Efficient Numerical Methods for Adaptive Quantile Regression

Christian Viller Hansen

Kongens Lyngby 2007
IMM-M.Sc-2007-62

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc: ISSN 1601-233x

Abstract

The efficiency and profitability of wind power relies heavily on having precise productivity forecasts in order to predict the need for other power sources.

A significant addition to power production forecasts are uncertainty predictions, which is the focus of this Master thesis. A Matlab implementation of an adaptive quantile regression program has been converted to C for increased computational performance efficiency, and to break the grounds needed for it to become an integrated part of a commercial power prediction tool (WPPT).

The adaptive algorithm has been improved significantly by the introduction of a penalty based selection and apart from the simplex implementation for quantile regression, the program has been extended with the interior point method and with a flexible framework for additional algorithms. With these additions to what the program is capable of, the computational performance has still improved significantly.

Particular focus has been placed on qualitatively validating the calculated quantiles in terms of reliability. With the aid of both reliability and skill score tests it has been shown that the quality of quantiles predicted benefits from frequent updates, but this can be only weekly, and not hourly, as previously expected.

Resumé

For at kunne udnytte vindkraft til fulde og gøre det rentabelt, er det essentielt med pålidelige strømproduktions forudsigelser, da disse gør det muligt at forudse behovet for andre energikilder.

En betydningsfuld tilføjelse til disse forudsigelser er estimering af usikkerhedsparametre, hvilket er fokus for dette speciale. En Matlab version af en adaptiv fraktilregressions algoritme er blevet konverteret til C for hurtigere afvikling, og for at gøre en implementering i et kommercielt effekt forudsigelsesprogram (WPPT) mulig.

Den adaptive algoritme er blevet væsenligt forbedret gennem introduktionen af en strafbaseret udvælgelsesalgoritme. Udover simplex metoden er interior point metoden også tilføjet som udskiftelige moduler, med mulighed for lette udvidelser med andre algoritmer. Afviklingstiden er blevet væsentligt reduceret i forhold til Matlab versionen, der endda ikke har samme omfangsrige funktionalitet.

Stor vægt er lagt på kvalitativt og funktionelt at validere de beregnede fraktilers pålidelighed. Ved hjælp af både pålidelighedsmål og skill score test er det blevet vist, at kvaliteten af fraktilforudsigelserne forbedres ved jævnlig opdatering af estimatoren, men ugentlige opdateringer er fuldt ud tilstrækkeligt for at opnå gode resultater.

Preface

This Master thesis was carried out at Informatics and Mathematical Modelling, the Technical University of Denmark, in the period from September 1st 2006 to July 1st 2007.

I wish to thank the following people for their help throughout my thesis: Henrik Madsen, Bernd Dammann, Jan Kloppenborg Møller and Pierre Pinson.

Kgs. Lyngby, July 2007

Christian Viller Hansen

Contents

Chapter 1

Introduction

1.1 Power Prediction

Wind power is a very popular energy source, and with the increasing oil prices, the importance of wind energy only becomes greater. One problem with wind power, however, is the uncertainty in availability. For a stable power supply, the production much match the demand very closely, otherwise the constant frequency cannot be retained.

The system is stabilized by buying and selling power, but since big power plants cannot simply turn their production up and down instantaneously, this power needs to be traded on beforehand, which requires both demand and prediction forecasts. Energy derived from natural sources such as wind, water and sun power are not as simple to forecast as energy from power plants, as the power production from these relies on the ever changing weather.

To stabilize the power grid with windmills increasing in both number and power, very reliable power predictions are needed, alternatively, the energy produced from the windmills will just be wasted. In order to predict wind power, good weather predictions are required. Weather predictions are out of the scope of this thesis, so it will simply be assumed that predictions are available[1].

[1] In reality these weather forecasts are bought from companies and institutions that spe-

1.1.1 Power Curve

In terms of predicting wind power, wind speed is the most important factor to consider. As most things in nature, the power production is not a perfect linear function of the wind, but instead the relation can be described by the *power curve*.

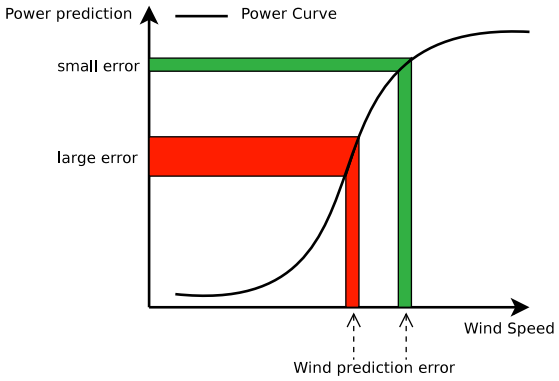

Figure 1.1: The relation between wind speed and produced power is what will be referred to when speaking about the *power curve*. This is simply a rough sketch of how an actual power curve might look like. The red and green area show how errors in wind speed predictions affect the actual produced power or prediction error in the case of power prediction.

In Figure 1.1 is a rough sketch of how a traditional power curve looks. It makes very good sense that the wind needs to be at a certain level before any power is produced, and that the windmill has a certain maximum power producing capacity.

The figure also indicates that the prediction uncertainty is a function of wind speed or predicted power since small errors in the wind speed predictions will cause larger errors in the steep middle part of the curve than at the extremes. A power prediction based on the power curve will predict wind power with larger uncertainties for medium capacity than for zero and max power predictions.

cialize in meteorological forecasts. In Denmark, such companies might be the Danish Meteorological Institute (DMI) or Deutscher Wetterdienst (DWD). Very good local forecasts are required, so the choice of weather forecast provider is based on the location of the windmills.

1.2 WPPT

Wind Power Prediction Tool (WPPT) is a program developed at DTU for predicting wind power based on weather forecasts. On the basis of WPPT and a few other tools, the company ENFOR A/S was established in 2006 to sell and service these tools.

The program is very successful and has been employed in many of the major Danish power production and power trade facilities.

What WPPT does is that it gives the people trading power a tool for predicting how much power can be expected to arrive from windmills. This information is of great importance, because shortage of power is fatal and would lead to power outage. To prevent this from happening, enough power must be produced at all times. Furthermore, with the large start up times of power plants, unpredictable power from windmills is close to useless.

The power predictions in WPPT are naturally carried out by using the previously mentioned power curve. The actual power production is not only dependent on wind speed, but also on other factors such as wind direction, season, temperature, landscape, vegetation, maintenance and more. Some of these factors are incorporated in the model of WPPT, whilst others, such as maintenance, will contribute to prediction error.

In order to adjust the model and power curve in particular, the program needs training in form of measured power versus the factors on which the prediction should depend. These factors are known as the explanatory variables.

The company ENFOR A/S have been kind enough to provide different data sets from one of its customers, a Danish windmill park called Klim. These data have been used throughout the thesis for tests and validation.

1.3 Prediction Errors and Quantiles

Predictions will inevitably include some kind of error margin. Windmill power predictions, which rely on weather forecasts, will in particular be subject to accumulated error. The power curve might not be completely accurate, or it might depend on factors not included in the model. But even if this relation between weather and windmill power was completely characterized, the uncertainty in the weather forecasts would still result in power prediction errors. The mod-

els are, however, undergoing constant improvement through research and more powerful computers, which can handle more complex models. By all means, the prediction error will get smaller and smaller, but by merely looking at weather forecasts in the media, you will know that there is a long way to go.

Relying on weather forecasts to plan a barbecue party or a trip to the beach is common to most people, but in the event of rain, the consequences are usually not worse than a mild annoyance to those involved. Using weather and ultimately wind power predictions to buy and sell energy is a completely different story. If some major coal power plants are shut down in the expectation of a windy few days, the lack of wind could potentially shut down the whole country. Once they are shut down, they cannot simply be turned on. Fortunately, power can be bought from other countries, but the price may be very high. The best thing would be to have perfectly reliable power forecasts, but since *perfect* is hard to achieve, the next best thing would be to know *how certain the prediction is.*

This thesis aims at providing a tool for *adaptively estimating the prediction uncertainty* based on previous data and relaying these estimated prediction uncertainties in terms of *quantiles*. As was seen in Figure 1.1, a small prediction error in wind speed results in a larger power prediction error, if the prediction is in the steeper part of the power curve as opposed to the flatter parts of the curve.

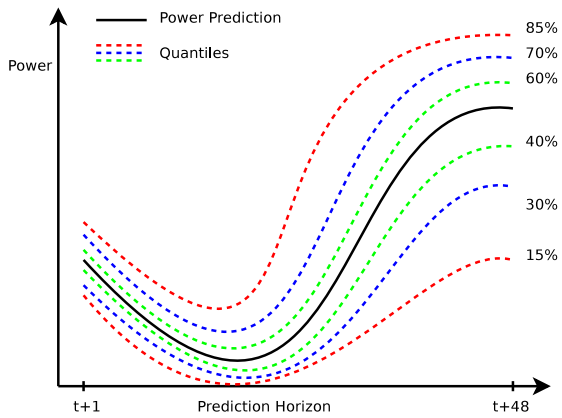

Figure 1.2: A simple sketch of how quantiles can support the power predictions with probabilistic information. A narrow span of the quantiles means a great confidence in the predicted power, whilst a wide span means the prediction is uncertain. Such information is very valuable when doing decisions based on the predictions.

The sketch in Figure 1.2 show how the quantiles might graphically be represented to the operator which needs to base his decision based on the prediction. At the time $t = 0$ the prediction program (WPPT) predicts the power for a given horizon, 48 hours in the sketch, and the quantile curves describing the estimated uncertainty are superimposed on the power predictions. With a knowledge of how predictable the prediction probably is, the operator will have more knowledge and will be able to make bolder decisions and thus utilize the windmills more effectively. Using the windmills more effectively is favorable for the environment, and money can be saved by making the windmills more profitable.

1.4 Computational Performance

Even with very fast computers, the importance of making programs run fast cannot be overlooked. In an online situation, where decisions must be made fast, it is of very little interest to have a program providing perfect predictions a couple of hours late.

A good way of looking at this is to assume the hardware computer power is constant, but the better the program performs, the higher the number of factors that can be taken into consideration, and at a higher resolution.

It has been a desire from the beginning of this thesis that the program developed should be able to run as efficiently as possible, so a large section of this report is devoted to explaining the theory and techniques involved in achieving a high level of computational performance.

1.5 Focus of Thesis

This thesis is very inter-disciplinary in the sense that it involves math, computer science, statistics and a focus on developing something useful for the industry. A very large part of the work involved with the thesis has been writing the software capable of predicting the quantiles. Although a working version was available through a previous master thesis by Møller [9], the translation from Matlab to C is so complex that it requires exact knowledge of what is going on in the algorithm. This meant that much of the time was spent studying underlying mathematical operations, before a translation could be successful. Although much time was spent getting deep into the the math behind methods for quantile regression, this report will only contain a brief introduction to the mathematical concepts and how they work - seen from above.

In the planning phases of this master thesis, it was decided that the main focus should be on the high performance aspects of the program. This has not changed, but the actual work of translating and improving the core program to interoperate nicely with WPPT has taken a large portion of the time. Performance considerations and tuning have not been neglected, but have become integral parts of the design and redesign of the implementation. The fairly large chapter on theory behind computational performance might seem isolated in terms of what is interesting to write about in the report, but this does not reflect the balance of the work done prior to this report.

A general knowledge of programming is essential to understanding the sections involving computational performance and implementation. The programming language C will used for examples so in order to understand the details of these examples, knowledge of C is a prerequisite.

Chapter 2

Quantile Regression

2.1 Introduction

The purpose of this chapter is to present the ideas behind quantiles and quantile regression as well as to describe some of the fundamental mathematical theory which makes quantile regression possible. For further information on the details and mathematical formulas involved, the book "Quantile Regression" by Koenker [6] is an excellent reference.

The short explanation of quantile regression is that everything is about optimization - the program in this thesis is presented with a number of points and the purpose is to find optimal uncertainly bands by minimizing the product of residuals and an asymmetric penalty or *loss function*. While this sounds very simple, the process of optimization involves a large range of techniques within linear algebra.

With all the different techniques that go into quantile regression, the overview easily gets lost. The ideal goal of this chapter remains focused on providing the reader with the essential overview of quantile regression which is required for understanding the evaluation of the algorithms in the last chapters of this report.

2.2 Quantiles

In statistics, the goal is usually to deduct a meaning from a set of data using a range of mathematical tools. The basic statistical toolbox contains well known tools such as mean, deviation, median and many others. Quantiles, as the name suggests, are a quantification of the data set. The quantile lines separate the data set in such a way that the number of observations below the "line" corresponds to a particular ratio which determines the quantile.'

Quantiles are a general term for this separation of the observation set, but for convenience, special cases have been given other names. *Quartiles* separates the set into four areas containing an equal amount of observations within each band. The lower quarter of the observation is thus called the 1st quartile and so forth. The second quartile is actually the well known median, which should not be too surprising.

Another very useful special case is the *percentiles* and as the name suggests, this simply divides the observation space into 100 quanta with an equal amount of observations within each segment.

The previously shown Figure 1.2 is a simple example of how the quantiles can describe the uncertainty of power predictions. This could be the kind of picture that the user of WPPT would be looking at when deciding how much power to buy or sell.

The program developed in this thesis will use quantiles in its general form and if the frontend application is referring to percentiles or quartiles, it can easily be mapped back to general quantiles $\tau \in [0..1]\mathbb{R}$. τ will be used as the symbol for the ratio corresponding to a particular quantile, and of course this means that eg. the 75th percentile corresponds to $\tau = 0.75$.

2.3 Regression

The process of finding quantiles is called Quantile Regression. It can be carried out in many ways and the basic idea is not much different than linear regression with least squares, but the quantification of the data set requires slightly more complicated methods in order to find the "best lines".

When trying to fit a line between some points, unless the line fits perfectly through all points, there will be difference between each point and the line,

which is will be called the *residual* or error. Least squares techniques minimizes the square of the residuals by finding the line through the points, which gives the least squared error. In quantile regression, the line must be placed so that τN observations are placed below the line and these τN points should be chosen to give the smallest total error.

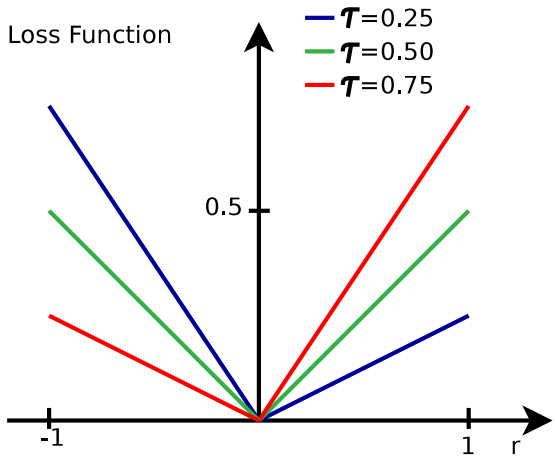

Figure 2.1: A sketch of the loss function for three different values of τ

The principle of the technique is actually quite simple, the idea is to multiply the error vector with an asymmetric loss function (Figure 2.1), based on τ. The sum of the resulting vector is then minimized using standard optimization techniques. The asymmetric penalty or loss function will be called ρ and is defined as

$$\rho(\tau, r) = \begin{cases} \tau r & \text{if } r \geq 0, \\ (\tau - 1)\, r & \text{if } r < 0. \end{cases} \tag{2.1}$$

The error or residual r is calculated simply by

$$r = y - \mathbf{X}\hat{\beta}^{<\tau>}, \tag{2.2}$$

where y is the observation, $\mathbf{X}$ being the explanatory variables and $\hat{\beta}^{<\tau>}$ is the current solution or *quantile predictor*.

The way quantile regression works is by optimizing $\hat{\beta}^{<\tau>}$ so that the sum of the loss function (2.1) on the data set $y, \mathbf{X}$ is minimized. Many different optimization algorithms can do this and the two major techniques interior point method

and simplex have been used in this thesis. The methods and implementations are described in detail by Koenker [6].

2.3.1 Simplex Method

The quantile regression optimization problem is a linear convex problem (Koenker [6], Bruun Nielsen [12]), which means that the path between two points in the feasible region or on the boundary will never go through an infeasible region. This might be hard to visualize, but one might look at the feasible region confined by the leather (boundary) of a football[1]. If the ball is filled with air, the straight line between two feasible points (inside or on the boundary of the ball) would never leave the boundary. Had the ball been punctured it might have a concave dent in it and it would no longer be purely convex.

Quantile regression has little to do with football, but the convex nature of the problem ensures that the optimal solution will be found on the boundary and that there are no local optima with a less optimal *objective*[2] is than the global optimum.

The simplex algorithm is actually very easily described by the football analogy. The points on the surface where the stitches meet is called a *vertex*, which in linear programming refers to a place where a number K of the variables are zero. K defines the order of the problem, or in the context of the implemented program, the number of coefficients used to describe a given quantile curve. These variables which are zero at the vertex are known as the *non-basic* variables, while the remaining variables are known as the *basic* variables.

The optimal solution is found by evaluating the increased objective associated with moving to the adjacent vertices and if the objective is improved, a step will be taken in that direction. Each of these steps is known as a *simplex iteration*. When there is nowhere better to go, the optimum has been found[3].

[1] A conventional European soccer football. An American football is also convex, but it does not describe vertices very well.

[2] The "objective" is a term used in linear programming for the function which is sought optimized.

[3] In some problems it might be necessary to move steps in direction with no gain in order to reach the optimum, but the convex nature of the problem ensures that steps which lowers the objective score are never needed to reach optimum.

2.3.1.1 Vertex Rank

In a simplex implementation the matrix with the K non-basic variables will need to be invertible in order to calculate the derivative of the objective associated with moving to the adjacent vertices.

Using a strong algorithm for evaluating the quality of the vertex matrix is paramount for getting the simplex method to work well on problems outside the world of text book examples.

It was chosen to use singular value decomposition for this. Despite the fact that it is computationally more demanding that other algorithms, it offers the possibility to reliably check not only if a given rank is obtained, but also how close the matrix is to being singular.

2.3.2 Interior Point Method

For large problems without an initial near optimal solution, the interior point method is a very efficient way of optimizing convex problems.

The principle is very different from simplex, because instead of working its way around on the boundary between feasible and infeasible, it moves directly in the interior from the initial starting point to the optimum.

The interior point method is good for large problems because the steps taken can be large in the beginning and then smaller when the boundary and optimum are reached.

One limitation of the interior point method in the case of adaptive quantile regression is that it is generally not very good at using a previous solution and only altering it slightly. The step size for each iteration is controlled by how far the current point is from the boundary, and since the previous solution will be very close to or even on the boundary, the step size will be so small that it cannot adjust towards improved objective.

2.4 Spline Fitting

The methods described for quantile regression are all linear methods, but for the purpose of this thesis, simple linear models are not adequate for describing

the uncertainty in wind power predictions. Piecewise linear would be a simple improvement and could be implemented simply by mapping each explanatory variable to piecewise defined ramp functions. This would increase the number of constraints in the optimization problem, but then it would be possible to make the quantiles better describe the non-linear relation between predicted power and prediction uncertainty.

A better approximation to unknown non-linear functions can be obtained with the use of *splines*. Instead of being piecewise linear functions, splines are piecewise polynomial functions, so they will are better at describing non-linear functions since they can describe curves. The order of polynomial defining the splines can be anything from one (piecewise linear) to an arbitrarily large degree. For most applications an order of three is sufficient, and these are called cubic splines.

The placement of the polynomial segments is controlled by what is known as *knots*. Knots can be thought of as control points for the splines. The number of spline coefficients is directly related to the number of knots.

Several different classes of splines exists and they are suited for different purposes. *Basic* splines are special because their sum is 1 in the range in which they are defined. *Natural* splines must have a constant first derivative outside of the boundary knots. Finally,*Periodic* splines are suitable for describing periodic functions because the splines at the lower and upper boundary can be joined continuously with matching derivatives.

In this thesis, all explanatory variables will be mapped to cubic splines in order to provide quantiles that are as precise as possible. More in depth information can be found in eg. lecture note by Nielsen [11].

Chapter 3

Computational Performance

3.1 Introduction

As Gordon E. Moore predicted in 1965, the number of transistors (and indirectly the performance) in main stream processors has roughly doubled every 18 months, so the question might be why spend time on increasing the performance of code, when a processor twice as fast might come around shortly. One answer lies in the fact that demand for computer power increases at least as fast as the speed of computers. As computers get faster, the demand will increase for both the number and resolution of what is to be computed.

A reason why performance tuning is worth spending time on, as opposed to simply buying new hardware, is illustrated in the sketch in Figure 3.1. A program which performs poorly on a slow processor will most likely also perform bad on a new processor. With bad performance is meant to what extent it utilizes the processor and not simply how fast it runs. The sketch shows that if a piece of code utilizes the processor badly, a doubling in performance would ideally halve the execution time, but due to the modern cache based architecture, latency and performance gap between processor and memory will realistically limit the impact on doubling in processor speed. Interestingly, the bad performing program might be tuned so it outperforms the un-tuned code on the new hardware. The sketch is not based on actual numbers, but throughout the thesis, several

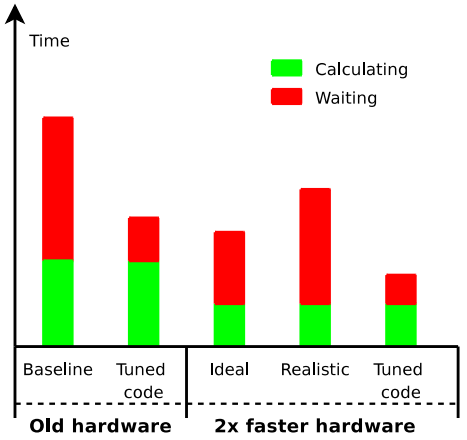

Figure 3.1: This sketch illustrates how a poorly performing baseline implementation can be made faster, by purchasing new hardware, tuning the code or both. Bad performance is often caused by bad memory access, which forces the CPU to wait instead of processing. Tuning refers to making the CPU more efficient and spend more time processing than waiting. The performance might increase more simply by tuning the code as opposed to buying new expensive hardware. The best performance is obviously obtained by tuning the code *and* buying new hardware.

examples of these performance gains will be presented.

Price versus performance is also worth having in mind, when you need large problems solved. There is a significant price difference between cutting edge and technology that is a few months old. In respect to how much performance tuning and parallelization are worth compared to simply buying faster hardware, an interesting observation can be seen simply by comparing processor prices. At the time of this writing, the price for one of the top model AMD high end processor "Dual-Core Opteron 290 2.8GHz" is 5028DKK [1], but the slightly slower "Dual-Core Opteron 275 2.2GHz" costs only 1695DKK. Since the processors are of the same family and with equal amount of cache, it is safe to assume that their performance scales roughly proportional to the clock speed. A performance increase of approximately $\frac{2.6}{2.2} \approx 1.18\times$ can be expected from a $\frac{5028}{1695} \approx 300\%$ increase in purchase price. The price of being the cutting edge of hardware technology is very high and this motivates parallel computers, clusters and performance tuning of existing code. This relation between cutting edge technology and high prices does not only apply to AMD processors or even processors, the trend is general for most hardware components.

When changing a parameter in order to gain performance, the term *speedup* is often used. This is a simple number, which can be calculated by dividing the new performance with the old performance as shown in (3.1).

$$speedup = \frac{Performance_{new}}{Performance_{old}} = \frac{\frac{Instructions}{Time_{new}}}{\frac{Instructions}{Time_{old}}} \tag{3.1}$$

The purpose of this chapter is to give an introduction to the performance tuning techniques and parallelization techniques used in this thesis. Performance tuning requires a good understanding of how a computer works, so the focus will be on how the computer works and then what that requires of the programmer in order to obtain good performance. A few simple C code examples will be used to explain some of the details, which are more easily explained by code than through words and drawings. Some of the examples will show how easy it is to get much more than the 1.18 times increase in performance mentioned above. The primary reference for this chapter is Hennessy and Patterson [2].

[1] According to www.edbpriser.dk - a site similar to other price comparison pages such as www.pricerunner.com.

3.2 Computer Architecture

Even though computers are considered a very modern piece of equipment, the basic ideas for the architecture used today is quite old. The first computers were based on mechanical relays or tubes and the program which they could perform were hardwired into their system. By using punch cards or other mechanical or magnetic storages, the machines were becoming more useful, but they still required a rewiring in order to perform other tasks. The need for a machine with easily changeable programs became apparent.

In 1945, John Von Neumann released his first draft of the "Von Neumann Architecture". In this *publication*[2] he described how the processing unit should be able to carry out instructions based on a program stored on the same media as the data. This was quite revolutionary at the time, and what we now call a *general purpose computer* is based on these ideas.

Computers have obviously come a long way since then, but the general structure of most computers is still based on Von Neumann's principles. Other architectures such as those found in simple devices, low end calculators and some dedicated hardware controllers, video codecs and other specialized processors are not based on the Von Neumann Architecture since their program cannot be changed.

3.2.0.1 The Von Neumann Bottleneck

Back in the days when Von Neumann proposed the architecture with the program written in memory, the storage memory was mechanical media such as punch cards or simple electromagnetic configurations. With the physical punch cards moving around the system in order to invoke instructions and supply data to a valve or relay based processor, the difference in speed of punch cards and processing circuit was obvious and the delivery of instructions and data did become a significant bottleneck in the system. This is known as the *Von Neumann bottleneck*.

With the introduction of planar processing, the computers turned from house size structures to small integrated chips. With everything made small and compact, it is compelling to think that every bandwidth problem would be solved simply by the down sizing of electronics. Unfortunately planar processing does

[2]There exists some controversy about this publication. All the people involved in the development are not credited and the disclosure of the ideas meant it was impossible to patent the architecture.

not solve relative performance problems, it has only increased the absolute performance and functionality.

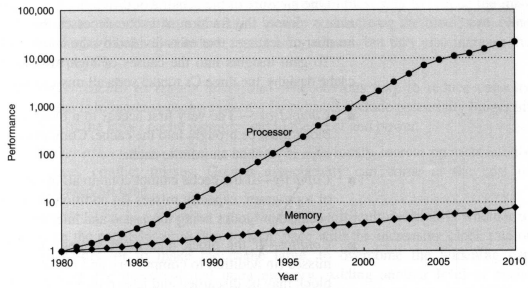

Figure 3.2: This graph shows the historical improvement in processor performance along with bandwidth of memory. From this graph it can be seen that the gap in performance between processor and memory is increasing quite rapidly due to the exponential growth in processor performance and the much slower increase in memory bandwidth. The picture is from Hennessey and Patterson [2].

Over the last few decades, the speed of processors have increased almost exponentiallly according to the famous Moore's Law stating that the number of transistors packed on a die would double every 18 months [2]. As can be seen in Figure 3.2, the bandwidth and latency of memory have however not been able to keep up with the increased processor performance. The gap between the two are steadily increasing, so time does not seem to solve the Van Neumann bottleneck. This gap might be caused by the simplified assumption of many consumers that clock speed equals performance. Who can blame manufactures for making what people want?!

3.2.1 Cache Based Systems

To overcome the Von Neumann Bottleneck, cache is used between processor and RAM. The cache is a small amount of specialized memory which is faster than the main memory. By copying the needed data to the cache, the CPU can access the data directly in the cache and thus minimize the need to communicate through the much slower bus to the main memory.

Depending on the architecture, there are a number of layers of cache. Most common is to have a level 1 (L1) cache running at clock speed and a larger amount of level 2 cache (L2), which runs at a particular fraction of the CPU

speed, but still much faster than the main RAM. Although not part of what is generally referred to when speaking about cache, inside the CPU there are a number of registers that serve as storage during computations, as loop counters and other time critical storage operations. Ideally the multi-layered cache scheme should enable the processor work efficiently using data from L1 cache and copy the needed data from L2 and RAM while it is working on something else. Unfortunately, this is not necessarily the case for most programs. With algorithms applied to large data sets, where only a few operations are performed each time, the main memory bandwidth and latency will restrict the speed quite substantially, if the implementation has not been adequately tuned.

L1 and L2 caches are very expensive, so consumer grade processors often have very limited cache available. High end processors have relatively large amounts of L1 and L2, while budget processors are sold cheaply due to the almost non existing cache. A high end processor such as UltraSPARC IV has 8MB cache per core while a normal Pentium 4 has somewhere between 1MB and 2MB of L2, depending on the specific model. The very popular budget model from Intel called Celeron only has between 128kB and 512kB L2 cache. A few years ago Celeron processors were sold even without L2 cache. The amount of L1 cache is very dependent on the particular processor family, but you will often find more L1 cache in a more expensive processor.

The memory hierarchy of a UltraSPARC IIICu can be seen in Figure 3.3. Approximate numbers for bandwidth and latency are also shown in the figure. It is clear to see that even from L2 cache, the latency is large enough to be a performance killer, if the processor needs to wait 19 cycles for every element it fetches from L2 cache. Although the figure shows a particular processor, this layout can be found in virtually all processors. Some key differences which might be between this and other processors are size of cache, particular bandwidth and some processors have embedded L2 cache on the CPU die.

With the large amounts of data often used in scientific computing, the more expensive processors with more cache are usually better suited for doing the calculations quickly since more of the data can be fitted in L2 cache.

Simply having a cache system does unfortunately not solve the memory bottleneck for good. It works perfectly if the problem size or memory footprint fits into cache, but since only L1 has low latency and high bandwidth, everything larger than L1 cache will need to be transferred through a slow bus, thus leaving the CPU waiting.

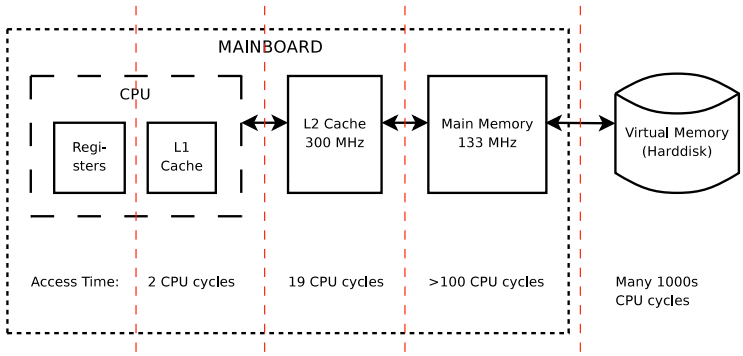

Figure 3.3: Memory hierarchy of the UltraSPARC III processor figure modified from [7]. What is important to see in this figure is the latency and bandwidth throughout the memory hierarchy. Fast memory is very expensive and thus only a small amount can be available. The larger the memory, the longer the latency, which makes good sense since the addressing is more time consuming when there are more elements to pick from. Furthermore, the lower clock rate further increases the number of cycles that the processor needs to wait. Even L1 cache requires two clock cycles to load a value into a register. A program with optimal performance would schedule the instructions so that something is carried out while it waits for the number to be available.

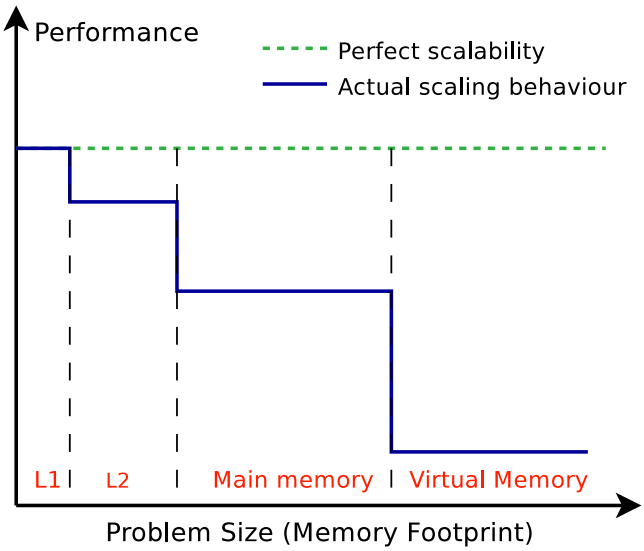

Figure 3.4: This sketch shows the typical relation between performance, measured in core algorithm instructions executed over time. Intuitively, there should be no performance penalty for increasing the size of the problem, and the performance should be constant such as the dashed green line indicates. The actual typical performance is indicated with the solid blue line and the performance decreases in steps, each time the memory footprint exceeds the capacity of a particular layer in the memory model.

3.2.2 Cache and Scaling Problems

In Figure 3.4, a rough sketch is shown of how performance, measured in instructions over time, actually decreases with problem size. This might seem odd since problem and algorithm is the same. When humans do calculations or other repetitive tasks, the efficiency often increases, when a large number of similar tasks needs to be done, but why is it completely opposite in a computer? The answer has already been touched upon and is due to the memory footprint of the problem, when the memory footprint exceeds cache size, more communication is needed through the slow bus to the memory at the level above. Instead of executing instructions that solve the problem, the CPU is forced to wait until the data becomes available and hence the slow down.

This decreased performance is not only restricted to the transition between L2 cache and memory, but is a general event throughout the whole memory arrangement. The worst slow down is obviously when the hard-disk needs to be used as virtual memory, and with an approximate factor 100 difference in bandwidth between main memory and hard-disk, the performance impact is indeed noticeable. Fortunately, Dynamic RAM is relatively cheep and the need for using virtual memory in calculations should rarely be necessary. Very large problems might need to be split in smaller portions to avoid using virtual memory.

If the performance drop of virtual memory can be avoided by partitioning the problem to fit in main memory, the next question is obviously if this technique can be used for all the layers. This is indeed possible and one of the main focus points of performance tuning is to make the problem fit in the faster caches, preferably L1. This is however not always as easy as it may sound, because some algorithms are hard to split up efficiently and an obvious penalty is some amount of overhead.

Before going into the details on how to modify a program to perform better with respect to memory, a little knowledge about the cache hierarchy is needed. In Figure 3.5, a sketch can be seen on how memory blocks are transferred. What is important to understand from this figure is that cache does not work on single bytes, but on small chunks of memory called cache lines. The latency and limited bandwidth available from the cache levels with more memory makes it necessary to move larger chunks of memory at a time in order to gain any improvement by having cache. If only single bytes were fetched from memory, the processor would have to wait for each and every single byte and thus render the cache useless unless the same byte is used many times. Cache lines are the smallest instance which can be fetched and if the program is written in such a way that the next elements in the cache line are needed, the cache does its job and the processor does not have to wait.

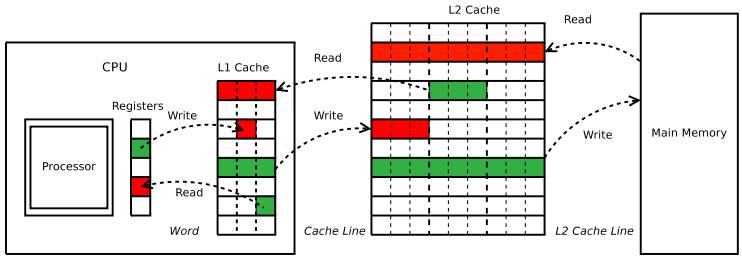

Figure 3.5: This sketch shows how the memory hierarchy of a normal cache based computer operates. From the left is the CPU with processor, registers and some L1 cache. The processor works directly with the registers, and if it wants to read a value from L1, it copies this particular memory location to a register. Writing works in the same way by copying a register value to the L1 cache. When an address, which is not in L1 cache, is needed, a whole *cache line* is copied from L2 cache (given that the address is already L2 cached). The length of a cache line is defined by the hardware architecture (64bytes on AMD Athlon X2 and 32 Bytes on UltraSPARC III). Also the L2 cache communicates with the main memory in lines or chunks on UltraSPARC III with 8MB cache, this length is 512 Bytes. (This has changed to 128 Bytes in UltraSPARC IV.)

The arrangement of which addresses gets copied into which slots in the cache might seem completely random in Figure 3.5, but the hardware implementation is obviously more structured. If the cache lines were simply placed in no particular order, the hardware memory manager would have to search every single cache location in order to check if the needed data was available, and that would be horribly expensive in terms of latency. How this is solved is to define that each and every memory address can only be mapped to small number of particular slots in the cache. Cache is often referred to as "2-way" or "4-way", and this is exactly the number of places each memory location can be in cache. The more "ways", the cache is, the more flexible the caching will be, but this flexibility is expensive to implement without suffering a larger latency.

The next section will cover some of the basic techniques to improve performance by better memory access.

3.3 Memory and Cache Management

Memory management is more complex than simply buying sufficient RAM. As mentioned, the CPU processes the data very fast compared to the bandwidth

from memory and storage. This means that one of the main goals for performance optimization is to make sure that the processor is constantly fed with data to operate on. When the processor is starving, nothing gets done.

It might seem impossible to make the processor work efficiently, and it sometimes is, but if used intelligently, the different layers of cache can make the operations very efficient on large data sets. Instead of moving around single words needed for a particular operation, a chunk of memory is taken from the RAM and moved to L2 cache. Even to L1 cache, only whole cache lines are transferred from L2 cache. Since L1 cache is running inside the CPU, the CPU does not have to wait long for the data already stored there. The main task is to ensure that as much as possible of the memory already copied to L2 and L1 cache can be reused. A good understanding of the memory structure is needed to be able to make programs perform optimally. Substantial performance improvements can be obtained by using favorable general optimization strategies, but to really use the full potential of the processor, very hardware specific tuning is needed.

This section will start theoretically and slowly progress to actual tuning techniques.

3.3.1 Cache Hits and Misses

In Figure 3.5, the system of cache lines were illustrated. As mentioned above, these cache lines or chunks of memory are essential to obtain any performance gain from the cache layers, because it becomes possible for the CPU to work on data which requires less latency to acquire. When the data requested by the CPU is in cache it is called a *hit*, and a *miss* when the data is not in cache. The penalty for a cache miss is the latency required to fetch a new cache line, so if this happens too often, performance will suffer.

The illustration in Figure 3.6 shows data read from memory in different ways and the cache hits and misses which occur. It should be fairly clear from the figure that the number of hits are much higher when the memory is accessed in a sequenced order corresponding to the hardware addresses.

At this point the focus has been on data caching, but instructions are also cached. This is very important since the processor must have instructions in order to actually do something. Programs are stored in memory along with the data so the same problems with latency and bandwidth also apply to the stream of instructions. Processors typically have the L1 cache split in two, one with instructions and one with data. The question of whether the processor is starving for data or instructions is a result of the algorithm and implementation

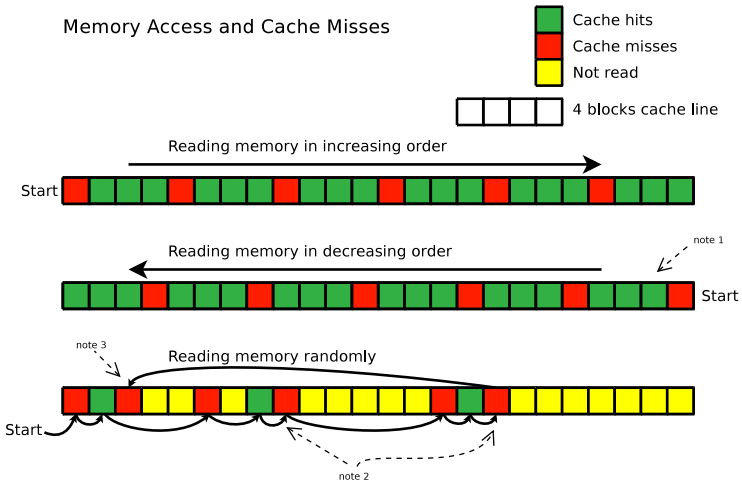

Figure 3.6: This is a simplified sketch of the cache hits and misses that occur when the memory is accessed in different ways. The memory is represented as a string of small blocks which correspond well to how actual hardware memory is addressed. When accessing in the direction of increasing address, the first block will be a miss, since nothing is cached, and the next three will be cached due to the cache line length of four. After these three hits, the next block is not cached and the cache miss results in the hardware fetching the next cache line. In the case of decreasing direction, the first will again be a miss. At *note 1*, there is a hit because cache lines are aligned in a fixed way so that the one which gets fetched is the one with the three elements with a lower address number. Things get slightly more complicated when the memory is accessed randomly as illustrated in the last case. Everything is just as before until *note 2*, where there is a cache miss due to the cache lines being aligned and hence the cached lines are only partially reused. At *note 3* one might think that a hit should occur, but since the program has been surfing around in memory, the first block has been pushed out of the limited cache area and it must be fetched again.

in question, so this is something which one needs to be aware of, but the focus will remain on data access except in the cases where there is a trade off between instructions and data caching.

3.3.2 Arrays in Memory

The importance of reusing cache has been described, but to understand how it is done, an understanding of the data storage is needed. Most scientific computing applications deal with data stored in vectors, matrices and cubic or higher dimension data structures. Memory in a computer, on the other hand, can be seen as a long string with consecutive addresses. Mapping a one dimensional vector or list into memory is quite straight forward since the elements can just be placed sequentially in memory with no trouble at all. All that needs to be taken care of are the start point and the step length between elements - if the elements are real numbers they will take up more space than more simple numbers such as integers, but this step length is taken care of by most programming languages. The end point or number of elements would also be a good thing to know, but in C, this is lead to the programmer to keep track of.

It is not as clear how to organize a matrix in memory. One possibility would be to "bend" the string back and forth for each row, but that would be terribly impractical. There are only two ways to store dense[3] matrices, *row major* or *column major* ordering. These types of ordering describe the direction in which the major jumps in address space are taken. The ordering can be seen in Figure 3.7.

These two different ways of structuring multidimensional arrays are used in Fortran and C, where Fortran uses column major and C uses row major. It might not seem like a big deal, but this ordering is exactly what needs to be understood to improve cache reuse. A good example of a very difficult task performance wise is to transpose a large matrix. If elements are read row-wise and copied to the column, the read sequence will perform nicely in C, whilst writing would be very inefficient due to large jumps in address space. In Fortran, this would be the other way around. Understanding Figure 3.7 in relation to Figure 3.6, the one with cache misses, is essential to obtaining good performance in matrix operations. The next sections will describe techniques to actually do this.

[3]Sparse, diagonal and other common structures can be stored differently for performance and memory reasons.

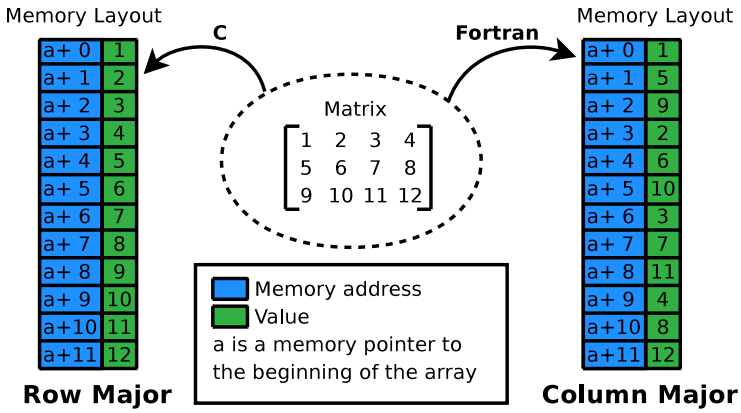

Figure 3.7: A diagram illustrating how a matrix is stored in memory using row and column major ordering. The matrix is in the middle and the blocks on the left and right represent the memory storage of this particular matrix. The standard way of describing a matrix in C is with row major ordering as shown on the left. The matrix is simply placed in memory one row at a time. Column major ordering as Fortran uses is just the opposite where each column is placed after the other in memory. With the chosen numbers in the matrix, the Fortran way seems much more complicated, but it is merely a different approach. The memory ordering is important to have in mind when doing performance tuning and interfacing Fortran routines from C or vice versa. To directly use methods from the other language, the matrix actually needs to be transposed - this will give the correct memory alignment.

3.3.3 Loop Ordering

The previous section has lead to a very important topic of performance tuning - the loop ordering. Most larger computations or memory operations can be done in a number of ways. A good example is the matrix multiplication $\left(\mathbb{R}^{M\times N}\right) = \left(\mathbb{R}^{M\times K}\right)\left(\mathbb{R}^{K\times N}\right)$ which can be solved using three loops ordered six different ways. Due to the cache ordering, some of these six possible implementations will perform significantly better than others. The matrix multiplication of $C = AB$ can be done in C as Listing 3.1 illustrates below.

Listing 3.1: Straight forward matrix multiplication

```
1  for(i = 0; i < M; i++)
2    for(j = 0; j < N; j++)
3      for(t = 0; t < K; t++)
4        C[i][j] = C[i][j] + A[i][t] * B[t][j];
```

This is basically as simple as it gets. The loop ordering reflects how you might normally do matrix multiplications using pen and paper. This ordering, however, is not the most optimal ordering, as mentioned there are six possibilities, which can be seen in Figure 3.8.

The arrows in Figure 3.8 clearly indicate that some implementations are more sensible than others. The MNK implementation written in C code above, which matches how matrix multiplication is done by hand, is not the worst possible implementation, but not the best either. Sometimes, the most intuitive implementation does not lead to the best performance, which is why it is crucial to think in terms of memory as well as math when implementing mathematical operations on large amounts of data in any form.

A better alternative to the matrix multiplication in Listing 3.1 is shown below.

Listing 3.2: Matrix multiplication, MKN ordering

```
1  for(i = 0; i < M; i++)
2    for(t = 0; t < K; t++)
3      for(j = 0; j < N; j++)
4        C[i][j] = C[i][j] + A[i][t] * B[t][j];
```

In Figure 3.9 the results[4] of the two loop orderings (Listing 3.1 and Listing 3.2) are plotted together with the NKM, which should theoretically show the worst performance. The results match the theory very closely. When problem size

[4]This test was done on an UltraSPARC IIIcu and the compiler "Sun C 5.8 Patch 121015-04 2007/01/10". To force the compiler not to do unwanted optimization, the flags "-fast -xprefetch=no -xdepend=no -xarch=v8plusb -xchip=ultra3" where used.

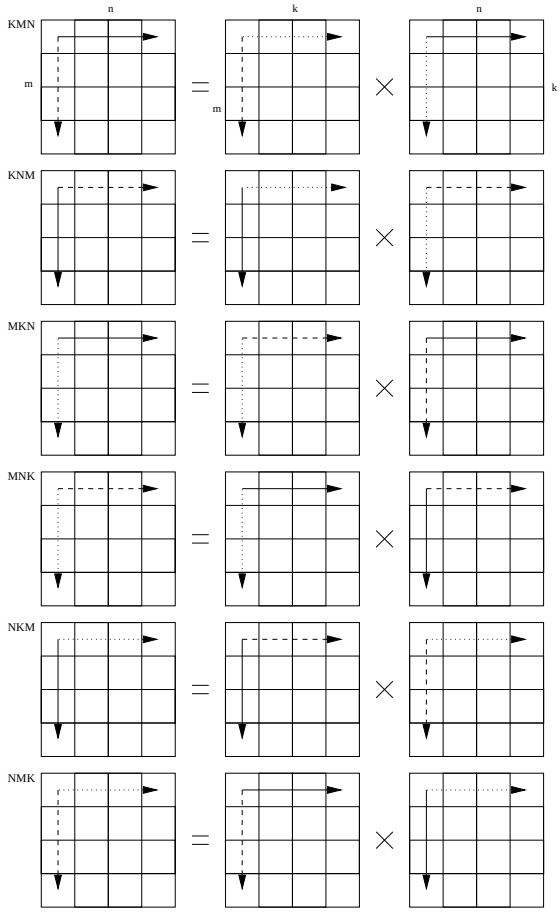

Figure 3.8: This chart shows the loop structure of the six different implementations of matrix multiplication. The solid line represents the innermost loop, the dashed line the intermediate loop and the dotted line shows the direction of the outermost loop. In a C implementation, the solid lines must be traversing a row at a time in order to achieve the best performance. The best performance should be obtainable from the implementation MKN, where two rows are accessed in the inner loop and the middle loop traverses a row and column and the last two column shiftings are kept in the outer loop. The worst performance is expected to be from the implementation marked NKM, where access in inner loop and two thirds of the middle loop are accessing the matrices in the most inefficient direction in respect to memory address space.

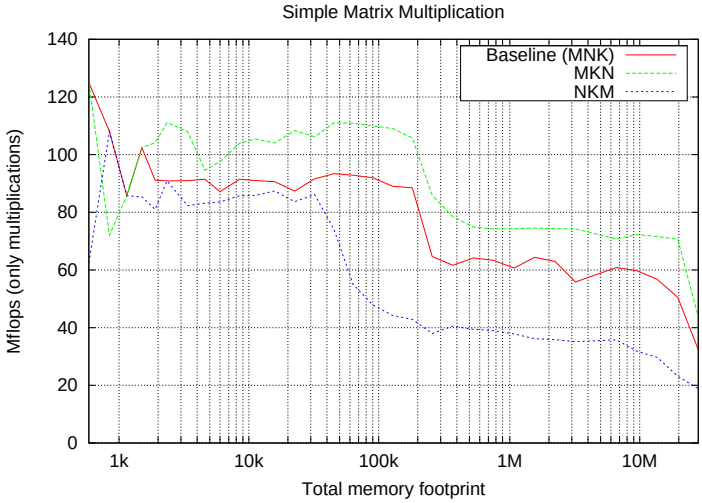

Figure 3.9: A plot of the performance of Listing 3.1,Listing 3.2 and the implementation NKM, which should theoretically be the worst performer of the three. The result illustrates clearly that there is a solid relation between loop ordering and performance. The program was tested on a UltraSPARC IIIcu processor and the steps corresponding to cache sizes described in Section 3.2.2 are very clear, but notice the loop ordering has an impact on the horizontal location of those steps too.

increases in size, the better implementation works at almost twice the performance, when compared to the worst implementation. This graph also supports the described stepped scaling behavior from Section 3.2.2.

It is interesting to see that the performance dips of MKN is just around 200kB of memory footprint. This is much larger than the L1 cache, but when looking at how the loops are arranged, it is clear that only the matrix $\mathbb{B}$ and single rows of $\mathbb{A}$ and $\mathbb{C}$ need to be in memory during the execution. This means that if $\mathbb{B}$ can stay in L1 cache, it will be perfectly reused, hence the performance drops significantly when a footprint of approximately $192kB$ is reached since then $\mathbb{B} = 64kB$ cannot stay cached throughout the whole calculation.

With the worst possible loop ordering, the footprint of all three matrices must be in cache at the same time in order to gain good performance. This can be seen with the steep drop immediately after the memory footprint reaches $32kB$[5]

3.3.4 Blocking

A common problem when doing calculations like for instance matrix multiplications on large matrices, is that data in cache is badly reused because it gets pushed out before it is needed again. This problem is easily identified in matrix multiplication from before because the whole of $\mathbb{B}$ is accessed once for each row in $\mathbb{A}$ and unless the problem is very small, the entire $\mathbb{B}$ and one row of $\mathbb{A}$ and $\mathbb{C}$ cannot fit in L1 cache at the same time.

The solution to this problem is to utilize a technique known as *blocking*. Although the code might sometimes be slightly more complicated than baseline implementation, the idea of blocking is quite simple. The problem is simply changed so that only one block of the problem is computed at a time. This may sound inefficient, but bear in mind that it is very common in matrix operations to reuse parts of the involved matrices several times.

A fixed block size is selected, this block size must be related to the problem and the amount of L1 cache in order get the best performance. If might be a good idea to try a range of different values and select the one which yields the best performance. The actual blocking implementation consists of two steps, working the algorithm in blocks and cleanup. Unless an integer multiple of the block size is equal to the problem size, cleanup is needed. This can be done before or after the blocked calculations, depending on the particular algorithm.

[5]The fact that this happens at 32kB and not 64kB, which would be expected, can be a result of how the memory accessed by the CPU, or something which restricts perfect L1 cache usage.

Some overhead is introduced from blocking and the clean up phase needs to be implemented efficiently too, in order to avoid losing the performance gained by blocking.

Below in Listing 3.3 is an example on how the matrix multiplication algorithm from Listing 3.2 can be blocked.

Listing 3.3: Matrix multiplication MKN

```
#ifndef BLOCKSIZE
#define BLOCKSIZE 36
#endif

  const int M_EDGE = M-M%BLOCKSIZE;
  const int N_EDGE = N-N%BLOCKSIZE;
  const int K_EDGE = K-K%BLOCKSIZE;

  register double *da,*db,*dc;

  for(bi = 0; bi < M; bi+=BLOCKSIZE)
    for(bt = 0; bt < K; bt+=BLOCKSIZE)
      for(bj = 0; bj < N; bj+=BLOCKSIZE){
        da = &(A[bi][bt]);
        db = &(B[bt][bj]);
        dc = &(C[bi][bj]);
        if(bi < M_EDGE && bt < K_EDGE && bj < N_EDGE){{
          // FULL BLOCK
          for(i = 0; i < BLOCKSIZE;i++){
            for(t = 0; t < BLOCKSIZE;t++){
              for(j = 0; j < BLOCKSIZE;j++)
                dc[i*M+j] = dc[i*M+j] + da[i*M+t] * db[t*K+j];
            }
          }
        }
        else{
          // CLEANUP
          for(i = 0; i < MIN(BLOCKSIZE,M-bi);i++){
            for(t = 0; t <  MIN(BLOCKSIZE,K-bt);t++){
              for(j = 0; j <  MIN(BLOCKSIZE,N-bj);j++){{
                dc[i*M+j] = dc[i*M+j] + da[i*M+t] * db[t*K+j];
            }
          }
        }
      }
```

There is a significant amount of overhead involved in the blocked implementation in Listing 3.3. The first three lines are preprocessor macros which make it possible to control the macro BLOCKSIZE at compile time. This way of controlling a parameter makes it easy to write scripts to test a wide range of possibilities without having to manually edit the values.

The constants defined afterwards are used to branch off between normal blocked

operation or cleanup, these are only calculated once, so they can just as well be constants. In line 9, three double pointers are placed in registers, by the keyword `register`. These will be used to point to the beginning of the blocks being multiplied[6].

The loops started from line 11 to 13 are where the actual blocking is made. These loops iterate in steps of the block size and the inner loops are simply working as if they were multiplying two square matrices with a length equal to `BLOCKSIZE`. To get optimal speed, the actual multiplication has been split in two different cases, controlled by the branch in line 17, one with complete blocks and one which accepts partial blocks. The reason for splitting this in two cases are the calls to `MIN` in the partial version, which makes this less efficient than the version with fixed block size[7].

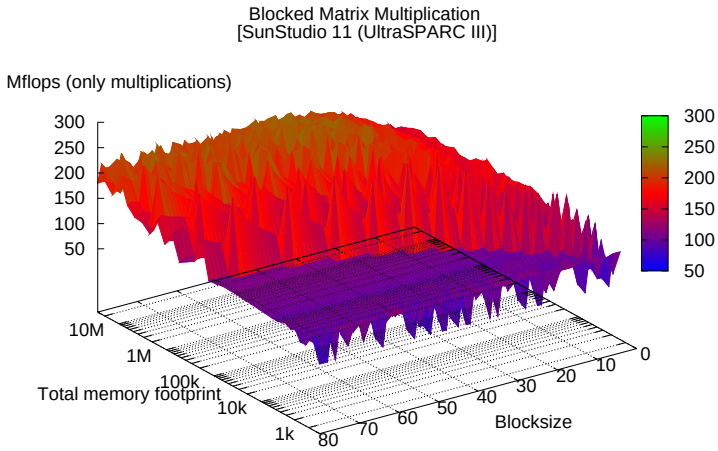

Figure 3.10: A 3D surface plot showing performance of Listing 3.3 as a function of block size and memory footprint. This plot shows a completely different relation between problem memory footprint and performance. The performance is very good even when L2 cache limit is exceeded. In general, the performance is much better than the simple MKN implementation.

In Figure 3.10 is a 3D surface plot of the performance measured in Mflops

[6]The performance difference between indirect addressing using normal matrix structure and these pointer registers was quite substantial.

[7]The call to `MIN` will be carried out for every iteration, so some performance might be gained by calculating these values even before entering the loops.

when changing the problem size and most importantly the block size[8]. The performance gain over the MKN implementation from previous is substantial. Especially the performance with large problem sizes is much better.

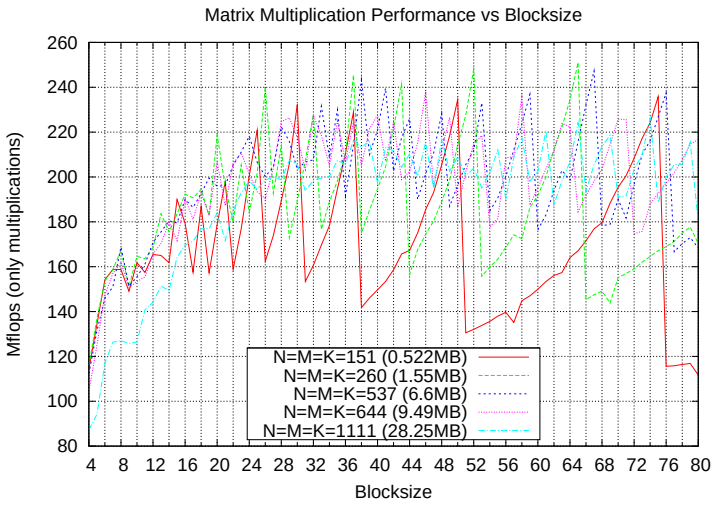

Figure 3.11: A few selected problem sizes plotted with performance against block size. This plot can be used to select the best block size for this particular implementation. The area around 36 seems like a good option for the blocking size, since the performance seems high and stable for most problem sizes.

The plot in Figure 3.11 can be used to select the optimal block size for this implementation. As mentioned in the figure text, a choice of 36 will be quite reasonable, since the performance is high and stable here. Interestingly, the memory footprint of the blocked multiplication is $\frac{3 \cdot 36^2 \cdot 8B}{1024} \approx 30kB$, which is half of the L1 cache available in this particular processor. The performance is also quite good except for N=151 when the whole L1 cache is filled at the block size 52. An imperfection of this implementation is also illustrated by this plot - the correlation between how much clean up is performed and the speed is very strong. What this essentially means is that the clean up loop part is not efficiently implemented compared to the full block part.

Blocking is an essential technique to gain optimal performance when working on problems which are larger than the L1 cache can contain. It does however require an amount of overhead and the clean up algorithm is just as important

[8]Again the code was run on a UltraSPARC IIIcu 1050MHz and the compiler "Sun C 5.8 Patch 121015-04 2007/01/10" with the flags "-fast -xprefetch=no -xdepend=no -xarch=v8plusb -xchip=ultra3".

as the full blocked algorithm.

3.4 Processor and Instructions

The tuning techniques described so far have focused on re-structuring the program to optimize the use of cache. Modern processors offer other features which can further help speed things up. Dedicated floating point units for adding and multiplying are a standard, and most processors also have scheduling logic that optimizes the use of the processors' special features. These features include prefetching, pipelining and instruction level parallelism.

The focus in this section will be on how to keep the scheduler happy in order to utilize the full potential of the processor and describe prefetching.

3.4.1 Branches in Inner Loops

As mentioned in the comments to the blocked algorithm above, the clean up algorithm is not very good because of the branching. Some of the efficient calculating features of processors are dependent on guessing what is going to happen next. When there are branches in the inner loops, the processor cannot predict what will happen and cannot prepare the usage of its high performing calculation flow (more on this in Section 3.4.3).

In general it is a bad idea to have branching, most commonly general `if` statements and function calls. In C programming it is possible to instruct the compiler (if it supports it) to inline particular functions either by compile flags or through explicit statements in the code. Doing this makes the compiled program larger, but scheduling is easier for the processor, so in many cases, this will improve performance.

The blocked code in Listing 3.3 had a poor performing clean up section, and it is interesting to see how much branching in inner loops is to blame for this. Particularly line 9 of Listing 3.3 is hard on the CPU scheduler, because the macro `MIN(a,b)` expands to `((b¡a)? b : a)`, which is basically an `if` statement and this happens at every iteration. This only needs to be evaluated once on entry, so below is an alternative clean up implementation, without unnecessary branching.

Listing 3.4: New clean up section for Listing 3.3 with less branching

```
1       else{
```

```
2           // CLEANUP
3           const int minm = MIN(BLOCKSIZE,M-bi);
4           const int mink = MIN(BLOCKSIZE,K-bt);
5           const int minn = MIN(BLOCKSIZE,N-bj);
6           for(i = 0; i < minm;i++){
7             for(t = 0; t <  mink;t++){
8               for(j = 0; j < minn;j++)
9                 dc[i*M+j] = dc[i*M+j] + da[i*M+t] * db[t*K+j];
10            }
11          }
12        }
```

The only thing changed in Listing 3.4 are three new constants which holds the value of whether this direction can be a full block step or a clean up step. The result can be seen in Figure 3.12.

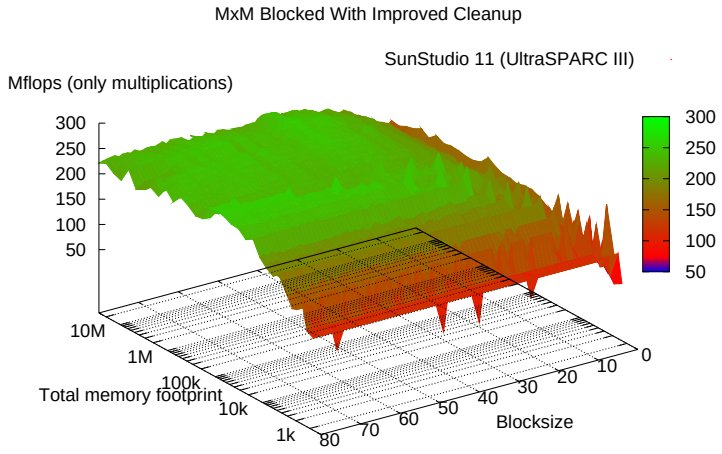

Figure 3.12: A 3D plot of performance versus block size and memory footprint of the matrix multiplication with improved clean up algorithm. The most noticeable aspect of this plot compared to the one in Figure 3.10 is how flat it is. This is directly related to a better clean up algorithm, which was reducing the performance before.

The results in Figure 3.12 clearly show a much more reliable performance from the blocking algorithm when the unnecessary branching is taken out as expressed in Listing 3.4. The performance still seems to be best around a block size 36. Comparing the performance of the two blocked algorithms Listing 3.3 and Listing 3.4, one with branching and one without, there is an average increase of

40%[9] by eliminating the branches.

3.4.2 Prefetch

Instead of waiting for cache misses, it makes good sense to get the data before hand, which is what prefetching does. This performance feature can be found in most processors. It has been standard in SPARC processors for many years and Intel and AMD have likewise implemented prefetching instructions with the SSE and 3Dnow! additions [4] [3]. Prefetching can either be done in hardware or by instructions (software). Hardware prefetching can only work efficiently if the processor can guess what data will be needed next, and this requires essentially sequenced access. If more elaborate memory access patterns are necessary, the prefetch instructions are needed.

The prefetch instruction can be called directly through the commands in sun_prefetch.h using SunStudio or in GCC through assembler instructions. The most simple way to use prefetching is to instruct the compiler to automatically turn on prefetching. This usually requires the compiler to know exactly what instruction set architecture it should be compiled for.

Explicit prefetching for read can be done for a 3Dnow! AMD chip like this:

Listing 3.5: Explicit prefetch using 3Dnow!

```
1 __asm__ __volatile__ ("prefetch %0" : : "m" (*((char *)(ptr))));
2 __asm__ __volatile__ ("prefetchw %0" : : "m" (*((char *)(ptr))));
```

This calls the prefetch instruction directly by assembler commands and places a cache line pointed to by ptr in the prefetch buffer. The ideal way to do prefetching is by requesting the cache line which will be needed after the one already cached is going to be used. What this means is that cache line two should be prefetched before computations are started on line one; and before starting computations on line two, an instruction to fetch line three should be issued.

By including the header sun_prefetch.h and using the compiler flag -xprefetch=explicit, the compiler in SunStudio accepts the command below for prefetching for the SPARC architecture:

Listing 3.6: Explicit prefetch on SPARC

```
1 sparc_prefetch_read_many((void*)(ptr))
```

[9] Blocksize was 36 for both algorithms. The performance increase is larger the more clean up is needed, and ranges from 0% to 120%, so the average performance increase can be adjusted by selecting the matrix sizes used.

```
2 sparc_prefetch_write_many((void*)(ptr))
```

The "many" means the cache line should be fetched for being read more than once and has nothing to do with how long cache lines will be fetched. The fact that cache lines vary on different processors makes explicit prefetching code less portable.

Prefetching is an essential method for achieving optimal performance, but if the data is accessed efficiently, the hardware prefetcher does a good job.

3.4.3 Instruction Level Parallelism and Pipelining

In terms of scientific computing, some very important features are the ones that make it possible to do fast calculations. There is a limit to how high the clock frequency can be on a processor, so carrying out multiple computations each clock cycle is a good strategy for improving performance.

Only very few computer instruction can be carried out at a single clock tick. This is only possible for very simple operations such as integer addition. When a large operation can be split up into smaller operations, each taking the same amount of time (preferably a clock tick), the whole operation can be *pipelined* by connecting these small operations as shown in Figure 3.13.

A pipeline essentially works like a conveyor belt in a factory. Small equally time consuming operations are performed on each stage in such a way the input and output are constantly flowing. The sketch in Figure 3.13 shows how four tasks each requiring three clock ticks can be carried out in only seven clock ticks. Had it not been for the pipelined process, the four tasks would have been completed in 12 steps instead of seven.

Processors offer pipelines for doing special tasks and it is important to write the code in such a way that the processor can schedule good use of its pipelines.

Another performance feature often found in processors is *instruction level parallelism*. As the name suggests, this involves processors being able to process several instructions simultaneously. An UltraSPARC III is said to be *four way super scalar*, which refers to the fact that it supports instruction level parallelism with four operations at a time. Limitations exists to what operations can be performed in parallel together, but this is very specific for a given processor, so this will not be covered here.

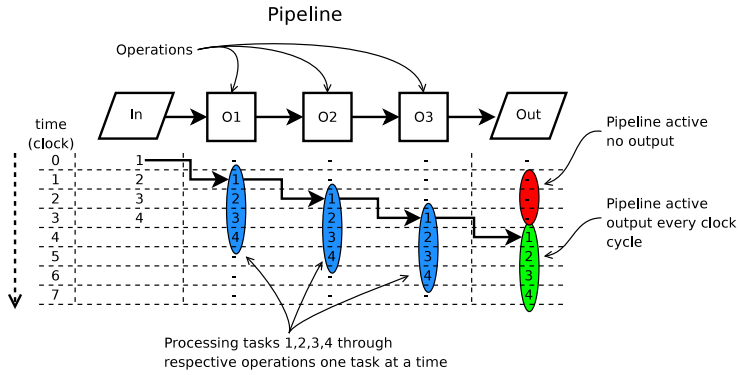

Figure 3.13: This sketch illustrates how a pipeline works. The three operations O1,O2 and O3 are performed on the tasks 1,2,3 and 4 by letting O1 operate on task 1 first and then passing the task on to O2 as well as receiving task 2 at the next clock tick. The tasks progress through the pipeline in this manner, and when the pipeline is full, one task is finished at each clock tick.

3.4.4 Fission and Fusion

One way to make instruction level parallelism easier for the compiler and the hardware instruction scheduler is to modify the loops. The theoretical approach to doing this is to obtain the specifications of the given processor or processor family and identify the pipelines and parallel instructions relevant to this algorithm. With a precise knowledge on what operations can be performed simultaneously, it is a simple matter of splitting up the loops or perhaps joining loops together to utilize the full capacity of the processor. Splitting or joining the loops is referred to as *loop fission* and *loop fusion*, respectively.

The more hands on approach of randomly trying with loop fission or fusion might give the same results and with an increasing level of experience, this process becomes less random.

Fusion can be used with simple loops, which might do inefficient store-load operations. Here is a simple example of loop fusion:

Listing 3.7: Loop Fusion

```
1 // Not good
2 {
3   for(i = 0; i < N; i++)
4     b[i] = b[i]*c[i];
```

```
5
6     for(i = 0; i < N; i++)
7       d[i] = b[i] + 1;
8   }
9   // Better with fusion
10  {
11    for(i = 0; i < N; i++){
12      b[i] = b[i]*c[i];
13      d[i] = b[i] + 1;
14    }
15  }
```

If N is large, the fused loops will have better cache reuse of `b` and instruction level parallelism is made possible.

Fission is splitting a complex loop into smaller loops. This is a good thing if the complex loop can be split into independent less complex loops. If the loop works on large arrays, the cache usage might be improved, unless the dependency is like in the fusion example in Listing 3.7. The more simple loops will make it easier for the compiler to do optimizations such as unrolling, which will be described next.

3.4.5 Loop Unrolling

What is known in programming as a loop is in reality only a branch and jump operation in the CPU. The evaluation of whether to go through the loop can be placed either before or after the loop content. A for-loop is, for instance, evaluated on entry, where a do-while loop is evaluated afterwards.

As already mentioned, branching makes it difficult for the processor to properly schedule its pipelines and super scalar capabilities. The solution for this is to do what is known as *loop unrolling*. What this loop unrolling means is to explicitly write down some of the loop iterations and thus create a bigger loop, which is run through fewer times. Below is a the main blocked loop from Listing 3.3, which has been unrolled four times.

Listing 3.8: MxM Loop Unrolled 4 times

```
1 #define BLOCKSIZE 36
2
3             // FULL BLOCK
4             for(i = 0; i < BLOCKSIZE;i++){
5               for(t = 0; t < BLOCKSIZE;t++){
6                 for(j = 0; j < BLOCKSIZE;j+=4){
7                   dc[i*M+j   ] = dc[i*M+j   ] + da[i*M+t] * db[t*K+j   ];
```

```
8                   dc[i*M+j+1] = dc[i*M+j+1] + da[i*M+t] * db[t*K+j+1];
9                   dc[i*M+j+2] = dc[i*M+j+2] + da[i*M+t] * db[t*K+j+2];
10                  dc[i*M+j+3] = dc[i*M+j+3] + da[i*M+t] * db[t*K+j+3];
11                }
12              }
13            }
14          }
```

This might not seem like something which makes the code run any faster, since it just introduces more lines of code, but actually, loop unrolling is an essential performance tuning technique. One important thing to remember when doing hand unrolling is to adjust the loop increments to the number of unrolling steps performed and make sure this number is a divisor of the total amount of steps to take. In Listing 3.8, this is done by manually ensuring that the BLOCKSIZE matches the unrolling at compile-time and as can be seen in line 6, the loop counter `j` is incremented in steps of 4.

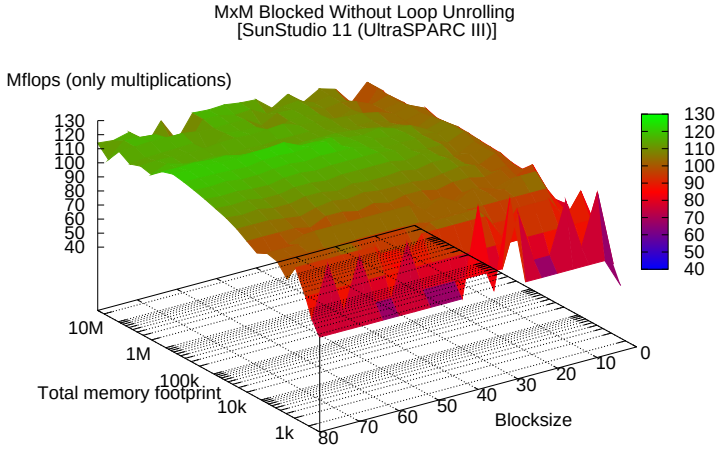

Figure 3.14: This is plot of the matrix multiplication performance as a function of block size and memory footprint. This is the exact same code as described in Section 3.4.1, but the compiler unrolling has been switched off. A comparison with this plot and the one with unrolling enabled (shown in in Figure 3.12) indicates that the performance is approximately halved when turning off unrolling.

It would be nice to show a convincing plot of the gain which can be achieved with the matrix multiplication algorithm by doing some hand unrolling, but this technique is so fundamental that it has been turned on by the compiler all along. In Figure 3.14 is a plot of how it looks when unrolling is turned off. The

speed is only half of what was observed in Section 3.4.1.

The compiler often does a very good job of unrolling since this is such an essential feature, and much work goes into this function. What the programmer seeking performance gain can do is to make it easier for the compiler to predict how to unroll. If, for instance, the compiler knows how many iterations a particular for-loop will carry out at compile time, the compiler can fully unroll the loop. This is really good in blocked algorithms, where the inner loop is already fixed in size. If the block size had been a variable and not a preprocessor macro, the compiler could not possibly know how many iterations the loop would do and there goes the chance of fully unrolling. Without the full unrolling, the processor needs to flush the pipeline before checking if another iteration in the loop is required.

Unrolled loops will make the compiled program larger, as well as most likely increase the amount of code which needs to be cached. Given no problem of supplying data, instruction cache misses are the limiting factor of to what extent loops can be unrolled. If the performance drops after unrolling very deeply, the culprit is most likely the instruction cache not being able to keep up. This should obviously be avoided by not unrolling too much.

3.5 Choosing the Right Compiler

As mentioned in the previous section, the compiler can do things to help you in terms of performance. What a compiler does is transform the C code to machine code, so the program can be run. One might think that a program is uniquely defined by the C code, but despite C being a relatively low-level language, the actual transformation to machine code offers plenty of room for interpretation. In order to get the best performance on a particular computer architecture, the selection of compiler is essential.

Here is a list of some of the things a compiler might do to improve performance:

- Use mathematical processing unit in CPU
- Loop interchange based on memory access
- Utilize processor pipeline and super scalar capabilities
- Insert prefetch instructions
- Use internal registers for counters and temporary values

- "Change" code to perform better

Most compilers use some very conservative optimization settings as standard, so in order to get the full set of features, a good knowledge of the compiler optimization flags is required.

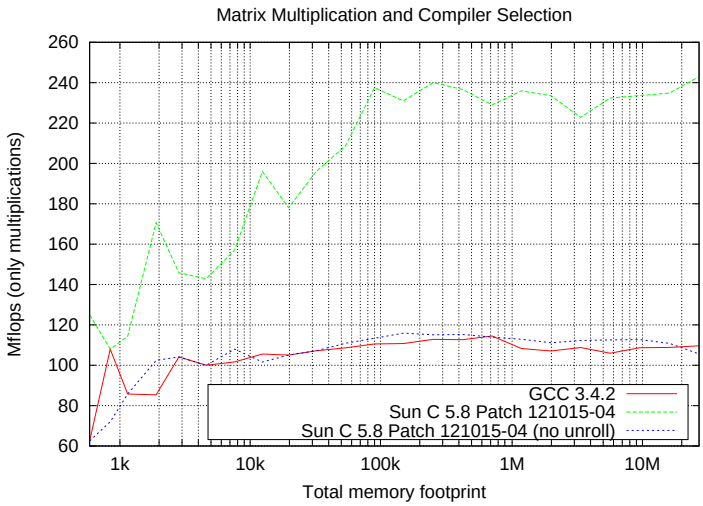

Figure 3.15: The well known matrix multiplication code tested using different compilers on a UltraSPARC III. The GCC compiled version performs significantly worse than the one compiled using unrolling on the Sun compiler. It is interesting to see that the version which is not unrolled performs nearly identically to the GCC version. This indicates that the GCC version does not use the four way super scalar capabilities of the processor.

Before selecting compiler flags, the actual compiler must be selected. For high performance computing, selecting the best compiler for a particular platform can have a tremendous impact on performance.

As an example, the best performing version of the matrix multiplication has been complied using GCC 3.4.3 and run on the same UltraSPARC III as all the other tests. The result can be seen in Figure 3.15[10]. This clearly shows that selecting the best compiler can make a huge difference, and on Solaris SPARC, the obvious choice is to use Sun's compiler. It makes good sense that Sun has

[10]GCC 3.4.3 was used with the recommended optimization flags for SPARC "gcc -g -O3 -mcpu=ultrasparc -funroll-loops" and the Sun compiler was the same as all the other experiments in this chapter.

the better performing compiler for their own platform. Intel likewise offers a compiler for their processors which they claim outperforms GCC. A lot of effort is required to make a compiler produce faster code, and since GCC is mostly used on x86 architectures, it is safe to assume that more time has been spent on getting optimal performance on this platform.

3.5.1 Compiler Flags

To get the most performance out of the final binary it is a very good idea to understand and use the performance flags available on the compiler. All the optimization capabilities are not turned on by default. It might seem strange that a compiler does not produce the optimal binaries by default, but some performance features such as unrolling increase the binary size which might be unwanted in some cases and it takes longer to compile if more options are turned on. The final binaries might also be intended for use on different machines (CPUs), and then it would be incompatible if a performance instruction subset was enabled, which was not available on all target machines.

The Sun C compiler offers many compiler flags and it is a very good idea to understand and use them if the program is to perform well. The performance flags are very well documented in Sun's C compiler and a list can be obtained by writing `cc -flags`. There are five level of optimization in Sun's compiler and these can be set by the flag `-xOn` where `n` is a number describing the level of optimization. A really nice feature with Sun C is the macro definitions of performance flags. There is a flag called `-fast`, which enables all the recommended performance flags. This is a very good starting point for getting the full potential out of the compiler.

GCC have similar options for selecting the optimization level, but only three levels are defined. Unfortunately, the documentation is not as complete as for the commercial Sun C and there is no `-fast` flag in GCC.

When selecting the correct flags, it is important in both Sun C and GCC to select the processor architecture, for which the binary is intended. Valuable performance instructions such as prefetching, pipelining and instruction level parallelism are only used optimally if the compiler knows about them.

3.6 Parallelization

In high performance computing, the computing demand is often higher than what a single processor can provide. The obvious solution to such a problem is to run more processors in parallel and thus gain a combined performance higher than the single processors. As mentioned in the introduction, Section 3.1, the price difference between the fastest processor and the next or third best is usually much higher than performance difference. In the example, tripling the purchase price only yielded a theoretical speedup of 1.18, but if three of the cheap processors were run in parallel, the performance would potentially be proportional to the purchase price. In general, this is a very good reason to run processors in parallel, the only problem with this approach is that motherboards capable of running several processors in parallel are much more expensive due to complexity and limited demand.

After realizing the power problems associated with the ever increasing clock-rate, the major processor companies have started to place two or more actual processors on the die and thus provide a much easier migration path to parallel systems.

Parallel systems consisting of a number of single core processors or multi-core processors are parallel in a significantly different way than processors with instruction level parallelism, as mentioned in Section 3.4.3. At instruction level parallelism, a single thread is running on the processor and within this thread, particular instructions might get executed in parallel on the processor. In a real parallel system, each processor runs its own instance of the program in a thread. Since the program needs to run in several threads, the program needs to be designed particularly for running on such a system. This is why people often talk about multi threaded programs when discussing parallel architectures.

3.6.1 Converting Code to Run in Parallel

Converting a program from running in a single thread to efficiently run in more threads is a very complex task involving communication between threads, synchronization and control of what needs to be done before other things. Each of these problems require many lines of code to be done safely and portability might become a major issue. Fortunately, this problem has already been solved with the use of adding an abstraction layer in the form of a toolkit for the programmer to use.

Two major toolkits for assisting the programmer in writing parallel applications

are MPI (Message Passing Interface) and OpenMP (Open Multi Processing). While MPI is a regular library toolkit with methods for sending data back and forth and controlling the process, OpenMP provides a number of preprocessor directives that automatically convert the specified parts of the code into multi threaded code. Since only preprocessor directives are used, it is possible to make a program which can be compiled for multi-threading if the compiler supports OpenMP and single-threaded if the compiler is unaware of the OpenMP directives, which would then simply appear to the compiler as comments.

Due to the shared memory hardware available personally and at DTU, the focus will be on OpenMP. The following sections will describe some of the issues general to parallelization and OpenMP in particular.

3.6.2 OpenMP

Although OpenMP might sound like magic, it still requires the programmer to avoid some general pitfalls, which might limit the speedup significantly. A limitation of OpenMP compared to MPI is that it is mainly supported on shared memory systems, which makes it unsuitable for clusters. OpenMP is easier for the programmer and less intrusive to the code, but it requires the compiler to support it. Sun Studio has supported OpenMP for years, which might seem natural since Sun sells high end shared memory servers; providing tools capable of taking full advantages of the systems is a must. OpenMP will definitely receive more attention with all the dual core processors being sold and in the recent release 4.2 of GCC, OpenMP is now a built in feature.

To use OpenMP with Sun C and GCC 4.2 it must be enabled with a flag. When it is enabled, the OpenMP implementation will go through the code, looking for `#pragma omp` statements and translating the code to multi threaded environments accordingly.

The program is made parallel by the `#pragma omp parallel` statement, but this simple statement alone does nothing more than run multiple instances of the same piece of code. To split the work, special statements must be inserted either before loops or by splitting the work by hand. The most simple work splitting is done with the `#pragma omp for` statement, which splits the following for-loop into equally sized portions, which are then split by the running threads.

An important thing to remember is to specify what variables should be shared amongst the workers and which one are private. Loop counters and temporary variables are naturally private, and the data being analyzed should most likely be shared. If the loop counter is accidentally marked as *shared* the program

would not work, because each thread would increment the counter and only parts of the loop would be carried out correctly. This and other pitfalls will be explained in Sections 3.6.5 and 3.6.6.

3.6.3 Amdahl's Law and Speedup

One might assume a proportional relation between the number of CPUs and performance, but this is not necessarily the case. The speedup associated with an increase in number of processors is highly dependent on the algorithm and implementation. Some algorithms are inherently incapable of running in parallel, and then no speedup can be obtained. Examples of such algorithms are some recursive processes as well as every time the threads need to share a single resource such as input and output devices.

The basic idea behind Amdahl's law in terms of parallelization is the assumption that a given program consists of a fraction F_{seq} which is sequential in nature and cannot benefit from parallelization. The rest of the program $(1 - F_{seq})$ experiences linear speedup with the number of dedicated threads $N_{threads}$. This law can easily be applied in order to predict how much speedup can be expected from running the program on a multiprocessor machine. The special case of Amdahl's law for parallelization speedup can be written like this:

$$speedup\,(N_{threads}) = \frac{1}{F_{seq} + \frac{(1-F_{seq})}{N_{threads}}} \tag{3.2}$$

This is a very simple equation and it makes good sense that if $F_{seq} = 1$, no speedup will be experienced and if $F_{seq} = 0$, the equation will simplify to $speedup\,(N_{threads}) = N_{threads}$. In Figure 3.16 is a plot of speedup according to (3.2) for different fractions of sequential code. The plot shows that even if a tiny part of the code needs to run sequentially, the obtainable speedup is affected.

When assessing the cost of processors according to speedup, the plot in Figure 3.17 gives a good picture of how much additional speedup is obtained by the last added processor.

The message from Amdahl's law is that one should not blindly think that improving one parameter (number of processors in this case) can provide linear speedup. In the case of parallel programs, one of the most important tasks will be to ensure that F_{seq} is as low as possible.

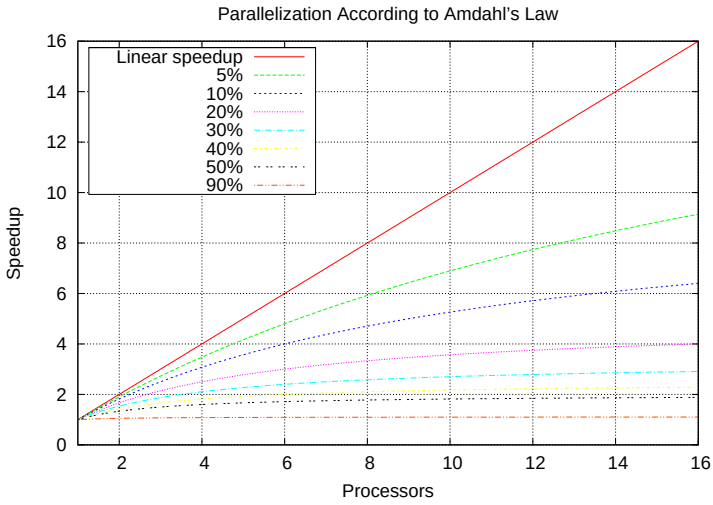

Figure 3.16: This plot illustrates that speedup is associated with an increase in the number of processors according to Amdahl's law. Linear or "perfect" speedup has been plotted as a reference. The graph clearly shows that even if only a small amount of the code is inherently sequential in nature, the speedup is far from perfect when the number of processors grows.

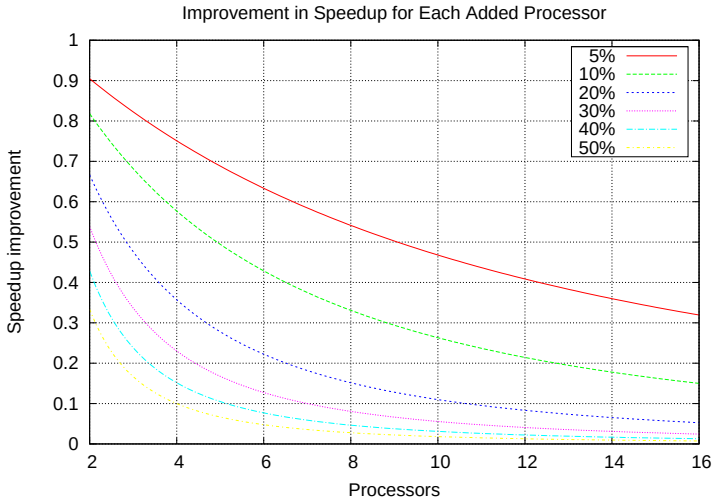

Figure 3.17: If the speedup is $speedup\,(p)$, this graph shows $speedup\,(p) - speedup\,(p-1)$, which gives a quantitative measure of how much the "last processor added" contributed to the speedup. This is essentially the same as differentiating, but only using the discrete number of processors. The plot shows that unless the algorithm is mostly parallelizable, the speedup delivered by the processors added after 4-8 is hardly noticeable.

3.6.4 Cache and Super linear Speedup

On the contrary to what Amdahl's law suggests, speedup can sometimes in practice be superior to proportional speedup. The reason for this is that the extra amount of cache in each processor might make the problem exactly fit in cache. For some problems it might be a very good idea to assign the number of processors with a total amount of cache capable of addressing the problem memory footprint in cache.

3.6.5 Race Condition

A very common problem with parallelization is handling the memory in a consistent manner. A very unfortunate situation is when two processors are trying to update the same memory address at the same time. This situation is know as *race condition*, because the processors are racing against each other to update the value. The problem can be illustrated by parallelizing the very simple example in Listing 3.9.

Listing 3.9: Simple vector sum

```
1 sum = 0;
2 for(i = 0; i < N; i++)
3   sum += A[i];
```

As described in Section 3.6.2, all variables must be declared either private or shared on entry to the parallel piece of the program. The code in Listing 3.9 has only two non trivial variables, `sum` and `A`. `A` must be shared since it is the actual input (which should not get reallocated) and sum must be shared too, because it holds the result. The code can be seen below.

Listing 3.10: Simple sum with OpenMP race condition.

```
1
2 #pragma omp parallel shared(A,sum) private(i)
3 {
4 sum = 0;
5 #pragma omp for
6   for(i = 0; i < N;i++)
7     sum = sum + A[i];
8 }
```

Listing 3.10 might look reasonable in terms of the OpenMP standard, but getting the correct result from this piece of code running in more than one thread is very unlikely. All threads would attempt to update the same address in memory, but what makes it even worse is that the previous value is required. There is no

control of which thread updates the value. The only hope one might have is that the hardware is constructed in such a manner that both the read and write operation is carried out in one thread at a time.

The problem can be solved easily using OpenMP's "critical region":

Listing 3.11: Simple sum with OpenMP critical region.

```
1 sum = 0;
2 #pragma omp parallel shared(A,sum) private(i,psum)
3 {
4 psum = 0;
5 #pragma omp for
6   for(i = 0; i < N;i++)
7     psum += A[i];
8 #pragma omp critical
9 {
10   sum += psum;
11 }
12 }
```

The "critical region" ensures that only a single thread executes the command at a time, so the race condition is avoided and the sum will be computed correctly. This region does, however, add complexity and expensive synchronization between the threads, but if the length of `A` is sufficiently large, the relative performance loss might be acceptable.

3.6.6 False Shared

A common parallelization pitfall on shared memory systems is a condition called *false shared*. The term refers to memory uniquely accessed by each thread, but due to the hardware implementation of cache coherency, the threads will be fighting over the same cache line in similarity to how threads fight to get access to a common memory location during race condition.

Understanding false shared requires a good understanding of how the cache system works on a hardware level. The problem arises as a consequence of the essential *cache coherency* features in cache coherent shared memory systems. The cache coherency system ensures that each processor is only working on a current version of memory in its cache. With a large number of processors, each with its own cache, it is clear that some control is needed in order to make sure that a read from a particular address yields the same result on all CPUs.

In Figure 3.18 is a sketch of how four processors work with each having their own cache, but all share the same memory address space. The cache coherency

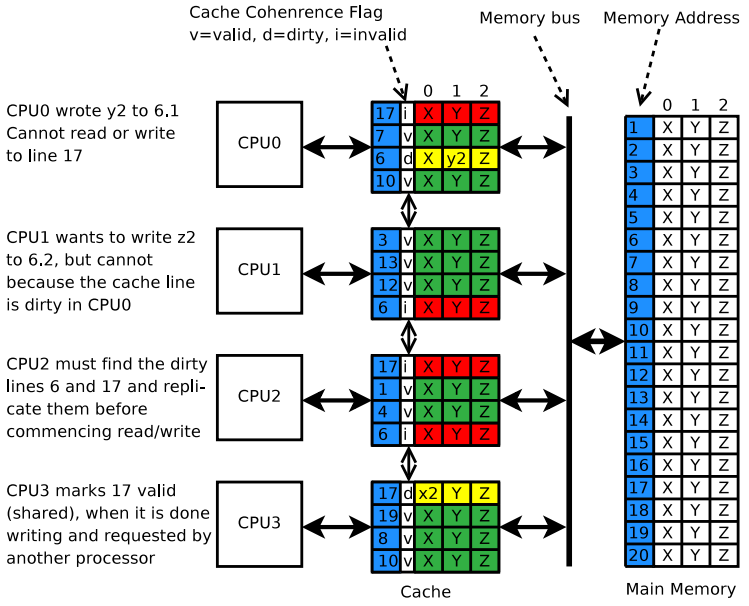

Figure 3.18: A simplified illustration of a cache coherency system. Four processors are sharing the same address space, but each have their own cached copy of the cache lines they want to work with. The arrows between the cache's cache coherency flags are illustrating a communication between these. Different techniques for this communication exist, but the idea of marking cache lines is the common way to do this.

works by marking the cache lines as valid, invalid or dirty. When a processor wants to write to a particular cache line, regardless of where in the line, it marks the line dirty and the other copies of this particular cache line are invalidated. If a processor with an invalid cache line wants to read or write to that line, a cache miss is placed and the dirty line will be located and distributed. False shared occurs when more than one processor are regularly updating a value in the same cache line.

In order to avoid the race condition in Listing 3.9, one might accidentally create a false shared instead, an example of this can be seen in Listing 3.12, where the small array sumv is subject to being a false shared.

Listing 3.12: Simple OpenMP sum with false shared.

```
1   double sumv = double[omp_num_threads()];
2 #pragma omp parallel shared(A,sumv,sum) private(i)
3 {
4   int t = omp_get_thread();
5   sumv[t] = 0;
6 #pragma omp for
7 {
8   for(i = 0; i < N;i++)
9     sumv[t] += A[i];
10 }
11 #pragma omp master do
12 {
13   for(i = 0; i < omp_num_threads(); i++)
14     sum += sumv[i];
15 }
16 }
```

Since the array sumv is very small, chances are that it will only be a single cache line, or at least its elements will share a number of cache lines. When the processors try to update their part of sumv, they will in fact be fighting over the same cache line and this will give very slow performance. The result should be correct, unlike the situation with race condition, but performance will suffer.

Below is a better version, where sumv is only accessed once by each thread and not at every iteration as the version above.

Listing 3.13: Simple sum with OpenMP and no false shared

```
1   double sumv = double[omp_num_threads()];
2 #pragma omp parallel shared(A,sum,sumv) private(i,tsum)
3 {
4   converted to parallel
5   double tsum = 0;
6 #pragma omp for
7 {
8   for(i = 0; i < N;i++)
9     tsum += A[i];
```

```
10 }
11   sumv[omp_get_thread()] = tsum;
12 #pragma omp master do
13 {
14   for(i = 0; i < omp_num_threads(); i++)
15     sum += sumv[i];
16 }
17 }
```

Avoiding false shared memory is very important when programming for systems with cache coherency mechanisms. Had the vector `A` or another large vector been updated during the loop (Listing 3.13 line 8), the vector would not be false shared because the processors do not work extensively on the same cache line because the large vector spans numerous cache lines. False shared occurs when a small shared vector is updated repeatedly by several processors.

3.6.7 Communication - Reuse or Recalculate

It is important to keep in mind that communication between the threads is often the main bottleneck in parallel programs. Even in shared memory systems, communication requires synchronization and distribution through a low bandwidth bus, compared to the local cache bus.

When converting code to run in parallel it might seem beneficial to share as much of the processed values as possible, so no processor does redundant work, but this is no recipe for optimal speedup. In many cases it can be more productive to let the processors do some redundant work if they already have the required knowledge in cache to do so. The question of whether to reuse or recalculate is important to ask when trying to convert programs to multi-thread, because much of the performance can be lost in needless communication between threads.

3.6.8 Data Dependency and Blocking

Some algorithms rely on other values being calculated before others and in these cases, parallelization is slightly more troublesome because the work may not be easily shared amongst the threads. These dependencies and how they are solved are very closely connected to the particular algorithms, so even though it would be very convenient to have a generalized solution, this is virtually impossible.

One way of solving dependencies in problems where neighboring elements are depending on each other (Gauss Seidel for instance), is to do what is know as

partial blocking. What this means is that the problem is partitioned into smaller parts with a well defined dependency scheme in respect to the other blocks, so the blocks which have their dependency sorted can be calculated first. This works well for iterative algorithms because the dependency issues can be sorted out by working in different iteration steps and thus dependency dead locks can be avoided. It adds some overhead to the implementation because a carefully planned message system is needed to signal which blocks have been calculated.

3.7 Performance Library

Performance tuning towards a particular platform can be very interesting and rewarding in terms of performance gain. Unfortunately, all the efforts that go into tuning a piece of code towards a specific chip and memory arrangement might not provide the optimal solution for a slightly different system. For instance a smaller L1 or L2 cache will result in a significantly decreased performance if the algorithm was tuned tightly using blocking towards the larger cache.

The solution to this problem is to have implementations for each platform, but doing this for every piece of code is time consuming and every time a new processor is released, the code needs to be tuned for that particular chip in order to take full advantage of the new features.

Fortunately, this problem of tuning scientific applications is well understood and the solution is simple - highly tuned basic mathematical operations are available through performance libraries.

3.7.1 BLAS and LAPACK

Two very popular standards of performance library interface are the Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra PACKage (LAPACK). Both interfaces are hosted on netlib.org, which is a repository for scientific computing tools and libraries. The interfaces were originally only for Fortran, but with the interfaces called CBLAS and CLAPACK, the methods have been easier to access through C.

The idea behind BLAS and LAPACK is to define a standard interface for linear algebra methods. The actual high performance implementations can be made by chip vendors, scientists, compiler vendors or basically anyone interested in doing so. The common interface makes the code portable to other platforms, as

long as the required BLAS and LAPACK routines are available.

3.7.2 ATLAS

Automatically Tuned Linear Algebra Software (ATLAS) is an Open Source C implementation of CBLAS and a key features of LAPACK. This is as close to a portable performance library as you can possible get. The idea is to have multiple implementations of every method and at compile time, the better performing implementations are selected for the library. This means that the package must be compiled on the system in order to provide optimal performance and hence, prepackaging is essentially impossible.

3.7.3 Sun Performance Library

Sun has a history of being a high performance platform for scientific computing, so naturally a very complete performance library is available. The implementation is primarily written in Fortran and only very few of the most common methods are available through the CBLAS interface. All methods can be accessed through C method calls, but the underlying data structure is column major, since the implementations are made in Fortran. This is important to remember when using the methods. One need not worry when the vector based methods are used since these are mapped identically in Fortran and C.

When using Fortran functions on matrices, the matrices must be transposed explicitly or through the transpose arguments in the function call. Some problems are of a nature where it does not matter if all the matrices are transposed. In that case the function can be used directly. Transposing matrices is normally a time consuming operation because it is very difficult to get very good L1 cache reuse, so if it can be avoided it will benefit the performance.

3.8 Profiling Tools

In the process of performance tuning on an application or in a small piece of code, tools to measure performance can be very helpful. A very capable tool for this is Suns Performance Analyzer, which is a part of the SunStudio programming environment.

3.8.1 Analyzer

A very valuable tool used when optimizing the code is a program called Analyzer, which is a part of SunStudio. If the binary is run through this program, practically every single line of code can be examined in terms of how much time was spent on that particular line. If hardware counters are enabled, the program can obtain information directly from the CPU concerning things such as instructions executed, cache misses and other highly relevant numbers. These hardware counts are also associated directly with the line of code in which they occurred.

The tool can also be used in Linux, but in order to get hardware counters, the kernel must be patched with a patch called `perfctr`. The patching process is very straight forward for people accustomed to compiling kernels, so this is a very easy way to achieve a really capable high performance analyzing tool in Linux.

The tool has been used extensively during development to ensure that the correct decisions were made in terms of performance. What is truly advantageous of the program is that it is able to quickly give an overview of where in the code, most of the time is spent. For performance tuning it is essential to know where it is beneficial to optimize to avoid spending excessive amounts of time on optimizing code that rarely gets executed or does not significantly limit the performance of the program.

3.8.2 Debuggers

C is a very strict language in the way that it creates code that does what you tell it to do. It does not, however, perform many checks if the things you instruct it to do is actually legal at runtime. Reading or writing to memory locations not allocated by the program is not checked, but the operating system will kill the program if such an operation is performed. This is very different to languages like Java, where such errors can occur without terminating the program, if the error is caught.

The simplicity of C does however pose a problem. If mistakes are made somewhere in a complicated code, the only message you get when the program dies is “segmentation fault”, meaning that the program attempted to do something outside of its segment. This does not tell you much about what exactly went wrong or even where the error occurred.

To get some more meaningful error reports, the failing program can be called through a debugger. The debugger will tell you what line of the code caused the error and what type of error occurred. This is a tremendous help since the error is usually obvious when the faulty line has been identified. An alternative to a debugger is to insert print statements in the code before and after the place you think produces the error and then step by step move the print statements closer to the faulty line. This is tedious work and requires many recompilations.

If the problem is hard to spot, the debugger can be used to find the faulty line of code and then some print statements can be inserted at that point in order to see what values are causing the problem. The problem is typically related to indexing beyond the allocated array.

During development of this program, a debugger called "Valgrind" was used extensively. It is a simple Open Source program and it offers good error messages and does not require the program to be recompiled with special debug features beyond the normal "-g" option. When run through Valgrind, the program runs much slower, which sometimes makes it practically impossible to use, but it reports memory leaks in a very clear way, so it is really good for tracking down those problems.

On Sun Solaris, a debugger called dbx can be used. It offers some of the same features, but unlike Valgrind it is a more advanced debugger with a command line interface.

Chapter 4

Program Design Requirements

4.1 Introduction

The foundation for this program came from a Matlab program written as part of a master thesis by Møller [9], in which the general algorithms were implemented and tested. Matlab is a very powerful tool for developing algorithms and prototypes, but when it comes to computing performance, but the fact that it is an interpreting language and most matrix operations result in a new assignment makes it unsuitable for performance-critical operations.

The work involved in the process of transforming a Matlab script to a native C program is highly dependent on the how well the data and algorithm flow of the Matlab implementation fit into a native C implementation. The benefit of a native C implementation is mainly speed, so it is important that the implementation is structured in such a way that the performance is not compromised. Matlab is a language much closer to math, which does not necessarily mean good performance, so programs written in Matlab typically need a complete rewrite to benefit from the performance.

During this thesis, the program was first translated line by line, and the program

worked, but it was not flexible enough to be combined with WPPT and the need for expensive sorting and indexing forced the need to completely rewrite the program. Due to the extensiveness of this report, the discarded implementation will not be presented.

This chapter will describe the general requirementes of the program. The actual implementation will be described in Chapter 5 and the source code is available at http://viller.eu/pep/.

4.2 General Program Specification

In this section the general specifications will be provided. This includes the basic functionality and some of the key design requirements.

The overall purpose of this program is to efficiently do adaptive quantile regression on a data set or stream and thus provide a predictor to use with the obtained or predicted explanatory variables of the future. Quantile regression is basically an optimization problem, so many different optimization techniques can be used. The two most popular methods for quantile regression, simplex and interior point method, have been implemented in this program.

The following points will give a more detailed overview of the main functional purposes of this program.

4.2.1 Interface

A sketch of the interface can be seen in Figure 4.1, where the three most frequently used function calls are shown. Elements which are not part of the online updates are not shown, these include configuration, intialization and load/save functionality. The shown "get prediction" method is actually several methods which take different kinds of arguments, but due to the lack of overloading, they must be seperate function calls in the C code.

Each of the functions shown in Figure 4.1 should be able to be called independently of each other, with the exception of predictions only being available when the predictor has been computed through `update beta`.

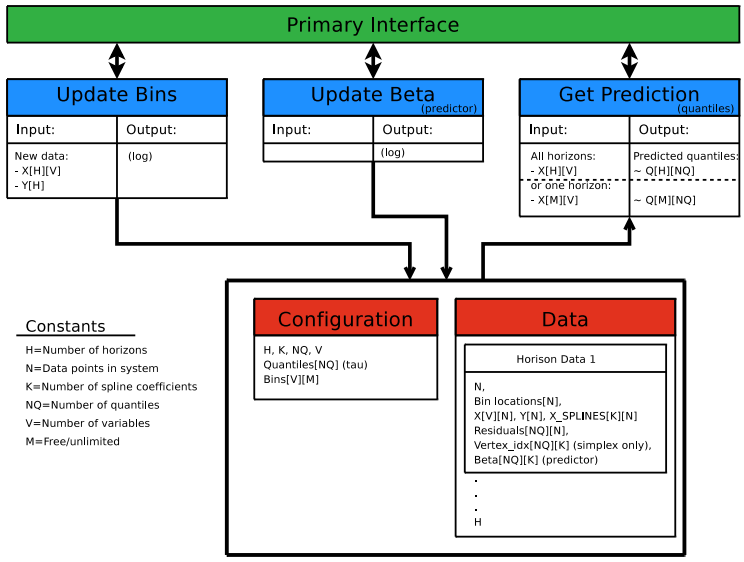

Figure 4.1: Diagram of the interface to the three main function calls and the data and configuration they share. The `log` is currently only a framework, but can be used to pass information from the algorithm back to the host application.

4.2.1.1 Interface to WPPT

The interface has been designed with integration into WPPT in mind. WPPT works primarily in an *online environment* which means that the predictions are calculated continuously. It is important that this program is suited for this, meaning that data is not allowed to accumulate in the data structure, because the program would then run out of memory after a while.

The modules hosted by WPPT are *event driven* which means that the modules are executed when they are needed or when an "event" occurs. The way this works in WPPT is by having a main loop checking for events and in case an event has happened, the module associated with this event will be called.

In order to make the program from this thesis work in this event driven online environment, the program has been split in the function calls shown in Figure 4.1. This was not a design requirement from the beginning of the thesis, but its necessity became apparent when the effort of fitting the module to WPPT was begun.

4.2.2 Modular Design

When the whole data structure is changed and thus much of the program needs to be rewritten, it is a good opportunity to make structural changes to the program as well. Modular programs are very popular because the programmer can focus on one part of the program at a time and this makes the code easier to comprehend. If the modules are also loosely connected, maintenance is further simplified.

Loosely connected modules means that each module is independent. A good, although much more complex, example on the difference between loosely connected and tightly connected are the two operating systems, Windows and UNIX. By nature, UNIX is made up of small independent modules and tools, which are basically connected by the interpretation of the user. Windows on the other hand is a very large and complex system of interconnections and integrated tools. This integration makes things like copy/paste between different programs much easier. This makes the experience nice and intuitive to the user, but it is very difficult and expensive to maintain because a seemingly simple change in one part of the program might have a catastrophical effect on something else.

In the context of this program, a loosely connected modular approach would make it possible to test different combinations of regression algorithms with

only the effort it takes to write the algorithm down in a module or method. It is hard, even in this small program, to make all the modules completely independent. The interior point method and the simplex method are so different in their requirements that if they are used as interchangeable modules, some rules of required side effects must be specified. In this particular case it would be a requirement of both the interior point method and the simplex method to update the vector of residuals.

4.2.3 Adaptive Input Data Handling

The adaptive behavior of the program means that the program should be able to receive an input stream of errors in terms of some explanatory variables, and adjust the prediction model based on these inputs.

In this program, the data structure is what guarantees this adaptive behavior, so unlike other types of adaptive programs, where the model is adjusted adaptively, this program bases it adaptiveness on replacing old data with new. This is essentially a sliding window, but in terms of windmill forecast data, a fixed sliding window would not work because informations of the tails of the distributions would get lost. The data structure ensuring the adaptive behavior is specified in 4.3.

4.2.4 Simplex Algorithm

The theory for quantile regression using the simplex method has been described in 2.3.1 and the details can be found in [6]. The method implemented is based directly on the Matlab implementation [8], but with a changed data handling and program structure. In effect, only the actual simplex algorithm is reused.

The simplex algorithm requires an initial guess to the solution, which needs to be relatively good in order to converge within a reasonable amount of time. When the program is running, this is ensured by remembering the vertex point from the previous solution.

The program must be able to handle not getting the actual vertex associated with the current predictor. This will be the case when a different method (not simplex) has been used to calculate the predictor. In this case, the program must be able to calculate the residuals associated with the predictor and select a non singular vertex matrix based on the least residuals. At this point it might be attempting to only select elements for the vertex with residual equal to zero,

but this should be avoided because there is no guarantee the previous quantile regression algorithm based the optimal solution on vertex points, so the solution might actually not be associated strictly with a vertex.

Selecting non singular vertex sets is a strict requirement of the simplex algorithm, because the vertex matrix is used in a solver. A common way of estimating the rank of a matrix is to do Singular Value Decomposition and compare the norm vectors, which are obtained from the decomposition process. This method is much more computationally expensive than more simple methods such as Gauss Elimination, but it is numerically stable and provides an easier way to set a threshold on close to singular is accepted. SVD must be used to select rank in this program since near singular vertex matrices would be fatal to the algorithm.

4.2.5 Interior Point Method

The program needs to be able to start from scratch, without a previous solution. It has already been mentioned that the simplex method does not work very well when "cold started". interior point method actually works most efficiently when started from scratch.

The interior point method implementation in this program is based on a Matlab implementation posted on Roger Koenker's website [5], which again is actually a translation of the method found in R[1], which is written in Fortran.

The interior point method should have an interface similar to the simplex method, so these two different algorithms can be used (almost) interchangeably.

4.2.6 Configuration

There are a lot of configurations for this type of program. Bins size and range must be definable and where the knots of the splines should be located must also be adjustable. Most of the configurations cannot be changed during program execution because the data structure is sized specifically for these settings. The spline knots have been made adjustable, as long as the number of knots does

[1] R (http://www.r-project.org/)is a program for statistical computing. It is an Open Source implementation of the proprietary statistical software program called S (Bell Laboratories).

not change. Exact bin locations can also be adjusted, but this could lead to problems with the data already stored in bins.

Some of the more advanced features, such as convergence criteria and allowed number of iterations will be made available through runtime configuration when the module is fully integrated in WPPT.

During program testing, driver programs have been implemented which read a list of configurations straight from the beginning of the data set. This is an easy way of providing configuration for simple test cases, but for WPPT, the configuration must be administered by the framework already present in WPPT.

4.2.7 Persistent Storage

Having an option for persistent storage in a program running in an online environment is essential because it makes it possible to restart the computer on which the program is running without losing the valuable trained model.

Since this is mostly important for WPPT, the method for persistent storage consists of serializing the data structure for easy binary storage and deserialization for retrieval. The process of serialization is simply to take all the data in the data structure and put it together in a block of memory. The actual storage will be done by the framework of WPPT.

The load save functionality is also useful when the program is running without WPPT, and with the serialization and deserialization in place, the whole data object can be stored simply by writing the serialized data to a file.

4.2.8 Internals

4.2.8.1 Linear Algebra

C offers little help in regards to matrix and array handling. In C, an array is simply a pointer to a chunk of memory and not much else. By giving the pointer a type, the compiler can help with adjusting the pointer algebra to fit the particular data structure. The size of the array itself is not easily obtainable simply from the pointer, this gets even more complicated when multi dimension arrays are needed.

Since most of the algorithms implemented in this program require matrix and vector structures, a standard set of structures have been made to store these. The goal for these structures is to be simple and clear to use and still offer easy access to the core memory block in a standard way suitable for performance library calls. One benefit from having a standard structure for matrices and vectors is the possibility of creating standard constructors and destructors and thus avoid memory leeks. Another advantage is the simplified function calls, where dimensions do not need to be explicitly passed since it is already a part of the structure.

A range of simple and more advanced matrix operations are needed in order to provide the functionality used in Matlab. These must be implemented as efficiently as possible.

4.2.8.2 Non Convergence Handling

The optimal solution might for some reason not be found during the specified number of iterations. When the solution does not converge before the maximum number of iterations, the solution cannot be trusted and something must be done. When the program is used in a production environment, it is very important to have a clear specification for what is done in this situation.

If only interior point method is used, and it does not converge, the particular solution must be discarded and the previous solution should be selected. The program should also in some way be able to communicate the error to the user or to a log file. A few mistakes should not be a problem, but if the program hardly ever converges, the operator should be informed.

In the case when the program is running using simplex and it does not converge, the situation is a little different. The convergence problem might be related to the previous solution, which for some reason is too far from the new optimal solution, and then it would not make much sense to continue restarting from a bad starting point. The non converged solution should still be discarded and the old solution can be used, but the program should raise an internal flag in order do a cold start on that particular data set.

4.2.8.3 Transform explanatory variables to splines

The program must be able to map the explanatory variables to splines in order to accommodate non-linear dependencies between explanatory variables and

quantile predictions. This has been chosen to be done using both natural splines and periodic bsplines, depending on the nature of the explanatory variables. Wind direction must for instance be mapped with periodic splines, to properly describe the cyclic behavior.

4.2.8.4 Handle Missing Values

The data passed to program are real data from actual measurement gauges and weather forecasts, so sometimes data points might be missing. These missing numbers are handled in WPPT by assigning the value NaN (Not a Number) from the IEEE floating point specification. This is a good way to handle these missing numbers as long as they are not used in calculations, because a real number multiplied by NaN would give NaN and thus the whole data structure might get contaminated.

Seen from a logical perspective, a set of explanatory variables in which some of the values are missing cannot be used to update the predictor and should simply be discarded. The "number" `NaN` offers a clean way of marking these missing values, without risking segmentation faults, but the program must check for them and ignore the set, if it discovers any NaN instances in the incoming data.

4.2.9 Platform and Performance

This program has originally been implemented on Linux (x86) and tested on Solaris (SPARC), but it is a goal for the implementation to be able to run on all platforms where performance libraries are available. Doing quantile regression is a demanding operation in terms of computing power, so the program should be optimized for performance as much as possible.

The code should be easily portable. This means that only standard libraries should be used and although the code is optimized, these optimizations should be done in a way that does not limit the portability. This can be done by using performance libaries.

For easier portability, the program will depend on as few libraries as possible. Most importantly the GNU Scientific Library will not be used, since this is most likely not easily available on all platforms. Fortunately, the library is Open Source, so the needed functionality will be integrated directly in the code base of the program.

4.3 Data Structure

The entire program was first written directly as a line by line translation of the Matlab implementation of adaptive quantile regression, but after this was done, it was clear that the data structure needed to be changed to be flexible enough for WPPT in an online environment.

The data structure is a very important part of the program, so the requirements for the data structure will be presented in detail in this section. The actual implementation will be explained in Section 5.3.1.

4.3.1 Design Objectives

When the data structure is changed, every part of the program which interacts directly with the data will be affected. The purpose of this program is to process a stream of data and calculate the quantiles. When the basic idea of how these data are fed through the program is changed, every part interacting with the data needs to be modified or rewritten.

Since changing the data structure is such a major change to this program, it is important to make sure that it is done correctly. The changes should provide more readable code, performance increase and more flexibility for future extensions. This section will describe the general requirements and things to consider, and finally list the requirements.

4.3.1.1 Continuous Updates

In an online situation, the algorithm needs to work as a black box, where measurements are continuously passed to the program when they become available and the results are returned. This has not been a direct requirement of the previous implementation, because the whole dataset was available at the beginning.

The old data structure could be used by simply expanding the arrays every time a new set of measurements was added, but then the structure would just grow. Deleting elements from this structure can be very difficult because the solution and algorithm data are indirectly linked between the data structures. Even if the correct elements for deletion were found, the representation in computer memory would pose a problem, because one element cannot just be released at

a random place in a memory block. This problem could be solved by copying the structure every time, but this would decrease the performance.

4.3.1.2 Indirect Addressing and Performance

Using indirect addressing can have some advantages in making the code look more like the mathematical expressions, but as described in Section 3.3, it tends to slow down performance significantly. If a program uses the memory randomly or processes the data in the wrong direction according to actual storage in memory, the program will perform poorly.

With indirect addressing it is virtually impossible to predict what data is needed and how to reuse the chunks already in cache close to the CPU. This leads to reduced performance and hence, indirect addressing should be avoided, especially in operations which are performed frequently.

4.3.1.3 Flexible Bin System

The previous implementation had only framework for bin separations in one of the variables. In the process of changing the structure and designing the interface with WPPT, it was decided that it would be a pleasant feature to have bin separations in every variable.

It is the operator or system designer who decides how many bins the data should be separated into and where these should be located. To avoid making the system of bins too complicated, the bin capacity is the same in all bins and is fixed when the system is designed. The flexibility does not suffer from this limitation since the number of bins in a particular segment can simply be expanded, there is no limitation of a uniform distribution of these bins.

4.3.1.4 Algorithm Properties

The simplex method used in this program takes advantage of the fact that the solution of the previous time step is very similar to the solution of the next time step. In order to gain full advantages of this knowledge, internal parameters of the simplex algorithm need to be stored between runs.

These properties pose a potential problem of cluttering the code since they are only used by the simplex method, but for performance reasons they should be

kept as an integrated part of the data structure. If the data is handled in an easily understandable way, adding these algorithm properties should not make the code any harder to maintain.

4.3.1.5 Limited Memory Overhead

It is almost self explanatory that it is a good thing if the program does not use excessive amounts of memory. The allocated memory should not grow over the years that the program is running, and the program should not allocate memory which is not used or keep doublets of the same data. Poor memory management can hurt the performance.

Allocating memory and freeing it again takes a significant amount of computation time and should not be done within critical inner loops or in other parts of the program that are executed repeatedly.

4.3.1.6 Summary and Data Requirements

In summary, there are a number of design objectives which need to be considered in the process of developing a new data structure. Besides general ideas to consider, the actual data also needs to be fitted into the structure.

Below is a list of the requirements and design goals of the data structure.

- Suitable for incremental add and delete.
- Flexible bin assignment in every variable.
- Capabilities for marking vertex corners and other algorithm properties.
- Limited indirect addressing for improved performance.
- Capabilities for modular algorithms.
- Minimal memory overhead and reallocation.

The capabilities for marking vertex corners and other algorithm properties, which is needed for the simplex implementation works against the idea that the program should have independent modular interchangeable quantile regression algorithms. This is, however, a must for the simplex method to work, so compromises must be made.

Data to Store The general requirements are mostly guidelines to how the structure should be crafted. The actual data that the structure must be able to hold is listed below. The dimensions of these vectors and matrices are only for one quantile in one horizon. This will be expanded in the description of the actual implementation.

- Explanatory variables ($\mathbf{X}$) $\mathbb{R}^{N\times V}$
- Variables splines mapped ($\mathbf{X}_{spl}$) $\mathbb{R}^{N\times K}$
- Observed value (Y) $\mathbb{R}^{N}$
- Bin in which elements are located $\mathbb{Z}^{N}$
- Solution ($\hat{\beta}^{<\tau>}$) $\mathbb{R}^{K}$

N is the number of elements in the system (all bins added), V is the number of explanatory variables and K is the number of spline coefficients.

The splines mapped values and bin locations could actually be calculated from $\mathbf{X}$, but these are not changed in a points lifetime, but are referred to often, so it makes sense to keep these.

An interesting observation regarding the sizes of these data is that they can be transposed in a way so that they all, except the solution, have the same length N. The solution only has K times the number of quantiles elements and its nature is so different that it makes no sense to try to match it with the rest of the data. The solution is also updated in the end of the algorithm whereas all the other values are updated as the first thing when a new point is added.

In order to avoid reallocations and to gain better control of memory, it is also sensible to treat the data needed for the simplex regression algorithm together with the data. The interior point method does not take advantage of these data, but it stores the residuals for later use by the simplex method.

- Absolute residuals $\mathbb{R}^{N\times NQ}$
- Sign of residuals $\mathbb{Z}^{N\times NQ}$
- Vertex flag $\mathbb{Z}^{NQ\times K}$

Where NQ is the number of quantiles. The vertex flag is used to mark the points which define the current vertex in the simplex algorithm. These values are closely related to the data points so the residuals and sign can be linked together with the data N data points.

4.4 Validation

The data calculated using this program might potentially be used as a guideline for making cost related decisions, so the functionality must be verified in order to provide the needed confidence in the tool. Some of the key elements that need validation will be described here. To give a better overview, these elements have been split into three categories: basic functionality, quantile regression and prediction quality which will be described in Sections 4.4.1, 4.4.2 and 4.4.3 respectively.

4.4.1 Basic Functionality

Validating the basic functionality requires testing the input output behavior, adaptive data handling and convergence failure strategy. Most importantly, the program should be able to behave as expected with different data sets and settings.

Most of the basic features will be tested automatically during development and when running other tests, so dedicated functional tests will not be presented. Ideally, to properly validate the program, each and every branch in the program would need to be tested, and it would still be possible that with a different compiler or optimization flag, the program might behave slightly differently. There are, however, some basic features which need to be validated, either explicitly or indirectly through other tests.

- Multiple explanatory variables
- Load and save of system data
- Adaptive bin system
- Not-a-number rejection
- Spline mapping

During development, the data set used only had one explanatory variable. This makes everything much easier, but the program was implemented in such a way that more variables could be used. The more variables used, the more complex everything gets, but the program is designed to take a virtually unlimited number of explanatory variables, so if it works for two or three, it will work for ten as well.

The load and save mechanism are mostly of importance to the WPPT interface, but the possibility of being able to save a complete state makes it possible to save data during execution and resume after failures such as power outage or scheduled reboots.

4.4.2 Quantile Regression Implementation

The fact that the quantile regression algorithm is functional and runs without crashes does not guarantee a useful output. Some tests to determine the quality of the quantile regression algorithm and adaptive behavior must be carried out. The two methods, simplex and interior point method also need to be qualitatively benchmarked against each other.

The most important test method which will be used to validate the quantile regression quality is simple point counting of how well the quantile curves describe the data from which they have been found. The method for this will be described in 7.3.1.

4.4.3 Prediction Quality

The third main area is the actual prediction capabilities. In this part of the validation, the system will be used to predict quantiles into the future, and the quality of these predictions will be assessed. Although the algorithm is adaptive, the program might only update its predictor every once in a while and how often this is required will be studied based on actual wind mill data sets.

To make the best prediction possible, it is necessary to have a good understanding of wind mills, how the forecasts change with time and how this affects the power production and prediction uncertainty. The tests in this thesis have been carried out mainly to validate the program, which is the main focus, so the test setups for prediction quality validation might not yield the absolute optimal of what the program is capable of. Further studies could help determine where to place the bins, knots and what explanatory variables best describe the uncertainty in the forecasts.

Chapter 5

Implementation

5.1 Introduction

In this chapter, the implementation of the adaptive quantile regression program will be described. The purpose of this chapter is to give an overview of how the program is implemented. This will not be a complete walk through the code base, but more of a presentation of the decisions that went into the design. The code can be found at http://viller.eu/pep/

As described in the previous chapter, the program has first been implemented by a direct translation and afterwards rewritten to better work with WPPT in an online environment. The presentation here will focus mostly on the current design.

5.1.1 Program or Module

The code written for this thesis will be referred to as both a *module* and a *program*. In terms of WPPT, it is only a module, but during development, driver programs have been made, which essentially makes it a program.

The prefix `mod_rq` has been given to all the external function calls, and in naming of code files, headers and so on. Only the matrix and vector structure have not adopted the naming scheme. The name `mod_rq` is an abbreviation of "module" and "quantile regression". Quantile regression is called "rq" in function calls by Roger Koenker, presumably to make a distinction between this and the more well known QR-factoring.

The header file `mod_rq.h` is the main header file for the project. It contains all the function calls which the host application should use.

5.2 Supporting Building Blocks And Methods

The power of Matlab is the simple ans easy access to different matrix and vector operations. When a program is translated from Matlab to a low level programming language like C, equivalents for all the matrix operations must be found or implemented. There are libraries for C which can be used to facilitate this, but some of these require using a particular matrix and vector structure, which might conflict with the program design. A very complete library which can be used is the GNU Scientific Library (GSL), which is an implementation of a wide range of commonly used operations. The problem with GSL, however, is that it uses its own data structure which can be difficult to work with, particularly when intermixing with high performance libraries.

Some parts of GSL have been taken out and rewritten to become an integral part of this program. The ability to do this is a very valuable feature of OpenSource because it means that an additional library requirement can be avoided.

5.2.1 Matrix and Vector Structure

Since most of the algorithms used for quantile regression are routed in linear algebra, many different matrices and vectors are used in this program. The array structures in C are, however, too simple in the sense that the programmer needs to carry around information concerning lengths, rows and columns of these structures in order to use them as vectors and matrices. Fortunately, C offers the possibility of defining *structures*[1], where these properties of the vector or matrix can be grouped together with the array pointer. It may not sound like a big deal, but being able to easily ask a matrix how many rows it contains makes the code much easier to read and the risk of introducing bugs is reduced.

[1] Very much like objects in modern programming languages.

5.2.1.1 Structure Definitions

The main data types used in this program are double precision floating points (`double`) and integers (`int`). Needless to say, C does not provide polymorphisms either, so the data structures for vector and matrix must be defined for each data type. Below are the matrix and vector structures for `double`.

Listing 5.1: Matrix and Vector structure for `double`

```
1 typedef struct{
2     double** M;
3     int rows;
4     int cols;
5 } dmatrix;
6
7 typedef struct{
8   double* V;
9   int length;
10  int mem;
11 } dvector;
```

The structures are named with the first letter of their type as prefix. This means the matrix and vector structures for integers are called `imatrix` and `ivector`. When reading a large program it is advantageous to be able to easily see what type is being refereed to and to help this, most instances of these structures are prefixed with `dm` if it is a `dmatrix`, `dv` for `dvector` and so on. The same applies to the functions.

5.2.1.2 Construction and Allocation

In the structure definition for the matrix the pointer `M` is in fact a pointer to a pointer. This is how C makes it possible to do indexing without specifically calculating the memory location. This is not carried out automatically, however, the `M` must point to an array of pointers, and these pointers point to one row of the matrix each. Since this pointer to pointer arrangement must be done for each matrix defined, it has been written in a support method for creating matrices.

Listing 5.2: Matrix constructor for `double`

```
1 dmatrix* get_dm_new(int rows,int cols){
2   if(rows > 0 && cols > 0){
3     dmatrix* dm = (dmatrix*)malloc(sizeof(dmatrix));
4
5     dm->M = (double **)malloc(rows*sizeof(double *));
6     dm->M[0] = (double *)malloc(rows*cols*sizeof(double));
7     int i;
```

```
8       for(i = 1; i  < rows;i++)
9         dm->M[i] = dm->M[i-1]+cols;
10
11      dm->rows = rows;
12      dm->cols = cols;
13      return dm;
14    }
15    else
16      return NULL;
17  }
```

As it can be seen from the code listing above, initializing a matrix does require more than one line of code. The allocation of first an array of pointers and then the actual data block is required for mentioned easy indexing. The loop in line 8 sets up the pointers to point to the place in memory where each row starts. Another way to do this would be to explicitly allocate each row and connect it to the pointer for that particular row. This method does however suffer from the possibility of poor performance since the memory block of the matrix might get scattered all over the ram, which would reduce cache efficiency.

Another major issue, and perhaps even more important, is the fact that the performance library interfaces BLAS and LAPACK require the memory representation of a matrix to simply be one large chunk of memory. This would be unlikely if each row was allocated independently.

5.2.1.3 Destruction

Memory allocated in C does not get freed unless this is explicitly done. Methods for releasing the memory when a particular structure is no longer needed are very handy in a large program like this, because as can be seen from Listing 5.2, two chunks of memory are actually allocated for each matrix, so implementing the destructions correctly once makes it easier to avoid memory leaks in the program.

The methods implemented for this are named `destroy_dm`, for `dmatrix` and similarly for the other data structures.

5.2.2 Linear Algebra Routines

In order to do quantile regression, a large range of different linear algebra routines are required. Many of those implemented have been taken directly from

GSL, but some adjustments were required so that they could work well with the data structures used in this thesis.

The methods implemented include Singular Value Decomposition for finding the rank, QR factorization for solving least squares problems and do transformations. In the end of the developing phase, it was decided to name these methods to start with `mod_rq_linalg`, so they would be easier to identify.

In the process of transforming the routines from GSL to become a part of the module, the mathematical operations were translated to performance library routines, where it was possible. A problem with the GSL package is that there are many places where double values are compared with strict equal, which is a numerically unstable approach. Due to rounding errors, these strict comparisons might lead to unwanted results. The typical case is: when is a number zero? In this program, there is a macro definition with a threshold, and this is used in the comparison instead.

5.3 Core Program Implementation Elements

In this section, the implementation main program elements will be described. First the definitions such as data structure and preprocessor macros will be presented. Next, the update algorithm and quantile regression implementations will be described.

The data structure definition together with the updating algorithm (Section 5.3.3) are essentially the backbone of the program. The quantile regression methods can easily be changed if better algorithms are proposed, but the basic data handling is defined by the data structure and the updating algorithm.

5.3.1 Data Structure

The data structure is a very central part of the program. The adaptiveness is implemented by remembering past predictions and observations in this structure. In the process of redesigning the program it was first thought to be effective with an FIFO data structure, but the large amount of memory operations made this quite inefficient compared to the present implementation where new elements are simply replace the leaving element. Below are the structure definitions for the data structure with comments above each line.

Listing 5.3: Data structure `mod_rq.h`.

```
1  // N  = total number of elements
2  // NQ = number of quantiles
3  // K  = number of spline coefficients
4  // V  = number of explanatory variables
5
6  /* mod_rq_horizon_data is a structure containing all the data from one
7     horizon. */
8  typedef struct{
9    // Measured values y [N]
10   dvector* dv_y;
11   // Explanatory variables X [V][N]
12   dmatrix* dm_x;
13   // X mapped to splines [K][N]
14   dmatrix* dm_x_splines;
15   // Bin index [N]
16   ivector* iv_bin;
17   // Vector with relative bin age of each element [N]
18   ivector* iv_bin_age;
19
20 #ifdef ENABLE_BIN_PENALTY
21   // Penalty vector for being too many points in the same segment of
22   // the bin [N]
23   ivector* iv_punishment;
24 #endif
25
26 #ifdef ENABLE_SIMPLEX
27   // Residuals for each quantile. [NQ][N]
28   dmatrix* dm_r_storage;
29   // Residual signs [NQ][N]
30   imatrix* im_sign;
31   // Matrix [quantiles][K] with the elements that define the vertex
32   imatrix* im_vertex_flag;
33 #endif
34
35   // [1..n_in_bin] containing the number of element in each bin.
36   ivector* iv_bin_fill;
37
38   // The actual result for each quantile. [NQ][K]
39   dmatrix* dm_beta;
40   // Initialization flag
41   int init_phase;
42   // Occupied space (N when full)
43   int length;
44 }mod_rq_horizon_data;
45
46
47
48 /* The data for a given state. Simply an array of
49    mod_rq_horizon_data.*/
50 typedef struct{
51   mod_rq_horizon_data** horizon;
52 }mod_rq_state_data;
```

The first structure defines all the data which are needed for one horizon and the `state_data` simply holds an array of `horizon_data` structures. The orientation of all the matrices with N elements in the row direction is a leftover from the FIFO structure, where this was most efficient. For some parts of the algorithms a different orientation would be preferred, but this is not generally so.

Some of the data structure is left optional with two preprocessor statements. When the simplex method is not used, some memory can be saved by not using these structures. Similar control statements are obviously placed in the appropriate place in other parts of the code.

5.3.2 Macro Definitions

Some settings are not provided through runtime configuration, but must be set before compiling in the header file `mod_rq.h`. Below is the code listing of the current definitions.

Listing 5.4: Macro definitions `mod_rq.h`.

```
1 // Iteration constants
2 #define MAX_SIMPLEX_ITERATIONS 5000
3 #define MAX_INTERIOR_ITERATIONS 1000
4
5 // Algorithm specific zero threshold for simplex and interior point
6 // methods
7 #define ZERO_THRESHOLD 1e-12
8
9 // Numbers below DOUBLE_ZERO are considered equal to zero
10 #define DOUBLE_ZERO 1e-30
11
12 // When a regular number is needed for infinity, this is used
13 #define DOUBLE_INF 1e30
14
15 // Performance library definitions
16 #define PERF_LIB_ATLAS 1
17 #define PERF_LIB_SUN 2
18
19 // Penalty algorithm settings
20 #define BIN_POINT_PUNISH_SEGMENTS 1000
21 #define BIN_POINT_PUNISH_RELAXER 1
22
23 // Comparison macros
24 #define MOD_RQ_ABS(a)    ((a<0)? -a : a)
25 #define MOD_RQ_MIN(a,b)  ((b<a)?  b : a)
26 #define MOD_RQ_MAX(a,b)  ((b>a)?  b : a)
```

Most of the settings except for the penalty settings are quite self explanatory. The maximum iterations will most likely be moved to runtime configuration

later, since the number of iterations required depends on how large the problem is.

The thresholds for zero are convenient macros since they offer the possibility of changing the thresholds in one place and not in all the comparisons where they are used.

During development, two performance libraries were used. ATLAS was used on the Linux test machine and Suns performance library was used on the UltraSPARC Solaris machines at DTU. These macros are basically just reminders of what numbers defined in `USE_PERF_LIB` enables which performance library.

The penalty system is controlled with the two definitions `BIN_POINT_PUNISH_SEGMENTS` and `BIN_POINT_PUNISH_RELAXER`. The segments mean how many segments each bin must be split into for the penalty comparison, and the relaxer is a constant which controls how high the penalty should be, compared to the age of an element. The penalty is divided by the relaxer, so this number should be 1 for high penalty and higher if the penalty should be reduced.

Other settings which can be controlled through macros include enabling the simplex algorithm, selecting performance library, debugging information, penalty algorithm and adjustable knots. These are conveniently controlled from the `Makefile` or as it was done in this thesis from a file called `makeplatform`, which was included in the `Makefile` to set platform specific settings.

5.3.3 Update Algorithm

The adaptive algorithm in this program is based on a sliding window technique, but with some modifications. A simple fixed sliding window would not be very good for wind power prediction applications because the tails of the distribution would be very poorly described, since the majority of predictions are in a small segment.

To provide better descriptions of the tails (less frequently occurring events) of the distribution, the updating system has been extended with multiple sliding windows, called bins, where the incoming explanatory variables only push out elements of the same bin as they belong to. The procedure for selecting the bin for the incoming data point can be seen in the code listing below.

Listing 5.5: Procedure for selecting bin on incoming data point.

```
1  // Find bin
```

```
2   char mybin[V];
3   for(i = 0; i < V; i++){
4     mybin[i] = 0;
5     for(j = 0; j < conf->dv_bins[i]->length-1;j++){
6       if(new_data->dm_x->M[h][i] >= conf->dv_bins[i]->V[ j ] &&
7          new_data->dm_x->M[h][i]  < conf->dv_bins[i]->V[j+1]){
8         mybin[i] = j;
9       }
10    }
11  }
12  int bin_idx = get_bin_index(conf,mybin);
```

From the code in Listing 5.5 it can be seen how the correct bin in each explanatory variable `V` is found for the incoming value data point. The indexer `h` is a loop counter for the horizons. The bins must be configured in such a way that all possible values of the explanatory variables are contained within the range. After the bin location for each explanatory variable has been established, the bin identifier is computed in `get_bin_index`. This identifier is a very simple mapping of the array `mybin` to an integer, which makes it possible to uniquely index all the combinations of bins of each explanatory variable.

When the *entering bin* has been established, it will be checked if this bin has already been filled. If the bin is not full, the entering element will simply be placed in the first available spot in the data structure. The data structure does not enforce any restrictions on where elements are placed according to their bin relation, the bin location is only defined by the bin indexer assigned to each element.

If the bin is full, the entering element will replace the oldest element in the entering bin. The relative age of each element is stored in a vector so the oldest value can be found. While searching for the oldest value, the other elements in the particular bin get their age incremented.

5.3.3.1 Recurrence Penalty

At a late state in the development, an additional selection scheme was added to the bin selection due to problems with too many zeros. The penalty algorithm is very much like a bin structure within the bin system. Each bin is split into a number[2] of smaller segments. The segment into which the entering elements fits is then checked to see how many other elements are in the same segment. A penalty proportional to the number of elements in that particular segment is

[2]The number is defined by the preprocessor definition `"#define BIN_POINT_PUNISH_SEGMENTS 1000"`.

then added to the elements in question.

When the decision of which element should leave the bin has been made, the penalty is added to the age, so that the leaving element is the one with the highest age and penalty. The importance of the penalty compared to the age can be decreased by increasing the value of the relaxer (`#define BIN_POINT_PUNISH_RELAXER`) control statement.

This algorithm requires an additional penalty vector with as many elements as the total capacity of the active data. To avoid a massive performance impact of this algorithm, the complete penalty vector is never updated, only elements matching the incoming data point are penalized.

5.3.3.2 Not a Number (NaN) Handling

It is important for the quantile regression algorithm that undefined numbers or NaNs, as they are called in C programming, do not enter the bins. If they do so, the results would be undefined for as long as they stay in the system.

The way NaNs should be handled is to simply throw them out before they enter the structure. In this program this is done by checking all the new data points for NaNs in y and explanatory variables with the C function `isnan()`. If a NaN is observed in a data point, it will be marked and the data point will not enter the data structure.

The implementation for checking for NaNs is very simple, but it is essential that these checks are carried out in order to obtain correct results.

5.3.4 Interior Point Method

As mentioned in the design chapter, the interior point method implementation is based directly on a Matlab implementation by Koenker [5]. It uses the principle of reducing the gap between the dual and primal optimization problem.

One limitation of the interior point method is that it is difficult to reuse knowledge of previous solutions. It is not impossible (Gondzio [1]) to warm-start the interior point method, but the steps to ensure that the method gets on track are rather complicated. One problem is that the step length decreases as the solution gets closer to the optimal solution, so large changes in a few constraints is made impossible. This situations is also known as blocking.

It was attempted to warm start the interior point method for the program in this thesis, but in the end the idea was discarded because it would make the implementation too complex and less reliable.

5.3.5 Simplex Method

The simplex implementation used in this program was originally a direct translation of the Matlab implementation, but it has been changed very much to improve compatibility with the data structure and in order to enhance performance.

5.3.5.1 Extracting a Previous Solution

An important support function for the simplex method is the solution extraction function, which finds a near optimal vertex from a previous solution without a vertex. The interior point method does not compute the vertex, so if the simplex method needs to carry on after the initial solution has been found with the interior point method, the vertex corresponding to the solution must be found.

5.3.5.2 Vertex Rank

In the simplex method, a selection of which elements the vertex should consist of is done for each iteration. With perfect text book examples, the vertex matrix will usually not become singular, but with large data sets, this is a common problem. A non-singular vertex matrix can be ensured by checking rank before the elements enter the vertex. Many methods have been tried for efficiently estimating the rank. Simple difference tests were quickly discarded, Gauss Elimination was fast but not reliable in the long run, so the rank method of choice is Singular Value Decomposition (SVD).

SVD is numerically stable and can provide a measure for the condition of the matrix rather than only the rank. If an entering element makes the vertex matrix very close to being singular, the element should not enter, because the simplex method relies on the vertex matrix being invertible.

5.3.6 External Interface

The main interface which WPPT and other host applications should use is defined in `mod_rq.h`. Here the function calls will be shown and commented. To avoid using the long function names, they will only be referred to by their suffix.

Listing 5.6: Main function calls.

```
1 int mod_rq_main_update_bins(mod_rq_conf* conf,
2                              mod_rq_state_data* data,
3                              mod_rq_log_data* log,
4                              mod_rq_data_point* new_data);
5
6 int mod_rq_main_update_beta(mod_rq_conf* conf,
7                              mod_rq_state_data* data,
8                              mod_rq_log_data* log);
9
10 dmatrix* mod_rq_main_get_prediction(mod_rq_conf* conf,
11                                     mod_rq_state_data* data,
12                                     mod_rq_log_data* log,
13                                     dmatrix* dm_x);
```

From the top, bins are updated with `update_bins`, which takes the configuration, data structure, log and new data point as arguments. Only the framework has been made for the log, and it has not been decided what informations are vital to get out through that channel. The function `update_bins` must be called for every single data point added. If more data points are bundled, they will have to be added separately. One call to `update_bins` does however update bins in all horizons.

The next method, `update_beta`, is the one that updates the predictor. No additional arguments are needed, since the data should already have been updated by `update_bins`. The selection between simplex and interior point method is done at compile time, but even if simplex is enabled, the initial calculation of the predictor will always be done with the interior point method.

Getting the predicted quantiles can be done through `get_prediction` if a complete horizon is required. The quantiles are returned as a matrix with the number of horizons as rows and the number of quantiles as columns. Other methods for calculating the quantile predictions with other inputs have been made.

Listing 5.7: Configuration generation and destruction.

```
1 mod_rq_conf* mod_rq_init_conf(int horizon, int n_variables,
2                               int n_in_bin, int warmlength,
3                               dvector** dv_bins, dvector** dv_knots,
4                               ivector* iv_spline_type, dvector* dv_quantiles);
5
```

```
6 int mod_rq_destroy_conf(mod_rq_conf* conf);
```

The configuration object can be created with the function `init_conf`, which ensures that everything is set up correctly according to the parameters. In the order of formal parameters, the settings are: number of horizons, number of explanatory variables, number of elements in each bin, number of elements that must be in system before the predictor can be calculated, bin definitions, knot placements, spline type for each explanatory variable and finally the nominal quantiles.

Since the configuration consists of many different data types, the convenient `destroy_conf` destruction call has been written.

Listing 5.8: Data object related methods.

```
1 mod_rq_state_data* mod_rq_init_data(mod_rq_conf* conf);
2
3 int mod_rq_destroy_data(mod_rq_conf* conf, mod_rq_state_data* data);
4
5 char* mod_rq_serialize(mod_rq_conf* conf, mod_rq_state_data* data, size_t *length);
6
7 mod_rq_state_data* mod_rq_deserialize_data(char* buf, size_t length);
```

From the configuration the state data object, which is the data structure that holds everything but the configuration, can be allocated using the `init_data`. As with the configuration object, the state data memory allocations can also be released through a simple call to `destroy_data`.

The functions `serialize` and `deserialize` can convert the data object to a consecutive memory block and back again. The pointer to a `size_t` (unsigned integer), is provided to pass the total length of the memory block back to the caller.

Chapter 6

Performance and Optimization

6.1 Introduction

One of the key objectives of this thesis was to improve the performance of the adaptive quantile regression Matlab implementation [8] by translating the code to C and tuning the program.

During this translation, it became apparent that the basic structure of the implementation done in Matlab did not suit integration with WPPT very well. The program was translated line by line and produced similar results to those of the Matlab implementation, but obviously at a higher speed. The problem, however, was the data structure, which was not sufficiently flexible to run "forever" without running out of memory. Performance wise, the old structure also limited the optimization, since many expensive sorting operations were required and the data structure was not compatible with the performance library routines.

With a complete rewrite of the program it was possible to limit the need for sorting and indirect indexing, as well as enable the possibility of using standard performance library routines.

Using performance library routines often results in significant increases in performance, since the routines are highly optimized. As shown in Chapter 3 about performance, significant improvements can also be made by using simple techniques such as blocking and unrolling, but the price in terms of portability, code complexity and readability is very high. These performance optimization techniques have been used as a foundation for decisions all the way through design and development rather than as a recipe to tune the program afterwards.

In this chapter some results and decisions in terms of performance of the program will be presented. Due to time and space constraints, this chapter will not contain detailed studies of how small changes affect the performance of the entire code base. In general, the best performance has been obtained by using the performance libraries as much as possible.

6.1.1 Test Machine

All tests have been carried out on the same machine. The relevant specifications for this machine are:

- Processor: AMD Athlon(TM) 64 X2 Dual Core Processor 3800+
- L1 cache: 2x64kB (data) + 2x64kB (instructions)
- L2 cache: 2x512kB
- RAM: 2GB DDR400
- OS: GNU/Linux (Gentoo)
- Compiler: GCC 3.4.6
- Performance Library: ATLAS 3.7.11

6.2 Memory Structure Performance

It is interesting to see how fast the program can update its memory structure, since this *must* be run every time new points are added. The first implementation of the program relied on a data structure where all points were added before the program started. The limitations related to this lead to the implementation of a FIFO structure for keeping the data. A FIFO structure is good because it is very easy to implement and the age of each element is an inherent part of

the structure. This simplicity does, however, come at the cost of performance. Pushing the all the data one step in memory for each added point does take a considerable amount of processing time.

A large dataset with 48 horizons with roughly 20000 points in each horizon was fed through the spline generation and FIFO and timed. The result was:

```
real    1m6.794s
user    1m6.592s
sys     0m0.188s
```

These times are not particularly bad. The fact that two years worth of data can be fed into the system in about one minute should not be a problem in most applications. It was, however, decided to try to optimize this by dropping the idea of a FIFO in favor of a random access pattern, where the discarded value simply gets overwritten. This was implemented and the result was:

```
real    0m7.488s
user    0m7.364s
sys     0m0.112s
```

A very dramatic improvement. It is important to have a very efficient data structure for the updates, because no matter how rarely the predictor is updated, all observations must go through the bin system.

One of the consequences of the reliability tests was the addition of a penalty based selection algorithm, which added complexity to the bin update system. Here are the timings for a slightly different setup, but with the same data:

```
Without Penalty:
real    0m14.684s
user    0m14.493s
sys     0m0.180s

With Penalty:
real    0m42.551s
user    0m42.287s
sys     0m0.212s
```

It can be seen that the penalty system adds quite a lot of overhead, but in terms

of how much time it takes to compute the predictor, this added complexity should not be a problem.

6.3 Parallelization

From the beginning of this thesis it was a desire that the program could be implemented in a way that could benefit from shared memory multiprocessor environments. With the use of OpenMP this is indeed possible and the quantile regression problem offers several candidate loops for parallelization. The most obvious ones are horizons and quantiles. If more horizons are being calculated, they can be split between the available processors, and since these operations are very much independent, a very nice speedup should be obtainable. As an alternative candidate, the number of quantiles calculated could also be split amongst the available processors.

The problem with parallelization on this problem is that the program is designed as modules for an *event driven* master application. What this means is that the quantile regression module is only called upon when there are new data to process, and not like in normal scientific computing where the whole data set is available and the algorithm is called and only returns when it is done. The latter situation offers much better conditions for parallel processing since the cost of starting and stopping multiple threads comes with a significant overhead.

If the event driven module offered parallelized function calls, these would have to start and stop the number of threads specified themselves. This overhead would most likely limit the speedup offered by the additional processors significantly. Naturally, this overhead must be compared to the time it takes this particular function call to finish its work. The solution to this problem is to provide the parallelization at a higher level, but this level means the host application or WPPT. In a real life online implementation, WPPT will most likely run multiple instances of the module in order to provide different models, so the whole effort of parallelization might even be diminished by the way the host application works. If the host application was multi threaded, it would provide a much more efficient parallelization than if one of its modules had a multi threaded function call.

Despite the obvious reasons not to do algorithm level parallelization on this module, this has been tested out in practice. A data set with 48 horizons was used on the dual core test machine, so each core would ideally work on only 24 horizons and should provide a speedup of 2.

OpenMP Threads	Time regression ms/point
1	123.0
2	78.6

Table 6.1: Performance timings of a multi threaded implementation of interior point method quantile regression. A data set with 48 horizons, each treated separately, was used. The number of knots was nine and a total of 2000 points in the bins. The timings does not include updating the bins.

The predictor update function call was parallelized using OpenMP and the new GCC 4.2 with OpenMP was used to compile the program. The results can be seen Table 6.1, where a speedup of $\frac{123}{78.6} \approx 1.56$ is observed.

It does not come as a surprise that the speedup was not better; under the given circumstances the results are actually quite good. A normal dual core machine as the one used is not optimized for this type of scientific computing, so it is very likely that having both cores running at full speed on a large data structure creates a bottleneck in the memory hierarchy. The memory bandwidth is not enough for two cores working at full capacity. The results might have been better if the machine was equipped with DDR2 RAM or with a dedicated memory bus for each core.

6.4 Performance Library Wrapper

The main tool for optimization is the performance library. Using methods from a performance library offers performance, which is close to impossible to obtain from manual C code optimizations, since performance library routines depend heavily on low level assembler[1] instructions and very processor-specific optimizations.

Although the performance library through the BLAS and LAPACK interfaces should be a common standard on all platforms, the reality is unfortunately slightly more complex. The different implementations might have slightly different required include files, and particularly when using the C interfaces, which is not the original standard, some tricks may need to be performed in order to achieve the same results on all platforms.

[1] Assembler is a programming language which is very close to machine code - it is basically a bunch of aliases to the machine opcodes.

The best way to deal with this is to make wrappers[2] for the interface calls. Different ways of doing this have been tested out during the thesis, but it can be difficult to find the balance between portability, readability and performance.

All vectors and matrices used in the program share the same structure, which have been defined for the purpose of this program only. Making wrapper calls to work directly on these structures would make the program easier to read, because operations like matrix multiplication could simply be called just that and the wrapper would make sure to unpack the low level memory structures from the arguments to pass to the performance library. However, one major problem with this approach is the fact that C does not support function overloading[3], and thus requires a confusing array of different functions names, which in essence do the same.

The traditional method calls in BLAS and LAPACK offer a great deal of flexibility through their low level memory access and many parameters. This flexibility makes it possible to work only on some types of subsets of data matrices and vectors without having to copy the entire data structure. The price for this flexibility is unfortunately paid for in terms of code readability. Subsetting matrices through the performance library interface requires pointer operations, which might seem strange, but it is very efficient, so the price of lost readability may be well worth it.

Keeping the original performance library was chosen in the end, but not all code has been converted to use the wrappers. Wrapping methods so directly might seem like a waste, but limits the problem with porting the program to other platforms, because with all platform related calls placed under wrappers, only the wrappers need to adapt to the platform. Particularly the needed transposing to work together with Suns Performance Library can be done in the wrapper, thus avoiding having the algorithm code cluttered with different versions for different platforms.

6.5 Interior Point Method Performance

One of the key differences between this implementation and the Matlab implementation [8] is that the interior point method has been included as an inte-

[2] A wrapper is essentially a function which calls another function or *wraps* it. The purpose of wrapping is to either provide a more general function call or to call the particular function in a specific way.

[3] Overloading is when you have multiple functions with the same name, and the function with the correct combination of formal parameters will be executed.

grated part of the module, so no other programs are needed to provide the initial solution for warm starting the simplex method.

Another benefit of this integration is that the interior point method can be used instead of simplex, when this is more convenient. Furthermore, it opens up for a direct comparison of the adaptive quantile regression performance using simplex and interior point method. A similar comparison has been made by Møller [10], but the algorithms were implemented in different languages, so the timings were not directly comparable.

In this section, the performance of interior point versus simplex of the C implementation will be shown but first, for a reference, the interior point method implementation will be compared to quantile regression in R.

6.5.1 Quantile Regression in R

Large efforts have been made to implement efficient methods for quantile regression in R [5], so it is natural to compare the current implementation with the methods offered by R. Unfortunately, due to time constraints, it was not possible to test the Frisch-Newton (also known as "fn") method in R, since it caused an error with the test set[4].

The standard Barrodale and Roberts "br" simplex based algorithm did however work perfectly on the test set. The test set used was very simple, 20000 random numbers as y and natural splines $\mathbf{x}$ with six columns - one was intercept or 1.0. Below is the R session where the quantile regression was tested.

```
> x<-read.table("xdata_spl.dat")
> x<-as.matrix(x)
> y<-read.table("yrand.dat")
> y<-as.vector(y)
> system.time(rq(y[1:20000,] ~ x[1:20000,],0.5))
[1] 2.06666667 0.08333333 2.13333333 0.00000000 0.00000000
```

The last line is a vector with the timing results, where the elements represent: `user cpu, system cpu, elapsed, subproc1 and subproc2`. The elapsed time (2.133s) is the best measure for performance. The test was run four times and the fastest have been kept.

[4]The error might be related to the type of data objects used, but due to lack of time, this could not be investigated further.

A driver program was written to do the same test with the C implementation of the interior point method. The best result of four can be seen below.

```
$ time ./perf_quantreg xdata_spl.dat yrand.dat 20000 6
Seconds IPM 0.586183

real    0m0.697s
user    0m0.676s
sys     0m0.020s
```

The UNIX tool `time` has been used as a reference. The output of the program states how many seconds have been used on the actual interior point method, where reading data from the files have not been included.

These tests show that on this particular problem size, the Barrodale and Roberts implementation in R takes $\frac{2.133}{0.586} \approx 3.6$ times longer to finish than the C implementation of interior point method. It would have been very interesting to see if the same would be true for the R implementation of interior point method. It would most likely perform better on this problem size, since the interior point method is generally better at large problems than simplex based algorithms.

6.5.2 Simplex Versus Interior Point Method

During development, there has been little doubt that the simplex method is much faster than the interior point method, when the methods are used on an adaptive data set where the simplex method can use its previous solution, while interior point method must start from scratch every time. A simple test has been run to see exactly how much difference can be observed.

It can be seen in Table 6.2 that the simplex implementation clearly outperforms the interior point method with a factor of six. This lead in performance can however easily be overcome by the fact that the interior point method does not need continual updates, so less frequent updates are possible. As soon as the predictor is only updated once a day, the interior point method implementation takes the lead by being four times faster than the simplex method. The tests except for "Simplex 2" were carried out with four bins with 500 points in each and nine spline knots.

The test "Simplex 2" in Table 6.2 has been been adjusted to the settings used by Møller [10] where the performance of the Matlab implementation is used.

Method	Time between updates (hours)	Time regression ms/point	Time total ms/point
IPM	1	119.2	119.3
	24 (daily)	4.956	5.048
	168 (weekly)	0.740	0.830
	672 (every 4 weeks)	0.193	0.283
Simplex	1	19.28	19.40
Simplex 2	1	14.14	14.37

Table 6.2: Performance timings of interior point method and simplex on a data set with one explanatory variable, nine spline knots and a total of 2000 points in the bins. The four quantiles 20%,40%,60% and 80% are being calculated for each update.The timings in the last column include the time spent on forming splines and putting data into the data structure. The test "Simplex 2" was carried out with other settings.

The adaptive algorithm with 5 bins, 1200 points in each bin, spline knots at bin boundaries and two quantiles (25% and 75%) are calculated in $0.14s$ per point. The C implementation of the same algorithm does the same in $0.01437s$. The data sets used are not the same, but this should not have a very big influence on the computing time. More importantly however is the fact that the tests have been carried out on two different machines. Since the article is very new, it is unlikely that the performance difference is greater than the factor 10 seen in the timings.

To validate this result, the test should be run again with the same data set and settings in both implementations and on the same processor.

Chapter 7

Tests and Validation

7.1 Introduction

The purpose of this chapter is to show the results of the validation tests described in Section 4.4. Since the module is going to be implemented in a commercial application, it is important to show that the functionality is working properly.

In the first section of this chapter the simple functional tests will be described. After this, the quantile reliability will be assessed, first in terms of how well it describes the data stored within the program. Afterwards the prediction capabilities of the program will be studied.

All the tests except for some of the simple functional tests will be carried out on the data sets obtained from ENFOR A/S with power predictions, measured production and weather forecasts. The main data set will be described in Section 7.3.2.

7.2 Functional Tests

Formal functional tests are an important phase of the development process. Different approaches to how this is supposed to be carried out exists. One idea is to completely write the program and then test it in the end. This is clearly a problematic approach, particularly if the program is large and complex. When everything is only tested in the end, everything might work or it will fail, but the risk of potential time loss, which could have been avoided by stepwise tests, is large.

During the development of this program, each component was tested individually before implementing it to the main program. Most of the complex algorithms implemented were available through Matlab or R, so the tests could be done by comparing output of the new C implementation with the corresponding method in another language.

The beauty of this type of software with many complex algorithms mixed together is that it only works if everything else works. A few minor problems with indexing might not be discovered in the most simple tests, but when a few years of data has passed through the system and the expected results are obtained, the confidence in the program's functioning is very good.

Tests of every single component will not be shown in this report, as this would simply fill up too many pages and probably be of little interest to the reader. In this section, a few tests of the more important basic functionalities which are not obviously tested through the quantile reliability (Section 7.3) and prediction (Section 7.5), will be presented. Finally in Section 7.2.4, key functionalities, which are tested indirectly in the later tests will be discussed.

7.2.1 Save and Load

The functionality for serializing the data structure is essential for integration with WPPT, since the users of WPPT will not be able to restart their computer or recover from hardware failures and still hold on to all the training data stored in the adaptive quantile regression module.

The feature has been tested and used many times. It has been a valuable help for debugging purposes, since saving and restoring the state right before a failure makes it possible to use performance slowing debuggers such as Valgrind without having to wait for the program to reach the state of failure.

A general problem with this load and save functionality is that it uses simple binary storage, so if a state is saved using one version of the program, this file cannot be read back into a new version if the new version includes changes to the data structure.

7.2.2 Spline Generation

The actual spline generation functionality is tested every time a point is added to the system, because all explanatory variables are mapped to spline domain in order to find the non-linear quantile curves with linear quantile regression methods. However, these splines are so fundamental that it needs to be shown here that they work.

The method for generating splines has been taken from R and converted directly to C. The translated implementation will be tested simply by forming a simple set of splines and comparing them to what R produces.

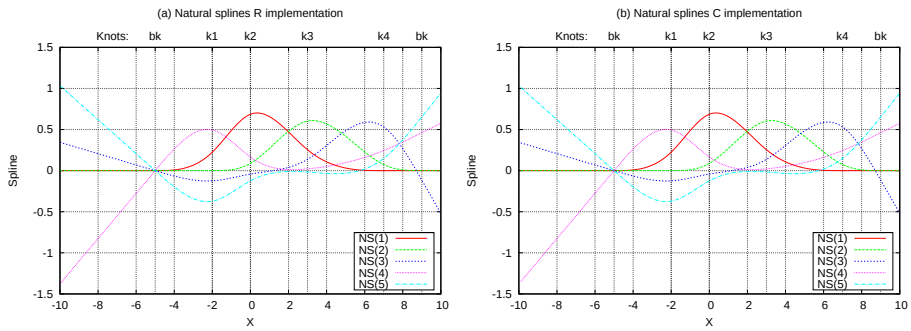

Figure 7.1: A comparison of natural spline generation using R (a) and the C implementation (b). The location of the knots has been shown on the axis just below the title. Boundary knots are marked "bk" and internal knots are marked with "k" and their number. The value of the knots can be read on the x-axis. The two plots look identical.

Figure 7.1 shows a test of the natural splines with knots in $-5, -2, 0, 3, 7, 9$, the end knots are special because they are boundary knots and the natural spline implementation automatically multiplies these knots to get the natural splines. In R, these splines are formed by the following command[1]:

[1] `x` might be formed more elegantly in R, but this works.

```
ns(x=c(-100:100)*0.1,knots=c(-2,0,3,7),Boundary.knots=c(-5,9),intercept=FALSE)
```

The statement `intercept=FALSE` ensures that the spline value is zero at the first boundary knot. The results in Figure 7.1 look exactly as they should and the two implementation appear to be identical.

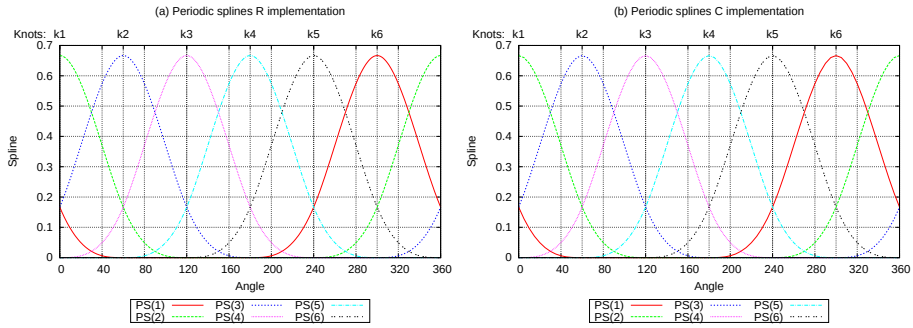

Figure 7.2: A comparison of periodic spline generation using R (a) and the C implementation (b). The 6 knots are marked with "k" and their number. The two plots look identical.

For explanatory variables with a periodic nature, the periodic splines must be used. In Figure 7.2 is a comparison between R and C implementations. This also seems to be correct.

R does not have a default method for calculating periodic splines, so a function `bs.per.ek` written by Henrik Aalborg Nielsen was used. The code can be seen in Appendix A.1. To make periodic splines in R with six knots and a period of 360 degrees, the method can be called like this:

```
bs.per.ek(c(0:359,0),360,3,6)
```

The first argument is the x values to be evaluated, and next the period, degree and number of knots are entered. A degree of 3 corresponds to cubic splines.

The spline implementation works perfectly, which is fundamental for getting anything else to work in this program. It is a very valuable feature to be able to let the module or program calculate the splines by itself.

7.2.3 Removing NaNs

The data received from WPPT contains NaN (Not a Number), when a measured value is missing or for some reason the prediction is invalid. Letting the NaNs into the system is very problematic, since they may contaminate the stored training set for a very long time. The strategy is to simply discard them upon entry, which is done in the `update_bins` (see Section 5.3.3.2) method.

NaN is a special number encoded in the IEEE floating point specifications. In C, the test `isnan(value)` will return true if `value` is in fact not a number. Whenever a NaN enters an equation, the result will also be NaN and the program can be compiled to catch this. Casting a number to an integer will remove the NaN flag, so it is important to use double or floats all the way through, when NaNs must be caught.

The fact that the program correctly discards NaNs, will be shown with a very simple test. A large data set with NaNs occurring in both measured value and explanatory variable, have been used for the test here:

```
functest $ grep NaN klim_complete.dat | wc -l
96429
functest $ ./val_prediction klim_complete.dat 20000 10 | grep "discarded nan"| wc
96429
```

The first step is a simple search with `grep`[2] through the data set for lines which have NaN in them. The number of lines is counted with `wc -l`[3]. In the bin algorithm, a print statement has been inserted, which prints "discarded nan" to the standard output every time a point is thrown away by the NaN checking rule. The occurrence of this is also *grepped* and counted. To save time, the program was run with parameters that effectively disables predictor calculations, by telling the program to only start doing quantile regression at "count" 20000, which is more than enters the system[4].

The numbers are identical, which means the program cached every single line infected with NaNs from the data set.

[2] Grep is a versatile command line tool for searching through files and streams.
[3] The command line utility `wc` counts lines, words and characters in files or streams.
[4] Each "count" consists of 48 horizons, which is why the found number of lines with NaNs are higher than the number of counts.

7.2.4 Indirect Testing

The foundation of the whole system is the data structure. It is the implementation of this data structure which makes it possible to do *adaptive* quantile regression. The workings of the bins have been tested thoroughly throughout the design phase, and the fact that everything works relies heavily on the bin system functioning properly. In Section 7.3, the qualitative contents of the bins are studied and the penalty based selection is added.

The ability to use more than one explanatory variable is tested directly in Section 7.7 in an experiment where the actual power and not just the uncertainty is predicted.

All the linear algebra methods, which include QR factoring, singular value decomposition (SVD), least squares solve and more, are indirectly tested in the sections where the quantile predictor is calculated. SVD is only used for rank tests in the simplex implementation, which will be tested in Section 7.4 and QR is mostly used in the interior point method, which is thoroughly tested. QR factorization is also an important part of the natural spline generation algorithm, so the core functionality has already been tested.

7.3 Reliability of Quantiles

In this section the reliability of the computed quantile curves will be tested. The quantile regression method which will be used in this section is the *interior point method*, because it offers the most flexibility in regards to changing settings at runtime and it calculates the quantiles from scratch every time, so the current solution will be completely independent of the previously computed solution.

The primary focus of the tests in this section are on the updating algorithm and number of spline knots, and how it affects the reliability for the computed quantiles. Before running any tests, the *reliability test method* will be described and the data set used throughout the tests will be presented.

7.3.1 Reliability Test Method

As described in Section 2.2, the quantiles are lines or curves with a fraction τ of the points below the line and $(1 - \tau)$ above. This realization leads to the

most simple validation test imaginable - counting the number of points below the line.

In practice, this is done by first taking the explanatory variables $\mathbf{X}$, mapped into splines $\mathbf{X}_{spl}$ and calculating the residual as

$$r = y - \mathbf{X}_{spl}\beta_{tau}, \tag{7.1}$$

where y is a vector of the measured values at the given explanatory variables $\mathbf{X}$ and β_{tau} is the coefficients in spline space found by the quantile regression.

The number of points below the given quantile curve can then be found by the summation

$$count = \sum_{i=1}^{N} (r_i > 0), \tag{7.2}$$

where the result of $(r_i > 0)$ is a normal boolean expression yielding 1 if true and 0 of false.

Dividing the counted number of points below the quantile curve with the total number of points yields the fraction of points below the curve which should be equal to τ. The "total" number of points in the denominator N_{den} is the number of non-zero elements of r. Hence the reliability for a particular quantile τ can be written as

$$reliability\,(r) = \frac{1}{\sum_{i=1}^{N} [(r_i > 0) + (r_i < 0)]} \sum_{i=1}^{N} (r_i > 0). \tag{7.3}$$

The relative reliability can obviously be found by subtracting the reliability found in (7.3) from the given quantile τ. The absolute reliabilities are, however, easier to plot together for different values of τ, so the relative reliability will not be used in the plots.

7.3.2 Test Data Set

The main data set which has been available for this thesis is an actual set of real recorded data from a windmill park called "Klim" in Denmark, with predictions made by WPPT. The data set consists of 48 complete sets of measured versus predicted power data for close to two years. The combination of a measured value and the predicted power will be referred to as a *point* or an *element*. The data set contains 19000 points in each of the 48 horizon. The time difference

between each horizon and each point is one hour, so many of the measured values will be reused, in fact, each measured value will be used 48 times, one for each horizon. This reuse of measured values is only done for convenience as it makes the data set more easy to read into a program. For online data, some additional system for fetching pairs of matching explanatory variables and measured results must be in place.

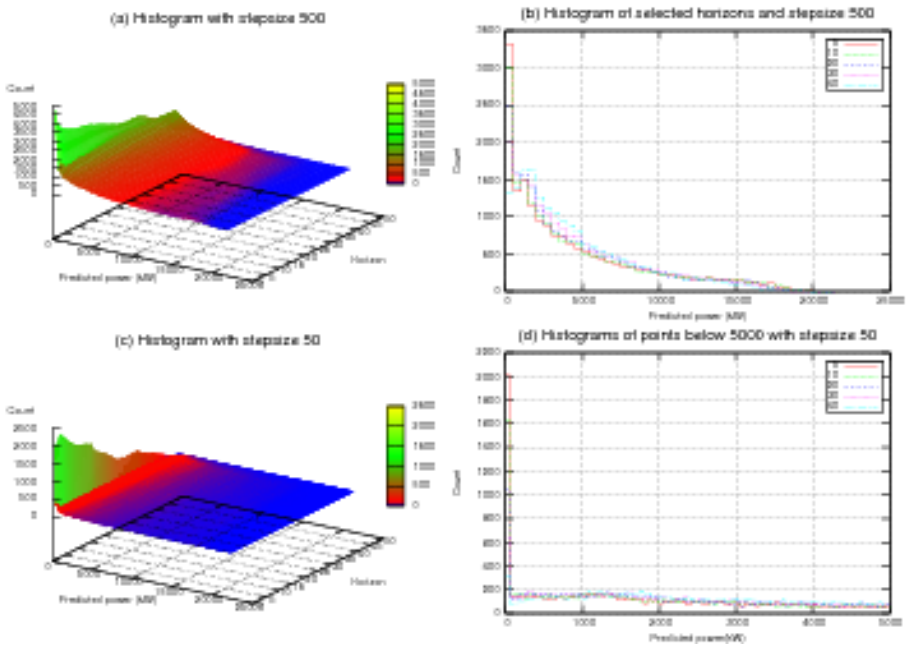

Figure 7.3: Plot (a) and (c) show a 3D histogram of an entire data set. The x axis is the predicted power and the z axis is the count of how many points are within a given segment (the resolutions are 500kW and 100kW for (a) and (c) respectively). The y-axis is the horizon, so for each horizon h, $y = h$ will give a complete histogram in that horizon. Five of these are plotted to the right in (b) and (d). From the 3D plots and 2D plots in particular, it can be seen that the distribution is not very even and plot (d) shows a large peak in the first segment (0-100kW).

Figure 7.3 illustrates different histograms of the data set. The 3D plot (a) and (c) show how the distribution of power predictions that have been made throughout the whole data set. From these plots and (b) in particular, it can be seen that the data are not at all uniformly distributed. Most of the data available are from predictions below 5000kW. On histogram (d), the high resolution reveals a very large peak of predictions close to 0kW. This should not be surprising

due to the nature of the *power curve* described in Section 1.1.1. All these zero predictions do however pose a problem with respect to the quantile regression, which will be observed in the following reliability tests.

Only the tenth horizon will be used in the tests. There is no particular reason for choosing this one over any of the other, but the longer the horizon, the more error in predictions will be expected, so taking the first horizon would not be interesting. The tenth has been chosen because it should offer some prediction error but not too much.

7.3.3 Preliminary Tests

During the development, some interesting problems with the data distribution were discovered. These problems will be illustrated here before trying to address them later.

Throughout the tests, the two bin settings below will be used.

Two Bins

0-8MW	8-21MW

Four Bins

0-3.5MW	3.5-8MW	8-12MW	12-21MW

The total number of points are kept the same at 2000 points, so in the first case with two bins, each of these will contain 1000 points when filled. The four bins must be of size 500.

In both cases, the knots are placed on the edges of the bins, so in the setup with four bins, the five knots are placed at $\{0, 3.5, 8, 12, 21\}$(MW) and the three knots are placed likewise in the case of two bins.

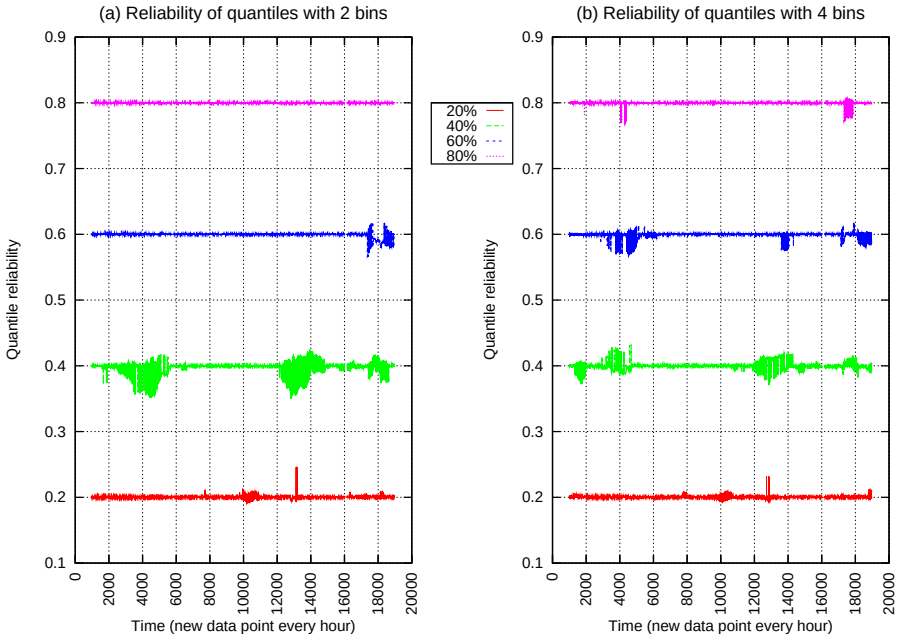

Figure 7.4: The two plots show the reliability of the quantiles over time. The time axis is a simple counter of how many points have been added, each with one hour in between. The reliability is plotted for the case with two and four bins. There is some unwanted noise in both cases, but apart from that the reliability is quite good (close to the wanted quantiles).

7.3.3.1 Reliability With Standard Setup

The plots in Figure 7.4 show the reliability of the four quantiles 20%, 40%, 60% and 80% as a function of points added from the data set. It is clear from both plots that the lines representing the reliability of the quantiles fits very well with what was expected. The lines are very nice and straight except for a few places where the reliability seems to fluctuate badly. In the worst cases, the reliability seems to move just more than 5% away from the particular quantile.

However, the results are not very good when considering the fact that the data on which the reliability are estimated are the exact same as the data used in the regression. The peaks on the plot with 4 bins and more knots seems to be smaller in amplitude, so it might help to adjust the number of knots. The two bin setup also does not ensure as high a degree of information about rare cases, because with only two bins, the typical forecasts will dominate more than with four bins, so in the following, four bins will be the primary test setup.

7.3.3.2 Changing The Number of Knots

In Figure 7.5, the "four bin"(page 107) locations have been used, just as in Figure 7.4(b), but the numbers of knots have been changed in order to be able to study what effect, the number and placement of knots have on the data.

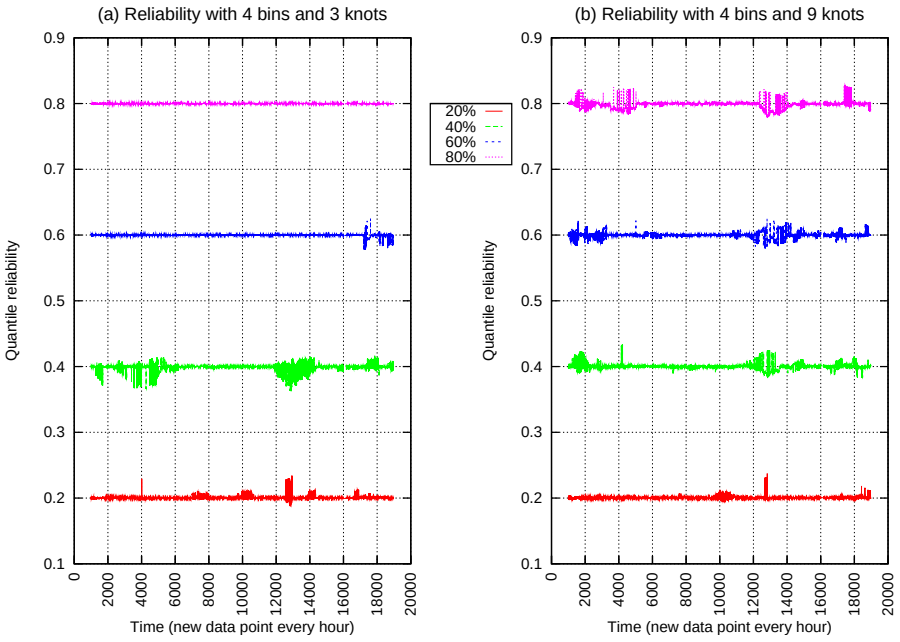

Figure 7.5: Reliability with four bins. In plot (a), the knots are placed at 0,8MW, 21MW as in Figure 7.4 (a). In plot (b), extra knots are placed in the midpoint of each bin, so the total knot sequence is $\{0, 1.75, 3.5, 5.75, 8, 10, 12, 17, 21\}$(MW). The plot does not seem to have changed much since the standard setup, if anything, the results are slightly worse.

The plots (a) and (b) in Figure 7.5 look almost identical to (a) and (b) in Figure 7.4. The new (a) with five bins and the same knots as Figure 7.4(a) seems to have the unwanted peaks placed the at the same location time wise.

The results shown in Figure 7.4 and Figure 7.5 might be reasonable or even quite good, if they were describing the reliability of *forecasted* quantiles in the future, but when the training set and test set are the same, these results are clearly unacceptable. This problem requires some further investigations.

7.3.4 Reliability Problem Analysis

The problem with the reliability of the quantiles found through regression might appear to simply be a problem with the interior point algorithm which has been used for these tests, but as will be shown in this section, the problem is with the data itself. Before going into details on what causes the problems, a variation of the reliability plot needs to be explained.

7.3.4.1 Local Bin Reliability

When the program is running, the data set is fed into an algorithm, which places the incoming observation points into bins, based on the explanatory variables. In this case, only one explanatory variable is used, which is the predicted power. The reliability plots shown above have only shown the average reliability of the entire training data stored all the bins defined in the program. As shown in the histogram in Figure 7.3, the data is not distributed evenly over the range of the predicted power, so perhaps, the cause of the problems with reliability must be sought in more refined intervals.

One way of getting a better picture of how the reliability problems relate to the explanatory variable, predicted power, the *local bin reliability* can be plotted separately. The calculations are still exactly the same as in (7.3), but instead of looking at the entire data as a whole, each bin is evaluated separately.

A plot of local bin reliability can be seen in Figure 7.6. This shows the local reliability of the exact same two runs shown in Figure 7.5 (four bins with 3 and 9 knots). These plots reveal some alarmingly unreliable quantiles locally, which was not visible in the average reliability measure.

Besides being quite bad, there is an interesting thing to observe in the plots of local reliability compared to the average before and that is the fact that the increased number of splines helps on the local reliability even though it cannot be seen on the average in Figure 7.5. This brings hope to the perspectives of the algorithm used to this kind of data, because a better placement of the splines might improve the systems capabilities of coping with problematic data sets. This will be studied further in Section 7.3.5.

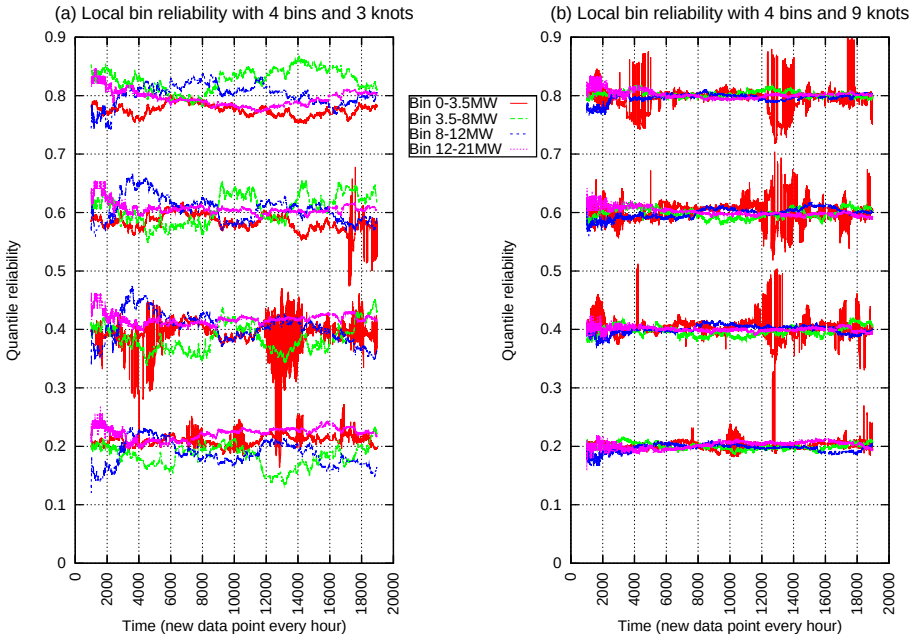

Figure 7.6: Local bin reliability with 4 bins same knot settings as used in Figure 7.5.

7.3.4.2 Bin Contents - Active Data

The adaptive nature of this program is, as previously described, implemented using a bin system in the explanatory variables, always throwing out the oldest member of a given bin. This is essentially a sliding window, but extended with several windows (the bins) sliding over their own segment of the explanatory variable.

The *active data* in the system are the data points currently located in bins. Old data points are simply thrown away and does never reenter the system.

In Figure 7.3 on page 106, histograms of the whole data set were shown, but when the program is running, only the part of this data currently located in bins will be used for the quantile regression. The number of active data points is set by the constant size of the bins times the number of them. As mentioned in the figure text of this histogram on page 106, the data points are far from evenly distributed over the range of the predicted power, and this might be the root cause of the reliability problems.

As it can be seen in Figure 7.7, the distribution of the active data changes over time, but the distribution of the active data is not much different from the total data set. The main reason to have a bin system instead of simple sliding window is to keep information of as much of the prediction space as possible. Ideally, the active data should represent as evenly as possible, the span of the explanatory variables, but this is clearly not the case here.

The peaks in the 0-100kW range are clearly related to the problem with reliability and in particular to the reliability in the first bin (0-3.5MW). If most of the observations are concentrated in a small area, these point will *force* the quantile curves through them and thus make it harder for the curve to describe the remaining points in the set. What makes this even more problematic is the fact that most of these numbers in the 0-100kW are in fact zero. Counting the number of zeros in the data set reveals that as much as 7.8% of the power predictions are zero.

Changing the updating algorithm so it will cope with all these zeros will add some complexity to the updating algorithm, but if the system is going to be used for critical applications, something must be done. The two obvious areas to seek improvements are spline locations and a better updating algorithm. With improved knot placement, the regression should be better at handling many identical points, so this will be tested next.

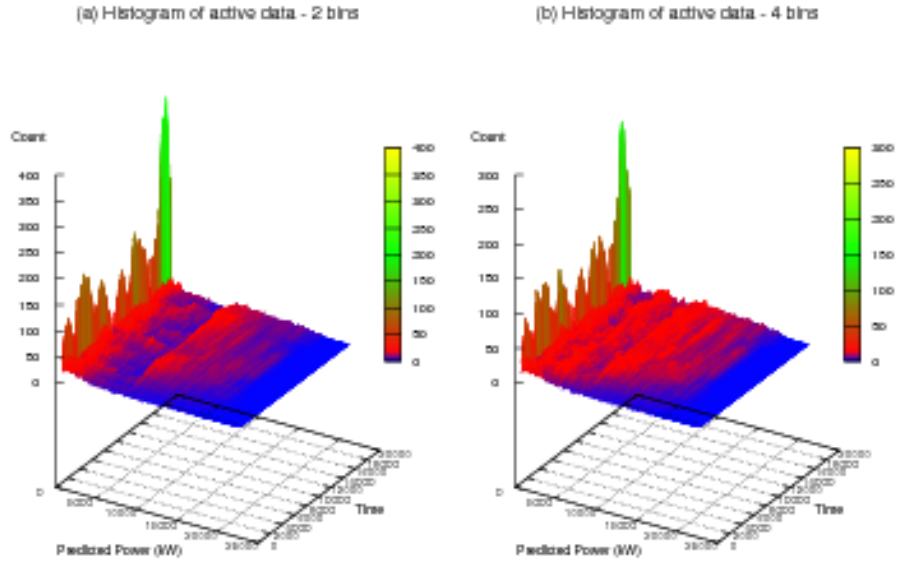

Figure 7.7: The two 3D plots show the histogram of the active data over time. The resolution of both histogram is 100kW The left plot (a) is generated with two bins, 0-4MW and 4-21MW and plot (b) is from the 5 bin run. In both plots there are very large peaks occurring in the 0-100kW range. The distribution of points seems to be slightly better in the 5 bin test, which is a direct consequence of the additional number of smaller bins. In both plots there is a very large peak at around 18500h, which must be due to a large number of similar values entering the bin. In the 5 bin case, the 0-1.5MW bin, with a total size of 300 values is almost completely full with points in the 0-100kW range, which leaves very few points to describe the rest of the range.

7.3.5 Adjustable Knots

The local bin reliability did not offer a solution on how to obtain better reliability, on the contrary, the plots only revealed a much bigger problem. From the histograms of the active data (Figure 7.7), the problem seems to be caused by too many points in a small segment of the bins.

The tests with knots in the middle of each bin seemed to significantly improve the local reliability, but there is no guarantee, that the midpoint of each bin is a good location for the knots. This is especially true in the bin containing all the zeros, where a knot placement closer to the majority of the points would make more sense.

A possible method for allowing the system to provide the most resolution, or flexibility of the splines, would be to place the knots adaptively based on some properties of the data. An obvious choice is to place knots in the median of each bin, because then knots will be placed closer in the areas of the data, where most points reside and thus letting the quantile regression curves get closer to the desired shape.

An algorithm for selecting the median, placing the knots and recalculating the splines of the current data set has been implemented and the results of the run with the same settings as Figure 7.6(b) can be seen in Figure 7.8. Besides the problem with the local reliability of the first bin (0-3.5MW), the results have improved noticeable.

An important feature of the implementation is a forced gap between the middle knot the edge, this gap has been set to 1% and it ensures that if the median of the first bin is in zero, the middle knot will not place it self in zero and thus removing some of the flexibility needed to fit the quantile curve to the rest of the data in this bin[5].

Although the adjustable knots did help slightly to the reliability, the reliability is still not impressive and offers little confidence, particularly in the area contained in the first bin. Since these predictions are also the most common, the system will not be very helpful for making critical decisions.

[5] Before this gap was implemented, the tests showed complete garbage around the 18500h area, which corresponds exactly to the peak in the histogram (Figure 7.7), where many zeros enter the system and shifts the balance of the bin.

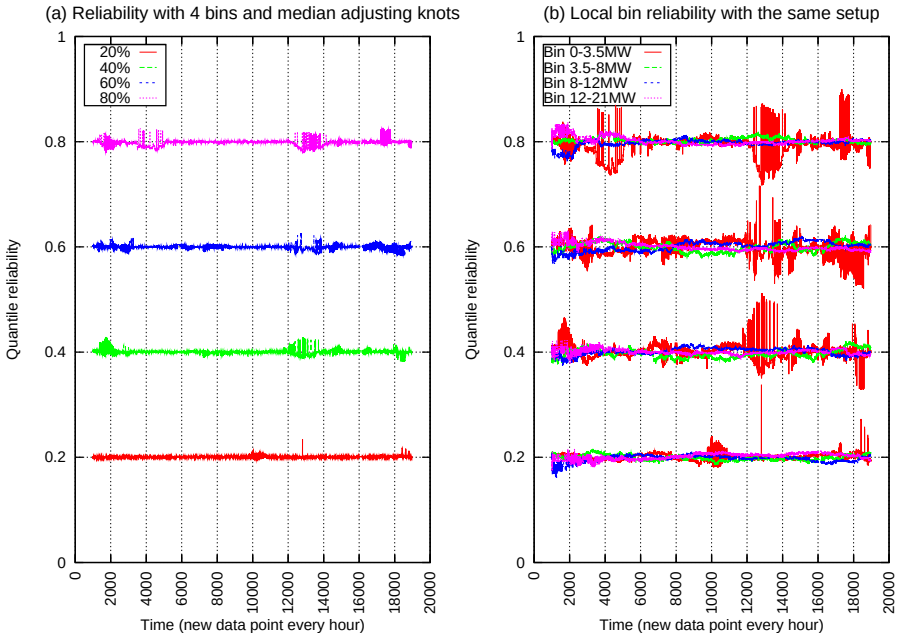

Figure 7.8: Local bin reliability with 4 bins same settings as before, but with 9 knots where the knots in the middle of bins are automatically adjusted to the median of the bin for each 50 points added. This adaptive knot placement have only improved the total reliability slightly, but the reliability in the first bin is still not good. The local reliability in the other bins are actually very good - the variations in the beginning are due to the fact that the bins have yet to be filled.

7.3.6 Penalty Based Replacement

As shown above, the problem with too many identical power predictions can cause the program to give unreliable quantile curves. Simply adjusting the knots to provide more flexible freedom of the splines does not provide sufficient improvements to the problem with reliability. Even the self adjusting knots could not cure the problem. It has previously been suggested that a change in the updating algorithm could help reject some of the many duplicate observations entering the system. Such an algorithm has been implemented, and in this section it will be described and tested.

Simply rejecting a number of zeros from the data set is indeed a possible solution, but given the fact that this program should be able to function as a general purpose adaptive quantile regression module, this strategy would be too simple. It would fail, if the program was used on data with duplicates at a different location than zero.

The method selected is to split up the bins in smaller segments and assign a *penalty* to each point in the active data set. This penalty based on how many data points are grouped in the same segment of the bin. The algorithm is very much like looking at the histogram and penalize the elements placed in peaks. The programming details of the algorithm is described in Section 5.3.3.

The selection of which element to replace with the entering element is simply the one having the maximum sum of age and penalty. It is important to still use the age when selecting the leaving data point, because if two elements have the same penalty assigned, the older one and not the younger should be thrown out of the active data set.

7.3.6.1 Standard Four Bin Setup

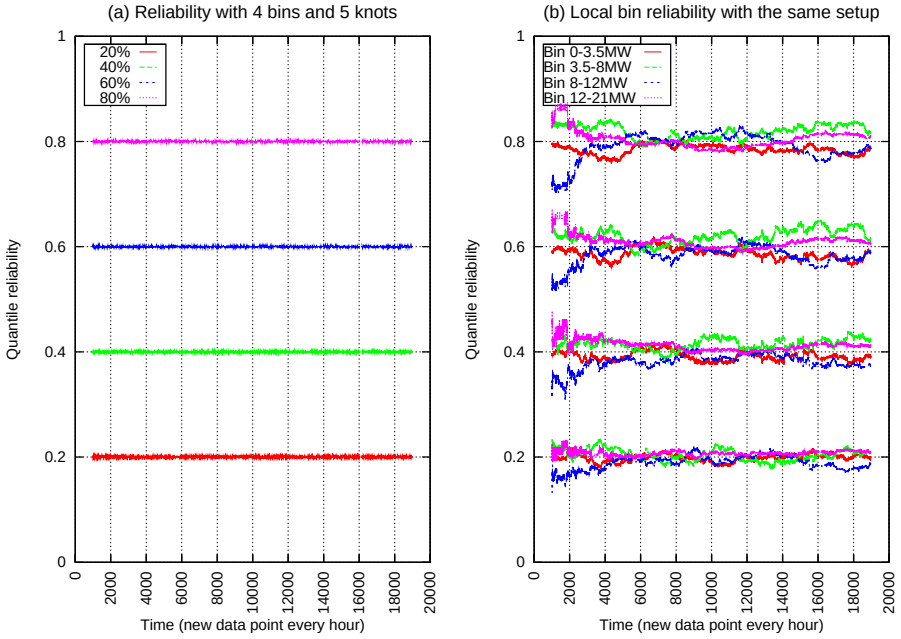

Figure 7.9: Average and local reliability of a run with the penalty based updating algorithm. Besides the new algorithm, the setup and data is exactly the same as used in preliminary reliability tests. The average reliability seems almost perfect spot on, but the local bin reliability shows some variation. The lower bin (0-3.5MW) finally seems to behave nicely.

The result of the new update algorithm can be seen in Figure 7.9. The average reliability have improved significantly and looks perfect. The local reliability have also improved in terms of extreme peaks, but the lines look far from straight. Besides the peaks, the local reliability seemed better in the previous runs with 9 knots, so in order to test if adding more knots helps, the next test will have additional knots in the center of each bin.

7.3.6.2 Additional Knots

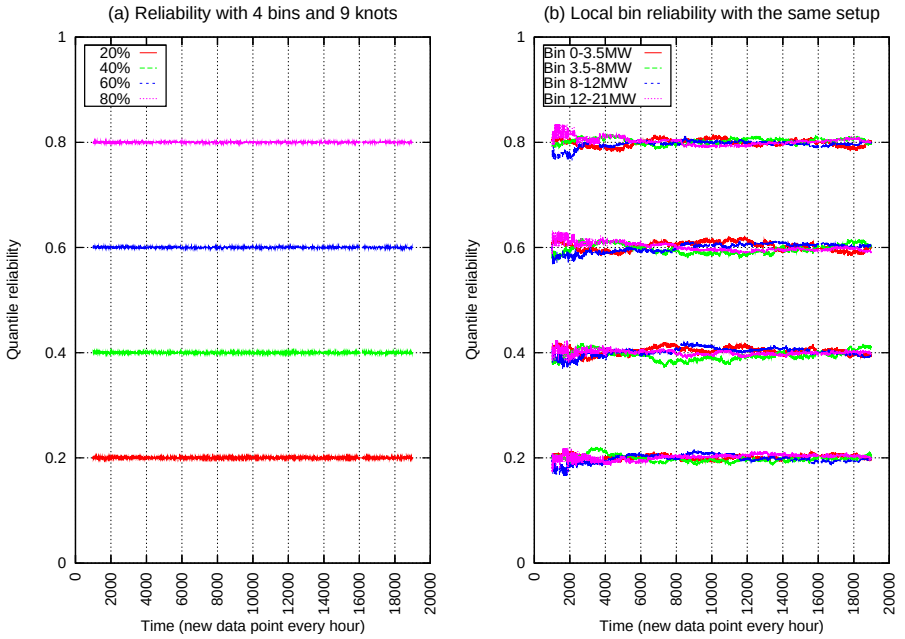

Figure 7.10: Average and local reliability with the standard 4 bins and 9 knots placed on edges and in center of all bins. The mean reliability is close to perfect and the variations of in local reliability are very small.

With the combination of a better updating algorithm and knots both on edges and in center of bins, the reliability plots in Figure 7.10 is finally looking really good. The bumps in the beginning (1000-4000h) are due to the bins not being reliably filled. In the beginning, the concentration of zeros will dominate, because the points in a bin are not starting to be replaced before the bin is full, so the bin needs to have been full for a while before the quantile curves are getting reliable.

7.3.6.3 Histogram Active Data

It is interesting to observe what effects the penalty based updating algorithm have on the distribution of active data as time goes. The goal of this penalty system was to avoid having too many duplicate points. In Figure 7.11 are the histograms with settings identical, except for the updating algorithm, to the other "active data histogram" in Figure 7.7.

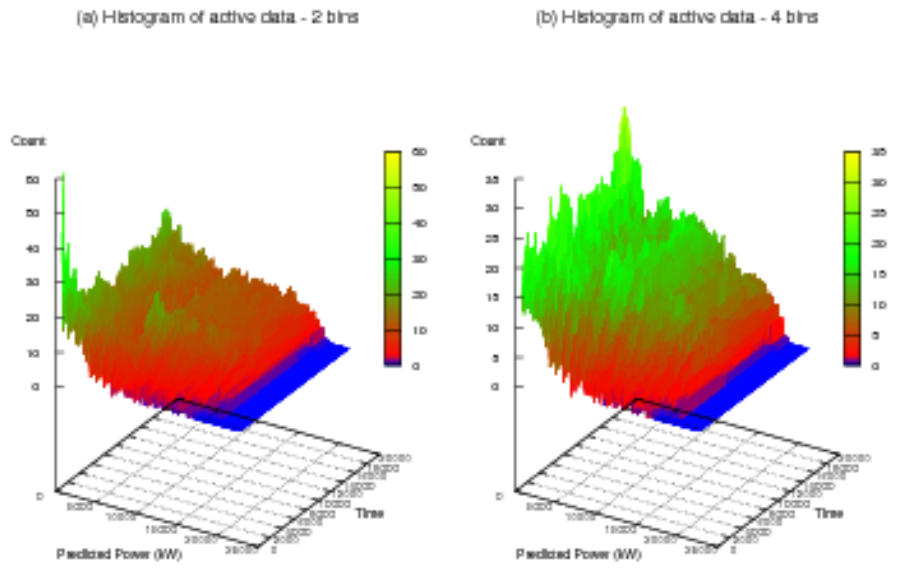

Figure 7.11: Histograms of the active data with penalty based updating algorithm in the cases with two and four bins. The resolution of the histogram is again 100kW. Plot (a) with two bins seems very flat, which is good. There is a big peak around zero predicted power in the beginning, but that is only to be expected because the bin might not even be filled at this point, so no duplicates have been thrown away. Due to the smaller bin size, this does not occur on the test with four bins.

The histograms in Figure 7.7 supports the penalty based strategy for selecting the leaving element from a given bin. There are some small peaks from time to time, but the effects are greatly attenuated, leaving a better variety of points to base the quantile regression on.

The big peak in the beginning of the two bins run (plot (a)) looks worse than

it actually is. As mentioned in the figure text, points only start getting thrown away when the bin is filled, so the bin needs to be full for a while before the distribution flattens.

7.3.6.4 Last Test - Putting it All Together

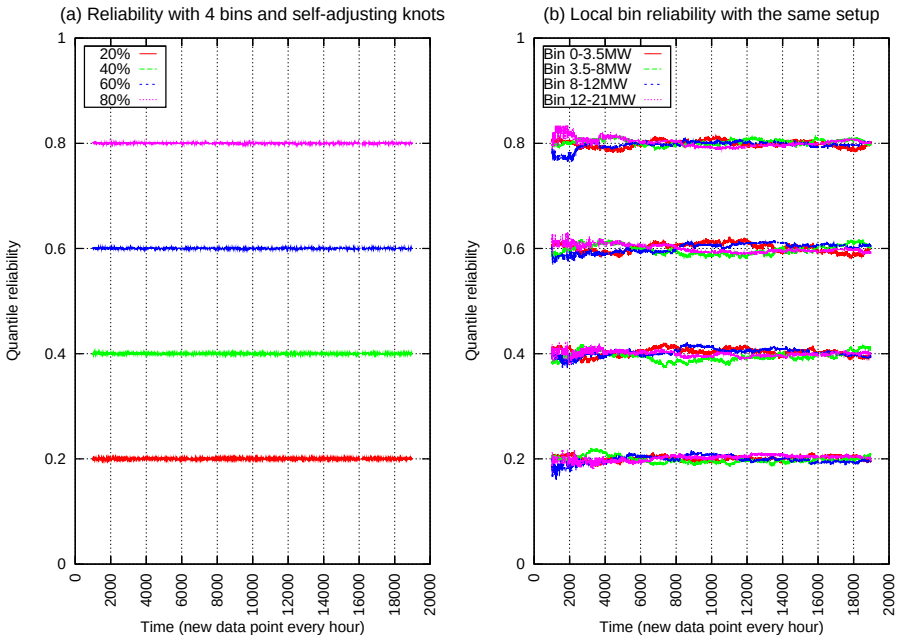

Figure 7.12: The total and local reliability of a run with four bins, penalty based updating algorithm and median adjusting knots. This looks almost exactly like Figure 7.10, but the more even distribution of the bin data should also move the medians closer to the middle of the bin and thus limiting the effect of moving towards the median.

The test shown in Figure 7.12 concludes these reliability improvement tests. The setup is based on the usual four bins and knots are now placed with the median adjusting algorithm. Both the average and local reliability are very good, but it is difficult to see much improvements compared to the run with knots placed on the edge and in the center of each bin (Figure 7.10).

Placing knots in the center or median of the bins does help significantly, but the quantiles are very data depended, if many identical predictions are allowed to enter the system. The penalty based updating algorithm offers a perfect simple, yet flexible solution to this problem and will be used as a default in the next tests. The adjustable knots can only be used when the interior point method is used and in the test case, not much was gained, so this feature will be left optional.

7.4 Simplex vs. Interior Point Method

In Section 7.3, the updating algorithm was tested together with the interior point method, and the combination provided good results in the end. Now with the updating algorithm and active data behaving nicely, the simplex method can be tested against the IMP implementation. This will be tested in this section.

An addtional benefit of having two different implementation and being able to test them head to head, is the added confidence in both algorithms, if the results they produce are the same. Together with the reliability tests, if both simplex and interior point method behaves as expected, the implementations can be considered correct.

Since the two methods use different techniques to get to the optimum, a perfect match between the two methods cannot be expected. In fact if they both calculate the exact same predictors, it will be hard to argue that notthing has been done to tamper with the tests. Small rounding errors and the fact that the simplex method only yields optimum on vertices whilst the interior point method can provide results anywhere on the boundary, minor differences can occur.

The test data will be the same as used in the reliability test (Section 7.3), and the setup will be based on 4 bins:

0-3.5MW	3.5-8MW	8-12MW	12-21MW

, where the 9 knots will be placed on the edges and in center of each bin.

The tests which will be performed are first the reliability test of the simplex method, which should give the same result as seen in Figure 7.10. If the reliability is acceptable, a direct comparison between the results from each of the algorithm will be carried out.

7.4.1 Reliability

The first and most simple validation test is the reliability. The test is carried out exactly as in Section 7.3, with only the algorithm switched to simplex.

The reliability plots from the simplex run, shown in Figure 7.13 looks exactly as they should, in fact they seem very identical to those of the IMP run (shown in Figure 7.10). The small variations in the local reliability at 6000-8000h are also found in the IMP version, which supports that these two algorithms are providing very similar results.

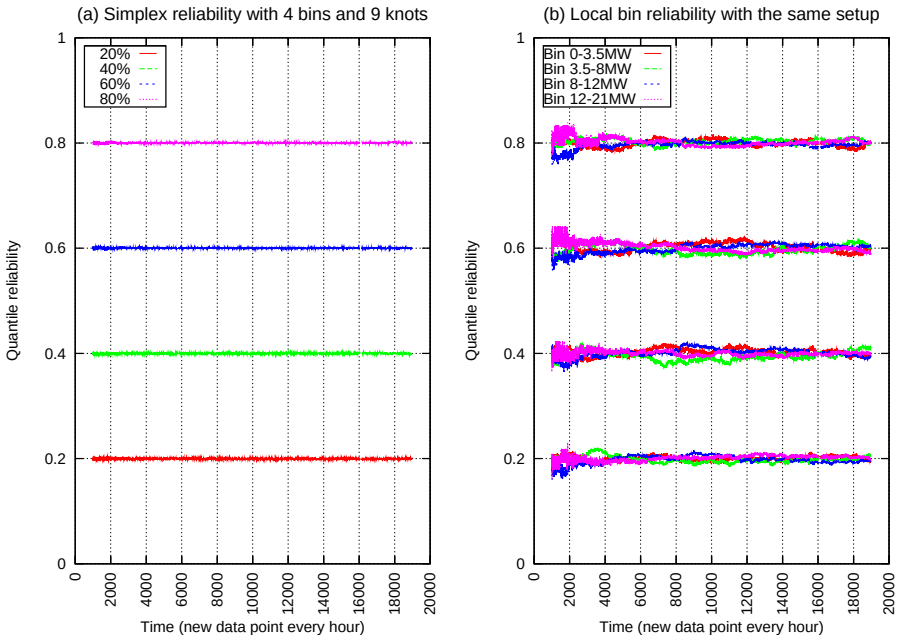

Figure 7.13: The plot shows the reliability of the four quantiles computed with the simplex method. The left plot shows the total or average reliability, which seems to be close to perfect. The local reliability plot to the right is not as perfect, but the lines behave quite well, which have been shown earlier is very good for such a plot.

Furthermore, the fact that these two algorithms result in reliability variations the same places support the conclusion in the reliability section that the problems were caused only by the updating algorithm and knot placement and not the quantile regression itself.

With the confidence that the reliability of the simplex method is acceptable, a more direct comparison will be the next step.

7.4.2 Graphical Comparison of Predictor Matrix

As mentioned before, the two algoriothms are very different in nature, although both of them are optimization algorithms, the strategy for finding the optimum differs very much. Never the less, the *predictor* produced by both algorithms will be directly compared here.

A special driver application have been implemented to dump the contents of the predictor matrix at every point added into a file. This results in large amounts of data because there is an individual predictor for each quantile and each of these predictors constist of a as many coefficients as there are knots. It is often a challenge to show such large amounts of data in a easily understandable form, so only the 20% quantile will be considered here.

The full page plot (Figure 7.14 page 126) shows the value of all the coefficients for the 20% quantile predictor and how they changes over time. This data is also what is referred to as the contents of β_τ where $\tau = 0.2$. The actual quantile curves $Q(\tau|\mathbf{X}_{spl})$ are found from this β_τ by the equation $Q(\tau|\mathbf{X}_{spl}) = \mathbf{X}_{spl}\beta_\tau$, where $\mathbf{X}_{spl}$ is the $N \times K$ matrix with spline mapped explanatory variables (K is related to the number of knots). In the plot, the result from the simplex method is shown with red lines and the green dots are from the IMP run. The lines and dots seem to coincide nicely.

The lines (simplex) actually seem to be completely covered by the dots showing the IMP calculated predictor. This looks exactly as it should, so there is no need to make visual inspections on the actual quantile curves, the raw predictor data matches perfectly.

7.4.3 More Detailed Study of Differences

As it was shown in Figure 7.14, the predictors found from both implementations are very much alike. In fact, they are so much a like that simple visual inspec-

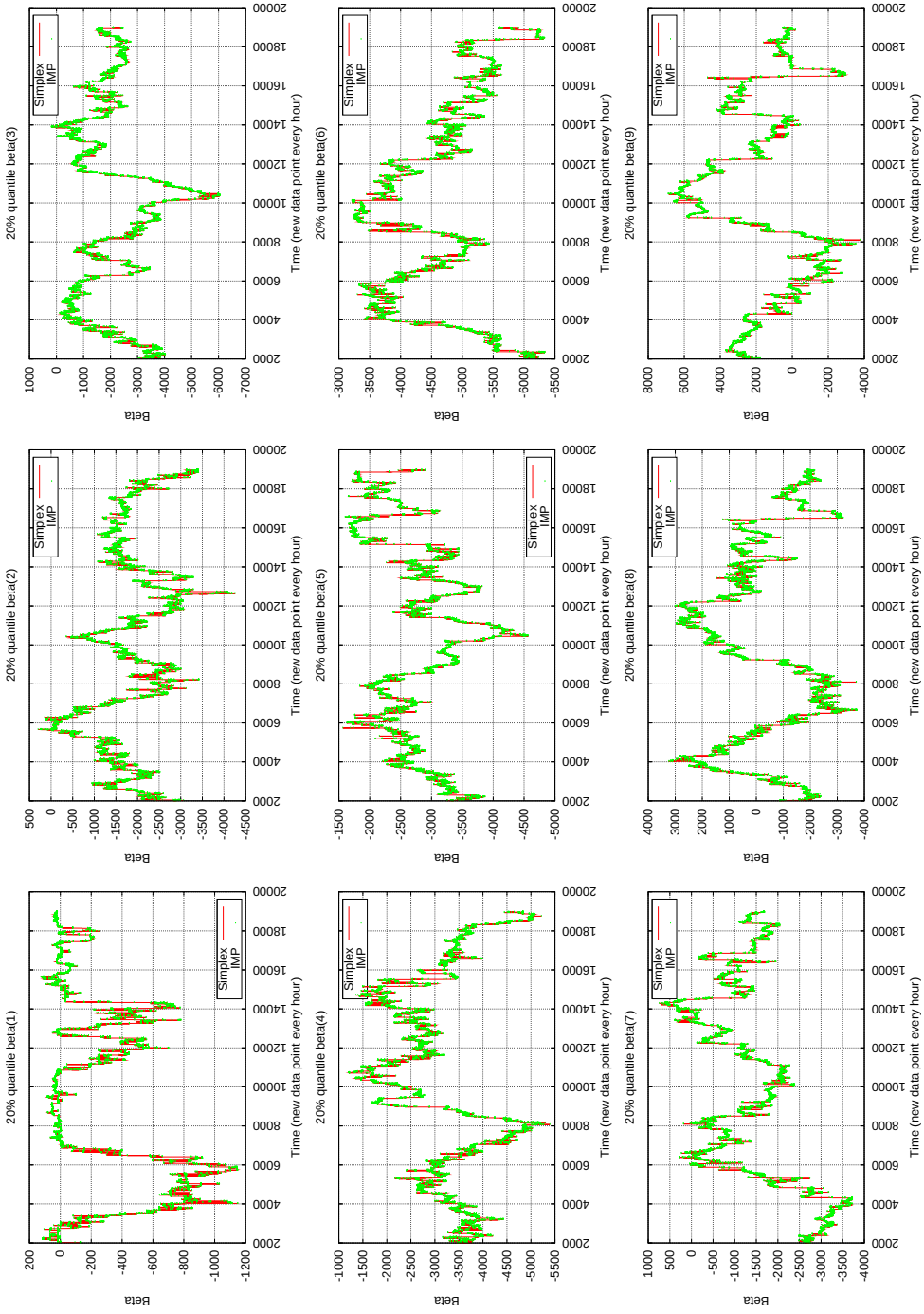

Figure 7.14: Simplex and IMP direct comparison of 20% quantile predictor

tions of the two predictors plotted on top of each other reveals nothing. The next step in the validation is nessarily to take a closer look at the difference - at least some rounding errors should emerge.

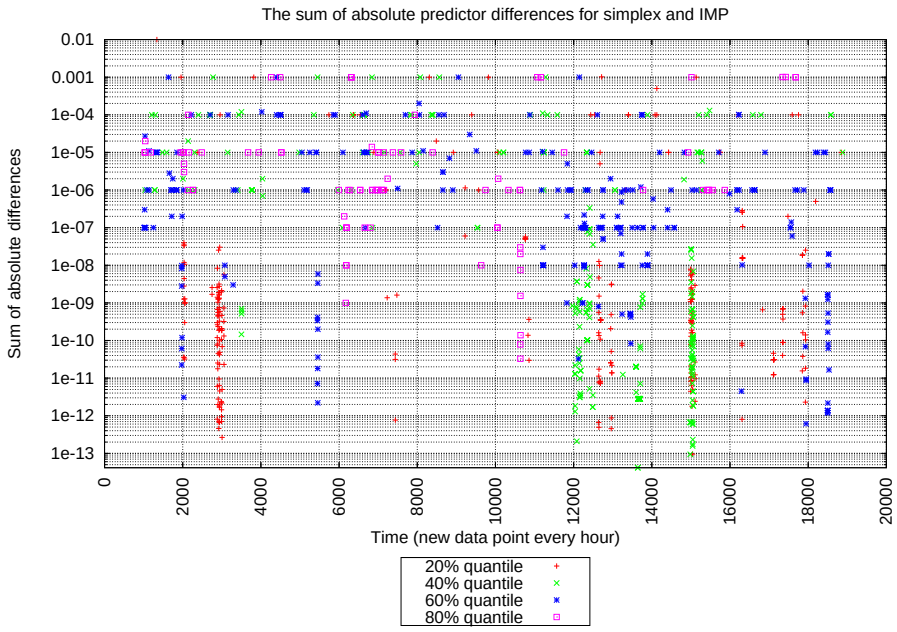

Figure 7.15: This plot is meant to show the magnitude of the differences between the algorithms. To reduce the amount of data, the absolute differences have been summed for each quantile at each data point added. The data have been converted to ASCII(scientific 7 digit notation) before comparison, so some differences may have been truncated.

The plot in Figure 7.15 show the sum of differences between the predictor calculated with the two methods for each data point. It seems like there are many errors, but they are fairly small and some differencies were expected. The number of points which result in difference in the predictor was only 851 out of 17983 points added. This means that there were no difference at all in most of the points and 5% of the times a very small difference was noticed.

7.5 Quantile Prediction

So far, only the functionality and quality of the quantiles in relation to the data stored in the program have been tested. In this section, the predictor will be used as it is supposed to - to predict quantiles of events in the future.

The point of having an adaptive algorithm for estimating quantiles is to be able to adjust for changes in the prediction error. Since the data used for these tests are real windmill data from the WPPT program, the error is expected to change with time and thus result in degraded quality of the quantile predictions as the "look-ahead" time increases. The purpose of these tests are first of all to see if quantiles can be predicted using this program, and second, how far into the future they will remain valid.

7.5.1 Test Setup

Yet another driver program has been developed to facilitate this test. The principle of the test is also yet again very simple, instead of feeding the most recent data to the updating algorithm, the elements are stored in a FIFO[6] in order to have a time shift between quantile prediction and updates of the predictor.

The idea behind the driver program can be seen in Figure 7.16. The FIFO makes it possible to have a constant relation between time shift and *prediction age*[7]. With this time shift in place, it is a simple matter of multiplying the content of the FIFO with the predictor in order to obtain the quantiles.

For each added point, the quantiles of all requested prediction ages are compared to their corresponding prediction error. Counting the number of points below the quantile estimates and dividing this number by the total number of points gives the reliability, for each quantile prediction age.

[6] First In First Out buffer, a simple core logic design block, but in general purpose programming, it is implemented by moving the data one step in memory for each new element added.

[7] Age is used here instead of *horizon*, which would be more appropriate, to avoid confusion between quantile prediction horizon and wind power prediction horizon.

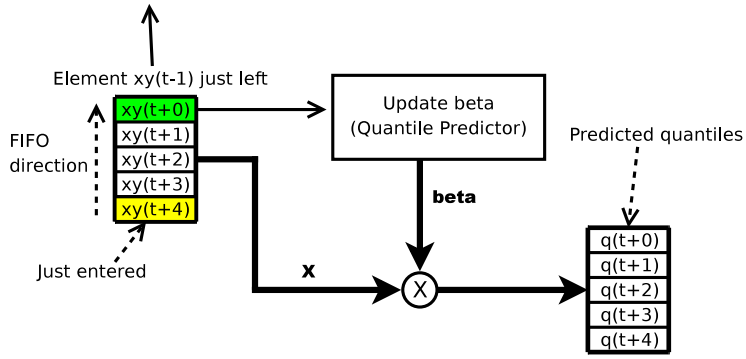

Figure 7.16: This is a sketch of how the data time shift needed to predict future quantiles is made possible by the use of a FIFO. The data elements are represented as "xy" to emphasize the strict relation between prediction error y and explanatory variable x (for simplicity, spline mapping is not show). The oldest point $xy(t+0)$ is used to update the predictor and quantile prediction is then done by the multiplication of explanatory variables and the new predictor. The resulting quantiles are predicted for all requested ages, in this case 0 to 4 hours into the future.

7.5.2 Test Data and Setup

The data used for these tests are the same as described in Section 7.3.2, but for these tests, the five horizons 5, 10, 20, 30 and 40 hours are used. Again there are no particular reason for taking exactly these horizons, but using them all would make it difficult to make clear plots, and the selected horizons covers most of the range offered by the data set. In this case where actual predictions are made, it is a good idea to vary the prediction uncertainty (indirectly done by selecting other horizons), because this ensures that the findings does not only apply to a particular degree of uncertainty in the predictions.

Four bins placed in the order:

0-3.5MW	3.5-8MW	8-12MW	12-21MW

And also the 9 knots are placed the usual way, five on the edges and the remaining four in the middle of each bin.

The quantiles calculated this time are 20%, 40%, 60% and 80%, so the computed reliabilities should also be placed on these percentages.

7.5.3 First Simple Test

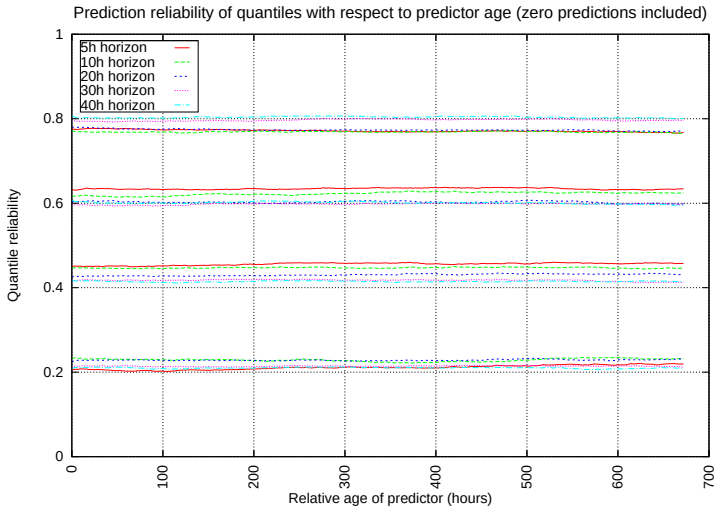

Figure 7.17: This plot shows the reliability of the quantile predictions as a function of the relative age of the predictor. Five different data sets have been used, and each have their own color. The optimal situation would be perfect horizontal lines placed at their corresponding quantiles - 20, 40, 60 and 80%. The lines are close to straight and horizontal, but particularly the 5 and 10 hour horizons are located far away from their optimal location. Very little time variations can be seen as the prediction age goes up.

The plot in Figure 7.17 show the prediction reliability of quantiles predicted 0 to 672 hours (4 weeks) into the future. There seems to be very little difference in reliability when the age of the predictor is increased. What this means is that there are practically no difference in the quality of the prediction, if it is a few hours old or 4 weeks. This is surprising, because it questions the need to continuously update the quantile predictor, at least for these data.

The actual quality of the predicted quantiles are however not as good as expected, there seem to be vertical offsets on many of the reliability tests. A constant offset through all the different prediction ages indicate a problem with the data and not the predictor. Most likely, these offsets originates from all the zero predictions in the data sets. With too many zeros, the averaging nature of the reliability test can easily be biased in a way such offsets occur.

7.5.4 Without Zero Predictions

The most simple method to establish, if the zeros are again to blame for strange results, is to take them out. Since the driver program for this test is a special instance and not part of the general purpose module, it is safe to simply pick out all zeros from the reliability evaluation[8] which is exactly what have been done for the next test.

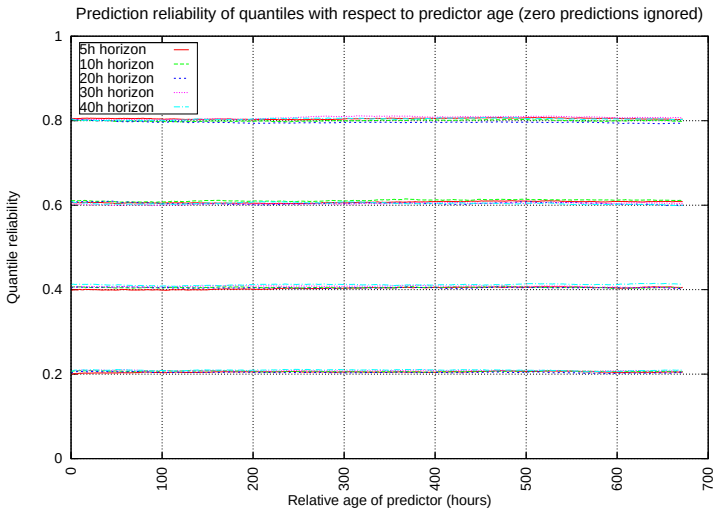

Figure 7.18: This plot shows the reliability of the quantile predictions as a function of the relative age of the predictor. The range is the same as in Figure 7.17, but here the zero predictions have been ignored. The reliability is very good for all four quantiles in all five horizons. Again there seems to be very little correlation between relative age of predictor and reliability.

Yet again, dealing with the zeros proves to be the key to get stable reliability measures from the quantiles. The plot in Figure 7.18 could hardly be any better.

It was expected to see the reliability getting worse with the age of the predictor, but it seems that within the period of four weeks, the predicted quantiles are fairly stable. Next step is obviously to go further with the predictor age.

[8]The zeros are not taken out of the quantile regression module, because the penalty based updating algorithm should be strong enough to cope with these multiple zeros.

7.5.5 Further Into Future

In the last reliability test, we will try to determine how far into the future, the predictor stays valid. Due to the amount of data needed, to stabilize the predictor, the size of the data set puts a limit on how much further the predictor age can be tested. The longer the period of prediction gets, the less points will be available for computing the reliability afterwards. A period of approximately a half year offers a reasonable compromise between time and number of points.

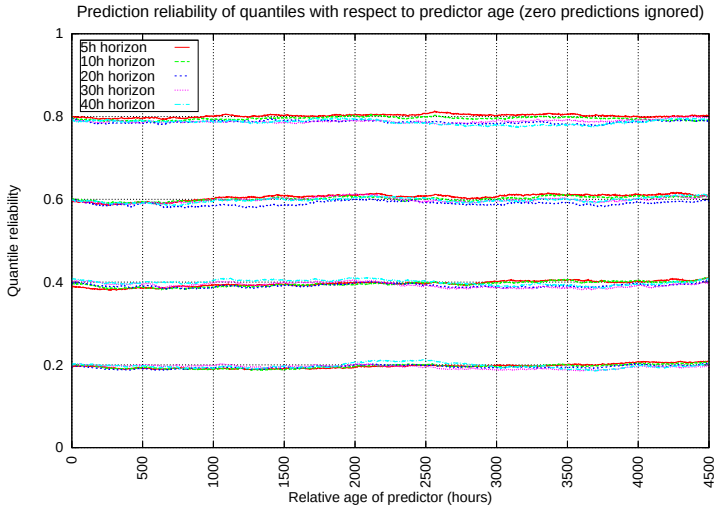

Figure 7.19: The reliability plotted against the predictor age from 0 to 4500 hours (approx. half a year). Five different horizon data sets have been used and they are plotted with different colors. The reliability seems to be reasonably good throughout the whole range of the prediction age. The slight variations in the reliability does not seem to be directly linked to the age of the predictor.

In Figure 7.19, the reliability is plotted for predictor ages from 0 to 4500 hours, which corresponds roughly to half a year. A slight variation with predictor age can be seen, but it does not seem to get continuously worse the way it was expected. The 20h horizon seem to show a slightly degraded reliability of the 60% quantile at 400h and again around 1000-2000h. Although the variations are not that big, it might still be a good idea to update the predictor frequently.

7.5.6 The Averaging Effect

The results shown here are from a very simple data set only taking measured power versus predicted power into account. The quantiles ideally show the error margin of the power predictions based on previous predictions and outcome. If the prediction errors do not change with time, the quantile predictor made years ago will still be valid. The adaptive algorithm is supposed to adjusts its uncertainty predictions of changing events, which are not modeled (perfectly) by the power forecast model.

At first glance, what these tests, Figure 7.19 in particular, show is that the error does not change significantly over a long period of time. One explanation could be that the prediction model in WPPT, perfectly adjusts itself for all possible time depended variations.

Another, less intriguing explanation, but probably more correct is the averaging nature of the quantile reliability test smooths out any variations over time. If the quantiles are positively offset by a certain amount for a while and then negatively for a similar period, the net result would look perfect, according to the reliability tests at least. The problem here is that with the reliability test, it is impossible to test if a single point is correctly characterized by quantiles, because the nature of quantiles describes distributions and not single points. In the next section, a different validation technique will be used. This might shed some light on question whether the shown lack of relation between prediction age and quality of uncertainty predictions are correct.

7.6 Prediction Skill Score

According to Pinson et al. [13], a valuable method for testing the quality of quantiles is the measure called *skill score*. It is calculated from the residuals very much like the reliability shown in (7.3). The difference between the two methods is that the skill score is not found by counting, but by applying the loss function to the residuals. The residual vector $r^{<\tau_i>}$ for each quantile τ_i is defined as

$$r^{<\tau_i>} = y - \mathbf{X_{spl}}\hat{\beta}^{<\tau_i>}, \tag{7.4}$$

where $\hat{\beta}^{<\tau_i>}$ is the predictor and $\mathbf{X_{spl}} \in \mathbb{R}^{N\times K}$ contains the N spline mapped explanatory variables under consideration. K is the total number of spline coefficients.

Then by applying the piecewise linear loss function $\rho(\tau_i, r)$ defined in (2.1) on

the residuals $r \in \mathbb{R}^N$, the average skill score can be calculated as

$$skillscore(\tau_i, r^{<\tau_i>}) = \frac{1}{N}\sum_{j=1}^{N} -\rho\left(\tau_i, r_j^{<\tau_i>}\right). \tag{7.5}$$

To get a better understanding of how the skill score in (7.5) works, the the loss function can be expanded, and the average skill score will become

$$skillscore(\tau_i, r^{<\tau_i>}) = \frac{1}{N}\sum_{j=1}^{N} \begin{cases} (1-\tau_i)\, r_j^{<\tau_i>} & \text{if } r_j^{<\tau_i>} < 0 \\ (-\tau_i)\, r_j^{<\tau_i>} & \text{if } r_j^{<\tau_i>} \geq 0 \end{cases}. \tag{7.6}$$

By looking at (7.6), it is clear that the skill score will always be less than or equal to zero. A high skill score is a good skill score, and this is in fact exactly the evaluation function optimized during quantile regression with both implementations (simplex and IMP).

The skill score might seem like an arbitrary number, and in some sense it is. For the skill score of these tests to be comparable to results obtained from other windmill farms, the results must be normalized using the maximum power capacity, which is 21MW at the Klim windmill farm. In all but Figure 7.26 and 7.27 , this has been done on the skill score afterwards (during plotting)[9].

Test setup will be the same as in Secntion 7.5.2, unless otherwise stated.

7.6.1 Skill Score vs Prediction Age

It will be interesting to see if the skill score stays constant with prediction age, just like the reliability did in the previous tests. To test this, the same driver program only needs to be adjusted slightly, to calculate the skill score and not the reliability. Since the skill score does not offer the same kind of easy plotting due to segmentation at the nominal quantile ratios, only the average skill score of the four predictions will be considered for now. The data set will be Klim data set used previously.

The plot in Figure 7.20 show that the skill score has some dependence on prediction age, but it actually gets better when the predictor is 500-700 hours old. The quantile predictor age might fit with a season variation or perhaps it is pure luck. There are variations with prediction age, but nothing final can be concluded from this test.

[9] Normalizing the data before processing was also tested, but showed no difference in results.

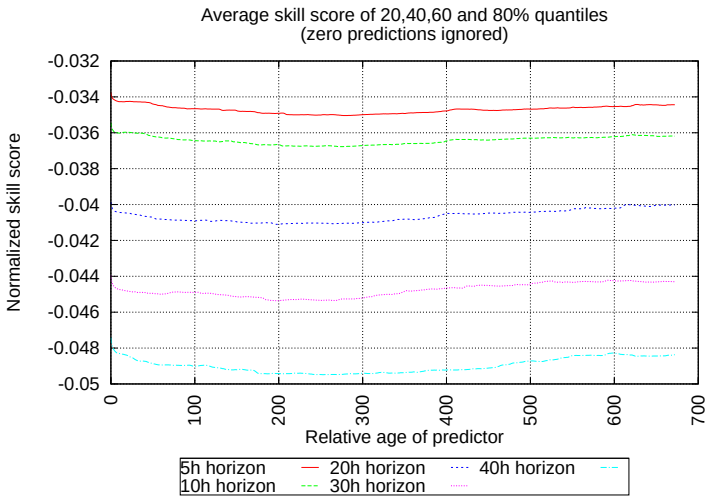

Figure 7.20: The average skill score of the four predicted quantiles in each horizon. The skill score have been normalized with the maximum power (21MW).

However, the tendency towards worse skill score as a function of horizon predicted by WPPT seems to be very clear. This relation with horizon was also shown by Pinson et al. [13]. It makes perfect sense that the skill score will go down as the horizon and thus uncertainty goes up. The residuals will be larger if the spread is larger between the quantile curves, and thus lead to a worse skill score.

7.6.2 Prediction Updates

In the previous test, the predictor has been updated every hour, but it has been used on data in the future in order to show how the age of the quantile predictor affects the predictions. In a real life situation, this way of using the predictor would never occur. A situation where an old predictor would be used would be if the predictor for some reason does not get updated every time a new data point enters the system.

Situations where the predictor might not be updated every time new data enters, could be if the predictor update is considered too computationally intensive to do very often. It would also be the case if the measured values were bundled and only delivered to WPPT once a day or even less frequently. Then the predictor

would not have the data needed to get updated continually, and thus be forced to only update every time a data bundle arrives. In such situations, the quantile predictor would get updated and then used for a while until it would be updated again. This situation is what the next test will try to investigate.

Different update frequencies will be considered by looking at the skill score. Ideally, there should be a clear relation between update frequency and skill score, but the tests so far have shown little correlation between age of the predictor and prediction quality.

To reduce the amount of data to plot, only the 40 hours horizon will be considered for the next few tests. This horizon has been selected because its skill score is worse than the others and the a longer horizons are more interesting in terms of power trade. The 10 hours horizon however, will be shown in Figure 7.25 for comparison.

From the plots in Figure 7.21 a slight difference in skill score can be seen for the different predictor update frequencies. The difference is however not very large, except for the case with only one update. Either the data is better after 10000 points or the system with updates gets better trained. The peak in plot (d) between 16000 and 18000 hours indicates that the system benefits largely from training beyond the 6000 hours, which is the only time the predictor gets updated in plot (d).

It can be hard to spot small differences across several plots, so in Figure 7.22 all the tests have been assembled in one plot, but instead of looking at the skill score for separate quantiles, the average value have been calculated.

The results shown averaged together in Figure 7.22 does not reveal much more information. For this setup it is clear that updating the predictor does help to improve the skill score, but it does not need to be done every hour. Including other explanatory variables might be a better route to lowering the skill score than increasing the frequency of updates.

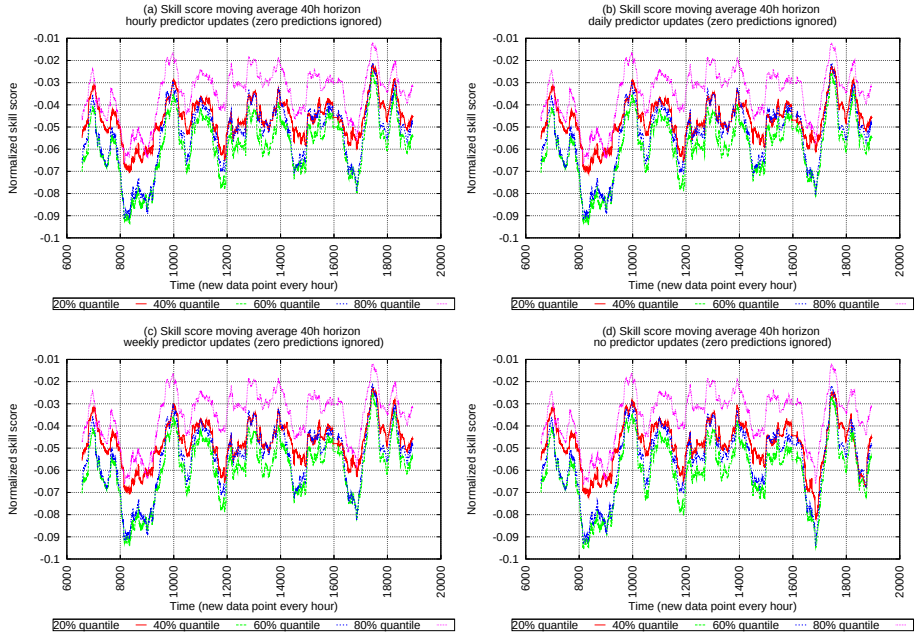

Figure 7.21: The 500 point moving average of the skill score with four different update frequencies. The predictor is updated hourly, daily and weekly in the plots (a),(b) and (c) respectively. In plot (d), the predictor is only updated once at 6000h. The low score between 8000h and 10000h might indicate that the system is not perfectly trained, since all but (d) keep the score better (almost) than -0.08 after 10000h.

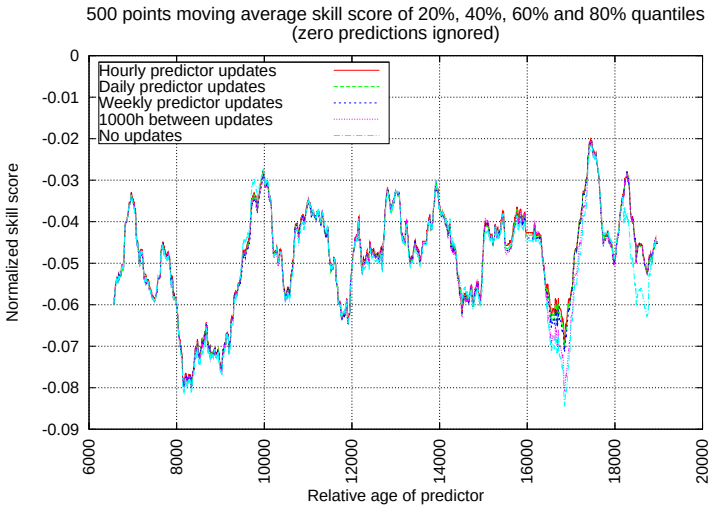

Figure 7.22: Evaluation of different quantile predictor update frequencies. The 500 point moving average for each quantile (20%, 40%, 60% and 80%) have been averaged for each prediction update frequency. Each line represent the total average skill score for each update frequency. In general the skill score seems to get better, the more frequent the predictor is updated, but the differences in skill score (except for "no updates") seem to be very small.

7.6.3 More Examples

A few corners were cut in the previous tests due to previous findings. The main area of concern is the fact that zeros have been left out due to the problems they caused when evaluating the reliability. Here a few examples will be shown that hopefully reveals that the inclusion of zeros does not diminish the results found with the skill score. The plots will only be briefly commented, since they should not show anything new.

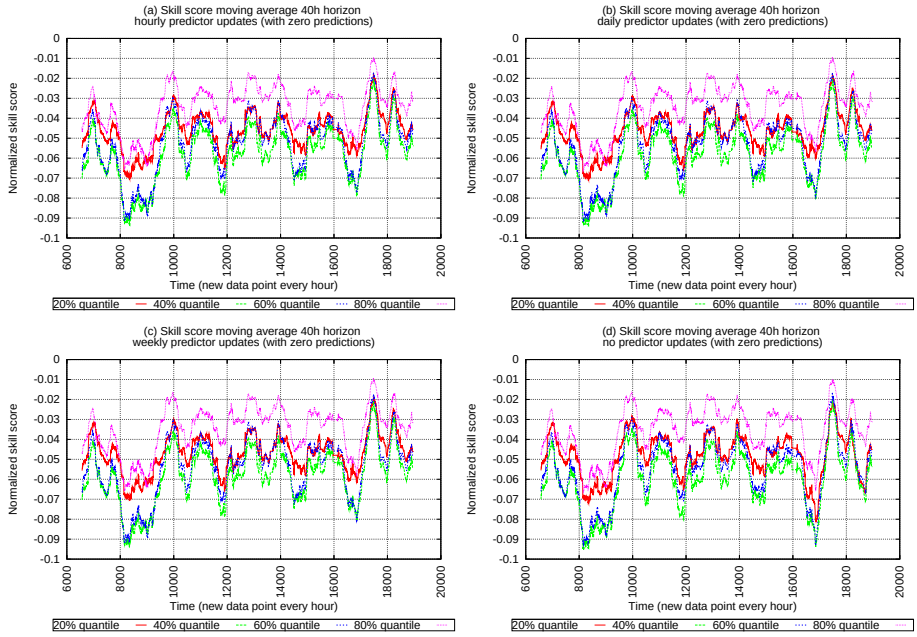

Figure 7.23: Test run identical to the one in Figure 7.21, but with zeros included.

From Figure 7.23 and 7.24 in can be seen that including zero in the skill score does not seem to have such a dramatic effect on the skill score as it had on reliability. This might come as a surprise, since the skill score is thought to give a more precise measure for the correctness of the quantiles, but it bases its evaluation on the exact loss function which is used for the quantile regression. This does not mean that the skill score is a bad measure for quality of the quantiles, on the contrary, it shows that reliability might be a too simple evaluation.

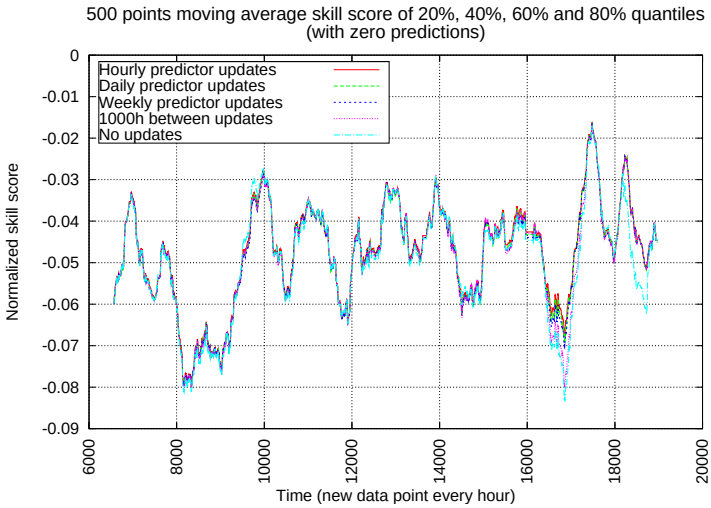

Figure 7.24: Test run identical to the one in Figure 7.22, but with zeros included.

7.6.3.1 Another Data Set

The 40hours horizon was selected as a reference data set to see the effects of update frequency. Here the 10 hours data set will be used to verify that the findings were not only related to the relatively long 40 hours horizon.

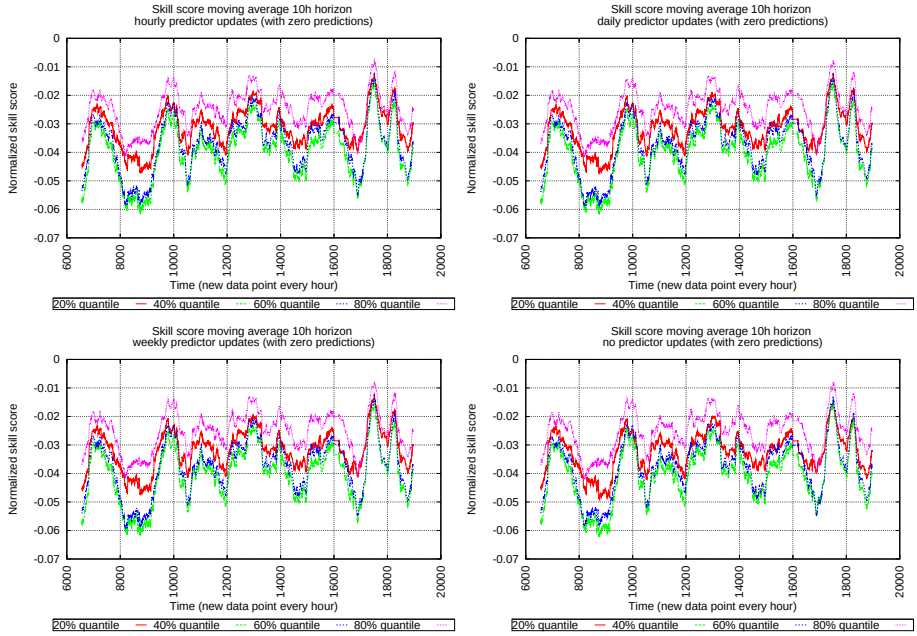

Figure 7.25: Test run identical to the one in Figure 7.23, but with the 10 hours horizon data set as input.

As can be seen in Figure 7.25, the update frequency of the quantile predictor does not seem to be hourly. This test even seems to favorize the test with no prediction updates beyond the 6000h point.

7.6.3.2 More Quantiles

The test setup used by Pinson et al [13] included more quantiles and instead of looking at the average skill score related to each quantile, the sum of all the skill scores was considered. For the 40hours horizon, the skill score was

approximately[10] -0.77.

It is not the intention to compete with the results in [13], but instead to show that the implementation done in C performs comparably well. The quantiles to be calculated are from 0.05 to 0.95 with steps of 0.05 between, a total of 19 quantiles.

Another difference with the test setup for this test is that the data set have been normalized prior to entering the system. The bin an knot locations are also normalized, so the setup should be completely identical to the previous runs.

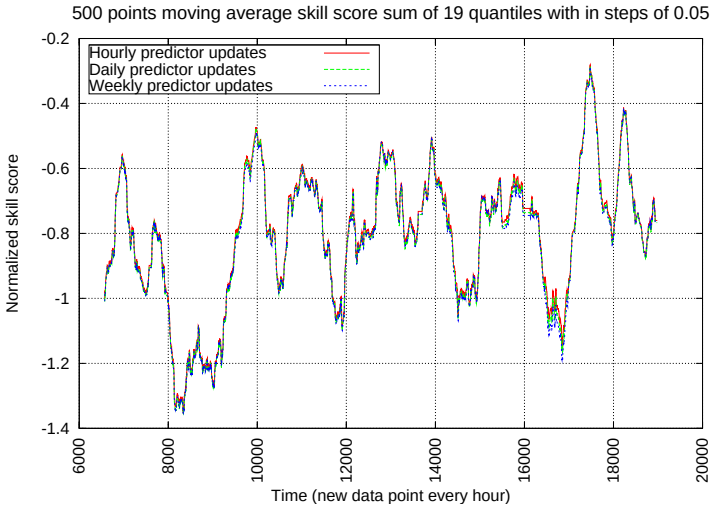

Figure 7.26: The graph shows the sum of skills scores from the 19 evaluated quantiles. The 40 hours horizon data set has been used to generate this plot. This looks almost identical to the previous figures. Three update frequencies have been tested and some small variations can be seen.

The plot in Figure 7.26 show the moving average of the skill score when evaluated like it was done by Pinson et al [13], and the size of the skill score seems to be in the same range. The bin and knot locations used in the article might be different than used here, which could result in other results.

This test showed the moving average, but it might also be interesting to see the average skill score of the entire data set. With an adaptive algorithm, which should get better all the time, it is difficult to select a certain point which should

[10] The number is read from a graph (Figure 6.) in the article.

define where the training set stops and the test begins. It will be unfair to the algorithm to consider the whole data set a test set, since a non-adaptive method would likely dedicate more of the data to training.

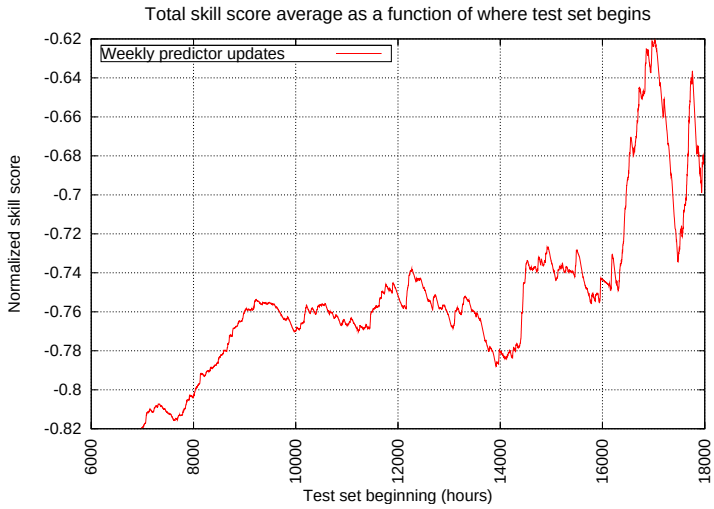

Figure 7.27: Evaluation of the average skill score of the whole test set as a function of where the test set starts.

Figure 7.27 shows the average skill score of the test set with weekly updates as a function of when the test set starts. As it is expected with an adaptive algorithm, the quality of the prediction gets better when the system have received more training. The value of -0.77 mentioned earlier seems to fit very nicely with the findings of this test - at least when data from 8500 hours (roughly a year) is used as training set.

7.7 Multiple Variables and Power Predictions

One feature of this program, which was not present in the Matlab implementation by Møller [8], was a simple scheme for using multiple explanatory variables. It was indeed possible, but the bins were not definable in all explanatory variables, which is the case for the current implementation. Since this feature is new, it is important to test it, but instead of constructing an uninteresting functional test to prove this works - the feature will just be assumed to work and a new prediction test, which needs this feature will be tried instead.

The test will be using multiple explanatory variables, so if it works, the changeable number of explanatory variables also functions properly. To fully validate such an implementation works, every possible outcome of all branches obviously need to be validated, but even with a fairly simple program like this, such a exhaustive validation remains a theoretical possibility only.

Besides testing the multiple explanatory variable functioning, this is a good opportunity to also test if the periodic splines works together with the non-periodic natural splines.

7.7.1 The Power Prediction Test

So far the data set have been consisting of only measured power and predicted power. Due to the nature of the power curve model, it is safe to assume that the prediction error is indeed dependent on predicted power. However, by only using the predicted power as explanatory variable, the quantile predictions cannot react on minor factors such as season and wind direction, since the wind speed will be dominating the predicted power.

It would be interesting to see what happens to the quantile predictions when the other factors are added. At the end of this thesis it was however not possible to obtain a data set large enough with both meteorologic forecasts and WPPT predicted power to do such a test[11]. It was however possible to obtain a fairly large data set with meteorological forecasts and power measurements for the same time at the same windmill site (Klim).

The obvious thing to do with this data is to use the program to estimate the actual power and not only the error. For this test, the explanatory variables will

[11] It has already been shown that with real data, it takes quite a while for the quantile predictions to stabilize, so with only 600 points it was not possible to fill the bins properly and get good quantile predictions.

be the wind speed at 10m and wind direction forecasts. Since this test involves trying to estimate the actual power, only a single quantile is needed, which is $\tau = 0.5$ or better known as the median.

The median does not offer the best predictions in this situation, the mean would be preferred, because the median weights every outcome equally unlike the mean, which as the name suggests does averaging. However, this does not really matter that much, because the test is not designed to compete with traditional models or those of WPPT, but simply a test to show if the program is still useful when more explanatory variables are used.

Several test runs were made, but only the most simple and successful version will be shown here. The season was also taken in as a variable, to test if three explanatory variables would work and if it could give better predictions, but the changes were so subtle, the two variable version was selected. The setup will be shown here:

7.7.1.1 Explanatory Variables

- Wind speed in $\frac{m}{s}$ at 10m (min=0.0 max=38.4)
- Wind direction in degrees (min=0° max=360°)

7.7.1.2 Settings

- Bin size = 500
- Wind speed bins 0-4,4-10 and 10-40
- Wind speed knots 0,2,4,7,10,30 and 40 (natural splines)
- Wind direction bin 0-180 and 180-360
- Wind direction knots 0,90,180,270 and 360 (periodic splines)

7.7.1.3 Prediction Frequency

The data used for this test are unlike the previous data only updated four times, once every 6th hour. Each of these updates consists of 48 horizons with 1 hour in between, just like the previously used data set. This should not change anything for the program, but less points will be available each year.

7.7.2 The Results

The prediction run worked as it was supposed to, although with three explanatory variables and many spline knots in each, the interior point method sometimes have a hard time converging. All 48 horizons were calculated independently, but only the 30th, will be shown here. A horizon of 30 hours is a nice round number and it is close to what is required for power trading, but again, this is merely a test of using multiple explanatory variables and not a replacement for conventional prediction methods.

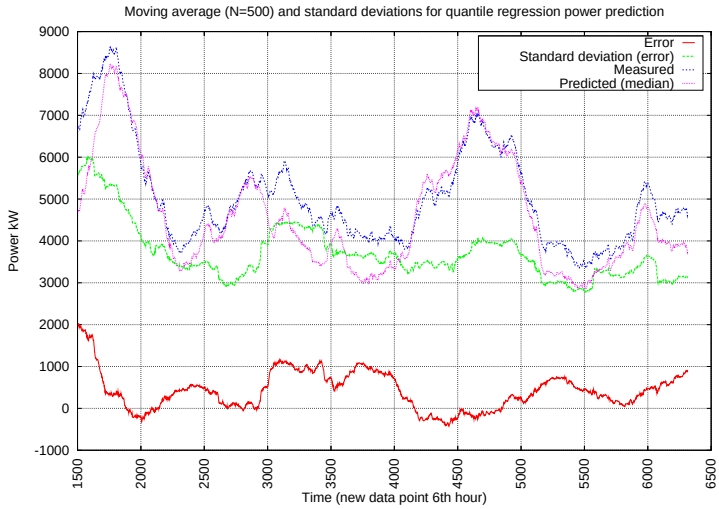

Figure 7.28: The plots shows the moving average prediction error and its associated standard deviation. As for a reference, the moving average of predicted and produced (measured) power are plotted for the same period. This moving average have been made with N=500, which corresponds to 125 days. The x-axis is no longer hours, but quarters of days, since the meteorological data arrive four times a day. The predictor is updated every 42 day or 168th point added.

The plot in Figure 7.28 show the 125 days moving average of the predictions. The predictions seems to be getting better over time, which is exactly what was hoped for. This shows that the system is adapting and getting better to predict. The prediction error is close to zero most of the time, but there is a slight tendency towards predicting less power than actually produced. A prediction method based on means, would most likely have an error more symmetrical around zero.

The standard deviation seems to be very high compared to the actual measured power, but the good thing is that it gets better with time. It will be interesting to see how WPPT handles this, and fortunately, although not for the whole period, power predictions from WPPT of the same data set was obtainable.

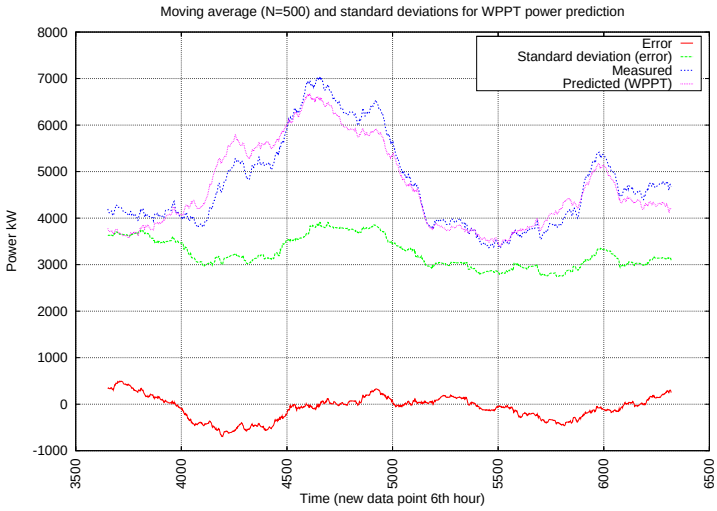

Figure 7.29: This plot shows the prediction error mean and standard deviation as well as predicted and measured power based on WPPT. The window size of the moving average is 500 points, which equals to 125 days. For easier comparison, the arbitrary "points added" x-axis have been shifted around two years to match approximately match the time in Figure 7.28. The standard deviation of the WPPT predictions are actually very similar in size to those found with the adaptive quantile regression median predictions.

The moving average and related standard deviation of the prediction errors of WPPT can be seen in Figure 7.29. The error mean seems slightly better than the one produced by the adaptive quantile regression prediction, but the deviations are still high compared to average measured power. It would be interesting to be able to compare the two algorithms over a longer period of time, because the adaptive regression method appears to be very demanding in terms of amount of data.

Traditional non-adaptive models use a training set to adjust the model and then the model is not changed on the test set. The adaptive method does however not really need that much training, it starts on the test set straight away, but the predictions are only as good as the data that has been through the algorithm. Since the adaptive quantile regression model has not been fed

with any prerequisites, the learning will naturally be slower than models like the one in WPPT, which relies on many years of research into wind power prediction models. The data point which are not shown on the WPPT graph are in fact those data points used for training.

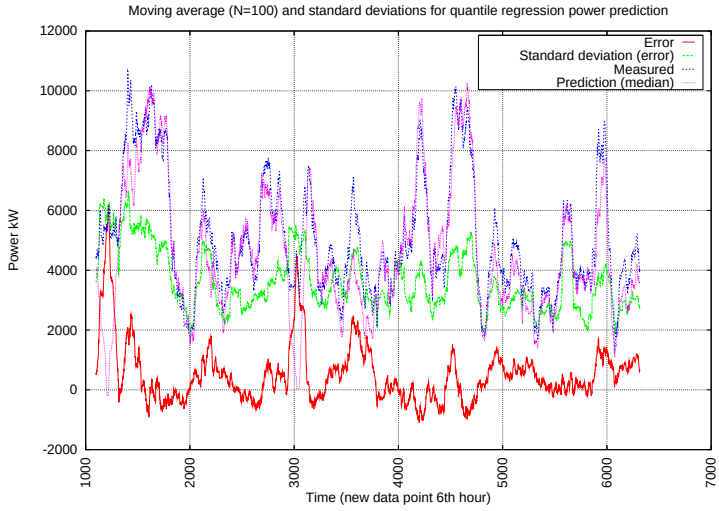

Figure 7.30: The plots show the moving average prediction error and its associated standard deviation. The moving average of predicted and produced (measured) power are plotted for the same period. This moving average have been made with N=100, which corresponds to 25 days. The increased resolution or decreased smoothing factor reveals spikes, but also a good correlation between mean estimated and produced power.

Moving average is very good for smoothing out time series, but depending on what information is sought from the technique, the window size must be adapted. In Figure 7.30 the same data have been plotted using a moving average of only 100 points, which corresponds to 25 days. As expected, the graph is now much less smooth and the improving trends is some what more difficult to see. The spikes of the mean prediction error do however seem to smaller after 4000 data points.

Interestingly to see, the smaller window size of the moving average of the WPPT data in Figure 7.31, reveals spikes that can also be found on in the quantile prediction plot. The spikes do not match perfectly, but it seems to support that two methods tend to agree on their power predictions.

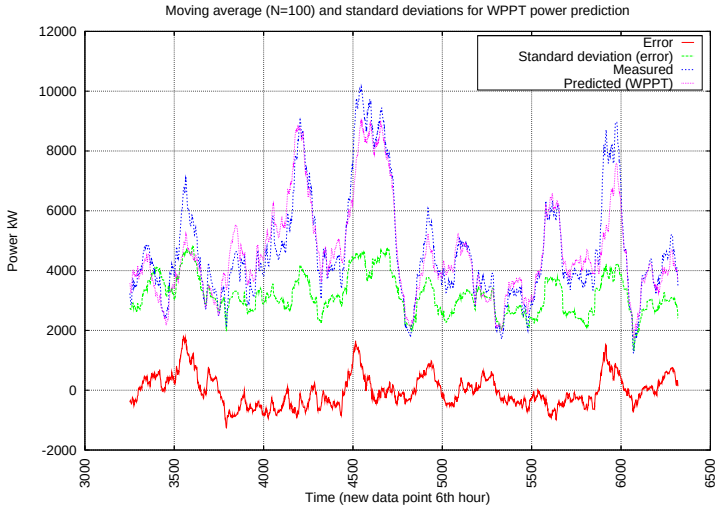

Figure 7.31: The moving averages from WPPT with a window size of 100 points or 25 days.

Testing the multiple variables went smoothly except for a single minor bug in the program that was revealed by selecting too small bins. In terms of improved quality of predictions, it is hard to say anything definitive from these test since it was completely different than the previous prediction reliability tests. Another test was run with only wind speed as explanatory variable, and the results were almost identical. The good thing is that the wind direction did not damage the predictions, but it could be shown that it had helped.

Many parameters were adjusted to get better results, but in the end, the initial and most simple setup offered just as good prediction results as more complicated setups with more variables, bins and knots. More frequent updates of the predictor were also tested, but the improvements could not justify the additional computational time required[12].

Problems related to *data thinning* where encountered at the tests with three explanatory variables and the limited amount of data. Some bins with an unlikely combination of explanatory variable segments can be very hard to fill, so the test designer needs to be aware of this. With the penalty based updating

[12]The odd update frequency of once every 168 data points where chosen from the idea that it would be one week, but since the data arrives four and not 24 times a day, the period corresponds to 42 days. Later it was realized that it did not matter and the 168 hour period was kept.

algorithm, a good strategy is to have less, but bigger bins.

Chapter 8

Discussion

8.1 Introduction

The results from performance tests and validations have been discussed in the context in which they were presented. Consequently, in this chapter, the results will be summarized and discussed in relation to each other. Furthermore, new light will be shed on the quality of the results, as well as on various factors that may have influenced the results.

This discussion has been divided up into three main sections. First, the performance aspects will be presented. Following this, the results in terms of quality and predictions will be discussed. The final section will cover the current and future perspective of the implemented program.

8.2 Implementation and Performance

One of the key goals of the translation of the adaptive quantile regression program from Matlab to C was an improvement in performance. At the time when the program was transferred to C, the performance was significantly better than in the Matlab implementation. However, throughout this thesis, the program

specifications have changed so dramatically that it is very difficult to make any direct comparisons.

8.2.1 Comparison With Matlab Implementation

According to the test in Section 6.5.2, the C implementation performs roughly 10 times better than the Matlab version, but there are some key differences between the tests, which make them not directly comparable although qualitatively a winner can still be found. The test machines used are different, but as stated, it is very unlikely that the difference in computational performance between the two machines are greater than the speed difference observed.

The data sets used are also different, but it should not have a large effect on the performance, because they are both produced from WPPT and the same explanatory variables have been chosen. If any, the gain from either of the data sets should be fairly small and nowhere near the factor 10 seen in speed difference.

Apart from the different test machines, two key differences do contribute to unequal timing of the implementations, and these are the fact that the Matlab implementation does not compute the initial solution and neither does it generate the splines within the measured time. The initial starting point and the splines are generated in R before being run in the Matlab adaptive quantile regression script. Since these timing differences are in favor of the Matlab implementation, even if the test machine used for the C implementation is faster, it is still safe to assume that the C implementation is significantly faster than the Matlab implementation.

8.2.2 Event Driven Approach

The C implementation has been changed so that it can be run in an event driven environment, which is how WPPT operates, and this is in direct contrast to the Matlab implementation. The transformation to an event driven program, as opposed to a regular Matlab script, has not limited the performance gains, and has opened up for interesting experiments in terms of how often the predictor needs updating and what quantile regression method should be used.

The event driven approach did, however, limit the possibilities for parallelizing the algorithm. To obtain optimal speedup from parallelization, it is good to carry out the parallelization on as high a level as possible. It would require

breaking with the requirements of WPPT if the parallelization should occur on a higher level. The optimal parallelization would be obtained by letting WPPT do it and perhaps change the update predictor function call so that multiple instances can easier be run in parallel.

8.2.3 Interior Point Method and Simplex

During the program development, there has been very little doubt that the simplex method performs much better than the interior point method in the case of the continious adaptive updates.

The comparative performance tests showed that the simplex method was indeed much faster than the interior point method, but at the cost of having to update the predictor each time new data enters the module. The tests were carried out under controlled circumstances, on a unloaded system and with identical setups. The reason why the simplex method is faster is because each iteration requires less computations than the interior point method and the reuse of the optimal solution from the previous run significantly lowers the number of iterations required to find the new optimal solution.

Even though the simplex method does not work very well with infrequent updates, it might be possible to obtain good results by calculating the vertex from the previous predictor rather than by relying on the vertex being delivered from the last run. This might potentially lead to even better performance since the less computationally expensive simplex method may be used in setups where the predictor is only updated weekly.

8.3 Evaluation of Quantile Quality

As part of the validation tests, the two measures, reliability and skill score, have been used in different ways to provide a picture of how well the calculated quantiles describe the active data in the system and how well the prediction describes the data to come.

8.3.1 Adaptivity and Quantile Reliability

Testing the reliability of the calculated quantiles has been an interesting exercise. The study on reliability of the data within the program throughout a time series showed that the reliability can be compromised by having too many zeros in the system. The distribution of predictions is not very uniform and a large number of zero predictions exist. This is well within the theory of power prediction, but for estimating the prediction uncertainty using linear quantile regression and splines, the zeros cause the system to become unreliable.

The first coefficient of the splines is the intercept. If there is a large concentration of errors at zero in the system, the optimization algorithm will try to limit the loss function, which causes the intercept to be placed either so it does not correctly describe what is going on at zero or so the offset makes it difficult to describe the rest of the distribution.

The problem with zeros leads to an adjustment of the algorithm in which a penalty was assigned to duplicate points. This effectively cured the problem with reliability, but other methods might work just as well. The penalty based replacement of old values is a general solution, which should perform well on data with frequently occurring inputs different from zero.

Another method for handling this would be to treat zero predictions as a special case using another model. The model would not need to be very advanced, but it would take the load off the rest of the system. The penalty based model does, however, still present a significant improvement towards the quality and robustness of the system and should not be removed unless something replaces it. The penalty rules might be slightly adjusted so point age would have a relatively larger importance, but this is highly application specific.

8.3.2 Predictions

The results from the tests where the predictor was used to predict the uncertainty showed a surprising lack of relation between predictor update frequency and quality of the probabilistic uncertainty. Even if the predictor was only updated once, the skill score was not significantly worse. This is very good news in terms of using the program as a module in WPPT, since the computational requirements can be overcome by updating the predictor less frequently.

A number of factors which may contribute to this lack of relation between predictor age and quantile quality have already been mentioned. If the prediction

errors are fairly symmetrical or around zero, the quantiles may show good performance in terms of reliability over the long run, due to the averaging nature of the test. The moving average skill score does, however, seem to contradict that this is caused by averaging effects, because the trend can be followed through most of the time series.

Yet another possibility is that the relatively large uncertainty or standard deviation of the wind power prediction, which can be seen in Section 7.7, limits the precision of the predicted uncertainty quantiles and thus diminishes the need for very advanced updating schemes to catch minor variations.

In order to determine whether or not this is true, much larger data sets need to be analyzed. The data set for most of the tests consisted of approximately two years worth of data, but it was shown that it took almost a year for the adaptive algorithm to settle and provide optimal uncertainty predictions. It would be very interesting to see if the uncertainty predictions could get better by using other and more explanatory variables.

8.3.3 Selecting Other Explanatory Variables

If wind predictions had been used instead of power predictions, the problem with reliability might never have been identified. The problems encountered during evaluation of the reliability were mainly caused by the threshold in the power curve model. It would be interesting to observe if the uncertainty predictions could be improved if the variables available to WPPT were used instead of just the predicted power. This is indeed possible with the implemented algorithm, but there is a risk of data thinning if too many bins are defined in each explanatory variable. The penalty based replacement model does remove some of the reasons for using bins, so perhaps tests with many explanatory variables can be performed easily by using only few bins.

The reason for using additional explanatory variables apart from the predicted power is that if there is a correlation between prediction error and some other factor, this is impossible to include if only the predicted power is used to explain this. For instance if the power prediction reads zero because the wind prediction is just below a certain threshold, the chances for power to be produced would be larger than if the power prediction was zero due to zero wind. To the system only looking at predicted power, this information is completely lost.

Another very interesting parameter to take into account would be the uncertainty of weather predictions. If the company doing the weather forecasts had a similar system to the one implemented in this thesis for estimating probabilis-

tic uncertainties in their predictions, these uncertainty numbers could be taken into account when estimating the power prediction uncertainty, and this should ultimately lead to more accurate results.

8.3.4 Treatment of Horizons

It is well known, and has also been shown in the skill score tests, that the uncertainty of power predictions are related to the horizon in which they are predicted. From the Matlab implementation, the idea of splitting the data set in the number of horizons was continued, but there are some basic problems with this when other explanatory variables are considered. Treating each horizon as a separate data set offers the possibility of calculating a unique quantile predictor for each horizon and thus adjust for the increased uncertainty with longer prediction horizons.

There are some unfortunate limitations to this strategy, because the system, with its isolated data sets for each horizon, is oblivious to time skewed events in the incoming data. If for instance the weather prediction agency continually had a one-hour time skew in wind predictions for a particular windmill farm location, the program would only be able to adjust for this as an unknown error, but if system was structured substantially differently, it could potentially be able to adjust uncertainty estimates with this time skew.

Another event which goes completely unnoticed in the system is the relation between points in the time series - if a whole day with good power production is predicted, the program only sees this as a time series of independently "good power" predictions. It is clear that if there is a 50% power production at time t, and the predicted power at $t + 1$ hours is also 50%, this prediction will be more likely to be correct, than a 50% prediction one hour after having had 0% power production for a while.

Changing the behavior of the program to incorporate capabilities for time skewed events and relations between the individual points in the predicted time series would require large rewrites of the code - in fact it would most likely be a completely different program.

Simply including the horizon as an explanatory variable could, however, easily be done and it might provide some additional information. If weather forecasts were used as explanatory variables, the age of these could also be interesting to include. If the strict separations between horizons were dropped, the problem with the need for a large training set might also be removed, because points from all horizons would enter the same model.

8.4 Current and Future Perspective

The adaptive quantile regression module has been very interesting to work with. Especially the fact that this algorithm is not only theoretical, but can actually be tailored to be used in present applications, makes it very interesting.

8.4.1 WPPT Implementation

One of the long term goals of the translation of the adaptive quantile was to be able to implement the program into the WPPT application. The initial preparations for this process have been done, but due to time constraints, the actual integration will be carried out after completion of the thesis.

The module has been designed to be compatible with the event driven nature of WPPT, so the integration should be mainly a matter of defining events in WPPT which links to the methods in the adaptive quantile regression module.

With the module in place, WPPT should be able to provide better uncertainty estimates, which again will lead to more confidence in the predictions it performs. It will be extremely interesting to see how well the adaptive quantile uncertainty predictions perform on live data.

8.4.2 Alternative Uses For the Module

Probabilistic forecasts can be used in practically any situation where forecasts are quantifiable. This includes basic things such as water, power and heat supply and demand. Stock market forecasts might also benefit from better uncertainty measures.

The implemented program may not be an option for Wall Street anytime soon, but as a tool for general adaptive forecasts it offers an interesting alternative to static models.

If an effort was made to make a nice user friendly general purpose frontend, graphical or command line oriented, the program could be an interesting alternative to other statistical models available. With the adaptive behavior, the program would be perfect for analyzing timeseries with prediction in mind.

Chapter 9

Conclusion

The purpose of thesis was to efficiently implement adaptive quantile regression in a native C program, based on a previous Matlab implementation. The two main reasons for this translation was to improve performance and make the algorithm available for the commercial WPPT application.

The implementation in C was very successful in terms of versatility and performance. The work required to translate the program was quite substantial because C, being a low level programming language, does not offer the same array of mathematical operations found in programs like Matlab and R.

Integrated interior point method and spline generation have made the program very versatile and easy to use as a module in other applications. The driver programs written for the test cases only require a single text file with configuration in the top section and data below as input. A general purpose driver program for the module could make this adaptive quantile regression module very easy to use for non-programmers.

It has been shown very clearly that the performance of simplex is significantly better than the interior point method for adaptive quantile regression, since the knowledge of last optimum can be used to reduce the number of iterations needed to adjust towards the new optimum. Using simplex, however, does require the predictor to be updated very frequently, since the points defining the optimal

vertex may be thrown away thus rendering the previous solution invalid.

The performance techniques used throughout the development of the module have been described in detail in a theoretical context. Throughout development, these techniques have been an integral part of the design and have had a significant influence on achieving the shown performance. Performance libraries have been the method of choice for optimization, since using these ensures that the best performance is obtained on any given platform with an efficient performance library implementation. It was argued that the event driven modular approach was not ideal for function level parallelism and a speedup of 1.5 was recorded on a dual core machine, which may not have had the memory throughput needed for running this program at full speed on both cores.

A comparison between the simplex implementation done in Matlab and in C has argumented that the new implementation performs significantly better than the Matlab script. This does, however, need to be validated due to the different conditions in which the tests were run.

In response to problems with reliabilities, the adaptive algorithm was extended with a highly efficient penalty-based selection criteria. This proved to effectively remove problems related to the large number of data points located at zero. This system is a general solution to problems arising from peaks in distributions at any given value.

The quality of predictions has been tested in several ways, and the general picture points towards the adaptive algorithm being quite successful, but the update frequency does apparently not need to be hourly or even daily. This is good news for using the module in an online environment, where predictions and observations may not be available in a constant stream, but in daily or weekly bundles, which effectively prohibits constant updates.

The adaptive quantile regression algorithm has a significant potential for being a simple yet valuable addition to conventional static models for predicting uncertainties or as it was shown to actually calculate the prediction itself. The focus of the program's usage in this thesis has been on wind power production, but there is nothing in the algorithm or module that restricts it from being used on any other given problem with similar properties.

Appendix A

Code Listing

A.1 Periodic Splines - R implementation

The following R function has been written by Henrik Aalborg Nielsen, Lektor / Associate Professor at DTU. The function was found at http://www.biostat.wustl.edu/

```
bs.per.ek <- function(x, period=24, degree=3, nknots=6)
{
    ## EXPERIMENTAL
    ## The strange name means: B-spline, periodic, equidistant knots
    ## x must lie in [0 ; period) (if x == period it should be recorded as 0).

    if(any(x < 0 | x >= period)) stop("x out of bounds")

    if(degree < 1) stop("degree must a positive integer")

    ## Construct knots
    knots.x <- seq(0, period, length=nknots+1)
    delta <- diff(knots.x[1:2])
    knots.left <- seq(from = -delta, by = -delta, length = degree)
```

```
    knots.right <- seq(from = period + delta, by = delta, length = degree)
    knots <- c(knots.left, knots.x, knots.right)

    ## Design matrix:
    X <- spline.des(x=x, knots=knots, ord=degree+1)$design
    X[, 1:degree] <- X[, 1:degree] + X[, ncol(X) - (degree-1):0]
    X <- X[, 1:(ncol(X)-degree)]

    return(X)
}
```

Bibliography

[1] Jacek Gondzio and Andreas Grothey. Reoptimization with the primal-dual interior point method. *SIAM Journal on Optimization*, 13(3):842–864, 2002.

[2] John L. Hennesay and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2006.

[3] Advanced Micro Devices Inc. *3DNow! Technology Manual*, 2000. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf.

[4] Intel. *How to Choose between Hardware and Software Prefetch on 32-Bit Intel Architecture*. http://www.intel.com/cd/ids/developer/asmo-na/eng/82917.htm.

[5] Roger Koenker. Website: Quantile regression. http://www.econ.uiuc.edu/~roger/research/rq/rq.html.

[6] Roger Koenker. *Quantile Regression*. Cambridge University Press, 2005.

[7] SUN Microsystems. *An Overview of UltraSPARC III Cu (rev 1.1)*, 2003. http://www.sun.com/processors/whitepapers/USIIICuoverview.pdf.

[8] Jan Kloppenborg Møller. Time-adaptive quantile regression (matlab implementation and documentation). http://www2.imm.dtu.dk/~jkm/.

[9] Jan Kloppenborg Møller. Modeling of uncertainty in wind energy forecast. Master's thesis, The Technical University of Denmark (DTU), 2006.

[10] Jan Kloppenborg Møller, Henrik Aalborg Nielsen, and Henrik Madsen. Time-adaptive quantile regression. *Computational Statistics and Data Analysis*, 2006 (submitted).

[11] Hans Bruun Nielsen. Cubic splines. Technical University of Denmark, IMM http://www2.imm.dtu.dk/~hbn/publ/, 1998.

[12] Hans Bruun Nielsen. Algorithms for linear optimization. Technical University of Denmark, IMM http://www2.imm.dtu.dk/~hbn/publ/, 1999.

[13] Pierre Pinson, Henrik Aalborg Nielsen, Jan Kloppenborg Møller, Henrik Madsen, and George Kariniotakis. Nonparametric probabilistic forecasts of wind power: required properties and evaluation. *Wind Energy*, 2007.