Algorithms for Adaptive Quantile Regression

- and a Matlab Implementation

Jan Kloppenborg Møller, Henrik Aalborg Nielsen and Henrik Madsen
Informatics and Mathematical Modelling
Richard Pedersens Plads
DTU-Bygning 321
2800 Lyngby
Denmark

August 8, 2006

Contents

Chapter 1

Introduction

This presentation go through the algorithms needed to implement time adaptive quantile regression. Quantile regression was first introduced by Koenker in [4]. Quantile regression is a linear optimization problem and in [7] combine quantile regression and the simplex method to an adaptive method for quantile regression.

The algorithms presented in this presentation is also presented in [7], but these are presented in much more detail here and the scope is here to point out how the actual implementation in Matlab (see `http://www.mathworks.com`) was performed, while [7] argue that the method give great improvements in the performance of the presented quantile models. The algorithms will be presented in details along with the actual Matlab implementations. The Matlab implementations are given in Appendix A, and include some comments that should help to understand the code.

Chapter 2 explains the setting of the problem and give an overview of the algorithm, but not directly as it is implemented. Chapter 3 goes through the Matlab implementation line by line, and explain what is going on in each line.

The algorithm presented is is based on the simplex algorithm, see [2]. This is well suited for "*warm start*" algorithms (see [8] p. 75), where there exists knowledge about a solution which is believed to be close to the optimal solution. It is however not well suited to find a solution if no prior knowledge is available. This implementation therefore assumes that a solution to the quantile regression problem is given at the time point where the simulation start. Such a solution can be found by e.g. the "`rq`" command in "R" (see http://www.r-project.org).

Simulation refers to the fact that the algorithms are implemented such that simulation of an online setting based on known data sets can be performed. This makes the implementation suited for testing the algorithm with different data sets. Chapter 3 point out what is important in an online setting and what is not.

Before the adaptive algorithms and the linear programming setting is presented, a short presentation of quantile regression is given.

1.1 Quantile Regression

The quantile regression problem as presented in [4] and [5] is a linear model

$$\begin{aligned} \mathbf{y} &= Q(\mathbf{X};\tau) + \mathbf{r} && (1.1) \\ &= \mathbf{X}\boldsymbol{\beta}(\tau) + \mathbf{r} && (1.2) \end{aligned}$$

where $\mathbf{X}$ is a design matrix with observations of explanatory variables and $\mathbf{y}$ is a vector of observations of the response variable. The loss function for this problem is

$$\rho_\tau(r_i) = \begin{cases} \tau r_i & , \quad r_i \geq 0 \\ (\tau - 1) r_i & , \quad r_i < 0 \end{cases} \tag{1.3}$$

The best estimate $\hat{\boldsymbol{\beta}}$ of $\boldsymbol{\beta}$ is

$$\hat{\boldsymbol{\beta}}(\tau) = \arg\min_{\beta} \sum_{i=1}^{N} \rho_\tau(r_i) = \arg\min_{\boldsymbol{\beta}} S(\boldsymbol{\beta};\tau,\mathbf{r}) \tag{1.4}$$

This is a linear programming problem and the solution to this is a quantile. It can be shown that a solution to such a problem is

$$\hat{\boldsymbol{\beta}}(\tau) = \mathbf{X}(h)^{-1}\mathbf{y}(h) \tag{1.5}$$

where h is an index set and $\mathbf{X}(h)$ refers to rows of the matrix $\mathbf{X}$. The example given below illustrate how the linear setting presented above can be used to approximate nonlinear models, and give the setting in which the adaptive method was developed.

1.2 An Example

The algorithms presented here are given in a general setting and will work for all kind of problems, where the systems are believed to change slowly over time. A system that is known to have such properties is wind power prediction systems, this example will build a static model for such a system.

More example of quantile regression models in the wind power case is presented in [7] where the force of the developed adaptive method is illustrated with the wind power example.

The data presented come from a wind power plant near Samsø in Denmark, the data set consist of prediction error and prediction of power production from WPPT (Wind Power Prediction Tool) see [6], and a number of meteorological data from DMI (Danish Meteorological Institute).

The aim is now to model quantiles of the prediction error as a function of predicted power and meteorological forecasts. The quantiles given as a function of these variables is not expected to be linear, therefore two approximations is now made to get to the setting of a linear model. The first approximation is that the effects are assumed to be additive such that the quantile models can be written as

$$Q(\mathbf{x};\tau) = \alpha(\tau) + \sum_{j=1}^{p} f_j(x_j,\tau) \tag{1.6}$$

where x_j is an explanatory variable, and α is a common intercept. The functions f_j is restricted to ensure uniqueness of the solution, see [3]. The functions f_j are now approximated with spline basis functions such that

$$f_j(x_j) = \sum_{k=1}^{n_j} b_{j,k}(x_j)\beta_{j,k} \tag{1.7}$$

where $b_{j,k}$ are spline basis functions and $\beta_{j,k}$ are the coefficients to estimate, with this the model is linear in the coefficients $\beta_{j,k}$ and α. The explanatory variables for the model presented here is

`pow.fc`: Forecasted power production by WPPT

`horizon`: The prediction horizon for the forecasted power, between 18 and 36 hours in the presented data set.

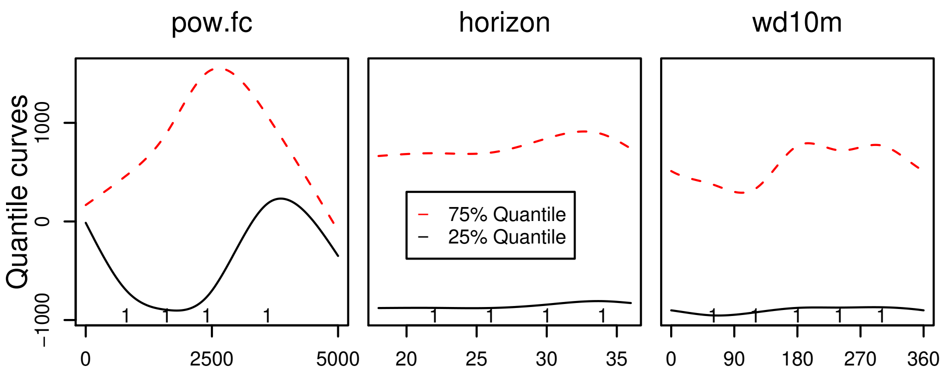

Figure 1.1: The figure show a additive quantile regression model with splines. The rugs on the first axis mark the knots for spline basis functions.

`wd10m`: The wind direction 10 meters above ground level.

With this the quantile model is

$$\begin{aligned} Q([\texttt{pow.fc}, \texttt{horizon}, \texttt{wd10m}]; \tau) &= \alpha(\tau) + \sum_{k=1}^{n_1} b_{1,k}(\texttt{pow.fc})\beta_{1,k} \\ &+ \sum_{k=1}^{n_2} b_{2,k}(\texttt{horizon})\beta_{2,k} + \sum_{k=1}^{n_1} b_{3,k}(\texttt{wd10m})\beta_{3,k} \end{aligned} \tag{1.8}$$

$b_{1,k}$ and $b_{2,k}$ are natural spline basis functions, which are forced through zero at the left boundary knot for the spline basis functions. $b_{3,k}$ is periodic spline basis function with the integral over one period equal to zero. This construction ensure the uniqueness pointed out above. The knots for the spline basis functions is indicated on the first axis of the plots in Figure 1.1. With these knots we have $n_1 = n_2 = n_3 = 5$ and the total number of degrees of freedom for the model is 16.

The effect in of the different components is shown in Figure 1.1. The figure illustrate how nonlinear effects can be approximated with a linear model. This construction motivate the updating procedure described in Chapter 2.

Chapter 2

Adaptive Quantile Regression

This Chapter will present the algorithms that are implemented. The chapter will focus on an overview and avoid to many details, proofs are also avoided, for a treatment of these aspects see [7] and [5]. The linear optimization formulation of the quantile regression problem is central for the presented algorithms, this is therefore presented in the first section and the following section will present the algorithms.

2.1 The Linear Optimization Formulation

The linear programming formulation of the quantile regression problem outlined Section 1.1 is

$$\min\{\tau \mathbf{1}^T \mathbf{r}^+ + (1-\tau)\mathbf{1}^T \mathbf{r}^- : \mathbf{X}\beta + \mathbf{r}^+ - \mathbf{r}^- = \mathbf{y}, (\mathbf{r}^+, \mathbf{r}^-) \in \mathbb{R}_0^{2n}, \beta \in \mathbb{R}^K\}$$

with $r_i^+ = I(r_i \geq 0)r_i$ and $r_i^- = -I(r_i \leq 0)r_i$. In a more compact notation this is

$$\min\{\mathbf{c}^T \mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{y}, (\mathbf{r}^+, \mathbf{r}^-) \in \mathbb{R}_0^{2n}, \beta \in \mathbb{R}^K\} \tag{2.1}$$

where

$$\mathbf{c} = \begin{bmatrix} \mathbf{0}_K \\ \tau \mathbf{e} \\ (1-\tau)\mathbf{e} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} \beta \\ \mathbf{r}^+ \\ \mathbf{r}^- \end{bmatrix} \quad \mathbf{A} = [\mathbf{X}, \mathbf{I}, -\mathbf{I}] \tag{2.2}$$

at least $n + K$ elements in $\mathbf{x}$ must be zero since these are the active constraints. These elements must be in the vector

$$\begin{bmatrix} \mathbf{r}^+ \\ \mathbf{r}^- \end{bmatrix} \tag{2.3}$$

which implies that there are K elements of $\mathbf{r}$ which are zero. The indices of these are collected in the index set h. With this $\mathbf{x}$ can be split into two vectors by the index sets $\mathcal{B}$ and $\mathcal{C}$ s.t. $\mathbf{x}_\mathcal{B} \geq 0$ and $\mathbf{x}_\mathcal{C} = 0$. $\mathbf{x}_\mathcal{B}$ can be written

$$\mathbf{x}_\mathcal{B} = \begin{bmatrix} \beta \\ |\mathbf{r}(\bar{h})| \end{bmatrix} \tag{2.4}$$

$\mathbf{A}$ is split into columns multiplied with $\mathbf{x}_\mathcal{B} \geq 0$ and $\mathbf{x}_\mathcal{C} = 0$, respectively, and these two parts are denoted $\mathbf{B}$ and $\mathbf{C}$. By interchanging rows in $\mathbf{B}$ the constraints at a solution can be written as

$$\mathbf{B}\mathbf{x}_\mathcal{B} = \begin{bmatrix} \mathbf{X}(h) & \mathbf{0} \\ \mathbf{X}(\bar{h}) & \mathbf{P} \end{bmatrix} \mathbf{x}_\mathcal{B} = \begin{bmatrix} \mathbf{y}(h) \\ \mathbf{y}(\bar{h}) \end{bmatrix} \tag{2.5}$$

where $\mathbf{P}$ is a diagonal matrix with diagonal elements

$$\mathbf{p} = \operatorname{sign}(\mathbf{r}(\bar{h})) = [I(r_i \geq 0)] - [I(r_i < 0)] \tag{2.6}$$

This is the notation needed for a description of the algorithm.

2.2 The Algorithms

The algorithm now uses knowledge about the solution at time t to find the solution at time $t + 1$. This is done with the simplex method and the updating algorithms presented below.

2.2.1 The Simplex Method for Quantile Regression

The simplex algorithm assumes that we are at a vertex, so we must have a method for getting to a vertex before starting the simplex algorithm. Here a solution from the "`rq`" command in "R" is used.

The main components for the simplex algorithm is

$$\begin{array}{lcl@{\quad;\quad}lcl} \mathbf{d} & = & \mathbf{c}_\mathcal{C} - \mathbf{C}^T\mathbf{g} & \mathbf{g} & = & \mathbf{B}^{-T}\mathbf{c}_\mathcal{B} \\ \mathbf{h} & = & \mathbf{B}^{-1}\mathbf{C}_{:,s} & \alpha & = & \min\{\sigma_1, ..., \sigma_m\} \end{array} \tag{2.7}$$

with σ_j given by

$$\sigma_j = \begin{cases} (\mathbf{x}_{\mathcal{B}}^{(k)})_j/h_j & \text{if} \quad h_j > 0 \\ \infty & \text{if} \quad h_j \leq 0 \end{cases} \tag{2.8}$$

If $\mathbf{d} \geq \mathbf{0}$ then the solution is optimal. If $\mathbf{x}^{(k)}$ is not optimal then choose a negative element d_s in $\mathbf{d}$, and change one element $(\mathbf{x}_{\mathcal{C}})_s$ of $\mathbf{x}_{\mathcal{C}}$, while keeping the other elements at zero. The basic variables, i.e. $\mathbf{x}_{\mathcal{B}}$, are changed in direction $\mathbf{h} = \mathbf{B}^{-1}\mathbf{C}_{:,s}$, $\mathbf{x}_{\mathcal{B}}$ is changed in this direction until we meet a new vertex. The objective function $\mathbf{c}^T\mathbf{x}$ is decreased by αd_s.

If $\alpha = \infty$ then the problem is unbounded. This can however not happen in the case of quantile regression, see e.g. [7]. α can be zero and then the objective function is not improved, but the step should be taken anyway since we could move to a position with a decent direction and an $\alpha > 0$. $\mathbf{x}_{\mathcal{B}}$ is now changed in two steps

$$\begin{aligned} \mathbf{x}_{\mathcal{B}}^{(k+1)} &= \mathbf{x}_{\mathcal{B}}^{(k)} - \alpha\mathbf{h} && (2.9) \\ (\mathbf{x}_{\mathcal{B}}^{(k+1)})_q &= \alpha && (2.10) \end{aligned}$$

where $q = \arg\min_j \sigma_j$. Further the q'th element of $\mathcal{B}$ is swapped with the s'th of $\mathcal{C}$ and the algorithm starts over again. If α is zero, then it can happen that we move back and forth between two vertices with equal objective functions, so we should keep track where we have been and then be sure not to go back.

The expensive part of the simplex algorithm is to calculate the inverse of $\mathbf{B}$. However in the special case of quantile regression it is possible to write $\mathbf{B}^{-1}$ as products of known matrices and the inverse of $\mathbf{X}(h)$. As stated $\mathbf{B}$ can be written as

$$\mathbf{B} = \begin{bmatrix} \mathbf{X}(h) & \mathbf{0} \\ \mathbf{X}(\bar{h}) & \mathbf{P} \end{bmatrix} \tag{2.11}$$

this lead (see [**?**]) to

$$\mathbf{B}^{-1} = \begin{bmatrix} \mathbf{X}(h)^{-1} & \mathbf{0} \\ -\mathbf{P}\mathbf{X}(\bar{h})\mathbf{X}(h)^{-1} & \mathbf{P} \end{bmatrix} \tag{2.12}$$

This is very important if the number of observations is large. The formulation above makes it possible to calculate the inverse of $\mathbf{B}$ by calculating the inverse of $\mathbf{X}(h)$, matrix multiplication with $\mathbf{X}(\bar{h})$ and element-wise multiplication with the vector $\mathbf{p} = \mathrm{diag}(\mathbf{P})$.

The algorithm described below is the same as the one described in [8], but the specialties w.r.t. quantile regression is used and described for the algorithm. The algorithm assumes that we are at a vertex, i.e. we have $\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1}\mathbf{y}$ and $\mathbf{x}_\mathcal{C} = \mathbf{0}$. An overview of the algorithm is given first and we go through each step below.

The simplex algorithm:

1. Compute $\mathbf{d}$. If $\mathbf{d} \geq \mathbf{0}$ stop $\mathbf{x}$ is optimal.
   Otherwise choose s s.t. $d_s < 0$
2. Compute $\mathbf{h} = \mathbf{B}^{-1}\mathbf{A}_{:,\mathcal{C}(s)}$. If $\mathbf{h} \leq \mathbf{0}$, stop the problem is unbounded.
3. Compute α and choose q such that $\sigma_q = \alpha$
4. Swap $\mathcal{B}(q)$ and $\mathcal{C}(s)$ and set $\mathbf{x}_\mathcal{B} := \mathbf{x}_\mathcal{B} - \alpha\mathbf{h}$; $(\mathbf{x}_\mathcal{B})_q := \alpha$

Step 1

In general the number of directions to calculate is equal to the number of elements in $\mathcal{C}$. This number is $N + K$, but the only directions which can be decent directions in the case of quantile regression is elements corresponding to h. It is therefore sufficient to examine $2K$ directions, corresponding to moving elements of $\mathbf{r}(h)$ in a positive or negative direction.

$\mathbf{d}$ is given by $\mathbf{d} = \mathbf{c}_\mathcal{C} - \mathbf{C}^T\mathbf{g}$ with $\mathbf{g} = \mathbf{B}^{-T}\mathbf{c}_\mathcal{B}$. If we have ordered $\mathbf{C}$ in the same way as $\mathbf{B}$ then the structure is

$$\mathbf{C} = \begin{bmatrix} \mathbf{I}_K & -\mathbf{I}_K & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\mathbf{P} \end{bmatrix} = \begin{bmatrix} \mathbf{P}_C & \mathbf{0} \\ \mathbf{0} & -\mathbf{P} \end{bmatrix} \tag{2.13}$$

Therefore we only need the first K elements of $\mathbf{g}$, which are

$$\begin{aligned} \mathbf{g}(h) &= \mathbf{B}^{-T}(1:K)\mathbf{c}_\mathcal{C} && (2.14)\\ &= [\mathbf{X}(h)^{-T} \quad - (\mathbf{P}\mathbf{X}(\bar{h})\mathbf{X}(h)^{-1})^T]\mathbf{c}_\mathcal{B} && (2.15)\\ &= -\mathbf{X}(h)^{-T}\mathbf{X}(\bar{h})^T\mathbf{P}\rho_\tau(\mathrm{sign}(\mathbf{r}(\bar{h}))) && (2.16) \end{aligned}$$

Further more $\mathbf{d}$ is given by

$$\mathbf{d} = \begin{bmatrix} \tau\mathbf{e}_K - \mathbf{g}(h) \\ (1-\tau)\mathbf{e}_K + \mathbf{g}(h) \end{bmatrix} \tag{2.17}$$

$\mathbf{d}_{K+1:2K} = \mathbf{e}_K - \mathbf{d}_{1:K}$, $d_1 < 0$ implies $d_{K+1} > 1$. This gives us at most K decent directions, and in the optimal solution we have $0 \leq \mathbf{d} \leq 1$.

There are different ways to choose s. One approach is to choose s as the steepest direction. Since $\mathbf{h}$ and therefore α depends on the direction we take, this does not guarantee that we get the greatest improvement in the objective function.

The approach chosen here is to compute $\mathbf{h}$ in all decent directions and then choose s s.t. $\alpha_s d_s$ is minimal, which is possible here because only a very limited number of decent directions exist.

Step one is completed by choosing $\tilde{s} = \{s | d_s < 0\}$ and $\mathbf{P}_s = \mathbf{P}_{C,:,\tilde{s}}$. I.e. $\tilde{s}$ is an index set.

Step 2

Compute $\mathbf{h}$ according to the strategy in step one, i.e. $\mathbf{h}$ will be a matrix with each column being equal to one of the first K columns in $\mathbf{B}^{-1}$ times the sign of the residual produced by going in this direction. Set

$$\tilde{s}_i = \begin{cases} \tilde{s}_i & \text{if} \quad \tilde{s}_i \leq K \\ \tilde{s}_i - K & \text{if} \quad \tilde{s}_i > K \end{cases} \tag{2.18}$$

and $\mathbf{h} = \mathbf{B}^{-1}_{:,\tilde{s}} \mathbf{P}_s$.

Step 3

The choice of α is selected to prevent any of the variables to pass to the infeasible region. In our case this means that a residual can not change sign without passing through the index set h. In this problem α can in theory not be ∞, since the problem is bounded. If this is returned anyway, then the algorithm terminates without a solution and go directly to the updating procedure described below with the old solution.

This should possibly be handled in some other way in an online implementation.

Step 4

This is just what is stated in the overview.

Back to step 1

Two changes or modifications of the simplex algorithm described above are implemented. The first change is that a maximum of simplex steps is set. This is set to 24. This number is quite arbitrary, but for the stable models very few iterations are observed where the number of simplex steps is equal to 24. This might be restricted to the example treated in [**?**] and should probably also be treated in an other way in an online implementation.

The second change is that in each step the condition number of the matrix $\mathbf{X}(h^{(t+1)})$ is calculated and the simplex step will only be taken if this number is less than a specified value. It is not obvious what such a value should be, but as long as we just have some value, the algorithm will not terminate. It will however not update the solution in this step either. Here the maximum condition number is set to 10^6. This is not a very elegant way to treat this and other ways should be considered.

This conclude the presentation of the simplex algorithm and the updating procedure is now presented.

2.2.2 The Updating Procedure

The updating procedure for the design matrix can be established in many different ways. The one used here is to let one observation leave the design matrix when a new observation become available. The decision of which observation to leave the design matrix is based on a variable, not necessarily in the design matrix. Other procedures could be implemented, e.g. updating could be done only after a number of new observations become available.

How the updating is done or implemented does not effect the simplex algorithm used on the updated problem. The steps in the updating procedure are given below and then each of the steps are explained in some more detail. It is assumed that we have the solution at time t.

1. Decide which row, l, that must leave the design matrix.
2. If $l \in h_t$ take one simplex step s.t. $l \notin h_t$.
3. Update $\mathbf{X}$, $\mathbf{P}$, h and $\mathbf{c}_{\mathcal{B}}$
4. Perform the simplex steps needed to get to the optimal solution at time $t+1$.

Step one is the choice that can be made here, while step two have to be implemented somehow.

Step 1

The algorithm implemented here assumes that a vector $\mathbf{z}$ with elements corresponding to the rows of the design matrix is given. E.g. as in Section 1.2 $\mathbf{z}$ could be forecasted power production and the rows of $\mathbf{X}$ spline basis functions of $\mathbf{z}$ (and other explanatory variables).

The variable $\mathbf{z}$ is now the decision variable. A new observation will belong to some bin that have been defined before hand, this could e.g. be the knots for the spline basis functions in direction of forecasted power in the example presented in 1.2. The oldest observation from this bin is now marked for deletion. The row number of the leaving variable is denoted l.

The number of elements allowed in each bin is defined before hand (the same number for all bins). If the bin of the leaving variable is not filled then step 3 (and 2) are skipped, this can of course not happen after some running time.

Step 2

If the leaving variable is in h, then we can not remove it, since the solution depend on the inverse of $\mathbf{X}(h)$. Therefore if $l \in h$ we perform one simplex step with a new objective function, where the loss on r_l is set equal to zero. This corresponds to setting $\mathbf{c}_{\{l,l+N\}} = \mathbf{0}$. This result is a change of two elements in the simplex vector $\mathbf{d}$. The elements we have to change are the elements corresponding to l. Denote these $\mathbf{d}^l$, then

$$\mathbf{d}^l = \begin{bmatrix} -\mathbf{g}(l) \\ \mathbf{g}(l) \end{bmatrix} \tag{2.19}$$

Therefore only the two elements of $\mathbf{d}$ corresponding to l have to be calculated. This determines the decent direction and gives us s. If $\mathbf{g}(l) = 0$ then this will not be a decent direction. The simplex step should be taken anyway, but the direction is then not important. With this we can find $\mathbf{h}$ and thereby σ and q, which was the variable that had to enter h. Having swapped l and q we are ready to update the design matrix and the other matrices needed for the simplex algorithm.

In the implementation presented here, this step will be taken without checking the condition number of $\mathbf{X}(h)$; this means that the algorithm can break down in some situations. So this should be dealt with in an online implementation.

Step 3

Let $\Omega = \{1, 2, ..., N\}$ where N is the number of observations in $\mathbf{X}$ and $r_{t+1} = y_{t+1} - \mathbf{x}_{t+1}^T \hat{\beta}_t$ then the updated versions of the simplex variables will be

$$\mathbf{X}^{(t+1)} = \begin{bmatrix} \mathbf{X}_{\Omega \setminus l}^{(t)} \\ \mathbf{x}_{t+1} \end{bmatrix} \tag{2.20}$$

$$\mathbf{P}^{(t+1)} = \begin{bmatrix} \mathbf{P}_{\Omega \setminus l}^{(t)} \\ \mathrm{sign}(r_{t+1}) \end{bmatrix} \tag{2.21}$$

$$\mathbf{c}_{\mathcal{B}}^{(t+1)} = \begin{bmatrix} \mathbf{c}_{\mathcal{B};\Omega \setminus l}^{(t)} \\ \rho_\tau(r_{t+1}) \end{bmatrix} \tag{2.22}$$

For $j = 1, 2, ..., K$, we update the index set h as

$$h_j^{t+1} = \begin{cases} h_j^t & \text{if} \quad h_j < l \\ h_j^t - 1 & \text{if} \quad h_j > l \end{cases} \tag{2.23}$$

With the elements needed for the simplex algorithm in place, we go on to Step 4.

Step 4

This is the steps explained in Section 2.2.1.

The simplex algorithm will sum up small numerical errors in each step. It is therefore necessary to deal with this once in a while. [8] suggest an algorithm for doing this in the general LO case. In the quantile regression setting we just recalculate $\hat{\beta}$ from $\mathbf{X}(h)$ after each time step and find $\mathbf{r}$, $\mathbf{P}$, etc. from this solution.

With the main steps of the algorithm explained, this presentation will go on with a detailed presentation of the Matlab implementation. This will be done by going through the most important functions line by line.

Chapter 3

The Matlab implementation Line by Line

This chapter go through the most important code line by line. All rutines are listed in Appendix A, where the commented code is listed. This chapter will focus on the parts of the code that is central to an online implementation. The main rutine is `rq_simplex_final.m`, see Section A.1, this rutine calls the other rutines. This chapter will go through this rutine line by line, and the rutines that this calls when these are important for an online implemantation.

The next section provide an overview of this chapter, and emphasize which sections that are important for an online implemantation.

3.1 The Fast Guide to this Chapter

The purpose of this section is to provide an overview of the algorithms and to emphasize which parts of the algorithms that runs like in a real time situation. This information is contained in the commented algorithm summary, given below

1. **Initialize the variables needed for the simplex algorithm** (Section 3.2.1 -3.2.2). This part is not important for the online implementation, but the sections do provide some of the notation used in later sections.

2. `while:` **"Data to simulate"** (Section 3.2.3). This part of the program run very much like in an online setting, with the variables being updated just as would be the case in an online situation.

   (a) **Update the design matrix and other variables needed for the simplex algorithm** (Section 3.3). This is done in the same way as would be done in an online situation, i.e. one new observation become available at the time. The updating procedure return the initial guess based on available data at time t, along with the available data at time $t+1$.

   (b) `while:` **"Solution not optimal"** (Section 3.4 -3.5). This section use the optimal solution at time t, to get the optimal solution at time $t+1$. I.e. the simplex algorithm given in Section 2.2.1.

      i. **Step 1-3 of the simplex algorithm** (Section 3.4). This makes ready to perform the simplex step.

      ii. **Step 4 of the simplex algorithm** (Section 3.5). The actual simplex step is performed if the solution is not optimal.

   (c) **The solution is recorded**. The algorithm store solutions and update the design matrix etc.

3.2 The Main Function (rq_simplex_final.m)

The main function and the functions it calls, are all given in Appendix A. This chapter will in addition to the main function go through the implemented updating procedure and the simplex algorithm.

The call for the main function in Matlab is

```
[N,BETA,GAIN,Ld,Rny,Mx,Re,CON1,T]...
    =rq_simplex_final(X,IX,Iy,Iex,r,beta,n,tau,bins,n_in_bin)
```

The code is given in Appendix A.

3.2.1 The Input Variables

To initialize the variables needed for the simplex algorithm, a solution to the quantile regression problem should be given as input.

The matrix `X` contains the design matrix, the response variable and a variable used for the updating procedure. The design matrix is the columns of `X` defined in the index set `IX`, the response variable is the columns of `X` defined by `Iy` and the column used for the updating procedure is defined by the index `Iex`.

`r` and `beta` denote the residuals and the best estimate of a solution based on the first `n` elements in `X`, `tau` is the required quantile.

In the description of the algorithm the following notation will be used

$$\mathbf{X}^{(k)} \quad : \quad \text{The design matrix at time } k \tag{3.1}$$
$$\mathbf{y}^{(k)} \quad : \quad \text{The vector of response variables corresponding to } \mathbf{X}^{(k)} \tag{3.2}$$
$$\hat{\boldsymbol{\beta}}^{(\boldsymbol{k})} \quad : \quad \text{The solution based on } \mathbf{X}^{(k)} \text{ and } \mathbf{y}^{(k)} \tag{3.3}$$

In this notation the start point of the algorithm is

$$\mathbf{X}^{(0)} = \texttt{X(IX,1:n)} \tag{3.4}$$
$$\mathbf{y}^{(0)} = \texttt{X(Iy,1:n)} \tag{3.5}$$

`beta`$= \hat{\beta}^{(0)}$ is the solution to the quantile regression problem based on $\mathbf{X}^{(0)}$, $\mathbf{y}^{(0)}$, and `tau`$= \tau$. The residuals are `r`$= \mathbf{r}^{(0)} = \mathbf{y}^{(0)} - \mathbf{X}^{(0)}\hat{\beta}^{(0)}$.

The solution is given to the function in terms of $\mathbf{r}^{(0)}$ and $\hat{\beta}^{(0)}$. This is not exactly what the simplex algorithm, the simplex algorithm needs the index set h, therefore a Matlab function which finds h on the basis of $\hat{\boldsymbol{\beta}}^{(0)} \in \mathbb{R}^K$ and $\mathbf{r}^{(0)}$ is implemented. The important note is here that this function should have at least K zeros.

The input variables `bins` and `n_in_bins` are used for the updating procedure, `bins` is a vector that defines a partition of a sample space covering the set of numbers in `X(1:end,Iex)`. `n_in_bins` is a natural number defining the number of elements allowed in each bin.

3.2.2 Initialization

The first part of the code is

```
GAIN=0;
Rny=0;
Ld=0;
mx=0;
```

```
T=0;
Mx=0;
N=zeros(1,2);
CON1=0;
CON2=[0,0];
BETA=beta';
```

This is just a trivial initializations of the output variables. An explanation of the is given in the code for the Matlab function `rq_simplex_final` given in Appendix A.1.

The next lines determines the dimension, `K`, of the problem (i.e. $\hat{\boldsymbol{\beta}} \in \mathbb{R}^K$) and set `Xny` s.t. `Xny(1:end,IX)`$= \mathbf{X}^{(0)}$, `Xny` will contain $\mathbf{X}^{(k)}$ through out the program.

```
K=length(beta);
Xny=X(1:n,1:end);
H=zeros(1,K);
tolmx=10^-15;
j=0;
LX=length(X(1:end,Iy));
i=2;
k=0;
```

`H` is used to track the simplex algorithm, i.e. to make sure that the algorithm does not go back and fourth between two solutions, `tolmx` determine when to deal with infeasible points and when just to set these equal to zero. `LX` is the stop criteria for the simulation, `i` is an internal counter. `k` is the time counter for the system and, this is what could be referred to as real time.

The best estimate of $\boldsymbol{\beta}^{(0)}(\tau)$ can be written as $\boldsymbol{\beta}^{(0)}(\tau) = \mathbf{X}^{(0)}(h)^{-1}\mathbf{y}^{(0)}(h)$ or equivalent $\mathbf{r}^{(0)}(h) = \mathbf{0}$. Therefore the index set h can be found by checking which elements of $\mathbf{r}^{(0)}$ is zero. If the number of zero elements in $\mathbf{r}_0$ is larger than K, then h is not given directly by this.

The function `rq_initialiser_final.m` deals with more than K zeros in $\mathbf{r}^{(0)}$, since is not really a part of the adaptive method it will not be dealt with further, the code is given in Section A.2. The important thing to note here is that `r` must contain at least K zeros and that there is an index set h s.t. $\mathbf{r}_{(0)}(h) = 0$ and $\text{rank}\mathbf{X}_{(0)}(h) = K$. The call to `rq_initialiser_final.m` is

```
[xB,Ih,Ihc,P]=rq_initialiser_final(Xny(1:end,IX),r,beta,n)
```

based on the first n observations this function will return the index sets, and vectors to give the matrix formulation, with the notation from Chapter 1. These are

$$\begin{aligned}
\texttt{Ih} &= h && (3.6)\\
\texttt{Ihc} &= \bar{h} && (3.7)\\
\texttt{xB} &= \mathbf{x}_{\mathcal{B}} && (3.8)\\
\texttt{P} &= \mathbf{p} = \text{sign}(\mathbf{r}(\bar{h})) && (3.9)
\end{aligned}$$

This presentation does not go through this function since it does not relate to an online situation.

With everything needed for the simplex algorithm in place the simulation can begin.

3.2.3 The Adaptive Simulation

The simulation starts with the `while` loop declared by

```
while i+n<LX
    k=k+1
    t=cputime;
    Re(k)=sum(P<0)/n;
    mx=min(xB(K+1:end));
```

The while loop will continue until the last element in the observation matrix `X`. `k` is the time counter i.e. simulation time. `t` is the CPU time used by the algorithm in each step. `Re` is the reliability in each step, and if the algorithm runs without problems this number is very close to τ; therefore `Re` can be used as a control of the algorithm. `mx` is used as a feasibility check if `mx` is less than zero then the solution contain infeasible points, this can happen because of small errors being summed up in each simplex step.

If the simplex algorithm have pushed the solution to an infeasible point something should be done. In a general LO problem, infeasible point have to be dealt with, by some kind of modified simplex algorithm. In the case of unrestricted quantile regression an infeasible point is a residual which is accounted for by the wrong sign, so basically it is just a matter of changing the sign of one or more residuals. What is done here is just to recalculate $\mathbf{x}_{\mathcal{B}}$ and $\mathbf{p}$ under h and $\bar{h}$. This is done with by the statement

```
if j>0 & mx<-tolmx;
    [xB,P]=rq_purify_final(xB,Ih,Ihc,P,K,Xny(1:end,IX),Xny(1:end,Iy));
    mx=min(xB(K+1:end));
end
```

The code for the Matlab function `rq_purify_final` is not given here but it can be seen in Appendix A.3. It basically use h and $\bar{h}$ and then recalculate $\mathbf{x}_{\mathcal{B}}$ and $\mathbf{p}$ as

$$\begin{aligned} \boldsymbol{\beta} &= \mathbf{X}(h)^{-1}\mathbf{y}(h) && (3.10)\\ \mathbf{r}(\bar{h}) &= \mathbf{y}(h) - \mathbf{X}(\bar{h})\boldsymbol{\beta} && (3.11)\\ \mathbf{x}_{\mathcal{B}} &= \begin{bmatrix} \boldsymbol{\beta} \\ |\mathbf{r}(\bar{h})| \end{bmatrix} && (3.12) \end{aligned}$$

The next lines

```
Mx(k)=mx;
j=0;
beta=xB(1:K);
BETA=[BETA;beta'];
```

Store the solution in `BETA`, the simplex counter `j` is reset and the minimum of the basic solution is stored in `Mx(k)`; `Mx` is only used for analyzing the algorithm.

3.3 Updating the Variables: `opdatering_final.m`

When the optimal solution at time k is found the algorithm updates the design matrix and the variables needed for the simplex algorithm. This is done with the call

```
[Ih,Ihc,xB,Xny,Rny,P,n,i]=opdatering_final(X,Xny,IX,Iy,Iex,...
    Ih,Ihc,beta,Rny,K,n,xB,P,tau,i,bins,n_in_bin);
```

the updating procedure is a central part of the simulation and it is of course also important for an online implementation. This function will therefore be presented here line by line. The function is given in Appendix A.4.

The function starts with

```
Xny=[Xny;X(n+i-1,1:end)];
Index=1:length(Xny(1:end-1,Iy));
```

The first line do the first part of the updating, a new observation is added to the observation matrix `Xny`, `Index` contain the index of the old observation matrix. The observation matrix is ordered according to age of the observations s.t. that the oldest observation is first.

The updating procedure is such that the observation to leave the matrix `Xny` should be the oldest observation in the same bin as the observation just provided. The bins were given in the direction of an explanatory variable in column `Iex`. This may or may not be a part of the design matrix.

The lines:

```
j=1;
while Xny(end,Iex)>bins(j)
    j=j+1;
end
j=j-1;
In=Index(Xny(1:end-1,Iex)>bins(j) & Xny(1:end-1,Iex)<=bins(j+1));
Leav=min(In);
```

mark the row `Leav` from the observation matrix for deletion. The while loop determines what bin the new observation belongs to. `In` is the index of the observations in this bin, `Leav` is the oldest observation in this bin.

If the bin is already full, then the observation found above should leave the observation matrix, if the leaving observation in h then it can not be deleted. Therefore if this is the case one simplex step to ensure the leaving observation to leave h will be taken.

This can be viewed as a simplex step with zero loss on the residual corresponding to leaving observation.

The lines

```
[minIL,Inmin]=min(abs(Ih-Leav));
if minIL==0 & length(In)==n_in_bin
```

Determine if this simplex step have to be taken. If the leaving variable is in h and the bin was full then this modified step should be taken. As can be imagined this is a rare event when there are many observations in the observation matrix.

In an ordinary simplex step for quantile regression, there would be $2K$ directions to choose from when we want to perform a simplex step. Here the leaving variable have already been chosen and there are two candidates for the best directions, corresponding to moving this residual in a positive or negative direction. The lines

```
cB=P<0+P.*tau;
invXh=Xny(Ih,IX)^-1;
g=-(P.*(Xny(Ihc,IX)*invXh(1:end,Inmin)))'*cB;
```

give the first calculation for the simplex algorithm. `cB` is the cost vector. `invXh` is the inverse of $\mathbf{X}^{(k)}(h)$. In the normal simplex set up for quantile regression `g` would be a vector and on the basis of this vector the decent direction would be chosen. Here this part have already been passed and `g` is a number and the sign of this determine the decent direction. The change from simplex algorithm is that the index `Inmin` (see Section 3.4 below) have already been chosen. This correspond to s or $s+K$ in the simplex algorithm.

The vector that determines which direction to go in is

$$\mathbf{d} = \begin{bmatrix} -g \\ g \end{bmatrix} \tag{3.13}$$

The direction is $\mathbf{h} = \mathbf{B}^{-1}\mathbf{C}_{:,s}$, but a column of $\mathbf{C}$ is a vector with all but one element equal to zero, and this element will be equal to either one or minus one. So $\mathbf{h}$ is one column of $\mathbf{B}^{-1}$ multiplied with either one or minus one. The first element of $\mathbf{d}$ corresponds to moving the residual in a positive direction, and the second element to moving this residual in a negative direction. If g is negative then the column of $\mathbf{B}^{-1}$, should be multiplied with minus one and if g is positive then the column $\mathbf{B}^{-1}$ should be multiplied with one. This is done in

```
h=[invXh(1:end,Inmin);-P.*(Xny(Ihc,IX)*invXh(1:end,Inmin))]*sign(g);
```

with the decent direction in place the next point is to determine how far the solution can be moved in this direction. There is no restrictions on how far the first K elements of $\mathbf{x}_{\mathcal{B}}$ can be moved (this is $\boldsymbol{\beta}$). Therefore this number is needed only for the rest of the directions. This is calculated in the lines

```
    sigma=zeros(1,n-K);
    hm=h(K+1:end);
    xBm=xB(K+1:end);
    xBm(xBm<0)=0;
    sigma(hm>10^-10)=xBm(hm>10^-10)./hm(hm>10^-10);
    sigma(hm<=10^-10)=Inf;
    [alpha,q]=min(sigma);
```

`alpha` is the amount the solution can be moved and `q` is the vertex that is met at this point. The tolerance 10^{-10} is quite arbitrary, but some tolerance have to be set since solutions calculated from elements of $\mathbf{h}$ very close to zero cannot really be trusted.

The solution can now be updated by

```
    z=xB-alpha*h;
    Ihm=Ih(Inmin);
    Ih(Inmin)=Ihc(q);
    Ihc(q)=Ihm;
    xB=z;
    xB(q+K)=alpha;
    P(q)=sign(g)+g==0;
```

The last line here just gives the sign of the new residual. With this the solution is such that the variable marked for deletion can actually leave the observation matrix. The next lines order the index sets such that they are in increasing order.

```
    Ih=sort(Ih);
    [Ihc,IndexIhc]=sort(Ihc);
    P=P(IndexIhc);
    xBm=xB(K+1:end);
    xBm=xBm(IndexIhc);
    xB(K+1:end)=xBm;
    beta=xB(1:K);
end
```

Here is a possible bug since the condition number (see Section 2.2.1 for a discussion) for the next matrix $\mathbf{X}(h)$ is not calculated. That this matrix is singular is an extremely rare event, since both $l \in h$ is rare and that the simplex algorithm suggests to go to an singular matrix are rare, but somehow this should of course be dealt with.

Now that the variable marked for deletion can actually leave the design matrix the algorithm can continue with the updating procedure. The first task is to calculate the residual for the new observations. This is done by

```
rny=X(n+i-1,Iy)-X(n+i-1,IX)*beta;
Rny=[Rny,rny];
```

This one step ahead prediction error is also stored and given as output. $\mathbf{x}_{\mathcal{B}}$, $\mathbf{p}$ and $\bar{h}$ is now updated

```
if rny<0
    P=[P;-1];
    xB=[xB;-rny];
else
    P=[P;1];
    xB=[xB;rny];
end
Ihc(end+1)=n+1;
```

With this we have a solution with the new observation in the design matrix. The last task is to remove the leaving variable, if the system have been running long enough then all bins will be filled, but until this happens there is a chance that there is no leaving variable. If this happen then the size of the design matrix is updated; otherwise the step counter `i` is updated.

```
if length(In)==n_in_bin
    i=i+1;
    Stay=ones(length(Ihc),1);
    Stay(Ihc==Leav)=0;
    Stay=Stay==1;
```

The vector `Stay` is a logical vector which is true if the corresponding element should stay in the problem. This is used to update the parameters used for the simplex algorithm.

```
    P=P(Stay);
    Ihc=Ihc(Stay);
    Xny=Xny(sort([Ih,Ihc]),1:end);
    xBm=xB(K+1:end);
    xBm=xBm(Stay);
    xB=xB(1:end-1);
    xB(K+1:end)=xBm;
```

here the variables are updated, and what is left is to update the index sets, such that these fits the new variables.

```
    Ihc(Ihc>Leav)=Ihc(Ihc>Leav)-1;
    Ih(Ih>Leav)=Ih(Ih>Leav)-1;
```

If the bin was not filled then the only thing to do is to update the size of the design matrix.

```
else
    n=n+1;
end
```

with this the method is ready to use the simplex method to look for the best solution given the new design conditions.

3.4 The Simplex Algorithm: rq_simplex_alg_final.m

The call for the simplex algorithm is given with

```
    IH=Ih';
    [CON,s,q,gain,md,alpha,h,IH,cq]=...
        rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny(1:end,IX),IH,P,tau);
```

the first line initializes a matrix used for monitoring where the solution have been. The function rq_simplex_alg_final does not actually perform the simplex step, but it makes ready to perform this. This presentation will go through the function line by line.

The input variables is as defined throughout this chapter, the first lines of the function is

```
invXh=Xny(Ih,1:end)^-1;
cB=(P<0)+P.*tau;
cC=[ones(K,1)*tau;ones(K,1)*(1-tau)];
```

the first line calculate the inverse of $\mathbf{X}(h)$. cB corresponds to the loss of the residuals in such a way that $\texttt{cB}^T|\mathbf{r}(\bar{h})| = \sum_{i\in\bar{h}} \rho_\tau(r_i)$. cC is the loss of moving $\mathbf{r}(h)$ in either a positive or negative direction. The next task is to find the inverse of $\mathbf{B}$ or rather the part of this needed for the simplex step.

```
IB2=-(P*ones(1,K).*Xny(Ihc,1:end))*invXh;
```

IB2 is equal to $-\mathbf{P}\mathbf{X}(\bar{h})\mathbf{X}(h)^{-1}$ and this was exactly what was needed to calculate $\mathbf{g}(h)$. Note that even though $\mathbf{P}$ is a large matrix, this implementation only uses element wise multiplication between two matrices with K columns, and K would normally be small compared to the number of elements in $\mathbf{p}$.

The simplex vectors $\mathbf{g}$ and $\mathbf{d}$ can now be calculated. These are calculated exactly as was shown in Step 1 of the simplex algorithm.

```
g=IB2'*cB;
d=cC-[g;-g];
d(abs(d)<10^-15)=0;
```

Small elements in $\mathbf{d}$ is set equal to zero, which is done to prevent directions which actually have zero gain in the loss function from creating very large estimates of the gain. The problem is that $\boldsymbol{\sigma}$ is created by dividing with elements of $\mathbf{h}$, so this is actually an indirect way of addressing this problem and it is not enough to solve the problem either.

The next lines finds the index set $\tilde{s}$ and the corresponding slopes of the decent directions; this is the last part of step 1.

```
[md ,s]=sort(d);
s=s(md<0);
md=md(md<0);
c=ones(length(s),1);
c(s>K)=-1;
C=diag(c);
```

This conclude what is denoted as step 1 in the simplex algorithm presented in the introduction. The matrix C was what is denoted as $\mathbf{P}_s$ in Chapter 2.

Step two is concluded in the following two lines.

```
s(s>K)=s(s>K)-K;
h=[invXh(1:end,s);IB2(1:end,s)]*C;
```

`h` is now a matrix with each column corresponding to a decent direction.

Step 3 is a little more complicated the algorithm should choose the best direction, but here the problem of stability is central since there is a risk of dividing with zero.

```
alpha=0;
q=0;
```

These lines initialize the variables needed, `alpha` is a vector and `q` is a index set corresponding to `alpha` (taken from $\boldsymbol{\sigma}$) for each decent direction.

$\boldsymbol{\alpha}$ is based on the last $N-K$ elements of the vector $\mathbf{x}_{\mathcal{B}}$. The meaning of $\boldsymbol{\alpha}$ is to measure how far the solution can be moved in a decent direction before it reaches a vertex. The first K elements of $\mathbf{x}_{\mathcal{B}}$ is $\boldsymbol{\beta}$ and these are free parameters, so what is needed to calculate is the last $N-K$ elements of $\mathbf{x}_{\mathcal{B}}$.

```
xm=xB(K+1:end);
xm(xm<0)=0;
hm=h(K+1:end,1:end);
cq=0;
```

If there are elements in the last part of `xB` which are less than zero then these are set equal to zero. Such elements can only occur from numerical errors, but even small negative values here can lead to wrong decisions regarding the direction to go in. `cp` will be a vector containing the signs of the new residual if the corresponding direction is chosen.

The `for` loop below calculates α and q for each direction.

```
for k=1:length(s)
    sigma=xm;
    sigma(hm(1:end,k)>10^-12)=...
        xm(hm(1:end,k)>10^-12)./hm(hm(1:end,k)>10^-12,k);
    sigma(hm(1:end,k)<=10^-12)=Inf;
    [alpha(k),q(k)]=min(sigma);
    cq(k)=c(k);
end
```

The gain in the objective function can now be calculated in each direction.

```
gain=md'.*alpha;
[Mgain, IMgain]=sort(gain);
```

`Mgain` is a vector of the gain in increasing order, such that the best gain is the first element in this vector. `IMgain` contains the indices corresponding to `Mgain`.

If the solution is optimal then $\mathbf{d} \geq \mathbf{0}$, in the construction above this means that $s = \emptyset$. This is checked by the first `if` statement below. If this is the case then `gain` is set equal to one, which will terminate the algorithm, and the simulation will go on to the next update.

```
CON=Inf;
j=0;
if length(gain)==0
    gain=1;
```

`CON` is initialized to ∞, `CON` is a decision variable used in the `while` loop below, the initialization means that the algorithm will visit this the first time. `j` is a step counter to terminate the algorithm.

```
else
    while CON>10^6 & j<length(s)
        j=j+1;
        IhMid=Ih;
        IhMid(s(IMgain(j)))=Ihc(q(IMgain(j)));
        IhMid=sort(IhMid);
```

The point of the `while` loop is to make sure that the algorithm does not take a step to an h that have already been visited or to an h that produce a singular (or close to singular) matrix $\mathbf{X}(h)$. `IhMid` contains the suggestion for the next index set h. In the `if` statement below the condition number of $\mathbf{X}(h)$ in the case that the step is taken is calculated.

```
        if min(sum(abs(IH-IhMid'*ones(1,length(IH(1,1:end))))))==0
            CON=Inf;
        else
```

```
            CON=cond(Xny(IhMid,1:end));
        end
    end
```

The first part of the `if` statement checks if the algorithm have already visited the proposed h, and in this case `CON` is set to ∞, since the step should not be taken in this case. Otherwise the condition number in case of the step is calculated.

When the `while` loop terminates, the only task left is to set the chosen values such that these are ready for the actual simplex step. This is done in the following lines.

```
    s=s(IMgain(j));
    q=q(IMgain(j));
    cq=cq(IMgain(j));
    alpha=alpha(IMgain(j));
    IH=[IH,IhMid'];
    h=h(1:end,IMgain(j));
    gain=gain(IMgain(j));
    md=md(IMgain(j));
end
```

This is the last lines in `rq_simplex_alg_final` and this presentation returns to the main algorithm. The meaning of the variables become clear in the next section.

3.5 The Simplex Step

The next lines in `rq_simplex_final` is

```
    CON1=[CON1;CON];
    while gain<=0 & md<0 & j<24 & CON<10^6
```

The first line stores the condition number of $\mathbf{X}(h)$, this is only used for analysis of the algorithm in its own right. The logic statements determine when the algorithm should terminate. The only reason for `gain` to be there is that it could be set to one in the simplex function described above. The maximal number of simplex steps allowed in each iteration is set to 24. The optimal value of this number will probably depend on the problem, but if it is very high the calibration in `rq_purify_final` should probably be done inside the `while` loop.

```
        GAIN=[GAIN;gain];
        j=j+1;
```

The gain in the objective function in each simplex step is stored in `GAIN` and the simplex counter `j` is updated.

The next lines perform step 4 of the simplex algorithm. I.e. swap elements of h and $\bar{h}$, and update the variables.

```
        z=xB-alpha*h;
        IhM=Ih(s);
        IhcM=Ihc(q);
        Ih(s)=IhcM;
        Ihc(q)=IhM;
        P(q)=cq;
        xB=z;
        xB(q+K)=alpha;
```

with this the simplex step is done. What is left is to order the index set and the variables such that the index sets is in increasing order.

```
        [Ih,IndexIh]=sort(Ih);
        [Ihc, IndexIhc]=sort(Ihc);
        P=P(IndexIhc);
        xBm=xB(K+1:end);
        xBm=xBm(IndexIhc);
        xB(K+1:end)=xBm;
```

The algorithm now go back to step one, which is done with the call to `rq_simplex_alg_final`.

```
        [CON,s,q,gain,md,alpha,h,IH,cq]=...
            rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny(1:end,IX),IH,P,tau);
        CON1=[CON1;CON];
    end
```

When the optimal solution at time k is reached, the algorithm store performance parameters for the adaptive procedure, this are the number of simplex steps and the timing for the time step.

```
    N(k,1)=j;
    N(k,2)=n;
    T(k)=cputime-t;
end
```

The algorithm can now go back to the updating procedure. With this the complete algorithm is now described.

Appendix A

The Matlab functions

A.1 rq_simplex_final.m

```
function [N,BETA,GAIN,Ld,Rny,Mx,Re,CON1,T]...
    =rq_simplex_final(X,IX,Iy,Iex,r,beta,n,tau,bins,n_in_bin)

% Call:
%
% [N,BETA,GAIN,Ld,Rny,Mx,Re,CON1,T]...
%    =rq_simplex_final(X,IX,Iy,Iex,r,beta,n,tau,bins,n_in_bin)
%
% rq_simplex_final calculate the solution to an adaptive simplex
% algorithm for a quantile regression problem. The function use
% knowledge of the solution at time t to calculate the solution at
% time t+1.
%
% The basic idea is that the solution to the quantile regression
% problem can be written as
%
%    y(t)=X(t)'*beta+r(t)
%
% where beta=X(h)^(-1)*y(h) for some indexset h, simplex algorithm
% is now used to calculate the optimal h at time t+1 based on the solution
% at time t. So basically the function  uses the h(t) as a start guesss for
% the sipmplex algorithm to iterate to h(t+1). This function is described
```

```
% in the master thesis "Modelling of Uncertainty in Wind Energy Forecast"
% by Jan K. Moeller,  URL http://www.imm.dtu.dk/pubdb/p.php?4428.
%
%
% Iputs to the function
% X         : The design matrix for the linear quantile regression problem,
%             or rather it contains the design matrix, see the three next
%             input variables.
% IX        : An index set, it refer top the columns of X which is the
%             design matrix.
% Iy        : One index refer to the matrix X, the column Iy of X contains
%             the respons corresponding to the explanatory variables in the
%             design matrix
% Iex       : An index refering to a grouping variable, refer to a column of
%             X this may or may not be a part of the design matrix. This is
%             used in the updating algorithm.
% r         : The residuals from a sulotion to the quantile regression
%             problem based on the first n (see below) elements of X. Such a
%             solution could be obtained by the "rq" method in "R". This is
%             only used to initialize the solution.
% beta      : The solution corresponding to the residuals in the vector r
%             above
% n         : The number of elements in r, i.e y(1:n)=X(1:n,IX)*beta+r(1:n)
% tau       : The required probablity, the solution in r and beta should be
%             based on this probablity
% bins      : A vector defining a partion of an interval which covers all
%             elements in X(1:end,Iex). This is used for the updating
%             procedure of the design matrix. If bins=[-Inf,Inf] the the
%             updating procedure is on gliding window.
% n_in_bin : Number of elements in the bins defined above, this is one
%             number, so the number of elements will be the same in each of
%             the bin after some time.
%
% Remarks:
% This implementatins requires that the number of elements in each bin of
% the initial solution is less than or equal to n_in_bin.
%
% Output:
% N    : The number of simplex steps to get the solutions at time t+1
%         given the solution at time t. This is a vector.
```

```
% BETA : A matrix with each column being the solution to quantile
%        regression problem. Corresponnding to the matrix X(n+1:end,1:end).
% GAIN : This is a parameter used only to analyze the method. It is the
%        gain in the loss function in each simplex step, if this is very
%        large at some points it can be taken as a sign of the problem
%        being ill possed.
% Ld   : This is only used for analyzing the algorithm in its own right. It
%        gives the number of decent directions in each simplex step.
% Rny  : This is the rsidual of the one step ahead prediction of the
%        method.
% Mx   : This is only used for analyzing the algorithm in its own right. It
%        is the minimum of the constraint solution to the simplex
%        formulation. This should be larger than or equal to zero. If this
%        becomes large in absolute value then it is an indication of the
%        algorithm having problems at this point
% Re   : The reliability on the training set for each point along the
%        solution, if everything is good, then this should follow Theorem
%        2.3 in the reference mentioned above. I.e. this is close to tau at
%        all points.
% CON1 : The condition numeber of the matrices X(h(t))
% T    : The time used for each iteration
%
% References:
% [1] J. K. Møller (2006), Modeling of Unceartainty in Wind Energy
%        Forecast. Master Thesis, Informatics and Mathematical Modelling,
%        Technical University of Denmark. Available at
%        http://www.imm.dtu.dk/pubdb/p.php?4428.
%
% [2] H. B. Nielsen (1999), Algorithms for Linear Optimization, an
%       Introduction. Course note for the DTU course "Optimization and Data
%       fitting 2". Available at http://www.imm.dtu.dk/courses/02611/



  % Initialize output vectors and  matrices
GAIN=0;
Rny=0;
Ld=0;
mx=0;
T=0;
```

```
Mx=0;
N=zeros(1,2);
CON1=0;
CON2=[0,0];
BETA=beta';

 % Initialize internal variables for the function

K=length(beta); %K is the number of explanatory variables in X

Xny=X(1:n,1:end); %The design matrix at the starting point for the
                  %algorithm, this will be the observation matrix in the
                  %simulation. So each solution will be based on Xny at the
                  %given time point.

H=zeros(1,K);     %This is used to track solutions that the algorithm
                  %have visited

tolmx=10^-15;     %This is a tolerance to dertermine when an infeasible
                  %point should be fixed and when it should be onsidered
                  %zero.

j=0;              %Counter of the number of simplex steps in each
                  %iteration

LX=length(X(1:end,Iy)); %The stop crterium of the simulation

i=2;                    %Internal counter used by the opdating procedure
                        %decribed below

k=0;                    %Timecounter for the algortihm, k will be the time
                        %from the beginning of the algorithm


 %The function rq_initialiser_final initialize the parameters needed for
 %the simplex algorithm based on the on the solution given as input to the
 %function

[xB,Ih,Ihc,P]=rq_initialiser_final(Xny(1:end,IX),r,beta,n);
```

```
  %The while loop makes the simulation through the data set, in each step
  %either i or n is updated. n is the total number of elements in the
  %design matrix, in the first iterations n will be updated later it will
  %be i.

while i+n<LX
    k=k+1       %The time counter is updated
    t=cputime;   %Zero point for cpu-time


    Re(k)=sum(P<0)/n;     %Reliability at time k
    mx=min(xB(K+1:end));  %The minimum of the basic solution, this should
                          %be larger than zero

    %Infeasible points is taken care of by rq_purify_final

    if j>0 & mx<-tolmx;
        [xB,P]=rq_purify_final(xB,Ih,Ihc,P,K,Xny(1:end,IX),Xny(1:end,Iy));
        mx=min(xB(K+1:end));
    end
    Mx(k)=mx; %The minimum of the basic solution is stored
    j=0;      %Simplex step counter is set to zero to start the new
              %timestep

    beta=xB(1:K); %The solution at time k is ekstracted and stored

    BETA=[BETA;beta'];

    %The design matrix and other variables needed for the simplex
    %procedure is updated

    [Ih,Ihc,xB,Xny,Rny,P,n,i]=opdatering_final(X,Xny,IX,Iy,Iex,...
        Ih,Ihc,beta,Rny,K,n,xB,P,tau,i,bins,n_in_bin);

    %Infeasible points is set equal zero to avoid taken a simplex step in
    %this direction

    IH=Ih'; %The index set h is stored in each sipmlex step, this is done
            %to prevent the solution from going back and furth between
```

```
        %equally good solutions

%Numbers needed to perform the simplex step is calculated in
%rq_simplex_alg_final

[CON,s,q,gain,md,alpha,h,IH,cq]=...
    rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny(1:end,IX),IH,P,tau);
CON1=[CON1;CON]; %The condition number of the matrix X(h(t+1)) is
                 %stored

%The while loop continuie until the optimal solution is reached or one
%of the two stop criterias are violated. These are a maximal number of
%simplex step, and the condition number of the next soltion

while gain<=0 & md<0 & j<24 & CON<10^6
    GAIN=[GAIN;gain]; %The gain for the step in stored
    j=j+1;            %The simplex counter is updated
    z=xB-alpha*h;     %z is the new solution to the problem

    %The index sets defining the solution is updated
    IhM=Ih(s);
    IhcM=Ihc(q);
    Ih(s)=IhcM;
    Ihc(q)=IhM;
    P(q)=cq;    %The sign of the newest residual

    %The basic solution is updated.
    xB=z;
    xB(q+K)=alpha;

    %The indexsets are ordered s.t. they are in increasing order
    [Ih,IndexIh]=sort(Ih);
    [Ihc, IndexIhc]=sort(Ihc);
    P=P(IndexIhc);
    xBm=xB(K+1:end);
    xBm=xBm(IndexIhc);
    xB(K+1:end)=xBm;

    %rq_simplex_alg_final calculate the numbers needed to perform the
    %next simplex step
```

```
        [CON,s,q,gain,md,alpha,h,IH,cq]=...
            rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny(1:end,IX),IH,P,tau);

        CON1=[CON1;CON]; %The condition nember is stored
    end

    N(k,1)=j;       %Number of simplex steps at time k
    N(k,2)=n;       %size of the design matrix at time k
    T(k)=cputime-t; %cpu time used to udate the solution
end
```

A.2 rq_initialiser_final.m

```
function [xB,Ih,Ihc,P]=rq_initialiser_final(X,r,beta,n)
% Call:
%
% [xB,Ih,Ihc,P]=rq_initialiser_final(X,r,beta,K,n)
%
% rq_initialiser_final uses the a design matrix and the residuals from a
% quantile regression problem:
%
% y=X*beta+r
%
% I.e. to give something meaningfull r must contain at least rank(X) zeros.
%
% The input to the function is (with n equal to the number of elements in
% r)
%
% X    : the design matrix or X(1:n,1:end) is the design matrix
% r    : the residuals from the quantile regression problem
% beta : the solution to the quantile regression problem
%
% The output from the function is
%
% xB   : A vector containing [beta abs(r(Ihc))]
% Ih   : The indexset s.t. beta=X(Ih)^(-1)*y(Ih), i.e. r(Ih)=0
% Ihc  : The complement of Ih, Ihc={1,2,...,n}\Ih
% P    : A vector with the elements equal sign(r(Ihc)), (with sign(0)=1)
%
% The above give the formulation
%
% [X(Ih)   0 ]*xB = y
% [X(Ihc) P*I]
%
% for an explanation of this see [1].
%
% References:
% [1] J. K. Møller (2006), Modeling of Unceartainty in Wind Energy
%        Forecast. Master Thesis, Informatics and Mathematical Modelling,
%        Technical University of Denmark. Available at
```

```
%        http://www.imm.dtu.dk/pubdb/p.php?4428.
%
% [2] H. B. Nielsen (1999), Algorithms for Linear Optimization, an
%       Introduction. Course note for the DTU course "Optimization and Data
%       fitting 2". Available at http://www.imm.dtu.dk/courses/02611/


Index=1:n;   % Index is the indexset s.t. r=r(Index)

 % if the number of zero elements in r is equal rank(X), then the work is
 % essensially done, otherwise the if statement will find the index set Ih
 % s.t. X(Ih)*beta=y(Ih) and X(Ih)^(-1)*y(Ih)=beta, the important note being
 % that X(Ih) have an inverse. This is by using the LU transform of a
 % non-quadratic matrix.

if length(r(r==0))>length(beta)
    Lr0=sum(r==0);
    [rs,Irs]=sort(abs(r));
    Irs=Irs(1:Lr0);
    Xh=X(Irs,1:end);
    [L,U,P]=lu(Xh);
    In=1:Lr0;
    for i=length(beta)+1:Lr0
        rI(i-length(beta))=In(P(i,1:end)==1);
    end
    r(Irs(rI))=10^-15;
end

 % The indexsets, P and xB is defined
Ih=Index(r==0);
Ihc=Index(abs(r)>0);
P=sign(r(Ihc));
r(abs(r)<10^-15)=0;
xB=[beta;abs(r(Ihc))];
```

A.3 rq_purify_final.m

```
function [xB,P]=rq_purify_final(xB,Ih,Ihc,P,K,Xny,yny)
%
% Call:
%
%     [xB,P]=rq_purify_final(xB,Ih,Ihc,P,K,Xny,yny)
%
% This function takes care of infeasible points in a simplex formulation
% of a quantile regression problem. The underlying assumption in this
% function is that there are no restrictions in the in the problem.
%
% The updating can therefore be done by recalculatin all residuals and
% coefficients.
%
% The assumptions is further that we are in a position s.t.
%
%  Xny*Xny(Ih)^(-1)*yny(Ih)=yny+residuals
%
% K=rank(Xny)
% Input as in rq_simplex_final
invXh=Xny(Ih,1:end)^-1;
xB=[invXh*yny(Ih);yny(Ihc)-Xny(Ihc,1:end)*invXh*yny(Ih)];
P=sign(xB(K+1:end));
P(P==0)=ones(sum(P==0),1);
xB(K+1:end)=abs(xB(K+1:end));
```

A.4 opdatering_final.m

```
function [Ih,Ihc,xB,Xny,Rny,P,n,i]=opdatering_final(X,Xny,IX,Iy,Iex,Ih,...
    Ihc,beta,Rny,K,n,xB,P,tau,i,bins,n_in_bin)

% Call:
%
% [Ih,Ihc,xB,Xny,Rny,P,n,i]=opdatering_final(X,Xny,IX,Iy,Iex,Ih,...
%     Ihc,beta,Rny,K,n,xB,P,tau,i,bins,n_in_bin)
%
% This function update the elements needed for the adaptive simplex
% procedure in rq_simplex_final.
%
% Input    :
% X        : A matrix with all the data to be simulated, the columns IX are
%            the columns needeed use in the quantile regression, column is
%            the corresponding respons and column Iex is used for updating
%            in several bins.
% Xny      : The "design matrix" that have to be updated, Xny(1:end,IX) is
%            the actual design matrix, see "X" for the other columns
% IX       : Xny(1:end,IX) is the design matrix
% Iy       : Xny(1:end,Iy) is the respons
% Iex      : Xny(1:end,Iex) is the an explanatory variable used for updating
%            in several bins
% Ih       : An index set s.t. Xny(Ih,IX)^(-1)*Xny(Ih,Iy)=beta
% Ihc      : The complement of Ih, i.e. Ihc={1,2,...,n}\Ih where n is the
%            number of rows in Xny.
% beta     : the coefficients in the quantile regression problem
% Rny      : A vector containing previous one step ahead prediction errors
% K        : K=rank(X(1:end,IX))=rank(Xny(Ih,IX))
% n        : The number of observations in the design matrix Xny
% xB       : A vector s.t xB=[beta' abs(r(Ihc))']', where r is the residuals
%           for the quantile regression model, corresponding to Xny
% P        : P=sign(r(Ihc)), with sign(0)=1
% tau      : The required probability in the quantile regressioin problem
% i        : Internal counter for the rq methods s.t.
%            Xny(end,1:end)=X(n+i-2,1:end)
% bins     : A partion of the sample space of the variable in X(Iex, ), to
%            make sure this is the case bins should be a vector with
```

```
%            -Inf=bins(1)<bins(2)<...<bins(end)=Inf
% n_in_bin : the number of elements allowed in the bins (one number)
%
% The output is updated versions of (some of) the inputs.
%
% The algorithm follows the algorithms described in [1].
%
% References:
% [1] J. K. Møller (2006), Modeling of Unceartainty in Wind Energy
%        Forecast. Master Thesis, Informatics and Mathematical Modelling,
%        Technical University of Denmark. Available at
%        http://www.imm.dtu.dk/pubdb/p.php?4428.
%
% [2] H. B. Nielsen (1999), Algorithms for Linear Optimization, an
%       Introduction. Course note for the DTU course "Optimization and Data
%       fitting 2". Available at http://www.imm.dtu.dk/courses/02611/

Xny=[Xny;X(n+i-1,1:end)]; % The newest avaliable observation is added to
                          % the design matrix

Index=1:length(Xny(1:end-1,Iy)); % An indexset corresponding to the old
                                 % design matrix is created


j=1;                        % The while loop determines a number j s.t.
while Xny(end,Iex)>bins(j)  % the new explanatory variable is in the
    j=j+1;                  % interval (bins(j);bins(j+1)]
end
j=j-1;

 % The index set of the design matrix in this interval is determined and
 % the oldest element in the interval is marked for deletion.
In=Index(Xny(1:end-1,Iex)>bins(j) & Xny(1:end-1,Iex)<=bins(j+1));
Leav=min(In);

  % minIL is a marker if minIL=0 then the leaving variable is in Ih, and it
  % cannot be removed before a simplex step have been taken away from this.
[minIL,Inmin]=min(abs(Ih-Leav));

  % The if statement take one simplex step with an objective function with
```

```
  % zero loss on the leaving variable. This simplex step ensure that the
  % row marked for deletion can leave the design matrix.
if minIL==0 & length(In)==n_in_bin
    cB=P<0+P.*tau;
    invXh=Xny(Ih,IX)^-1;
    g=-(P.*(Xny(Ihc,IX)*invXh(1:end,Inmin)))'*cB;
    h=[invXh(1:end,Inmin);-P.*(Xny(Ihc,IX)*invXh(1:end,Inmin))]*sign(g);
    sigma=zeros(1,n-K);
    hm=h(K+1:end);
    xBm=xB(K+1:end);
    xBm(xBm<0)=0;
    sigma(hm>10^-10)=xBm(hm>10^-10)./hm(hm>10^-10);
    sigma(hm<=10^-10)=Inf;
    [alpha,q]=min(sigma);

    z=xB-alpha*h;
    Ihm=Ih(Inmin);
    Ih(Inmin)=Ihc(q);
    Ihc(q)=Ihm;

    xB=z;
    xB(q+K)=alpha;

    P(q)=sign(g)+g==0;
    Ih=sort(Ih);
    [Ihc,IndexIhc]=sort(Ihc);
    P=P(IndexIhc);
    xBm=xB(K+1:end);
    xBm=xBm(IndexIhc);
    xB(K+1:end)=xBm;
    beta=xB(1:K);
end

 % The residual of the one step ahead predition is calculated
rny=X(n+i-1,Iy)-X(n+i-1,IX)*beta;
Rny=[Rny,rny];

 % This residual is used to update P and xB
if rny<0
    P=[P;-1];
```

```
    xB=[xB;-rny];
else
    P=[P;1];
    xB=[xB;rny];
end
Ihc(end+1)=n+1;

 % If the bin is filled then the updating of the indexsets is a little
 % tricky since not all of the indexset is updated, if the bin is not
 % filled then updating is straight forward. See below.
if length(In)==n_in_bin
    i=i+1;
    Stay=ones(length(Ihc),1);
    Stay(Ihc==Leav)=0;
    Stay=Stay==1;
    P=P(Stay);
    Ihc=Ihc(Stay);
    Xny=Xny(sort([Ih,Ihc]),1:end);
    xBm=xB(K+1:end);
    xBm=xBm(Stay);
    xB=xB(1:end-1);
    xB(K+1:end)=xBm;
    Ihc(Ihc>Leav)=Ihc(Ihc>Leav)-1;
    Ih(Ih>Leav)=Ih(Ih>Leav)-1;
else
    n=n+1;
end
```

A.5 rq_simplex_alg_final.m

```
function [CON,s,q,gain,md,alpha,h,IH,cq]=...
    rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny,IH,P,tau)
% Call:
%
%      [CON,s,q,gain,md,alpha,h,IH,cq]=...
%           rq_simplex_alg_final(Ih,Ihc,n,K,xB,Xny,IH,P,tau)
%
% rq_simplex_alg_final calculate the variables needed for a simplex
% algorithm for quantile regression. The function does not do the actual
% simplex step, this is done in the .m function rq_simplex_final.m. The
% algorithm is based on the recipe in [2], and a description of the
% simplex algorithm for quantile regresion is given in [1].
%
% Input :
% Ih    : An index set that defines the guess on solution of the quantile
%         regression problem.
% Ihc   : The compliment of the index set Ih.
% n     : The number of observations in the design matrix.
% K     : The number of explanatory variables.
% xB    : The basic variables this is [beta, r(Ihc)*sign(r(Ihc))]'.
% Xny   : The design matrix which contain the the explanatory variables on
%         the training set
% IH    : A matrix containing indexsets which have been visited by the
%         algorithm in rq_simplex_final, the indexsets in this can not be
%         returned.
% P     : The sign of the nonzero residuals, i.e. r(Ihc).
% tau   : The quantile to fit to.
%
% Output:
% CON   : The sondition number of the suggested X(Ih)'
% s     : The index to leave the Ih
% q     : The index to enter Ih
% gain  : The improvement in the loss function from this step
% md    : The slope of the loss function from this direction.
% alpha : The amount that the solution can be moved in direction h.
% h     : The directional vector that the solution is changed in.
% IH    : An updated version of the input
```

```
% cq    : The sign of the new residual, if the solution is moved in
%         direction h.
%
% References:
% [1] J. K. Møller (2006), Modeling of Unceartainty in Wind Energy
%        Forecast. Master Thesis, Informatics and Mathematical Modelling,
%        Technical University of Denmark. Available at
%        http://www.imm.dtu.dk/pubdb/p.php?4428.
%
% [2] H. B. Nielsen (1999), Algorithms for Linear Optimization, an
%       Introduction. Course note for the DTU course "Optimization and Data
%       fitting 2". Available at http://www.imm.dtu.dk/courses/02611/

invXh=Xny(Ih,1:end)^-1;
cB=(P<0)+P.*tau;
cC=[ones(K,1)*tau;ones(K,1)*(1-tau)];

IB2=-(P*ones(1,K).*Xny(Ihc,1:end))*invXh;

g=IB2'*cB;
d=cC-[g;-g];
d(abs(d)<10^-15)=0;
[md ,s]=sort(d);
s=s(md<0);
md=md(md<0);
c=ones(length(s),1);
c(s>K)=-1;
C=diag(c);
s(s>K)=s(s>K)-K;
h=[invXh(1:end,s);IB2(1:end,s)]*C;

alpha=0;
q=0;
xm=xB(K+1:end);
xm(xm<0)=0;
hm=h(K+1:end,1:end);
cq=0;
for k=1:length(s)
    sigma=xm;
    sigma(hm(1:end,k)>10^-12)=...
```

```
        xm(hm(1:end,k)>10^-12)./hm(hm(1:end,k)>10^-12,k);
    sigma(hm(1:end,k)<=10^-12)=Inf;
    [alpha(k),q(k)]=min(sigma);
    cq(k)=c(k);
end
gain=md'.*alpha;
[Mgain, IMgain]=sort(gain);
CON=Inf;
j=0;
if length(gain)==0
    gain=1;
else
    while CON>10^6 & j<length(s)
        j=j+1;
        IhMid=Ih;
        IhMid(s(IMgain(j)))=Ihc(q(IMgain(j)));
        IhMid=sort(IhMid);
        if min(sum(abs(IH-IhMid'*ones(1,length(IH(1,1:end))))))==0
            CON=Inf;
        else
            CON=cond(Xny(IhMid,1:end));
        end
    end
    s=s(IMgain(j));
    q=q(IMgain(j));
    cq=cq(IMgain(j));
    alpha=alpha(IMgain(j));
    IH=[IH,IhMid'];
    h=h(1:end,IMgain(j));
    gain=gain(IMgain(j));
    md=md(IMgain(j));
end
```

Bibliography

[1] Carl de Boor *A Practical Guide to Splines* Springer-Verlag 1978

[2] Vašek Chvátal (1983) *Linear Programming* W. H. Freeman and Company.

[3] T.J. Hastie and R.J Tibshirani *Genralized Additive Models* Chapman and Hall

[4] Roger Koenker, Gilbert Bassett Jr *Regression Quantile* Econometrica, Vol. 46, No 1, Jan 1978 (33-50).

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

[6] Henrik Madsen, Henrik Aalborg Nielsen, Torben Skov Nielsen *A Toll for Predicting the Wind Power Production of Off-Shore Wind Plants*

[7] Jan K. Møller (2006) *Modeling of Uncertainty in Wind Energy Forecast*. Master Thesis, Informatics and Mathematical Modelling, Technical University of Denmark. URL:`http://www.imm.dtu.dk/pubdb/p.php?4428`.

[8] Hans Bruun Nielsen *Algorithms for Linear Optimization, an Introduction* (1999) Course note for the DTU course *Optimimization and Data fitting 2*