}$. Values in $\mathbf{b}$ were taken from the standard Normal distribution with zero mean. Then the NNLS algorithm was used to solve for $\mathbf{x}$, and the computation time and number of positive values were recorded. For each combination of the three parameters n, m and μ, six replications were performed and the results were aggregated by taking the mean computation time. Thus, a total of 48114 simulations were performed.

Results

Table 5.2 shows a summary of some results for each value of μ, with replications and results for all values of n and m aggregated. The maximum computation time for a single computation was 95.8 seconds. The proportion of positive values in the solution is given by $\hat{m}/m$. For $\mu = 0$ the solution $\mathbf{x}$ contained zeros and positive values in approximately equal proportions, on average. At other values of μ, there were fewer positive values than zeros, with the proportion of positive values decreasing for μ further from zero. Mean computation times increased for increased proportions $\hat{m}/m$, indicating that the computation time depends on both $\hat{m}$ and m. However, these two variables were found to be correlated, with the correlation approaching one at $\mu = 0$. Because of this correlation, a 'variance inflation factor' (VIF) was calculated, for use in a log-linear model. High values of this factor indicate difficulties in fitting a model with all three parameters included.

Figure 5.2 shows contour plots of mean NNLS computation times, varying by parameters n and m. For each value of $\mu_0 > 0$ that was investigated, the results for values of $\mu = \mu_0$ and

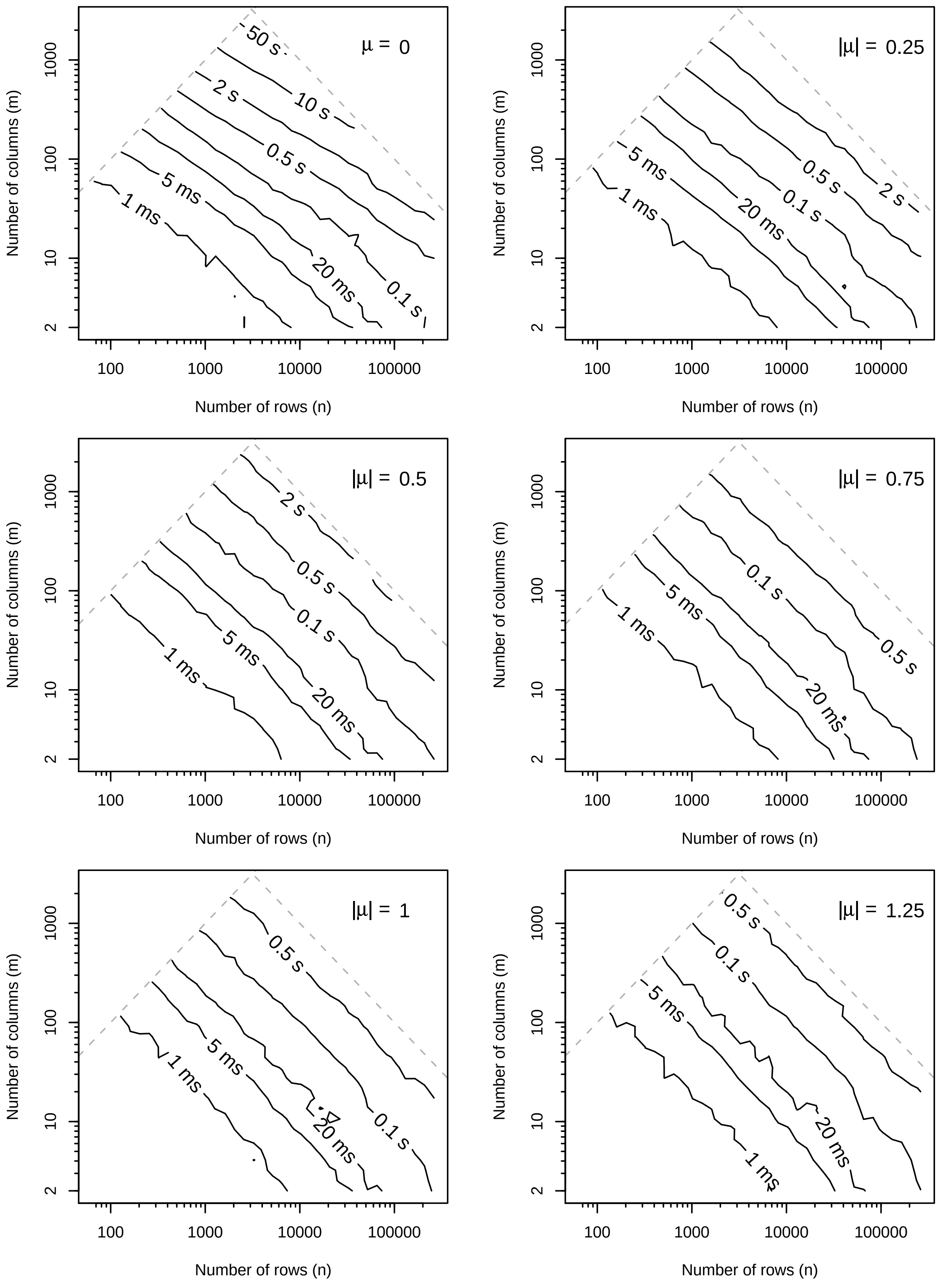


Figure 5.2: Contour plots of mean NNLS computation times by n and m, for six values of $|\mu|$.

Table 5.2: Summary of NNLS time complexity simulations, by μ.

μ	Mean $\hat{m}/m$	Mean time	Maximum time	cor(m,$\hat{m}$)	VIF
−1.25	0.11	0.11	1.66	0.48	1.32
−1.00	0.13	0.13	1.98	0.56	1.48
−0.75	0.16	0.19	3.50	0.64	1.75
−0.50	0.21	0.28	4.02	0.73	2.25
−0.25	0.30	0.55	10.11	0.85	3.75
0.00	0.50	2.46	95.82	1.00	646.87
0.25	0.30	0.54	9.92	0.84	3.62
0.50	0.21	0.27	4.65	0.73	2.24
0.75	0.16	0.19	2.77	0.64	1.73
1.00	0.13	0.15	2.23	0.57	1.50
1.25	0.11	0.12	1.76	0.46	1.29

$\mu = -\mu_0$ were very similar, so these results were combined under the label $|\mu|$. There are six contour plots, one for each value of $|\mu|$ that was used. The dotted grey lines indicate the extent of the simulations attempted, at $m = n$ and $nm = 10^7$. From the contour plots, the computation times (on a log scale) appear to be reasonably planar as a function n and m, both also on log scales. This suggests that a log-linear model is appropriate to model the computation time.

Figure 5.3, Figure 5.4 and Figure 5.5 show some log-log plots for specific values of m or n. Figure 5.3 shows three plots of time t against the number of rows n, for three specific values of $m \in \{32, 128, 512\}$. Figure 5.4 shows three plots of time t against the number of columns m, for three specific values of $n \in \{512, 3251, 32768\}$. Figure 5.5 shows three plots of time t against the number of positive values in the solution $\hat{m}$, for three specific values of $m \in \{32, 128, 512\}$. Each of these plots includes a simple estimate of the slope of the regression line through the points.

Because of the planar shape of the contour plots, the practical time complexity of NNLS is assumed to be of the form $O(n^\alpha m^\beta \hat{m}^\gamma)$. To estimate the parameters α, β and γ, the following log-linear model was used:

$$\log(t) = \alpha \log(n) + \beta \log(m) + \gamma \log(\hat{m}) + \delta$$

where t is the computation time, n and m describe the dimensions of the matrix A, $\hat{m}$ is the number of positive components in the solution and δ is an intercept term. Table 5.3 shows that the estimated parameter values, using data for all values of μ, were α=1.02, β=0.906 and γ=0.799.

Because m and $\hat{m}$ are highly correlated, an alternative form $O(n^\alpha m^\beta)$ can be used. Also, the slopes of the six planes in the contour plots do vary. So a separate model for each value of μ was fitted. Figure 5.6 shows how the coefficient estimates for the two-parameter and three-

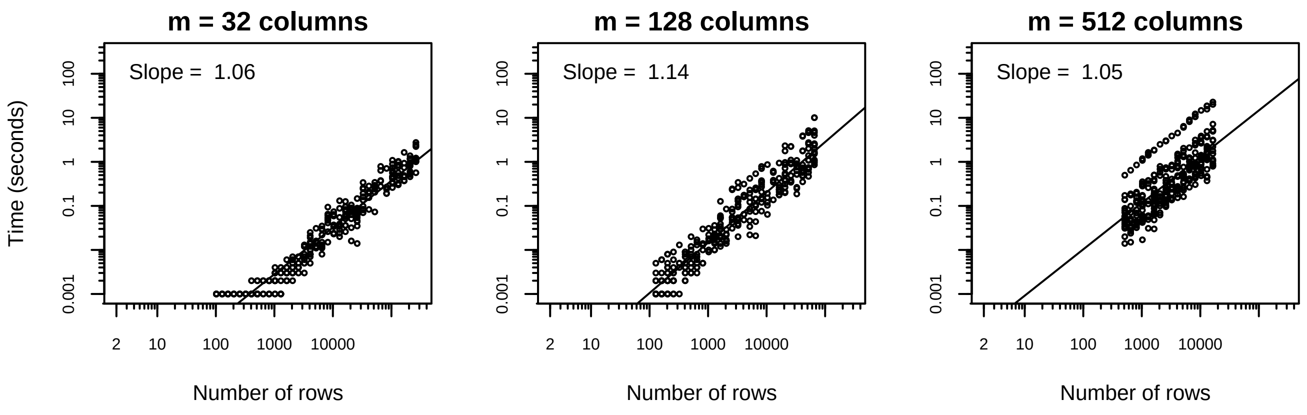


Figure 5.3: Plots of NNLS computation times by n, for specific values of $m \in \{32, 128, 512\}$.

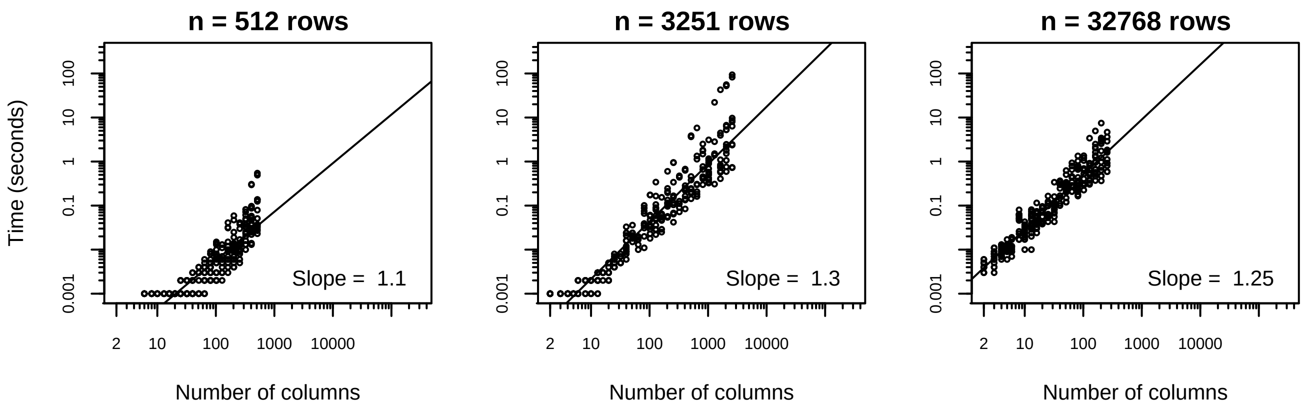


Figure 5.4: Plots of NNLS computation times by m, for specific values of $n \in \{512, 3251, 32768\}$.

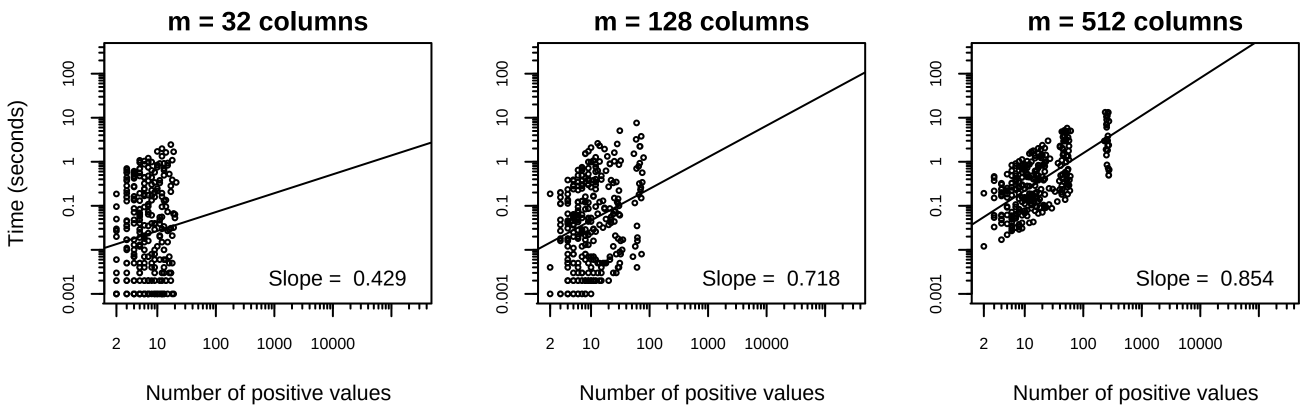


Figure 5.5: Plots of NNLS computation times by $\hat{m}$, for specific values of $m \in \{32, 128, 512\}$.

Table 5.3: Estimated coefficients of log-linear model of NNLS computation times.

	Estimate	Std. Error	t value	Pr(>|t|)
α	1.0227	0.0021	496.98	0.0000
β	0.9060	0.0020	448.84	0.0000
γ	0.7987	0.0018	453.95	0.0000
δ	−17.1347	0.0296	−578.44	0.0000

parameter models varied by μ. Situations where the high correlation between m and $\hat{m}$ prevented the coefficients from being estimated have been excluded. The highest computational complexity was at $\mu = 0$ where it is perhaps $O(nm^2)$. This is the situation that most closely resembles the use of NNLS within the HCNM algorithm.

In the three-parameter models, the time complexity is perhaps $O(nm\sqrt{\hat{m}})$, at least for values of μ sufficiently far from zero. Close to $\mu = 0$ the three-parameter model could not separate the effects of m and $\hat{m}$.

The conclusion from this section is that the time complexity of the NNLS algorithm takes on different forms, depending on what data is provided. The next section looks at the time complexity of NNLS using data extracted from the workings of the HCNM and CNM algorithms.

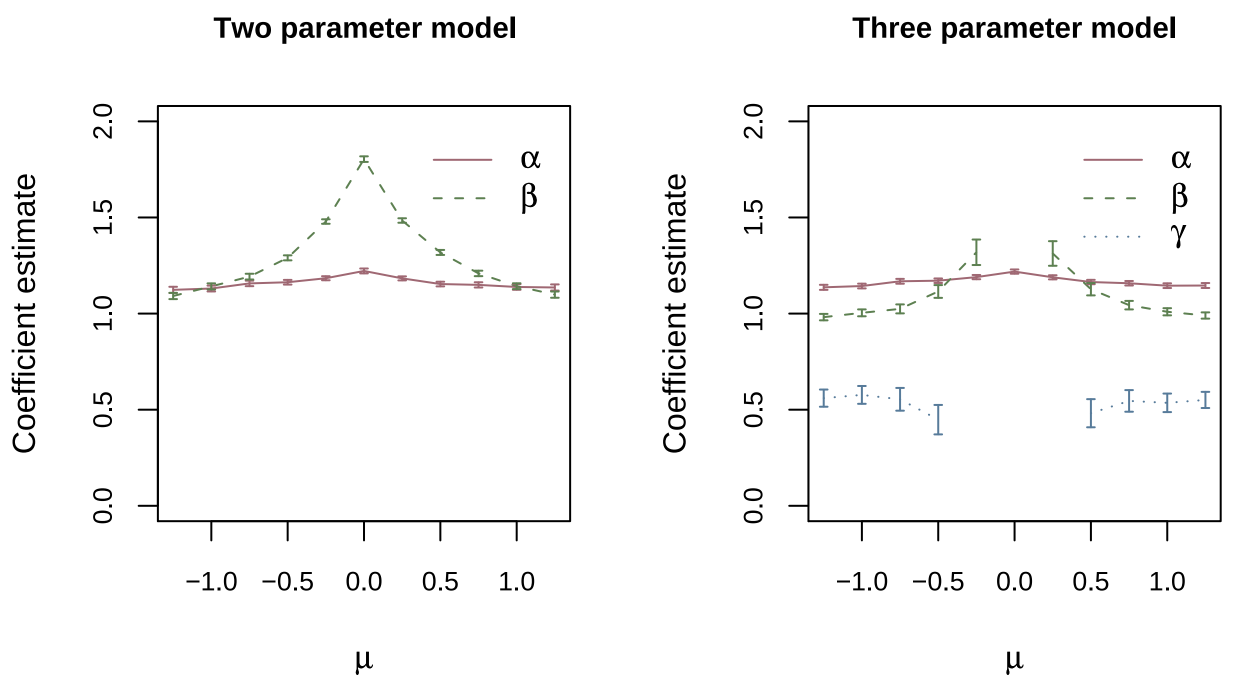


Figure 5.6: Coefficient estimates for NNLS time complexity models $O(n^\alpha m^\beta)$ and $O(n^\alpha m^\beta \hat{m}^\gamma)$ varying by μ.

5.3 Time complexity of NNLS within HCNM

This section describes an experiment to investigate the practical time complexity of the NNLS algorithm, specifically within the context of the HCNM and CNM algorithms. The NNLS algorithm appeared to contribute to the time complexity per iteration of the CNM algorithm. This gives the motivation for this experiment.

For this experiment, the implementation of the HCNM algorithm was modified slightly so that details of each and every call to NNLS were recorded. The details recorded were the dimensions of the matrix parameter given to NNLS (denoted in this section as n rows by m columns), the number of positive values in the solution ($\hat{m}$) and the computational time (t) taken by NNLS to return the solution. The CNM algorithm was implemented by using the HCNM algorithm and fixing the number of blocks to one. The aim of this experiment was to estimate the NNLS time complexity, expressing it in terms of the parameters n, m and $\hat{m}$.

When HCNM (or CNM) calls the NNLS algorithm, the number of rows n in the matrix is one more than the number of observations in the survival data set. This is because there is one additional constraint added to the non-negative least squares problem, namely that the probabilities must add up to one (in the case of CNM, as in Eq. (2.25)) or ω_b (in the case of HCNM, as in Eq. (3.10)).

The number of columns m in the matrix is equal to the number of support intervals under consideration at that point in the algorithm. Within HCNM this is defined by the size of each block in its turn. However, within CNM the number is equal to the size of the full set of support intervals being considered, which is determined by the dimension reduction technique (Section 2.3) from Wang (2008).

Rather than using the new default block size rule in this experiment, the HCNM algorithm was applied using a wider variety of block sizes. This approach deliberately included some block sizes that were far from optimal, from the viewpoint of minimising the computation time. Thus the experiment was able to explore more thoroughly the behavior of NNLS within HCNM.

Methods

Simulated data sets were created using the method described in Section 4.1. A total of 2240 data sets were created, with seven replications of each combination of the following parameters:

- Number of observations, values in {100, 126, 159, 200, 252, 317, 400, 504, 635, 800, 1008, 1270, 1600, 2016, 2540, 3200}.
- Proportion of exact observations, $r \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$
- Case-k censoring, $k \in \{2, 10\}$

For each data set, the HCNM algorithm was applied four times, each time varying the block parameter b. Values of the block parameter $b \in \{0, 0.3, 0.5, 0.7\}$ were used. This block parameter takes values $0 \leqslant b < 1$ and determines the number and size of blocks as follows. Suppose there are s support intervals under consideration in the current iteration, as determined by the dimension reduction technique. Then the number of blocks for that iteration is determined by $\lfloor s^b \rfloor$. Thus, the block size will be approximately s^{1-b}. For example, when $s = 400$ and $b = 0.3$ the number of blocks will be 6, each of size 66 or 67. This approach was used in preference to giving explicit block sizes so that a wider variety of block sizes would be used. The value $b = 0$ fixes the number of blocks to one, thereby implementing the CNM algorithm.

A limit of seven was imposed on the number of iterations, in order to limit the number of results collected. The CNM algorithm calls NNLS once per iteration, whereas the HCNM algorithm calls NNLS once per block per iteration. There was no reason to think that the behaviour of NNLS would be any different at iterations beyond this limit.

Results

The 8960 simulations (four per data set) resulted in a total of 699 236 observations. Each observation recorded the values $\{n, m, \hat{m}, t\}$ for a call to the NNLS algorithm from within HCNM. To reduce this large quantity of data, the results were aggregated by deriving the mean value of t at each unique combination of $\{n, m, \hat{m}\}$, resulting in 2977 aggregated observations.

The complete set of values of n that were used in this experiment was $n \in \{101, 127, 160, 201, 253, 318, 401, 505, 636, 801, 1009, 1271, 1601, 2017, 2541, 3201\}$. Values of m varied from one to 2880, with lower quartile at 80 and upper quartile at 382.

Figure 5.7 shows a contour plot of mean NNLS computation times, and a perspective view of the same data. The log-log relationship is again approximately planar, especially for times above about 10ms. This allows a log-linear model to be used to estimate the time complexity.

Figure 5.8 shows two plots of mean computation times for selected values of n and m, giving the slope of fitted log-linear models. The first plot gives the mean computation times varying by m for three specific values of n. The second plot gives the mean computation times varying by n for three specific ranges of m values. The fitted slope values provide an indication of how the time complexity depends on these two variables.

Within the context of CNM and HCNM, the number $\hat{m}$ of positive values returned in the NNLS solution was very highly correlated with the number of columns m, with a correlation of more than 0.99. The reason for this was due to the dimension reduction technique. This technique attempts to identify a small subset of the larger set of possible support intervals, selecting those that appear to be needed in the solution (i.e. those that would be expected to receive positive probability mass). The technique is very good at predicting which components should be positive, which explains the high correlation between m and $\hat{m}$.

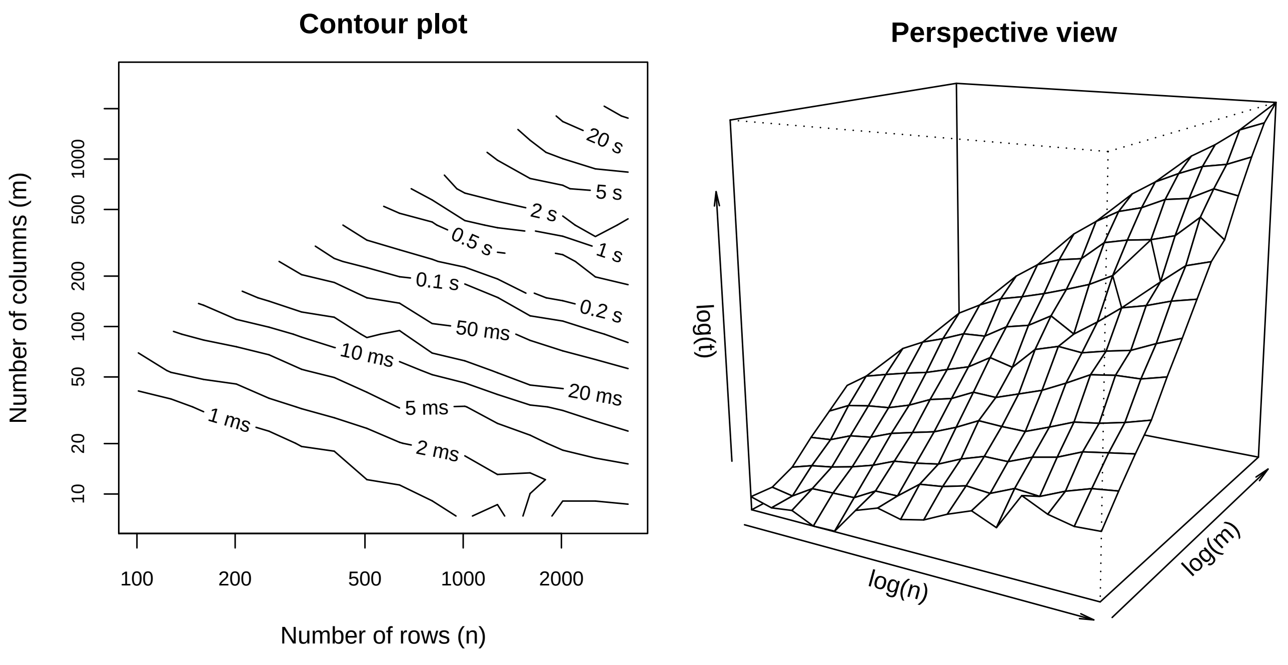


Figure 5.7: Mean NNLS computation times by n and m, within the context of the HCNM algorithm (contour plot and perspective view).

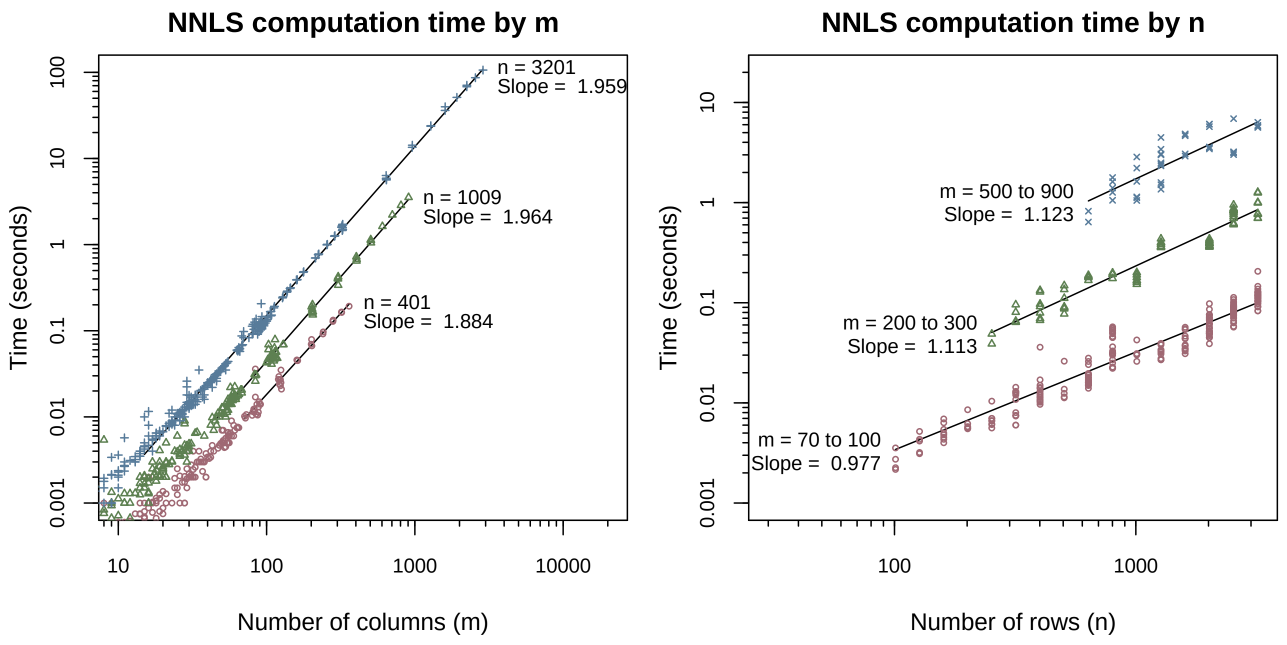


Figure 5.8: Computation times of NNLS within the HCNM algorithm, with fitted slopes.

Given the high correlation between m and $\hat{m}$, the log-linear model of time complexity cannot contain both those variables. This is because it would be very difficult to extract both coefficients. Therefore, the time complexity formula for NNLS within the context of HCNM is based only on n and m, and is expressed as $O(n^{\alpha}m^{\beta})$. To estimate the parameters α and β the following log-linear model was used:

$$\log(t) = \alpha \log(n) + \beta \log(m) + \gamma$$

where γ is the intercept term (of little interest). Table 5.4 shows that the estimated parameter values were α=1.03 and β=1.91.

Therefore, the practical time complexity of the NNLS algorithm, within the context of the HCNM algorithm, is estimated to be $O(nm^2)$.

Table 5.4: Estimated coefficients of log-linear model of NNLS times within HCNM.

	Estimate	Std. Error	t value	Pr(>|t|)
α	1.0323	0.0113	90.97	0.0000
β	1.9078	0.0087	218.36	0.0000
γ	−18.9807	0.0983	−193.13	0.0000

5.4 Time complexity of HCNM and CNM

The final experimental section of this chapter looks again at the HCNM and CNM algorithms to estimate the practical time complexity of both algorithms. The time complexity can be separated into the time complexity per iteration and the number of iterations required. This section focuses on the time complexity per iteration.

Table 4.3 showed the number of HCNM iterations required, assuming a sufficiently large block size has been chosen. It showed that the number of iterations remained approximately constant as n increased, and even decreased slightly for some values of r. However, the results for the case where $r = 0$ were different. In this case the HCNM algorithm did not create multiple blocks, and so was the same as the CNM algorithm. For the case when $r = 0$, the number of iterations increased slowly as n increased.

Both algorithms take as input a two-column matrix of pairs of values, with each row representing an observed censoring interval O_i. The two columns are the left and right end-points of the event time censoring intervals, denoted by L_i and R_i as in Section 2.1.1. The number of rows is denoted by n, which clearly affects the computation time.

During the set-up time, the algorithms derive the set of maximal intersections, which are the candidate support intervals I_j. The number of these is denoted by m_0 in this section. Then they derive the $n \times m_0$ clique matrix, which contains values $\delta_{ij} \in \{0, 1\}$ to specify whether or not I_j intersects with O_i (with $\delta_{ij} = 1$ meaning that it does). The final NPMLE solution is a subset of $\{I_j\}$ with size $\hat{m} \leqslant m_0$. Each iteration contains calculations that depend on m_0, such as calculating the matrix $\mathbf{S}$ and the vertex-directional gradient $\mathbf{d}$. Each iteration also contains calculations that depend on $\hat{m}$, such as creating the blocks and calls to NNLS. Note that the values of m_0 and $\hat{m}$ can be highly correlated. The computation time for a given value of n may also depend on $\hat{m}$ and/or m_0 in some manner that is to be determined.

Exact observations are identified by the fact that $L_i = R_i$. Some of these exactly observed values may be duplicated, due to rounding or other limitations in measuring the event times. Let the number of unique values from the exact observations be denoted by m_e. Each unique value that has been exactly observed must be included in the final support set, so $\hat{m} \geqslant m_e$. In practice, for large data sets with many exact observations, the exact observations often constituted the entire support set in the solution, meaning that $\hat{m} = m_e$. Since m_e has an impact on the final size of the solution, it also affects the time taken to derive the solution. For convenience, the proportion m_e/n of unique exact observations is denoted by r.

The aim of this experiment was to estimate the practical time complexity per iteration of the CNM and HCNM algorithms. This can be expressed in terms of one or more of the parameters n, m_0, m_e and $\hat{m}$. An example expression is that the computation time is of the order $O(n^\alpha \hat{m}^\beta)$, where the values of α and β are to be estimated from simulations.

At each iteration, the number of candidate support intervals being considered is determined by the dimension reduction technique. Although this number starts out small and then grows rapidly (Wang, 2008), the set of support intervals soon settles on precisely those that are in the solution. So, the number under consideration quickly settles on $\hat{m}$ and stays there. This gives good reason to use $\hat{m}$ in expressions of time complexity, rather than m_0 and m_e.

Methods

A total of 9760 simulated data sets were created, with parameters as follows:

- Number of observations, $n \in$ {200, 252, 317, 400, 504, 635, 800, 1008, 1270, 1600, 2016, 2540, 3200, 4032, 5080, 6400, 8063, 10159, 12800, 16127, 20319, 25600}
- Proportion of exact observations, $r \in$ {0, 0.025, 0.05, ..., 0.975}
- Case-k censoring, $k \in$ {2, 10}

Large numbers of different values for n and r were used to thoroughly explore the space of these two values. The HCNM algorithm was investigated with two replications of all combinations of the above parameters. Simulations using the CNM algorithm were restricted to values of $n \leqslant 3200$ and consisted of six replications.

Results

For each simulation, the computation time and number of iterations required were recorded. From those two values, the computation time per iteration was calculated. Let t denote the computation time per iteration of an algorithm.

Figure 5.9 shows perspective plots of $\log(t)$ against $\log(\hat{m})$ and $\log(n)$, for both CNM and HCNM. Both plots are approximately planar, which allows a log-linear model to be used in each case. The slight curve that is seen in both plots is noted but ignored, for the purpose of a rough estimation of the practical time complexity.

The time complexity per iteration for each algorithm is based on n and $\hat{m}$, and is expressed as $O(n^{\alpha}\hat{m}^{\beta})$. Note that $\hat{m} \approx r/n$ for larger values of n or r. Although each iteration contains some computations that depend on m_0, the high correlation between m_0 and $\hat{m}$ prevents this from being measured. Also, the more time-complex calls to NNLS depend more on $\hat{m}$.

To estimate the parameters α and β for each algorithm, the following model was used:

$$\log(t) = \alpha \log(n) + \beta \log(\hat{m}) + \gamma$$

where γ is the intercept term. Table 5.5 shows that the estimated parameter values were α=1.33 and β=1.39 for the CNM algorithm. In comparison, Table 5.6 shows that the estimated parameter values were α=1.38 and β=0.805 for the HCNM algorithm. This suggests that the HCNM algorithm reduces the dependency on $\hat{m}$ for the time complexity per iteration. The actual values are influenced by finite sample size effects. So they do not necessarily reflect the theoretical time complexity per iteration.

Table 5.5: Estimated coefficients of log-linear model of CNM computation times.

	Estimate	Std. Error	t value	Pr(>|t|)
α	1.33	0.01	260.51	0.00
β	1.39	0.00	370.48	0.00
γ	−17.22	0.02	−697.02	0.00

Table 5.6: Estimated coefficients of log-linear model of HCNM computation times.

	Estimate	Std. Error	t value	Pr(>|t|)
α	1.38	0.01	215.02	0.00
β	0.80	0.01	145.14	0.00
γ	−15.82	0.03	−572.79	0.00

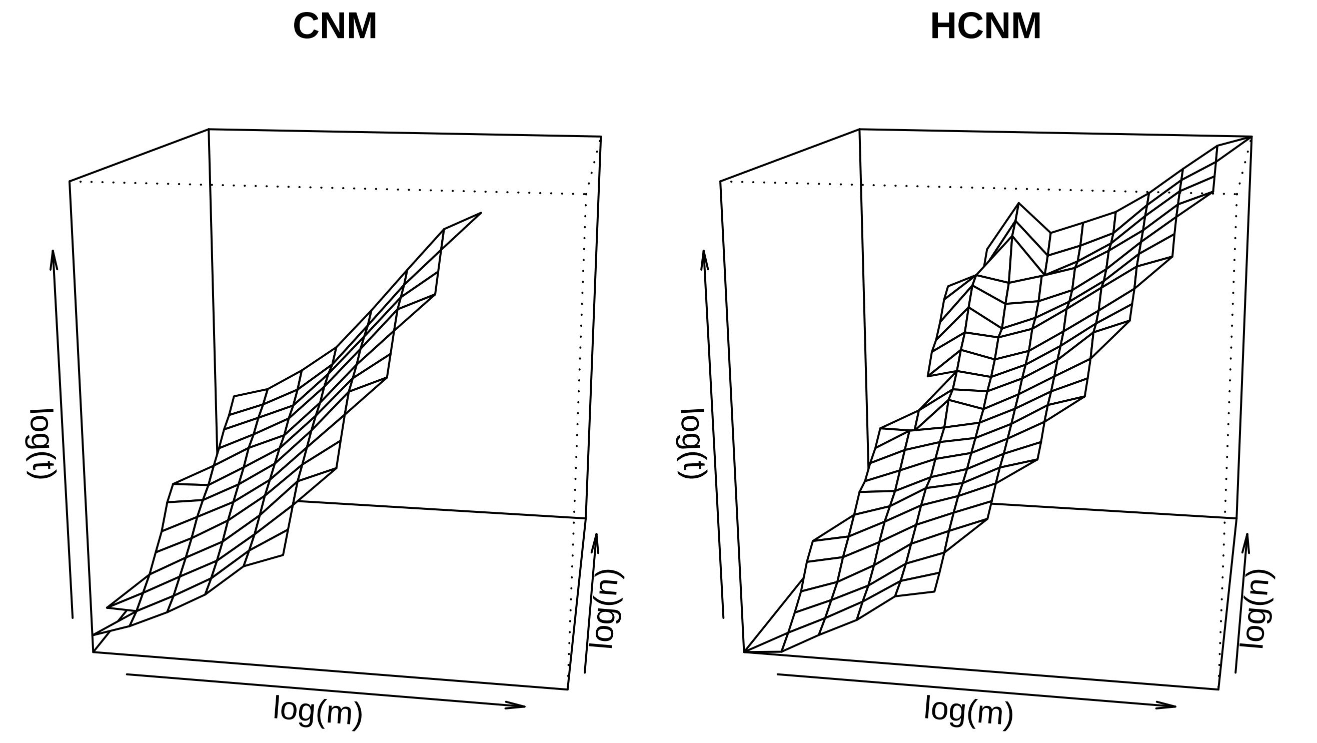


Figure 5.9: Perspective plots of CNM and HCNM computation time per iteration by $\hat{m}$ and n.

Theoretical time complexity

This part discusses theoretical expressions of the time complexity, per iteration, of the CNM and HCNM algorithms.

The time complexity of NNLS within the context of CNM (and HCNM) was estimated to be $O(nm^2)$, where n is the number of rows and m is the number of columns of the matrix provided to NNLS as input. In the limit, CNM requires $\hat{m}$ probabilities to be estimated, which means that m can be replaced by $\hat{m}$. The difference of one between the number of observations in the survival data set and the number of rows given to NNLS is insignificant in the limit as $n \to \infty$. The CNM algorithm uses NNLS exactly once per iteration and the NNLS task is the most time-complex task within CNM. This suggests that the time complexity, per iteration, of CNM is $O(n\hat{m}^2)$.

The default block size rule of HCNM (page 44) derives the block size b from the support set size. For time complexity in the limit, the support set size can be assumed to be $\hat{m}$, so the rule specifies $b = 15\log(\hat{m}) - 70$. The number of blocks κ is given by $\frac{\hat{m}}{b}$. Each iteration of HCNM calls NNLS κ times, with each call being of the order $O(nb^2)$. Therefore, the combined time complexity, per iteration, of this part of the HCNM algorithm is $O(nb^2\kappa)$, which simplifies to $O(n\hat{m}\log(\hat{m}))$, since $b \propto \log(\hat{m})$ and $\kappa \propto \hat{m}/\log(\hat{m})$.

Each iteration of HCNM must also perform the reallocation of probability among the κ blocks. It does this recursively, with time complexity at most $O(n\kappa \log(\kappa))$. Since $\kappa \propto \hat{m}/\log(\hat{m})$, this is less than the above expression for the task of redistributing probability within blocks. So it becomes insignificant in the limit, which was also seen in Figure 5.1.

Therefore the estimates of the time complexity, per iteration, are $O(n\hat{m}^2)$ for CNM and $O(n\hat{m}\log(\hat{m}))$ for HCNM. Results from simulations shown in Figure 4.2 indicate that the number of iterations stabilises, assuming a suitable block size has been chosen. Therefore, assuming the number of iterations remains constant, the overall time complexity of the HCNM algorithm is also $O(n\hat{m}\log(\hat{m}))$. Results from simulations do not contradict these theoretical derivations.

5.5 Summary

In this chapter, the practical time complexity of algorithms was estimated using empirical methods. Practical estimates of time complexity may be different to theoretical time complexity. Theoretically derived expressions capture only the most complex element of an algorithm, ignoring lesser components that contribute a vanishingly small proportion of computational time. In practical applications of an algorithm, data sets cannot be of unlimited size. There may be factors that affect the empirical estimates of time complexity that would be ignored by theoretical estimates.

The conclusion of this chapter is that the NNLS algorithm has an estimated time complexity of approximately $O(nm^2)$, particularly within the context of the CNM and HCNM algorithms. This provided the motivation for the enhancements that were made to CNM to create HCNM. HCNM divides the problem up into small blocks before calling NNLS, thereby keeping the value of m small.

Assuming that NNLS has time complexity $O(nm^2)$, the time complexity of the CNM and HCNM algorithms, per iteration, was derived. Since NNLS is the most time complex component of the CNM algorithm, and CNM makes use of NNLS exactly once per iteration, the time complexity of CNM, per iteration, was estimated to be $O(n\hat{m}^2)$. In contrast, the time complexity of the HCNM algorithm, per iteration, was estimated to be $O(n\hat{m}\log(\hat{m}))$, which shows a considerable improvement.

Chapter 6

Comparison of NPMLE algorithms

This chapter investigates the performance of the new HCNM algorithm against three existing methods for locating the solution to the nonparametric maximum likelihood estimate (NPMLE) problem in survival analysis. Performance is measured by the mean computation time over a number of simulations. The three existing methods are the Constrained Newton Method (CNM), the Subspace-based Newton method with dimension-reduction (SBNDR) and the ICM/EM hybrid algorithm with dimension-reduction (ICMDR-EM). More details about these methods are provided in Chapter 2. These three methods were studied by Wang (2008) and the conclusion was that no algorithm outperforms all the others in all situations, as shown in Table 2.1.

To design a robust test of HCNM against other algorithms, there was a need to create a new way of generating the simulated data, separate from the method that was used in Chapter 4. Furthermore, the simulated data needed to be as realistic as possible.

Section 6.1 describes a real-world example of a large survival data set, containing both exact observations and interval-censored observations. Section 6.2 describes a process of generating simulated data, which was developed specifically for the research presented in this chapter. Using these simulated data sets, the four algorithms were compared on their speed of computation, as described in Section 6.3. Finally, Section 6.4 provides the results from these simulations. The results indicate that the new HCNM algorithm succeeded in outperforming its competitors in nearly all situations that were attempted.

6.1 Real-life example

The study reported by Kumwenda et al. (2008) provides a real-life example of the type of data under consideration. This study looked at the infection-free survival of infants being breast-fed by mothers with the HIV-1 virus in Malawi. Infants who were unfortunate enough to be already infected at birth were excluded from the study.

More than 3000 infants were randomised into three groups and were monitored for up to two years. The three groups were a control group (receiving standard treatment) and two types of extended treatment. The aim of the study was to test whether these extended treatments were more effective in preventing the transmission of the virus from the mother to the infant in the breast milk.

In this example there are two types of event that the survival analysis must take into account; the first is infection and the other is death. Either of these events terminates the infection-free survival period for the infant. Infants were tested for infection at 11 prescribed follow-up times, starting at one week after birth, up to the age of 24 months. Although these times were scheduled, the actual times that individual mothers arrived varied from the schedule, and some follow-ups may have been missed altogether. When the infant was found to be infected at one of these follow-ups, all that is known about the actual time of infection is that it fell somewhere in the intervening time interval since the previous (infection-free) follow-up visit. This is an interval-censored event time. When an infant died, from whatever cause, the date of death was recorded and this represents an exact observation of the survival time.

6.2 Generating realistic random data

This section looks at a new method of randomly generating realistic data for the purpose of running simulations and testing algorithms. The method is based on the real-life example described above. Appendix A provides source code for the method, as an **R** function called `ricsa` (page 92).

The model for simulating data as seen in Kumwenda et al. (2008) assumes there are two independent types of event that can happen to each subject. One type of event is observed immediately, so its time of occurrence is known exactly. The other cannot be observed when it happens, so it must be discovered by a process of regular inspections. To model these two types of events, two Weibull distributions were used, one for exact and one for censored observations. The Weibull distribution was chosen because it is commonly used in parametric survival analysis. Each of the two Weibull distributions has two parameters, the shape a and the scale λ. Values of the shape parameter are used to determine the nature of the underlying hazard rate (the instantaneous rate at which events occur). Values of $a < 1$ represent a decreasing hazard rate, the value $a = 1$ means a constant hazard rate, and values $a > 1$ mean an increasing hazard rate. The Weibull distribution is often used in parametric survival analysis because it is able to model these three different situations. The scale parameter λ specifies the time scale in which the events tend to occur. When $a \geqslant 1$ the mean of the distribution is near the value of λ. The actual mean of the distribution also depends on the shape parameter and is given by $\lambda\Gamma(1+1/a)$.

For each subject, two event times are drawn, one from each of the Weibull distributions. If the event time from the first distribution has the smaller value then it is returned as the event time for that subject. This is an exact observation. Otherwise, it is the censored event that occurred first. In that case, the event time from the second distribution is used but is replaced by a censoring interval. The censoring interval is created from a series of inspections. The two consecutive inspection times that surround the actual value are returned as the censoring interval.

For the schedule of inspection times, Kumwenda et al. (2008) chose follow-ups at 1, 3, 6, 9 and 14 weeks and at 6, 9, 12, 15, 18 and 24 months after birth. These values (converted to days) were used in the simulated data method. To model the variation from scheduled times, an additional delay was added, independently for each follow-up visit for each subject. This delay was assumed to follow an exponential distribution with a mean equal to 5% of the time interval between followups.

When event times are measured, they are often expressed in whole numbers of a unit of measurement (such as seconds or days) or otherwise rounded. This rounding has an important effect on the number of unique exact observations. Rounding to fewer decimal places means more of the exact observations will coincide with one another. So the number of unique exactly measured times will be smaller. This has an impact on the size of the NPMLE solution, and therefore on computation times. To simulate this aspect of real data, an additional parameter was introduced to specify the number of digits that the event times should be rounded to.

An example simulated data set

To illustrate the simulated data that this method generates, an example data set containing 1000 observations has been created. Each observation consists of the two end-points L_i and R_i of its censoring interval. Of the 1000 observations, 245 were exact (with $L_i = R_i$) and the other 755 were interval-censored. Among the interval-censored observations, five were left-censored and 270 were right-censored.

The data set was created using parameters as follows. The distribution for the exactly measured events was a Weibull(2,1000) distribution. For the censored events, a Weibull(1,1000) distribution was used. Eleven inspection times were scheduled at $t \in \{7, 21, 42, 63, 98, 183, 274, 365, 456, 548, 730\}$ days, with additional random times added for delays, as described earlier. The resulting times were rounded to the nearest whole number of days.

Figure 6.1 illustrates the clique matrix for the example simulated data set. For this illustration, the data set has been sorted by L_i and R_i. There were 302 candidate support intervals constructed from the observations, which were also sorted. The thin portions in the figure are exact observations and the wider parts are censoring intervals. Details of the actual values of the end-points for the observed and support intervals have been omitted. This is because those values are not used in the allocation process; only the clique matrix is used.

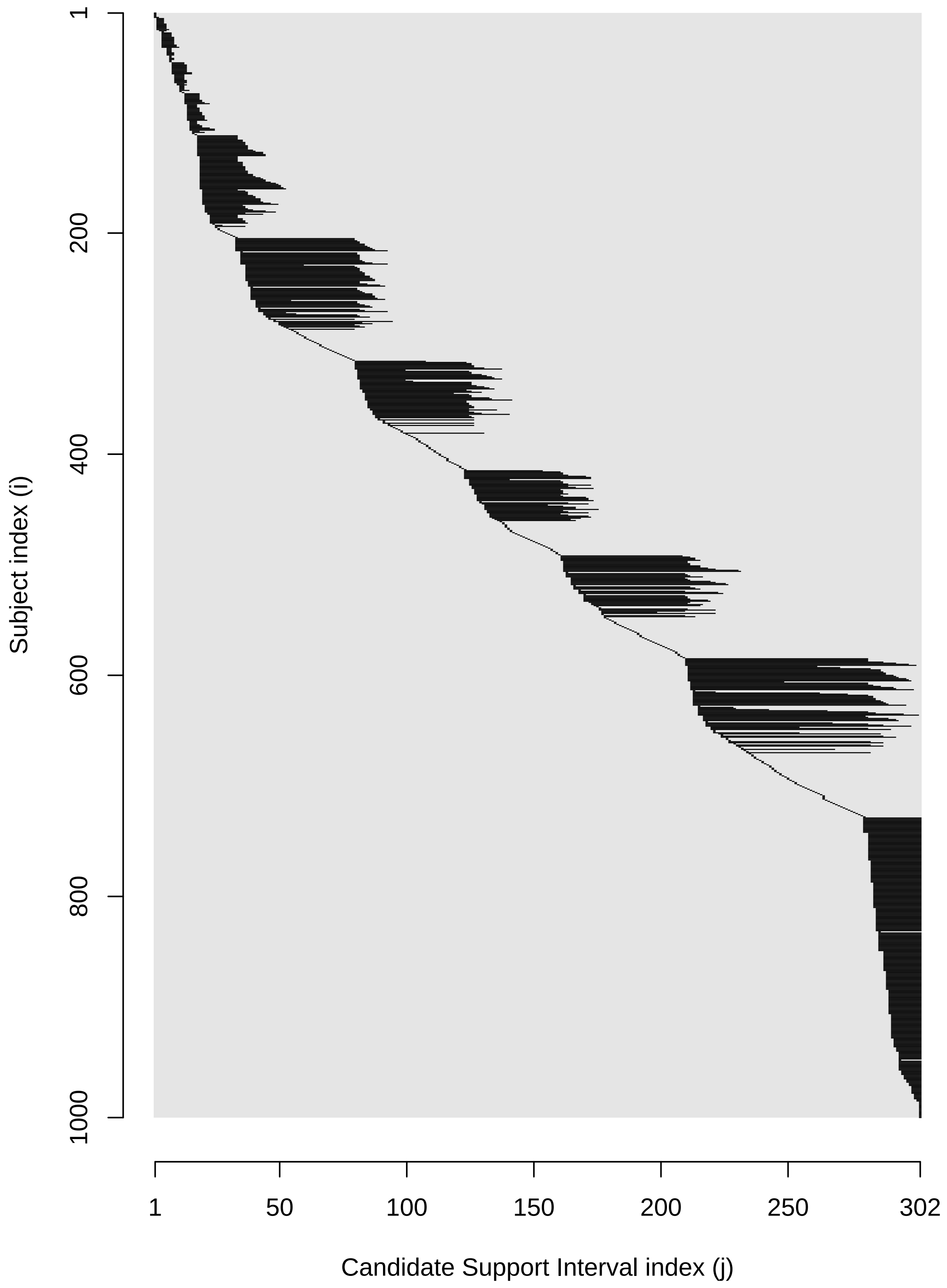


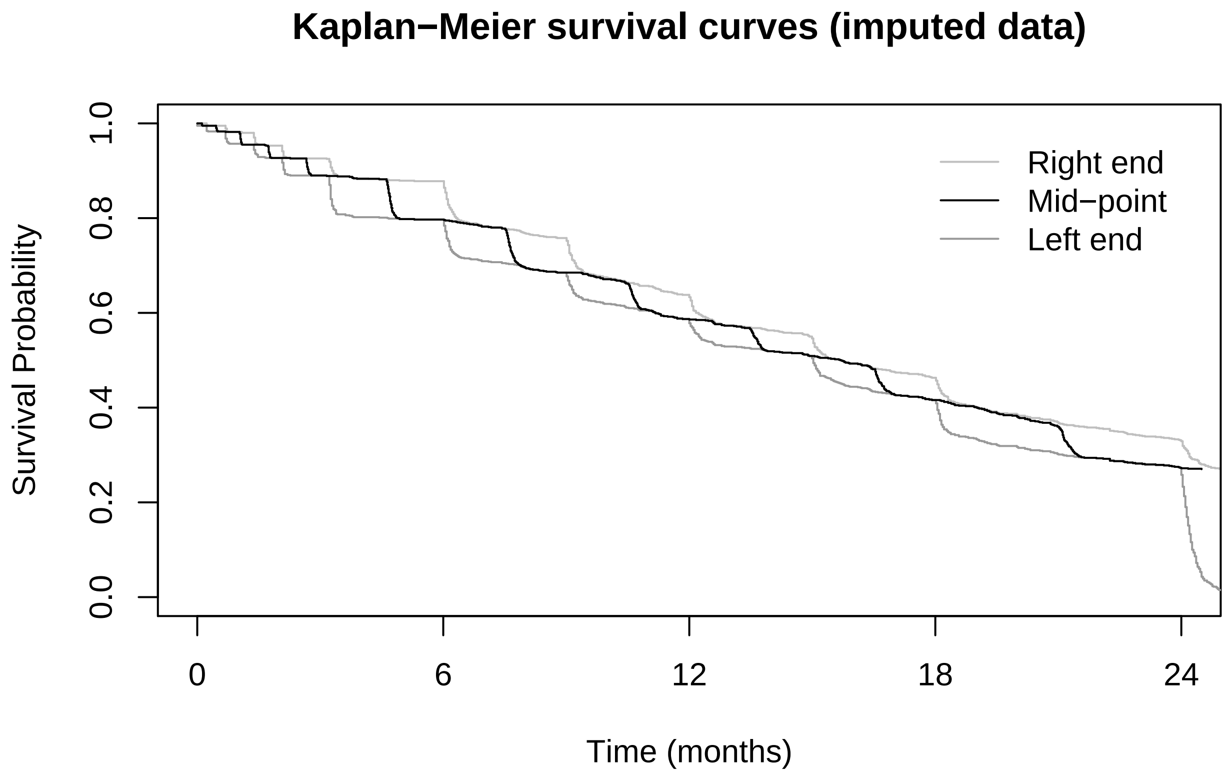
Figure 6.1: Clique matrix of example simulated interval-censored data. Black areas indicate where the subjects' censoring intervals intersect the candidate support intervals.

Figure 6.2 shows four estimates of the survival function, generated from the example simulated data set. Three methods of imputation were applied, so that the popular Kaplan-Meier product limit estimator (Kaplan and Meier, 1958) could be used to create the survival curve. Each of these methods of imputation replaces the censoring interval $(L_i, R_i]$ by a single value from that interval. The value chosen in the three cases was the left end-point, the mid-point and the right end-point. In all three cases, the censoring mechanism has an obvious unwanted impact on the shape of the estimated survival function.

The second plot in Figure 6.2 illustrates the nonparametric maximum likelihood estimate (NPMLE) of the survival function. The NPMLE provides a better, smoother estimate than the KM estimates do. This is because it uses more information contained in the data. For the same reason, the NPMLE will increase the power of statistical tests for detecting treatment effects, perhaps markedly, compared to tests based on imputed data.

Subsection 2.1.3 mentioned rectangular regions where the NPMLE was unable to pin down the survival function. There is one such region in Figure 6.2. It is quite small and can be found in the top left corner of the NPMLE plot. Generally, those rectangular regions are found in smaller data sets and data sets containing few exact observations. As the number of exact observations becomes large, the exact observations tend to dominate the set of support intervals that receive positive probability mass (as seen in Table 4.5 where $\hat{m} = nr$ for larger cases).

Section 4.1 described a different method of generating simulated data, which was used by Wang (2008) and also in Chapter 4 and Chapter 5. Since the inspection times were random in that method, the resulting data sets more closely resembled mixed case censoring, rather than case k censoring with the k inspection times fixed. In contrast, the censoring used in this chapter uses a schedule of inspections (varying only slightly), with the result that the censoring leaves a mark on the survival curve. The Kaplan-Meier estimates presented in Kumwenda et al. (2008, figure 2) also exhibit an artifact of the censoring mechanism.



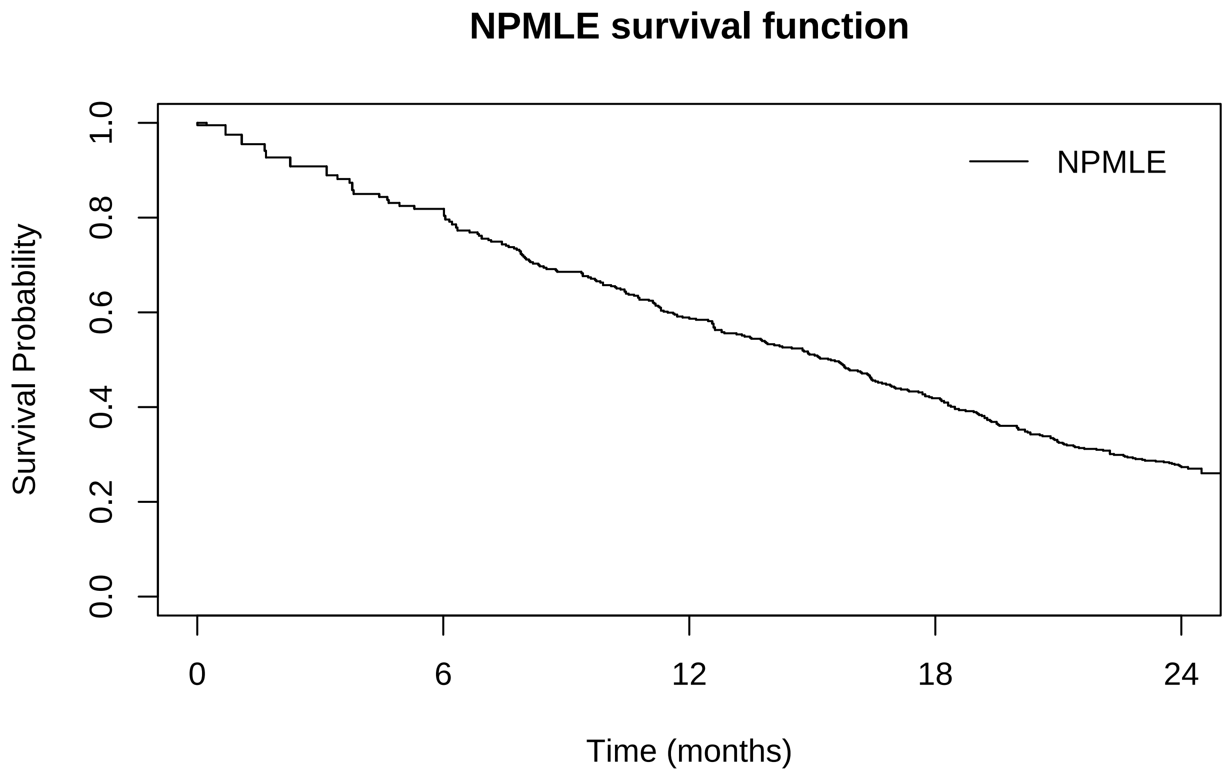


Figure 6.2: Survival functions for a simulated interval-censored data set, comparing Kaplan-Meier curves (derived from imputed data) to the NPMLE.

6.3 Comparison experiment

This section describes the main experiment to test the performance of the new HCNM algorithm against three existing methods. The four algorithms were compared using a wide variety of simulated data sets. Each of the four algorithms was used to compute the NPMLE solution on each data set. All these computations were timed and the mean computation times were collated.

A total of 8148 simulated data sets were randomly generated, using the process described in Section 6.2. Five different parameters were varied to create this variety. The parameters used to define the simulated data sets were as follows:

- Number of observations, $n \in \{200, 400, 800, 1600, 3200, 6400\}$
- Shape of Weibull distribution for exact observations, $a_e \in \{0.5, 1, 2\}$
- Scale of Weibull distribution for exact observations, $\lambda_e \in \{500, 1500, 4500\}$
- Scale of Weibull distribution for interval-censored observations, $\lambda_c \in \{500, 1500, 4500\}$
- Number of decimal places of rounding, each value from $\{0, 1, 2\}$

Each data set was a simulated survival analysis result containing both exact observations and interval-censored observations. The proportion of exact observations varied randomly, depending mainly on the two scale parameters, λ_e and λ_c. A smaller value of the scale parameter meant that more of that type of event occurred sooner. So higher proportions of exact observations were seen with smaller values of λ_e and larger values of λ_c.

The number of unique values among the exact observations was affected by the rounding parameter. For each simulated data set, the proportion of unique exact observations was calculated, and is denoted by r. Rounding to two decimal places was used to produce data sets within the space of higher r values.

For this experiment, the Weibull shape parameter for exact observations was varied. The nature of the NPMLE solution is that it is uninterested in the actual shape of the distribution of events. For this reason, the shape parameter of the censored event time distribution was not also varied.

6.4 Comparison results

Table 6.1 shows the results for a sample of ten of the individual simulated data sets. For each data set, the resulting computation time for each of the four algorithms is shown. In this table, most of the results for the HCNM algorithm can be seen as faster than the other three algorithms, sometimes remarkably so.

For the fourth data set, the CNM algorithm was slightly faster than HCNM. Referring to the default block size rule, that data set sits on the fence between the single block approach of CNM and the multiple block approach of HCNM. This would suggest that there is a transitional space where it is unclear which approach is clearly better. For the sixth data set the SBNDR algorithm was slightly faster.

Table 6.1: Computation time (seconds) for four algorithms on ten simulated data sets.

n	a_e	λ_e	λ_c	Round d.p.	r	ICMDR-EM	SBNDR	CNM	HCNM
400	2	1500	4500	2	0.2	0.36	0.18	0.20	0.13
400	1	500	4500	0	0.6	0.83	0.56	1.05	0.34
800	0.5	500	4500	1	0.6	3.39	2.04	6.54	1.30
1600	1	4500	500	1	0.1	8.47	2.02	1.45	1.98
1600	0.5	500	1500	0	0.2	9.89	4.24	19.66	3.98
3200	0.5	1500	500	0	0.1	37.20	7.56	25.65	7.90
3200	2	500	500	1	0.4	113.13	52.58	165.30	17.79
3200	0.5	500	1500	1	0.4	42.32	27.87	358.31	22.24
6400	1	1500	500	2	0.2	203.21	87.08	441.09	39.15
6400	0.5	500	1500	2	0.5	338.85	399.87	5766.34	156.11

Figure 6.3 shows the aggregated results of applying the four algorithms to all the simulated data sets. Results were aggregated by taking the mean computation time. A separate plot is given for each size of data set, $n \in \{200, 400, 800, 1600, 3200, 6400\}$. Mean computational time is shown on the y axis and the proportion of unique exact observations is shown on the x axis.

The HCNM results are shown as a solid black line in each plot. In all situations, the performance of HCNM was pleasing. For small data sets ($n \leqslant 400$) the HCNM algorithm performed no worse than the best of its competitors. The same can be seen for small values of r in the plots for larger values of n. In a few situations, the SBNDR algorithm was faster than HCNM by a narrow margin of at most 6%.

The HCNM algorithm was clearly the fastest algorithm for the largest data sets and in situations with a largest proportion of unique exact observations. Results for the CNM algorithm deteriorated rapidly as the parameter r increased. The two algorithms ICMDR-EM and SBNDR showed a similar, though less extreme, deterioration when applied to large data sets.

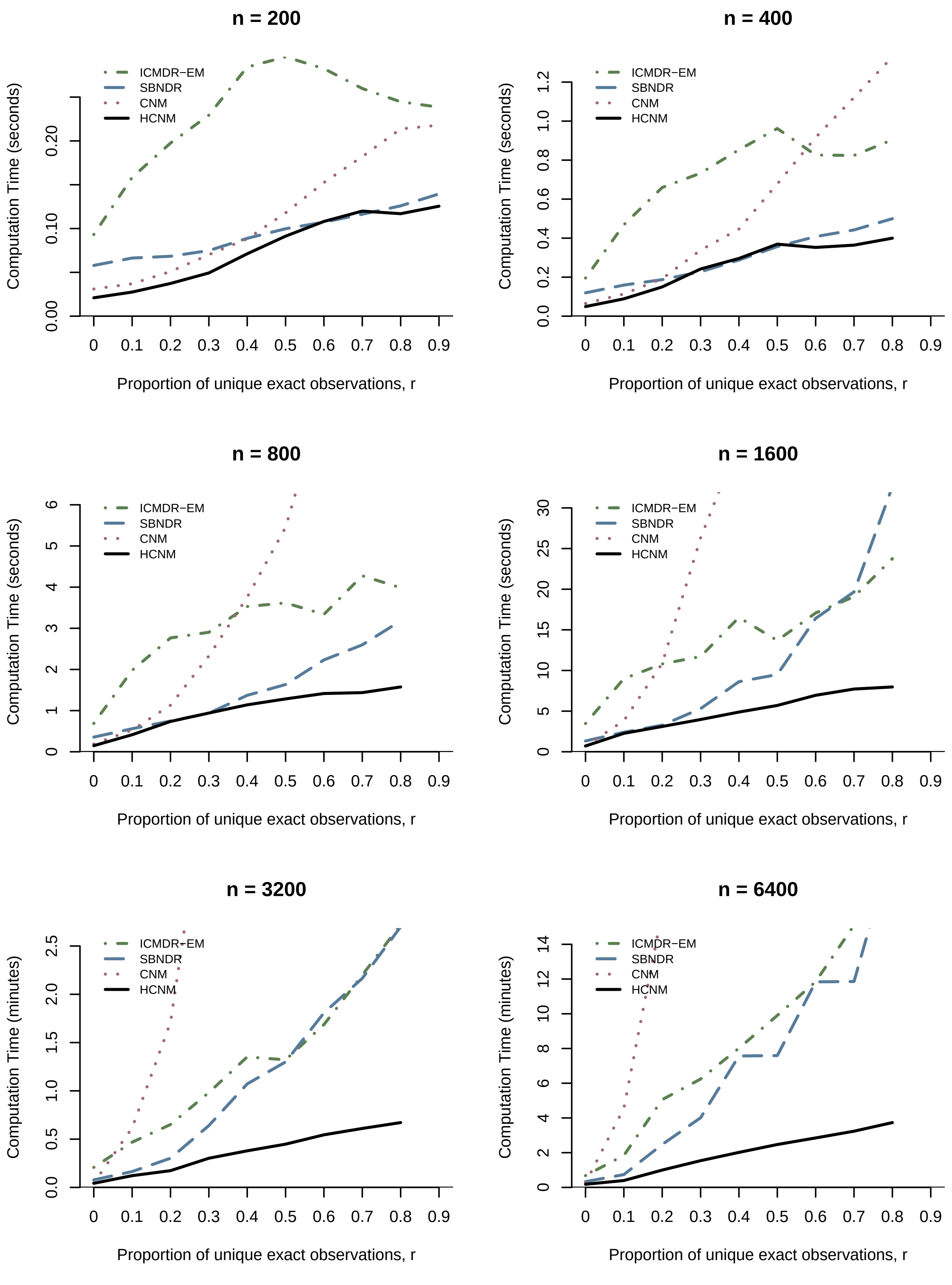


Figure 6.3: Mean computation time, comparing HCNM with competing algorithms.

Chapter 7

Conclusions

The research for this thesis focused on survival data with both exact and general interval-censored observations, in any proportion. The research problem was the task of finding the nonparametric maximum likelihood estimate (NPMLE) of the survival function for such a data set. Since there is no closed-form mathematical solution available for this problem, the best known approach is to use numerical optimisation techniques in an iterative algorithm.

Several algorithms for solving this problem were found in the literature. However, no single algorithm was the best in all situations. Further analysis of the problem suggested potential improvements that could be made to one of the algorithms. That algorithm was the Constrained Newton Method (CNM). The improvements made use of the mixture structure of the problem. More specifically, the improvements used an expression of the problem in the form of a hierarchical and recursive 'mixture of mixtures' model. This resulted in the Hierarchical Constrained Newton Method (HCNM).

7.1 Discussion

The Achilles' heel of CNM

The CNM algorithm possesses an impressive quadratic order of convergence, meaning that it requires a very small number of iterations to reach the solution. In terms of computation time, however, the performance of CNM was very good only on data sets with very few or no exact observations.

Exact observations are the Achilles' heel of the CNM algorithm; they are the reason for its disappointing results. As the proportion of exact observations increased, the performance of CNM deteriorated rapidly. The reason was seen to be the time complexity per iteration of the CNM algorithm, which was derived from the time complexity of the NNLS algorithm. Since each unique exact observation must be a support interval in the solution, having many of these means a larger number of probabilities must be found. The time complexity of NNLS is such that this number has a particularly detrimental effect on the computation time.

The new Hierarchical CNM algorithm was created to address this issue.

The research problem and the HCNM algorithm

The CNM and HCNM algorithms make use of a quadratic approximation to the log-likelihood function. This quadratic approximation is based on the Taylor series expansion, using the first two derivatives, the gradient vector and Hessian matrix. Computing the full Hessian matrix is computationally expensive and indeed unnecessary. The nature of a typical Hessian matrix was examined and this suggested a way of approximating it. The approximation is based on a diagonal band of values close to the diagonal of the matrix. One way of achieving this is to partition the set of support intervals into a number of separate blocks, which is what the HCNM algorithm does.

The numerical optimisation problem, of allocating probability mass to a large set of candidate support intervals, has been shown to have a mixture structure. Moreover, this mixture structure can be expressed hierarchically and recursively in terms of a mixture of mixtures model. The mixture of mixtures model allows the problem to be solved by using a divide and conquer approach. The set of mixture components can be partitioned into a number of separate blocks. Within each block, the optimisation of the mixing proportions can be performed, subject to keeping the probability mass of the block unchanged. Then the blocks can be treated as mixture components of the whole problem, and thus the mixing proportions of the blocks can be optimised. This two-step process can be repeated iteratively until a stopping criterion has been satisfied. Furthermore, the second step can be broken down further into blocks of blocks and so the problem can be solved in a hierarchical and recursive manner.

The new HCNM algorithm approaches the problem by partitioning a potentially large set of support intervals into a number of blocks. It redistributes probability mass within each block in turn. Then it calls itself recursively to redistribute probability mass among the blocks, making use of the mixture of mixtures model.

Fine-tuning the HCNM algorithm

Chapter 4 explored several aspects of the HCNM algorithm. Results from simulations assisted in the fine-tuning of the HCNM algorithm to achieve high performance.

The HCNM algorithm examines the data set it is given to decide how many blocks to use. This is achieved by means of a default block size rule, which was derived from the results of many simulations. When the number of exact observations is below a set threshold, HCNM attacks the problem as a whole (using a single block) and so it is the same as CNM. For the transition between CNM and HCNM, a threshold of 150 exact observations seemed to be best. Below this number, HCNM could not outperform CNM. Above this threshold HCNM improved on the performance of CNM, sometimes markedly. When the number of exact observations was 150 or more, the default block size was determined as follows. Suppose there are m_s support intervals under consideration at iteration s of the algorithm. Then the block size for this iteration was determined as the greater of 20 and $15\log(m_s) - 70$, rounded to the nearest integer.

Several methods of creating the blocks were investigated to explore different block structures. Some methods varied the positions of blocks at each iteration of the algorithm, which was achieved by shifting their positions by half a block width. This was done with the expectation that it would allow probability mass to be more efficiently transported around the entire support set. One method also varied the block sizes, depending on values in the vector $\mathbf{v} = \text{diag}(\mathbf{S}^T\mathbf{S})$. For this method, smaller blocks were created in regions of the support set with high values in $\mathbf{v}$. However, this approach was found to make the algorithm slower, perhaps due to the additional computation time required to calculate $\mathbf{v}$. Fuzzy blocks, which overlap one another in some proportions at each iteration, were also investigated. The performance of HCNM using fuzzy blocks was inferior to that with non-overlapping blocks. The reason for this is related to the time complexity of NNLS at each iteration. Since the overlapping blocks must necessarily be either larger or more numerous than non-overlapping blocks, this approach meant additional NNLS computation was required.

Results from this exploration of different block structures indicated that non-overlapping blocks of equal sizes was the best choice. The shifting of block positions by half a block width was not seen to significantly increase or decrease computation times.

The HCNM algorithm calls itself recursively to reallocate probability mass among the blocks. Results from simulations indicated that the number iterations for recursive calls to HCNM should be limited to two.

Time complexity

Chapter 5 explored the time complexity of algorithms. The focus was primarily on estimating the practical time complexity. The practical time complexity is an empirical estimate derived from simulations using data sets of varying sizes. Results from those simulations help to explain the poor performance of CNM and motivate the changes made to produce HCNM.

The time complexity of the NNLS algorithm, within the context of the CNM and HCNM algorithms, was estimated to be approximately $O(nm^2)$. The parameter m here represents the number of probabilities that CNM or HCNM needs to have estimated.

For iterations of CNM, the value of m quickly settles on $\hat{m}$, the size of the final solution set. The CNM algorithm calls NNLS once per iteration, using the entire dimension-reduced set of candidate support intervals. This can mean a large value of m and therefore a very time-consuming computation, because of the factor m^2 in the time complexity of NNLS. The NNLS time complexity per iteration of CNM was estimated to be $O(n\hat{m}^2)$.

By comparison, the HCNM algorithm breaks the support set up into κ blocks and calls NNLS once per block per iteration. Each block has approximately $b = \hat{m}/\kappa$ probabilities to estimate, and there are κ such blocks. So the NNLS time complexity per iteration of HCNM is $O(nb^2\kappa)$. The default block size rule of HCNM means that $b \propto \log(\hat{m})$ and so $\kappa \propto \hat{m}/\log(\hat{m})$. Therefore, the NNLS time complexity per iteration of HCNM can be expressed as $O(n\hat{m}\log(\hat{m}))$, which is a considerable improvement over that of CNM. Of course, there is an additional task in each iteration of HCNM, that of reallocating probability mass among the blocks. Simulations suggested that this contributed a decreasing, possibly vanishing, proportion of the computation time.

Algorithms in competition

Chapter 6 described a simulation study to compare the new algorithm against three other known algorithms for finding the NPMLE survival function.

The new HCNM algorithm was compared against the CNM algorithm and two other existing algorithms. A hybrid of the Iterative Convex Minorant algorithm with the Expectation Maximisation algorithm was found in the literature. It was further enhanced to include the dimension reduction technique (Wang, 2008). This enhanced hybrid was included in the comparisons, with the label ICMDR-EM. The fourth algorithm to be included in the comparison was the Subspace-based Newton method, also enhanced to include the dimension reduction technique. It was labelled SBNDR.

The comparisons were based on simulated survival data. A new method for generating simulated data was derived from a real-life example of survival data containing both exact and interval-censored observations (Kumwenda et al., 2008). A wide variety of simulated data sets was generated, based on this example. To each data set, each of the four algorithms was applied to find the NPMLE solution. The computation time of each calculation was recorded and results were aggregated by taking the mean computation times.

In nearly all situations, the HCNM algorithm was the best, having the smallest mean computation time. In some situations the SBNDR algorithm was faster than HCNM, but only by a narrow margin of at most 6%.

The results for HCNM indicated that a transitional phase existed, at about the number of exact observations that marks the switch from a single block to multiple blocks. For larger data sets and data sets containing larger numbers of exact observations, the HCNM algorithm was clearly the best (see Figure 6.3). As these numbers increased, the gap between HCNM and the next fastest algorithm widened. This suggests that the HCNM algorithm possesses a more favourable time complexity than the other algorithms.

7.2 Suggestions for further research

This section presents some ideas for further research that were identified during the study. Of the ideas listed below, there are two that seem particularly promising. The first is to extend the new algorithm to the situation of bivariate or even multivariate survival data with general censoring. The second is to apply the new algorithm to other problems that can be expressed by a mixture model, particularly those having many mixture components. This kind of problem could lend itself to the hierarchical mixture of mixtures model.

For data sets with no exact observations, HCNM never creates multiple blocks, meaning that its performance is the same as CNM. Results for large data sets in this situation indicated that the number of iterations required by the CNM algorithm increased as the size of the data set increased. This may indicate that the default block size rule of HCNM should be modified so that multiple blocks are used even when there are no exact observations. The number of exact observations is used in the default block size rule as a proxy for the unknown size of the final solution.

Multivariate survival data

During this research, the HCNM algorithm was applied to univariate survival data. The HCNM algorithm may also be applicable to bivariate and, more generally, multivariate survival data. In such a case, the support set would be broken up into blocks by dividing up the multidimensional space of event times into regions of similarity.

Maathuis (2005) studied the reduction step of the NPMLE problem for bivariate data, implementing a 'Height Map' algorithm. Once this has been done and the clique matrix has been created, the HCNM algorithm can then be used to find the NPMLE of the mixing proportions.

Other types of data having a mixture distribution

The HCNM algorithm, perhaps with some further modification, may be useful in solving the NPMLE problem in more general contexts. This suggestion comes from noting the way HCNM calls itself to solve the mixture problem of reallocating probability mass among its blocks. It does this by creating a matrix of component mixture densities and then solving for the NPMLE of the mixing proportions.

The performance of HCNM in such wider contexts has not been considered and the algorithm may require further fine-tuning before providing good results. In particular, the best block size may be very different in different contexts.

Apply the block idea to other algorithms

The dimension reduction technique was successfully used by Wang (2008) to enhance the ICM-EM hybrid algorithm and the SBN algorithm, resulting in the algorithms referred to in this thesis as ICMDR-EM and SBNDR. The hierarchical divide and conquer approach may also improve the performance of the ICMDR-EM hybrid algorithm and the SBNDR algorithm.

7.3 Conclusions

A new algorithm was developed during this research and was called the Hierarchical Constrained Newton Method (HCNM). It uses a divide and conquer approach to solve the NPMLE problem. The new algorithm was based on an existing algorithm, the Constrained Newton Method (CNM), to which two enhancements were implemented. The first enhancement was to break the problem up into small blocks. The numerical optimisation is performed for each block in turn. When this has been done for all blocks, the second enhancement was to perform the numerical optimisation among the blocks, treating each block as a mixture component. This task makes use of the 'mixture of mixtures' structure of the problem and was implemented by use of recursion in the HCNM algorithm.

Through a number of experiments, the new algorithm was fine-tuned so that it performed well on a wide variety of data sets. The most important experiment looked for the best block size to use in various situations. From those results, a default block size rule was derived.

The time complexity of algorithms was explored using empirical methods and simulated data. Results from this work helped to explain the poor performance of CNM, and thus motivated the enhancements that created the HCNM algorithm.

An experiment to test the new algorithm against three existing algorithms was conducted. Many and varied realistic simulated survival data sets were created, on which the four algorithms were tested for computational speed. The new HCNM algorithm generally outperformed the other three algorithms in nearly every case.

Appendix A

Source Code

This appendix provides source code for the new HCNM algorithm and for the simulations that were performed to test it.

Simulations were performed and algorithms were developed making extensive use of the statistical programming language **R** version 2.8.0 (R Development Core Team, 2008). Note that the source listings given below have passed through the `parse` function, which strips out comments and reformats the code according to its own rules. Original source files are available (contact the author). Files containing results from simulations are also available.

Hierarchical constrained Newton method, `HCNM`

Description

Implements the new Hierarchical Constrained Newton Method. Returns an object of class `icsurv`, as in the `Icens` package (Gentleman and Vandal, 2008). This object includes information about the support intervals $\{I_j\}$ and their assigned probability masses $\boldsymbol{\pi}$. Depends on the **R** package provided by Wang (2007a), which in turn makes use of Fortran code for the NNLS algorithm, provided by Lawson and Hanson (1974). The package provided by Mullen and van Stokkum (2007) also provides the Fortran NNLS algorithm, but this has not been used here.

Arguments

`LR`	Matrix ($n \times 2$) with columns of Left and Right endpoints of censoring intervals.
`CM`	Clique Matrix ($n \times m$) of logical values defining intersection of support intervals (columns) with original observations (rows). Or, more generally, a matrix of the conditional probabilities making up the mixture components of a finite mixture model.

`pguess`	Initial guess of the $\boldsymbol{\pi} \in \mathbb{R}^m$ probability vector.
`maxiter`	Maximum number of iterations to try before quitting.
`dtol`	Tolerance factor, used to compare the gradient with the log-likelihood value to determine the main stopping criterion.
`blockpar`	Parameter to control the number of blocks and their size. Zero means don't do blocks (i.e. the same as the CNM algorithm). Values greater than one specify the exact block size to use, if possible. Values less than one mean raise the current size of the support set to this power, e.g. if there are 400 support points and blockpar=0.4 then there will be $11 \approx 400^{0.4}$ blocks each of size 36 or 37.
`overlap`	Code (1 to 7) indicating the style of block structure. See Section 4.4 for details.
`timing`	Should detailed timing information be collected within the algorithm?
`depth`	For internal use only: the current depth of recursion.

Note that there are two different ways of calling this function. One is to supply the left and right endpoints directly, via the `LR` argument. In this case, the algorithm derives the clique matrix and initial $\boldsymbol{\pi}$ vector (called `p` in the code). The other usage is to supply the clique matrix (or a more general matrix of mixture component densities) directly, together with an initial guess of the $\boldsymbol{\pi}$ vector.

Code

```
R> HCNM

function (LR, CM = NULL, pguess = NULL, maxiter = 30, dtol = 1e-06,
    blockpar = NULL, overlap = 1, timing = FALSE, recurs.maxiter = 2,
    depth = 1)
{
    if (timing)
        times <- new.timing(c("setup", "misc", "nnls", "linesearch",
            "blocks", "alloc"))
    if (missing(CM)) {
        A.info <- A.matrix(LR)
        CM <- A.info$A
        intmap <- with(A.info, rbind(left, right))
    }
    else {
        if (missing(pguess))
            stop("Must provide 'pguess' with CM.")
        if (!missing(LR))
            warning("CM and LR both provided.  LR ignored!")
        intmap <- NULL
    }
    converge <- FALSE
```

```
n <- nrow(CM)
m <- ncol(CM)
m1 <- 1:m
nblocks <- 1
maxdepth <- depth
i <- rowSums(CM) == 1
r <- mean(i)
if (is.null(pguess)) {
    j <- colSums(CM[i, , drop = FALSE]) > 0
    while (any(c(FALSE, (i <- rowSums(CM[, j, drop = FALSE]) ==
        0)))) {
        j[which.max(colSums(CM[i, , drop = FALSE]))] <- TRUE
    }
    p <- colSums(CM) * j
}
else {
    if (length(p <- pguess) != m)
        stop("Argument 'pguess' is the wrong length.")
}
p <- p/sum(p)
P <- drop(CM %*% p)
ll <- sum(log(P))
evenstep <- FALSE
if (timing)
    times <- update(times, "setup")
for (iter in 1:maxiter) {
    p.old <- p
    ll.old <- ll
    S <- CM/P
    g <- colSums(S)
    dmax <- max(g) - n
    if (iter > 2)
        if (dmax/abs(ll) <= dtol) {
            converge <- TRUE
            break
        }
    j <- p > 0
    if (depth == 1) {
        s <- m1[j]
        if (length(s) > 1)
            for (l in 2:length(s)) {
              j[s[l - 1] + which.max(g[s[l - 1]:s[l]]) -
                1] <- TRUE
            }
    }
    sj <- sum(j)
    if (timing)
        times <- update(times, "misc")
    if (is.null(blockpar) || is.na(blockpar))
        iter.blockpar <- ifelse(sj < 30 | (depth == 1 & n *
            r <= 150), 0, 1 - log(max(20, 15 * log(sj) -
            70))/log(sj))
    else iter.blockpar <- blockpar
    if (iter.blockpar == 0 | sj < 30) {
        nblocks <- 1
        BW <- matrix(1, nr = sj, nc = 1)
    }
    else {
        nblocks <- if (iter.blockpar > 1)
            max(1, round(sj/iter.blockpar))
        else max(1, floor(min(sj/2, sj^iter.blockpar)))
        if (overlap >= 4) {
            if (overlap >= 6)
              nblocks <- min(nblocks * 2, floor(sj/2))
            BW <- outer(seq(1, nblocks, length = sj), 1:nblocks,
              function(u, v) {
                return(pmax(0, 1 - abs(u - v)))
              })
            if (overlap == 5 | overlap == 7) {
              BW[BW > 0 & BW < 1] <- 0.5
            }
        }
        else {
            if (overlap == 3) {
              i <- cumsum(colSums(S[, j] * S[, j]))
              i <- nblocks * (i/max(i))
            }
            else {
              i <- seq(0, nblocks, length = sj + 1)[-1]
            }
            if (evenstep & overlap != 2) {
              nblocks <- nblocks + 1
              BW <- outer(round(i) + 1, 1:nblocks, "==")
            }
            else BW <- outer(ceiling(i), 1:nblocks, "==")
        }
        storage.mode(BW) <- "numeric"
    }
    if (timing)
        times <- update(times, "blocks")
    for (block in 1:nblocks) {
        jj <- logical(m)
        jj[j] <- BW[, block] > 0
        sjj <- sum(jj)
        if (sjj > 1 && (delta <- sum(p.old[jj])) > 0) {
            Sj <- S[, jj]
            if (timing)
              times <- update(times, "blocks")
            a <- rbind(rep(1, sjj), (Sj - drop(Sj %*% p.old[jj] +
              1)/delta))
            b <- c(delta, rep(0, n))
            res <- nnls(a, b)
            if (res$mode > 1)
              warning("Problem in nnls(a,b)")
            if (timing)
              times <- update(times, "nnls")
            p[jj] <- p[jj] + BW[jj[j], block] * (res$x *
              (delta/sum(res$x)) - p.old[jj])
        }
    }
    if (overlap >= 4) {
        p[p <= .Machine$double.eps] <- 0
```

```
            p <- p/sum(p)
        }
        if (timing)
            times <- update(times, "blocks")
        p.gap <- p - p.old
        ll.rise.gap <- sum(g * p.gap)
        alpha <- 1
        p.alpha <- p
        ll.rise.alpha <- ll.rise.gap
        repeat {
            P <- drop(CM %*% p.alpha)
            ll <- sum(log(P))
            if (ll >= ll.old && ll + ll.rise.alpha <= ll.old) {
                p <- p.alpha
                converge <- TRUE
                break
            }
            if (ll > ll.old && ll >= (ll.old + ll.rise.alpha/3)) {
                p <- p.alpha
                break
            }
            if ((alpha <- alpha/2) < 1e-10) {
                p <- p.old
                P <- drop(CM %*% p)
                ll <- ll.old
                converge <- TRUE
                break
            }
            p.alpha <- p.old + alpha * p.gap
            ll.rise.alpha <- alpha * ll.rise.gap
        }
        if (timing)
            times <- update(times, "linesearch")
        if (converge)
            break
        if (nblocks > 1) {
            Q <- sweep(BW, 1, p[j], "*")
            q <- colSums(Q)
            Q <- sweep(CM[, j] %*% Q, 2, q, "/")
            if (timing)
                times <- update(times, "blocks")
            if (any(q == 0)) {
                warning("A block has zero probability!")
            }
            else {
                res <- HCNM(CM = Q, pguess = q, blockpar = iter.blockpar,
                  maxiter = recurs.maxiter, recurs.maxiter = recurs.maxiter,
                  depth = depth + 1)
                maxdepth <- max(maxdepth, res$maxdepth)
                if (timing)
                  times <- update(times, "alloc")
                if (res$lval > ll) {
                  p[j] <- p[j] * (BW %*% (res$pf/q))
                  P <- drop(CM %*% p)
                  ll <- sum(log(P))
                  if (timing)
                    times <- update(times, "blocks")
                }
            }
        }
        evenstep <- !evenstep
    }
    res <- list(pf = p, intmap = intmap, lval = ll, converge = converge,
        numiter = iter, dtol = signif(dmax/abs(ll), 3), nblocks = nblocks,
        maxdepth = maxdepth)
    if (timing)
        res$timing <- update(times, "misc")
    structure(res, class = "icsurv")
}
```

Blockpar simulations, `sim.blockpar.ricens`

Description

This function tests the HCNM algorithm in various ways. The first argument `bp` allows different block sizes and block parameters to be tried. The simulations involve randomly generating a number of simulated data sets, using the method from Wang (2008). The parameters for the random data sets are in vectors `n`, `k` and `r`. Every combination of these gets tried, with `reps` replications. The argument maxiter can be used to weed out badly performing cases early.

A similar version of this function was used to perform the block structure simulations, but it is not included. The only difference was that it varied the `overlap` parameter for the HCNM function above.

Arguments

`bp`	Values of HCNM's parameter `blockpar` to try.
`n`	Number of observations in the data set.
`k`	Case-k interval censoring (number of inspections).
`r`	Proportion of exact observations.
`reps`	Number of replications required.
`maxiter`	Maximum number of iterations allowed in HCNM.
`outfile`	Name of output file, to write results to.
`seed`	Starting random seed, if desired for reproducibility.

Code

```
R> sim.blockpar.ricens

function (bp, n, k, r, reps = 1, maxiter = 30, CNM.cutoff = 400,
    timing = FALSE, outfile = NULL, seed = NULL)
{
    require(lsei)
    bp <- sort(unique(bp), na.last = TRUE)
    nbp <- length(bp)
    R <- expand.grid(bp = bp, n = n, k = k, r = r, rep = 1:reps)
    R$seed <- NA
    R$nRCens <- NA
    R$nExact <- NA
    R$m0 <- NA
    R$nblock <- NA
    R$bsize <- NA
    R$mhat <- NA
    R$dtol <- NA
    R$llhood <- NA
    R$iter <- NA
    R$deep <- NA
    if (timing) {
        R$setup <- NA
        R$misc <- NA
        R$nnls <- NA
        R$linesearch <- NA
        R$blocks <- NA
        R$alloc <- NA
        timing.cols <- seq(which(names(R) == "setup"), length = 6)
    }
    R$time <- NA
    if (is.null(seed))
        seed <- round(runif(1, 0, .Machine$integer.max - nrow(R)))
    cat("Number of simulations: ", nrow(R), ".  Starting seed: ",
        seed, "\n", sep = "")
    seed <- seed - 1
    for (i in 1:nrow(R)) {
        if (nbp == 1 || (!is.na(R$bp[i]) & R$bp[i] == bp[1])) {
            seed <- seed + 1
            a <- ricens(n = R$n[i], k = R$k[i], r = R$r[i], seed = seed,
                sort = FALSE)
            nExact <- sum(a[, 1] == a[, 2])
            nRCens <- sum(a[, 2] == Inf)
            m0 <- ncol(clique.matrix(a))
        }
        R$seed[i] <- seed
        R$nExact[i] <- nExact
        R$nRCens[i] <- nRCens
        R$m0[i] <- m0
        if (!is.na(R$bp[i]) & R$bp[i] == 0 & nExact > CNM.cutoff) {
            calctime <- Inf
            res <- NULL
        }
        else {
            gc()
            calctime <- system.time(res <- try(HCNM(a, blockpar = R$bp[i],
                maxiter = maxiter, timing = timing)))[1]
        }
        if (is.list(res)) {
            R$time[i] <- round(calctime, 4)
            R$iter[i] <- res$numiter
            R$deep[i] <- res$maxdepth
            R$mhat[i] <- sum(res$pf > 0)
            R$nblock[i] <- res$nblock
            R$bsize[i] <- round(R$mhat[i]/R$nblock[i])
            R$dtol[i] <- res$dtol
            R$llhood[i] <- res$lval
            if (timing) {
                R[i, timing.cols] <- round(res$timing$times,
                  4)
            }
        }
        if (!is.null(outfile)) {
            if (i == 1)
                write.table(R[1, ], outfile)
            else write.table(R[i, ], outfile, append = TRUE,
                col = FALSE)
        }
    }
    if (is.null(outfile))
        return(R)
    else return(invisible(R))
}
```

Random interval-censored survival data, `ricsa`

Description

This algorithm randomly generates a realistic set of interval-censored survival data, as seen in the study by Kumwenda et al. (2008). It does that by generating random event times for two types of events, for each subject. Both types of event are assumed to have a Weibull distribution, with parameters supplied by the user. One type of event is recorded exactly (up to a level of precision defined by rounding). The other event is censored by a series of inspection times. These inspection times are jiggled slightly per subject by adding a random delay to each inspection visit.

The return value is an $n \times 2$ matrix. For the i^{th} row, the first column gives L_i and second column gives R_i.

Arguments

`n`	Number of observations in the data set.
`Epar`	Weibull parameters (shape and scale) for exact observation times.
`Cpar`	Weibull parameters (shape and scale) for censored observation times.
`fup`	Scheduled follow-up (inspection) times. The defaults are based on the study by Kumwenda et al. (2008).

`fupdelay`	Mean delay at each follow-up. The default values are 5% of the gaps between inspections.
`round.dp`	Number of decimal places, to which event times are rounded.
`seed`	Random seed, if desired, for reproducibility.
`sort`	Should the result be sorted? TRUE or FALSE.

Code

```
R> ricsa

function (n, Epar = c(shape = 1, scale = 1500), Cpar = c(shape = 1,
    scale = 1500), fup = c(7, 21, 42, 63, 98, 183, 274, 365,
    456, 548, 730), fupdelay = diff(c(0, fup))/20, round.dp = 0,
    seed = NULL, sort = TRUE)
{
    if (is.null(round.dp))
        round.dp <- NA
    if (!is.null(seed))
        set.seed(seed)
    nfup <- length(fup)
    fupdelay <- rep(fupdelay, length = nfup)
    fups <- outer(rep(0, n), fup, "+")
    if (any(fupdelay > 0))
        fups <- fups + rexp(n * nfup, 1/rep(fupdelay, each = n))
    if (!is.na(round.dp))
        fups <- round(fups, round.dp)
    if (all(is.finite(Epar))) {
        death <- rweibull(n, Epar[1], Epar[2])
        death[death > apply(fups, 1, max)] <- Inf
        if (!is.na(round.dp))
            death <- round(death, round.dp)
    }
    else {
        death <- rep(Inf, n)
    }
    if (all(is.finite(Cpar))) {
        hiv <- rweibull(n, Cpar[1], Cpar[2])
        if (!is.na(round.dp))
            hiv <- round(hiv, round.dp)
        hivL <- apply(ifelse(fups < hiv, fups, 0), 1, max)
        hivR <- apply(ifelse(fups >= hiv, fups, Inf), 1, min)
        c1 <- pmin(death, hivL)
        c2 <- pmin(death, hivR)
    }
    else {
        c1 <- c2 <- death
    }
    if (sort) {
        o <- order(c1, c2)
        cbind(L = c1[o], R = c2[o])
    }
    else cbind(L = c1, R = c2)
}
```

Functions for the `timing` class

Description

This section describes methods for creating, updating and printing objects of a new class `timing`. Objects of this class were used to record the computation time taken by specific parts within the HCNM algorithm.

Arguments

`tnames`	Names of timing components to keep track of.
`timing`	An object of class `timing` as created by `new.timing`.
`process`	The index or name of one of the named elements, the one to update.

Code

The code excerpt below provides three functions for using objects of the `timing` class. An example of how they work is also provided.

```
R> new.timing

function (tnames = "usertime")
{
    structure(list(times = structure(numeric(length(tnames)),
        names = tnames), calls = 0, now = proc.time()[1]), class = "timing")
}

R> update.timing

function (timing, process = 1)
{
    now <- proc.time()[1]
    timing$times[process] <- timing$times[process] + (now - timing$now)
    timing$now <- now
    timing$calls <- timing$calls + 1
    timing
}

R> print.timing

function (timing)
{
    cat("A timing object")
    if (timing$calls > 0)
        cat(" with calls=", timing$calls, sep = "")
    cat(".  Total=", sum(timing$times), " seconds.\n", sep = "")
    print(timing$times)
}

R> (example <- new.timing(c("create", "invert")))
```

```
A timing object.  Total=0 seconds.
create invert
     0      0

R> x <- matrix(rnorm(250000), 500)
R> (example <- update(example, "create"))

A timing object with calls=1.  Total=0.199 seconds.
create invert
 0.199  0.000

R> y <- solve(x)
R> (example <- update(example, "invert"))

A timing object with calls=2.  Total=1.186 seconds.
create invert
 0.199  0.987
```

Non-negative least squares

Description

This section presents the NNLS algorithm, provided by Lawson and Hanson (1974, page 161), in pseudo-code. Algorithm 2 solves the minimum least squares problem with non-negativity constraints.

Pseudo-code

Algorithm 2 Non-negative least squares (NNLS)

```
Require: an $n \times m$ matrix $\mathbf{A}$ and vector $\mathbf{b}$ of length $n$
Ensure: the vector $\boldsymbol{\pi}$ minimises the sum of squares $\|\mathbf{A}\boldsymbol{\pi} - \mathbf{b}\|^2$ subject to $\boldsymbol{\pi} \geqslant \mathbf{0}$
 1: $\boldsymbol{\pi} \leftarrow \mathbf{0}$ {Vector of $m$ values $(\pi_1, \ldots, \pi_m)^T$}
 2: $\mathbf{P} \leftarrow \emptyset$ {Set of indices of positive values in $\boldsymbol{\pi}$}
 3: $\mathbf{Z} \leftarrow \{1, \ldots, m\}$ {Set of indices of zero values in $\boldsymbol{\pi}$}
 4: loop {The outer loop}
 5:   $\mathbf{w} \leftarrow \mathbf{A}^T(\mathbf{b} - \mathbf{A}\boldsymbol{\pi})$ {Vector of $m$ values $(w_1, \ldots, w_m)^T$}
 6:   if ($\mathbf{Z} = \emptyset$ or $\forall j \in \mathbf{Z} : w_j \leqslant 0$) then
 7:     return $\boldsymbol{\pi}$ {Exit the algorithm}
 8:   end if
 9:   Find index $t \in \mathbf{Z}$ where $w_t = \max\{w_j : j \in \mathbf{Z}\}$
10:   Move the index $t$ from $\mathbf{Z}$ to $\mathbf{P}$ {Postulate that $\pi_t$ should be positive}
11:   loop {The inner loop}
12:     Let $\mathbf{A}'$ be a copy of $\mathbf{A}$ with columns indexed by $\mathbf{Z}$ set to zero.
13:     Compute $\mathbf{z}$ to minimise $\|\mathbf{A}'\mathbf{z} - \mathbf{b}\|^2$ {Using ordinary least squares}
14:       {Only $z_j : j \in \mathbf{P}$ are determined in this step, others are set to zero}
15:     if ($\forall j \in \mathbf{P} : z_j > 0$) then
16:       $\boldsymbol{\pi} \leftarrow \mathbf{z}$
17:       break from Loop B
18:     end if
19:     Find index $q \in \mathbf{P} : \pi_q/(\pi_q - z_q) = \min\{\pi_j/(\pi_j - z_j) : z_j \leqslant 0, j \in \mathbf{P}\}$
20:     $\alpha \leftarrow \pi_q/(\pi_q - z_q)$
21:     $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} + \alpha(\mathbf{z} - \boldsymbol{\pi})$
22:     Move all indices $\{j \in \mathbf{P} : \pi_j = 0\}$ from $\mathbf{P}$ to $\mathbf{Z}$
23:   end loop {Inner loop}
24: end loop {Outer loop}
```

Bibliography

Ayer, M., Brunk, H., Ewing, G., Reid, W., and Silverman, E. (1955). An Empirical Distribution Function for Sampling with Incomplete Information. *ANNALS OF MATHEMATICAL STATISTICS*, 26(4):641–647.

Betensky, R. (2000). On nonidentifiability and noninformative censoring for current status data. *BIOMETRIKA*, 87(1):218–221.

Böhning, D., Schlattmann, P., and Dietz, E. (1996). Interval censored data: A note on the nonparametric maximum likelihood estimator of the distribution function. *BIOMETRIKA*, 83(2):462–466.

Dax, A. (1990). The Smallest Point of a Polytope. *JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS*, 64(2):429–432.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via EM algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY SERIES B-METHODOLOGICAL*, 39(1):1–38.

Dümbgen, L., Freitag-Wolf, S., and Jongbloed, G. (2006). Estimating a unimodal distribution from interval-censored data. *JOURNAL OF THE AMERICAN STATISTICAL ASSOCIATION*, 101(475):1094–1106.

Gentleman, R. and Geyer, C. (1994). Maximum likelihood for interval censored data: Consistency and computation. *BIOMETRIKA*, 81(3):618–623.

Gentleman, R. and Vandal, A. (2001). Computational algorithms for censored-data problems using intersection graphs. *JOURNAL OF COMPUTATIONAL AND GRAPHICAL STATISTICS*, 10(3):403–421.

Gentleman, R. and Vandal, A. (2008). *Icens: NPMLE for Censored and Truncated Data.* R package version 1.2.0.

Groeneboom, P. (1991). Nonparametric maximum likelihood estimators for interval censoring and deconvolution. Technical Report 378, Department of Statistics, Stanford University.

Groeneboom, P. and Wellner, J. A. (1992). *Information Bounds and Nonparametric Maximum Likelihood Estimation.* Birkhäuser, New York.

Grotzinger, S. and Witzgall, C. (1984). Projections onto order simplexes. *APPLIED MATHEMATICS AND OPTIMIZATION*, 12(3):247–270.

Haskell, K. H. and Hanson, R. J. (1981). An algorithm for linear least squares problems with equality and nonnegativity constraints. *MATHEMATICAL PROGRAMMING*, 21:98–118.

Jongbloed, G. (1998). The iterative convex minorant algorithm for nonparametric estimation. *JOURNAL OF COMPUTATIONAL AND GRAPHICAL STATISTICS*, 7(3):310–321.

Jull, A., Walker, N., Parag, V., Molan, P., Rodgers, A., and on behalf of the Honey as Adjuvant Leg Ulcer Therapy trial collaborators. (2008). Randomized clinical trial of honey-impregnated dressings for venous leg ulcers. *BRITISH JOURNAL OF SURGERY*, 95(2):175–182.

Kalbfleisch, J. D. and Prentice, R. L. (2002). *The Statistical Analysis of Failure Time Data.* John Wiley New York, 2nd edition.

Kaplan, E. and Meier, P. (1958). Nonparametric-Estimation from Incomplete Observations. *JOURNAL OF THE AMERICAN STATISTICAL ASSOCIATION*, 53(282):457–481.

Kumwenda, N. I., Hoover, D. R., Mofenson, L. M., Thigpen, M. C., Kafulafula, G., Li, Q., Mipando, L., Nkanaunena, K., Mebrahtu, T., Bulterys, M., Fowler, M. G., and Taha, T. E. (2008). Extended antiretroviral prophylaxis to reduce breast-milk HIV-1 transmission. *NEW ENGLAND JOURNAL OF MEDICINE*, 359(2):119–129.

Lawson, C. L. and Hanson, R. J. (1974). *Solving Least Squares Problems.* Prentice-Hall, Englewood Cliffs, N.J.

Lindsay, B. G. (1995). *Mixture Models: Theory, Geometry, and Applications.* Hayward: Institute of Mathematical Statistics.

Maathuis, M. H. (2005). Reduction algorithm for the NPMLE for the distribution function of bivariate interval-censored data. *JOURNAL OF COMPUTATIONAL AND GRAPHICAL STATISTICS*, 14(2):352–362.

Mullen, K. M. and van Stokkum, I. H. M. (2007). *nnls: The Lawson-Hanson algorithm for non-negative least squares (NNLS).* R package version 1.1.

National Cancer Institute (2009). *Definition of Disease-free Survival.* `http://www.cancer.gov/templates/db_alpha.aspx?CdrID=44023` [Accessed on 12 January 2009].

Oller Piqué, R. (2006). *Survival analysis issues with interval-censored data.* PhD thesis, Universitat Politècnica de Catalunya.

Peto, R. (1973). Experimental Survival Curves for Interval-censored Data. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY SERIES C-APPLIED STATISTICS*, 22(1):86–91.

Pilla, R. S. and Lindsay, B. G. (2001). Alternative EM methods for nonparametric finite mixture models. *BIOMETRIKA*, 88(2):535–550.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Schick, A. and Yu, Q. (2000). Consistency of the GMLE with mixed case interval-censored data. *SCANDINAVIAN JOURNAL OF STATISTICS*, 27(1):45–55.

Sun, J. (2006). *The Statistical Analysis of Interval-censored Failure Time Data.* Statistics for Biology and Health. Springer-Verlag New York, LLC.

Turnbull, B. W. (1976). The empirical distribution with arbitrarily grouped censored and truncated data. *Journal of the Royal Statistical Society, Series B*, 38:290–295.

Wang, Y. (2007a). *lsei: Least Squares Linear Regression under Equality/Inequality Constraints.* R package version 1.0-2. `http://www.stat.auckland.ac.nz/~yongwang`.

Wang, Y. (2007b). Maximum likelihood computation based on the Fisher scoring and Gauss-Newton quadratic approximations. *COMPUTATIONAL STATISTICS & DATA ANALYSIS*, 51(8):3776–3787.

Wang, Y. (2007c). On fast computation of the non-parametric maximum likelihood estimate of a mixing distribution. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY SERIES B-STATISTICAL METHODOLOGY*, 69(Part 2):185–198.

Wang, Y. (2008). Dimension-reduced nonparametric maximum likelihood computation for interval-censored data. *COMPUTATIONAL STATISTICS & DATA ANALYSIS*, 52(5):2388–2402.

Wellner, J. A. (1995). Interval censoring, case 2: Alternative hypotheses. In Koul, HL and Deshpande, JV, editor, *ANALYSIS OF CENSORED DATA*, volume 27 of *INSTITUTE OF MATHEMATICAL STATISTICS LECTURE NOTES - MONOGRAPH SERIES*, pages 271–291. Workshop on Analysis of Censored Data, PUNE, INDIA, DEC 28, 1994-JAN 01, 1995.

Wellner, J. A. and Zhan, Y. H. (1997). A hybrid algorithm for computation of the nonparametric maximum likelihood estimator from censored data. *JOURNAL OF THE AMERICAN STATISTICAL ASSOCIATION*, 92(439):945–959.

Wong, G. and Yu, Q. (1999). Generalized MLE of a joint distribution function with multivariate interval-censored data. *JOURNAL OF MULTIVARIATE ANALYSIS*, 69(2):155–166.

Glossary

Definitions of key terms

Algorithm A prescribed method for solving a particular problem, composed of action steps, loops, decision branches and stopping criteria. A flowchart on sterroids.

Censoring The hiding of a value, preventing it from being directly measured.

Concave A function $f(\mathbf{x})$ is said to be concave iff for all $\mathbf{z}$ on a line between any two points $\mathbf{x}_1$ and $\mathbf{x}_2$, that is $\mathbf{z} = \mathbf{x}_1 + \alpha(\mathbf{x}_2 - \mathbf{x}_1)$, $\alpha \in [0, 1]$ the condition $f(\mathbf{z}) \geqslant f(\mathbf{x}_1) + \alpha(f(\mathbf{x}_2) - f(\mathbf{x}_1))$ holds. Looked at from below, the function is concave everywhere.

Hazard rate The instantaneous rate at which events occur in a survival model.

Likelihood A measure of how likely a set of observations is, as a means of comparing models. Usually, the model that maximises likelihood is selected. Often for convenience the log is taken, giving log-likelihood.

Likelihood Function The likelihood of a set of observations, which is expressed as a function of a statistical model, enabling different models to be compared on their likelihood.

Nonparametric Describes a flexible type of statistical model that does not make assumptions about the form of the distribution and does not contain parameters that need to be estimated.

Order of convergence Consider the absolute estimate errors for a converging algorithm. If the error $|\epsilon_i|$ is proportional to the previous error to the p^{th} power, that is $|\epsilon_i| \propto |\epsilon_{i-1}|^p$ in the limit, then the order of the algorithm is said to be p. For example, a quadratic order of convergence means $p = 2$.

Rate of convergence For an algorithm with a linear order of convergence, the rate of convergence is the limit of the ratio of the error of each successive pair of estimates.

Survival Analysis The study of time-to-event data and hence the progression of survival probabilities through time. It also looks at comparisons of these for different groups of subjects and based on other covariates (e.g. age).

Survival Function A description, such as a plot or a parametric function, of the progression of the probability of survival versus time. It usually starts at a value of one (at $t = 0$) and then continues in a non-increasing manner downwards, perhaps with discontinuous drops, towards 0 (at $t = \infty$).

Time Complexity The time taken to complete an algorithm, expressed as a function of the size of the inputs given to it (in the limit). The notation $O(n^p)$ means that as n increases, the computational time taken by the algorithm increases like n^p.

Unit Simplex An $m - 1$ dimensional space within $\mathbb{R}^m$ with the form of a constrained hyperplane. For a point $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_m)^T$ in the unit simplex, each component π_j is constrained to $\pi_j \in [0, 1]$ and the total of all components adds up to one. Thus, points in this space form vectors of probabilities.

List of Abbreviations

CNM	Constrained Newton Method †
CPU	Central Processing Unit
EM	Expectation Maximisation method †
HCNM	Hierarchical Constrained Newton Method †
ICM	Iterative Convex Minorant method †
ICMDR	Iterative Convex Minorant method, with dimension-reduction †
ICM-EM	A hybrid algorithm, combining ICM and EM †
ICMDR-EM	The ICM-EM algorithm, with dimension-reduction †
KM	Kaplan-Meier product-limit technique
MLE	Maximum Likelihood Estimate/Estimator
NNLS	Non-negative Least Squares
NPMLE	Nonparametric Maximum Likelihood Estimate/Estimator
R	R (the statistics software program)
SBN	Subspace-based Newton method †
SBNDR	Subspace-based Newton method, with dimension-reduction †

Those marked † are methods for solving the NPMLE problem, i.e. finding the NMPLE of the survival function for data containing interval-censored observations.

List of key mathematical symbols

T	Random variable of event times with distribution $F(t)$.
$S(t)$	Survival function for times $t \geqslant 0$. Starts at $S(0) = 1$, decreases monotonically and ends at $S(\infty) = 0$, since $S(t) = 1 - F(t)$.
n	Number of observations in a survival data set.
t_i	Event time of the i^{th} individual, for $i \in \{1, \ldots, n\}$ (not necessarily observed).
L_i and R_i	Left and right end points of a censoring interval around t_i.
O_i	Observed event time interval: either $\{R_i\}$ (exact) or $(L_i, R_i]$ (interval-censored). In each case, $t_i \in O_i$ is all that is known about t_i.
m	Number of candidate support intervals.
I_j	Candidate support interval, for $j \in \{1, \ldots, m\}$. Each is a maximal intersection derived from combinations of $\{O_i\}$. All are mutually disjoint.
δ_{ij}	Indicator that $I_j \subseteq O_i$.
$\mathbf{A}$	Clique matrix, $n \times m$ with values δ_{ij}.
$\mathbb{R}^m$	Real space, m-dimensional.
$\boldsymbol{\pi}$	Probability vector in $\mathbb{R}^m$ (more specifically, in the unit simplex) that allocates probability masses $(\pi_1, \ldots, \pi_m)$ to each of the candidate support intervals.
$\mathbf{p}$	Probabilities $(p_1, \ldots, p_n)$ of the observations O_i, with $p_i = \sum_j \delta_{ij}\pi_j$.
$\ell(\boldsymbol{\pi})$	The log-likelihood function, the maximum of which we seek.
$\hat{\boldsymbol{\pi}}$	The value of $\boldsymbol{\pi}$ that maximises $\ell(\boldsymbol{\pi})$ within the unit simplex.
κ	Number of blocks that partition $\{I_j\}$.
$\mathbf{B}$	The $m \times \kappa$ matrix of block memberships.
U_b	The b^{th} block, the union of the support intervals in that block.