Project Thesis in Study Programme
Mechanical Engineering

Implementation of a Parallel Nonnegative Least Squares Solver using ScaLAPACK

Project Thesis

Author: Christina Schwarz, 22567070

Examiners: Prof. Dr.-Ing. habil. P. Steinmann
PD Dr.-Ing. J. Mergheim

Advisor: Benjamin Brands, M.Sc.

Begin: 06.05.2020

End: 05.10.2020

Ich versichere, dass ich die Arbeit ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

.....	.....
Ort, Datum	Unterschrift

I hereby affirm, that I have written this thesis without the use of other than the given sources and that it has not yet been presented to any other examination office in the same or similar form and has been accepted by them as part of an examination. All statements that have been cited verbatim or in the general sense are marked as such.

....

Place, Date Signature

Contents

Chapter 1

Introduction

In many scientific fields as statistics, psychometrics, signal processing, image processing, pattern recognition or even machine learning, measured data is modeled in the form of linear functions depending on some underlying parameters. The data is composed of m observations, that are stored in a $m \times 1$ vector b, and the function's values at the data points, that are expressed as a $m \times n$ matrix A. The aim of data modeling is then the optimization of the underlying parameters in the $n \times 1$ vector x, that explains the observed values as well as possible.[1][2][3]

Usually, the linear system of equations $Ax = b$ is over determined, so that $m \geq n$, because of a sufficient large number of observations or rather data. In general, these systems are approximately solved using a least squares solver.

In practice, often an additional non-negativity constraint for the solution arises, to keep physical or chemical quantities from becoming negative. Examples for such quantities are mass, chemical concentrations, amounts of rainfall, probabilities, image pixel values or color intensities, that are never allowed to take values less than zero.[4][2][5] Another important application is model order reduction, that is inter alia used to reduce the computational costs of huge finite element models in mechanics or magneto-mechanics. Hereby the non-negativity constraint is used to maintain the stability.[6][7]

Currently, these kind of problems are solved via a serial solver, which restricts the size of the equation systems, because of the high computation time and consequently the high computational costs. Aim of this thesis is to implement a parallel version of the non-negative least square solver in C++ using ScaLAPACK. The working code will be finally integrated into the open-source library deal.II and therefore made accessible to the scientific community.

Chapter 2

Solving a Nonnegative Least Squares Problem

2.1 Least Squares Problem

A least squares problem is defined by the following:

> *"Given a real m x n matrix A of rank $k \leq min(m,n)$, and given a real m-vector b, find a real n-vector x minimizing the euclidean length of Ax-b."*[8]

In practical application the least squares method is used, for solving an over determined system of equations of the form

$$Ax = b. \tag{2.1}$$

Due to sufficient data, it usually isn't possible to solve the system correctly, but to look for an approximation that minimizes the error:

$$Minimize \qquad ||Ax - b||, \tag{2.2}$$

where $||\cdot||$ denotes the L_2 norm. "[L]least squares problems often occur as a component part of some larger computational problem." [8] A serious concern of the solving of such problems is the allocation of computer storage, because of very large volumes of data.[8]

A general way to solve least squares problems is by using the QR-decomposition of the matrix A where $A = QR$, with Q an orthogonal and R an upper-triangular matrix.[1] The resulting equation by inserting the decomposition in (2.1)

$$Ax = QRx = b \tag{2.3}$$

can be reformulated as

$$Q^T QRx = Rx = Q^T b = g, \tag{2.4}$$

as Q is orthogonal. Because of R being upper triangular, the linear System $Rx = g$ can be easily solved for x from below via back-substituting. To create the QR-decomposition, for example Housholder transformation, Givens rotation or Gram-Schmidt orthogonalization can be used.[1]

2.2 Non-negative Least Squares (NNLS) Problem

> "There are many applications in applied mathematics, physics, statistics, mathematical programming, economics, control theory, social science, and other fields where the usual least squares problem must be reformulated by the introduction of certain [equality or] inequality constraints."[8]

A common inequality constraint is the non-negativity for the solution vector in order to reflect real-world prior information. When data is for example corrupted by noise, the estimated parameters may not satisfy the constraints, and non-negativity has to be enforced explicitly.[1]

In this case, problem (2.2) is redefined in the following way:

$$Minimize \qquad ||Ax - b|| \ \ subject \ \ to \ \ x \geq 0, \tag{2.5}$$

where $x \geq 0$ means, that the vector x has to be non-negative in every component x_i, with $i = 1...n$.

2.3 Algorithms to solve a Non-negative Least Squares Problem

2.3.1 Active Set Algorithms

The first widely used method to solve the NNLS-problem is the active set algorithm, developed by Lawson and Hanson[8]. It partitions the set of variables x_i into the the passive set P and active set Z. Hereby the free variables in the passive set may take any value greater than zero, while the fixed variables in the active set are restricted to zero. The definition as 'active' is used, because the non-negative constraint is active for all x_i in Z. The idea of the method is to convert the inequality constrained least squares

problem (2.5) to an equality constrained least squares problem by fixing all variables in Z to zero [1][8][9]:

$$Minimize \qquad ||Ax - b|| \ subject \ to \ x_i = 0 \ \forall \ i \ in \ Z \tag{2.6}$$

At the beginning of the algorithm all variables x_i are set to zero and are therefore in the active set. During the algorithm certain variables - in the general case only one variable per iteration - are shifted to the passive set and an unconstrained LS problem is solved via QR-decomposition using only the variables corresponding to the set P [1][9]:

$$Minimize \qquad ||Ax - b|| \ \forall \ x_i \ in \ P \tag{2.7}$$

If the solution of (2.7) is feasible, that means that all x_i are non-negative, the active set is valid and one can identify further variables to be shifted to P in the next iteration. If the solution isn't feasible, the negative variables are set to zero and shifted back to Z.[8]

An improvements to the standard active-set method is for example the Fast NNLS (FNNLS) method[9], that avoids redundant computations and allows a pre-computation of a good initial active set[10]. Another method is the TNT-NN, a new dynamite strategy, that uses an efficient pre-conditioner for identifying the active set and a different strategy for solving the unconstrained least squares sub-problem.[10][11].

2.3.2 Gradient Based Iterative Methods

Iterative methods in general produce a sequence of feasible vectors $\{x_k\}$, $k = 0, 1, ...$ that shall converge to the exact solution.[6] In gradient based iterative methods, the solution x_{k+1} is updated using the previous solution x_k and its gradient $\nabla f(x_k)$. The gradient hereby is computed in the following way:

$$\nabla f(x) = A^T(Ax - b). \tag{2.8}$$

Most common is the Gradient Projection method [12][13], where the the previous solution is updated via the gradient and a suitable step size t_k and then projected onto $\Omega = R^n_+$,

$$x_{k+1} = P_\Omega(x_k + t_k(\nabla f(x_k))) \tag{2.9}$$

where the projection $P_\Omega(x) = max(x, 0)$ projects all negative values back to zero.[14][15]

A more developed variant is the Projected Quasi-Newton (PQN) approach, where it is possible to handle multiple active constraints per iteration. Here, the variables are

partitioned into the two sets P and Z in the same way as in the active set algorithms and the old solution x_k is updated iteratively with the following equation:

$$x_{k+1}^P = x_k^P + \alpha[P_\Omega(x_k^P - \beta S_k^P \nabla f(x_k^P)) - x_k^P], \tag{2.10}$$

where the superscript P denotes the passive set and P_Ω the orthogonal projection onto R_+^n. The matrix S is the approximation of the inverse Hessian $[\nabla^2 f(x)]^{-1} = [A^T A]^{-1}$ and is used to scale the gradient $\nabla f(x_k^P)$. The parameters α and β define the line search method, that is used to find an optimal step size.[6][2]

2.3.3 Other methods

There also exist various other methods to solve the NNLS problem, like the interior point method [16][17], the principle block pivoting method [17][18], the distance algorithm [5], a sequential coordinate wise algorithm [19] or the alternating NNLS [3][20].

2.4 Active Set Algorithm of Lawson and Hanson

In this section, the original active set algorithm from Lawson and Hanson [8], developed in 1974, is described precisely. As already mentioned, the algorithm partitions all variables x_i into the passive set P, that contains the free variables, and the active set Z, whose variables are fixed to zero. Both, P and Z are index sets, that store the indices i of the accompanying variables x_i.[1][8][9][6]

At termination of the algorithm, the following is valid: [1][8][9][6]

$$P: \qquad x_i > 0, \quad w_i = 0 \tag{2.11}$$

$$Z: \qquad x_i = 0, \quad w_i \leq 0 \tag{2.12}$$

The vector w hereby is the negative gradient of $f(x)$ and is also called dual vector: [8][6]

$$w = -\nabla f(x) = A^T(b - Ax) = A^T r, \tag{2.13}$$

where r is the residual vector [6]

$$r = b - Ax = Q_p \begin{bmatrix} 0_p \\ g_z \end{bmatrix}. \tag{2.14}$$

The matrix A and the vector x are partitioned and reordered according to the two sets:

$$x \rightarrow y = \begin{bmatrix} y_p \\ y_z \end{bmatrix} \tag{2.15}$$

$$A \rightarrow A_{sub} = [A_p \quad A_z] \tag{2.16}$$

The vector y corresponds to x, in other words it contains the same elements but in a reordered manner, according to P. To be more precisely, the variables x_i are arranged in $y_p \in \mathbb{R}^{p\times 1}$, where p is the number of indices in P, in the order of their indices in the Passive set, while $y_z \in \mathbb{R}^{n-p\times 1}$ contains all variables x_i with index i in Z. The matrix A_{sub} is comprised in the same way. Therefore the columns a_i of A are arranged in A_p according to their index i in P, while A_z contains all columns with index in Z.[6]

The method starts with initializing x to the zero-vector. Consequently, all indices are in the active set, while the passive set is empty, thus $p = 0$.[9]

The algorithm consists of an outer and an inner loop, which are described in the following sub-sections. Both loops converge after a finite number of iterations[8][9].

2.4.1 Outer Loop

In each outer loop iteration, one variable x_i is moved from the active to the passive set. Its index is selected as the one with the largest positive w_i, which is called w_{max}.[6] This means, the algorithm chooses the component with the most negative gradient, which reduces the residual the most.[1]

$$w_{max} = max\{w_i \mid i \in Z\} \tag{2.17}$$

Additionally, the index i is tested to generate a positive variable, if introduced into the solution, and its column to be linear independent of the previous ones.[8] In every iteration, the passive set is increased by adding one index to it.

Afterwards, an unconstrained least squares problem according to P [1]

$$A_p y_p = b_p, \qquad b_p = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix} \tag{2.18}$$

is solved using the QR-decomposition of A_p [6]

$$A_p = Q_p R_p \tag{2.19}$$

and the solving of the triangular system of equations for y_p, based on equation (2.4) [8][6]:

$$R_p y_p = g_p, \quad g_p = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_p \end{bmatrix} \tag{2.20}$$

If all components of the solution y_p are positive, the algorithms sets the vector y according to equation (2.15) and then goes on with the next iteration. Otherwise, if any variable takes a non positive value, the inner loop is entered.[8]

The algorithm terminates, when the active set Z cannot be decreased any further, because it is either empty or all w_i with $i \in Z$ are negative. Other possible termination criteria are the attainment of a solution with reasonably enough positive variables or with a sufficiently small residual compared to the residual of the right hand side vector b.[6]

2.4.2 Inner Loop

The inner loop is entered, when the computed solution y_p is infeasible, or in other words if there appears at least one non-positive variable in the solution vector y_p.[8][9] The loop condition is therefore '$y_{min} = min\{y_{p,i} \mid i = 1, 2, ..., p\} \leq 0$'.[6]

The infeasible solution y_p is then updated using the old but feasible solution from the previous iteration y [8][1][9][6]:

$$y = y + \alpha(y_p - y) \tag{2.21}$$

$$\alpha = min\{\frac{y_i}{y_i - y_{p,i}} \mid y_{p,i} \leq 0 \ \ and \ \ i \in \{1, 2, ..., p\}\} \tag{2.22}$$

Applying this update, all negative variables are either shifted to a positive value or else set to zero and removed from P.[8] In general, the variable y_{min} turns to zero while all other negative values become positive, on account of the way α is determined. The passive set is then decreased by moving the indices i of all variables with value zero back to Z.[9][6]

Subsequently the unconstrained least squares problem (2.18) based on the new passive

set is solved again by using the QR-decomposition (2.18) and solving for y_p (2.19) .[9][1]

The solution y_p is then checked again for non positive components and the inner loop is repeated until the solution finally becomes feasible. In this case the algorithm sets y according to equation (2.15) and returns to the outer loop.[6]

2.5 Methods to Update the QR-Factorization

In the course of both loops of the Lawson/Hanson active set algorithm, the QR-decomposition of the matrix A_p has to be computed several times. A_p is constructed from the matrix A_{p-1} from the previous iteration and only slightly modified by adding or deleting one single column, owing to the exchange of indices between the active and the passive set.[8][6]

Considering this simply small change in A_p, a full new QR-decomposition from scratch is unnecessary. Instead, the factorization can be incrementally updated "based upon retaining the QR decomposition of the previous problem"[8] and applying some so called up- and downdating methods.[1][6] Updating an existing factorization by using the structures we already created in the previous iteration is much faster, more efficient and thus leads to reduced computational cost, than solving the whole system afresh.[21][22][23][24]

2.5.1 Updating: Adding a column

During the outer loop of the active set algorithm, indices i are added to the passive set and therefore the matrix A_p is expanded by the corresponding columns a_i. To minimize computations the columns are generally inserted at the right most of A_p, in other words they are simply appended at the right side of the matrix.[1]

Considering the QR-decomposition $A_p = Q_p R_p$, the matrices Q_p and R_p can be reused for the factorization of $A_{p+1} = Q_{p+1} R_{p+1}$. Therefore the new column a_{p+1} has to be multiplied by the orthogonal rotation matrix from the previous iteration Q_p in order to obtain the vector

$$r^{\star}_{p+1} = Q_p^T a_{p+1}, \tag{2.23}$$

that can be appended to R_p.[1] Then, $r^{\star}_{p+1}$ has to be modified to fit into the upper triangular matrix by zeroing its elements $r_{p+2,p+1}, r_{p+3,p+1}, ..., r_{m,p+1}$, that are presented highlighted in the following formula [25]:

$$R^{\star}_{p+1} = \begin{bmatrix} R_p & r^{\star}_{p+1} \end{bmatrix} = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & \cdots & r_{1,p} & r^{\star}_{1,p+1} \\ 0 & r_{2,2} & r_{2,3} & \cdots & r_{2,p} & r^{\star}_{2,p+1} \\ 0 & 0 & r_{3,3} & \cdots & r_{3,p} & r^{\star}_{3,p+1} \\ \vdots & & \ddots & \ddots & \vdots & \vdots \\ \vdots & & & \ddots & r_{p,p} & r^{\star}_{p,p+1} \\ \vdots & & & & 0 & r^{\star}_{p+1,p+1} \\ \vdots & & & & \vdots & r^{\star}_{p+2,p+1} \\ \vdots & & & & \vdots & r^{\star}_{p+3,p+1} \\ \vdots & & & & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & r^{\star}_{m,p+1} \end{bmatrix} \tag{2.24}$$

The construction of the matrix Q_{p+1} depends on the method, that is chosen, to eliminate those elements:

The first possibility is using Givens rotations, that can be applied to a matrix to eliminate one specific element. To zero for example one sub diagonal element $r^{\star}_{i,p+1}$ in the last column of the matrix $R^{\star}_{p+1}$ with all other columns already triangularized, only one other component in the same column is changed (usually the diagonal element $r^{\star}_{p+1,p+1}$). The rotation matrix G_i of dimensions $m \times m$ is an identity matrix with only four other additional elements positioned at the intersections of the columns and rows $p+1$ and i [8][26]:

$$G_i = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \overset{p+1}{} \quad \overset{i}{} \\ \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & & & & \vdots \\ \vdots & \ddots & 1 & \ddots & & & & & \vdots \\ \vdots & & \ddots & c & \ddots & s & & & \vdots \\ \vdots & & & \ddots & 1 & \ddots & & & \vdots \\ \vdots & & & -s & \ddots & c & \ddots & & \vdots \\ \vdots & & & & & \ddots & 1 & \ddots & \vdots \\ \vdots & & & & & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ \\ \\ \\ p+1 \\ \\ i \\ \\ \\ \end{array} \tag{2.25}$$

with

$$c = \frac{r^{\star}_{p+1,p+1}}{\sqrt{r^{\star\,2}_{p+1,p+1} + r^{\star\,2}_{i,p+1}}} \tag{2.26}$$

$$s = \frac{r^{\star}_{i,p+1}}{\sqrt{r^{\star\,2}_{p+1,p+1} + r^{\star\,2}_{i,p+1}}} \tag{2.27}$$

Consequently one matrix G_i needs to be computed for every element, that has to be annihilated, and multiplied with the matrix $R^{\star}_{p+1} = \left[R_p \;\; r^{\star}_{p+1}\right]$ [25][22][26]:

$$R_{p+1} = G_m^T \cdots G_{p+4}^T G_{p+3}^T G_{p+2}^T \left[R_p \;\; r^{\star}_{p+1}\right] = Q_{p+1}^T A_{p+1} \tag{2.28}$$

or rather just with the column vector $r^{\star}_{p+1}$ as the other columns won't be affected by the multiplication.

$$r_{p+1} = G_m^T \cdots G_{p+4}^T G_{p+3}^T G_{p+2}^T r^{\star}_{p+1} = Q_{p+1}^T a_{p+1}, \qquad R_{p+1} = [R_p \;\; r_{p+1}] \tag{2.29}$$

$$Q_{p+1} = Q_p G_{p+2} G_{p+3} G_{p+4} \cdots G_m \tag{2.30}$$

Instead of using several Givens rotations, another possibility is to apply one single Housholder transformation. To zero all entries below the element $r^{\star}_{p+1,p+1}$ of the vector $r^{\star}_{p+1}$, the following symmetric Housholder matrix H_i can be constructed [8][23][21][27]:

$$H_i = I - \frac{2uu^T}{u^T u}, \qquad u = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ r^{\star}_{p+1,p+1} - sgn(r^{\star}_{p+1,p+1})\sqrt{\sum_{j=i}^{m} r^{\star}_{j,p+1}} \\ r^{\star}_{p+2,p+1} \\ r^{\star}_{p+3,p+1} \\ \vdots \\ r^{\star}_{m,p+1} \end{bmatrix} \begin{matrix} \\ \\ \\ \textcircled{p+1} \\ \\ \\ \\ \\ \end{matrix} \tag{2.31}$$

and then applied to the matrix $R^{\star}_{p+1} = \left[R_p \;\; r^{\star}_{p+1}\right]$ [27]

$$R_{p+1} = H_{p+1}^T \left[R_p \;\; r^{\star}_{p+1}\right] = Q_{p+1}^T A_{p+1} \tag{2.32}$$

or to the vector $r^{\star}_{p+1}$ [6][8].

$$r_{p+1} = H_{p+1}^T r^{\star}_{p+1} = Q_{p+1}^T a_{p+1}, \qquad R_{p+1} = [R_p \;\; r_{p+1}] \tag{2.33}$$

$$Q_{p+1} = Q_p H_{p+1} = H_1 H_2 \cdots H_p H_{p+1} \tag{2.34}$$

Another approach is called 'modified Gram-Schmidt' procedure, where Q_{p+1} is calculated by orthogonalizing the column a_{p+1} with all columns of Q_p via vector projections.[1]

Other methods are for example the combination of Givens rotation and Housholder transformation[24] or the application of semi-normal equations[1].

2.5.2 Downdating: Deleting a column

In the course of the inner loop, indices i are shifted back from the passive to the active set and thus the matrix A_p is modified by deleting the corresponding columns a_i. The matrix R_p of the previous iteration can be reused for the computation of R_{p+1} by likewise deleting the i-th column r_i. Subsequently, the matrix "is no longer upper triangular as the columns to right of index $[i]$ have shifted left."[1] To regain the required form, all sub-diagonal elements $r_{k+1,k+1}$ with $k = i+1, ..., m$ (see equation (2.35)) need to be eliminated.[25]

$$R^{\star}_{p+1} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,i-1} & r_{1,i+1} & r_{1,i+2} & r_{1,i+3} & \cdots & r_{1,p} \\ 0 & r_{2,2} & \cdots & r_{2,i-1} & r_{2,i+1} & r_{2,i+2} & r_{2,i+3} & \cdots & r_{2,p} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & r_{i-1,i-1} & r_{i-1,i+1} & r_{i-1,i+2} & r_{i-1,i+3} & \cdots & r_{i-1,p} \\ \vdots & & & 0 & r_{i,i+1} & r_{i,i+2} & r_{i,i+3} & \cdots & r_{i,p} \\ \vdots & & & & r_{i+1,i+1} & r_{i+1,i+2} & r_{i+1,i+3} & \cdots & r_{i+1,p} \\ \vdots & & & & 0 & r_{i+2,i+2} & r_{i+2,i+3} & \cdots & r_{i+2,p} \\ \vdots & & & & \vdots & 0 & r_{i+3,i+3} & \cdots & r_{i+3,p} \\ \vdots & & & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \end{bmatrix} \tag{2.35}$$

This elimination can be easily done by Givens rotations as it concerns only one element per column [8]:

$$R_{p+1} = G_m^T \cdots G_{i+3}^T G_{i+2}^T G_{i+1}^T R^{\star}_{p+1} = Q_{p+1}^T R_p \tag{2.36}$$

$$Q_{p+1} = Q_p G_{p+1} G_{p+2} G_{p+3} \cdots G_m \tag{2.37}$$

Another way is to replace the whole columns of R_p right of index i by the corresponding columns of original the matrix A. In order that these columns can be appended to the first $i-1$ columns of R_p, they previously need to be multiplied by the orthogonal rotation matrix belonging to the first $i-1$ columns of R_p: [6][8]

$$R^{\star} = Q_{i-1}^T \left[a_{i+1} \;\; a_{i+2} \;\; \cdots \;\; a_p \right], \tag{2.38}$$

$$R^{\star}_{p+1} = [R_{i-1} \;\; R^{\star}] \tag{2.39}$$

Subsequently several Housholder transformations can be applied to zero the sub-diagonal elements of these new appended columns $r^{\star}_{i+1}, \cdots, r^{\star}_{p}$ [8]:

$$R^{\star}_{p+1} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,i-1} & r^{\star}_{1,i+1} & \cdots & r^{\star}_{1,p} \\ 0 & r_{2,2} & \cdots & r_{2,i-1} & r^{\star}_{2,i+1} & \cdots & r^{\star}_{2,p} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & & \ddots & r_{i-1,i-1} & r^{\star}_{i-1,i+1} & \cdots & r^{\star}_{i-1,p} \\ \vdots & & & 0 & r^{\star}_{i,i+1} & \cdots & r^{\star}_{i,p} \\ \vdots & & & \vdots & r^{\star}_{i+1,i+1} & \cdots & r^{\star}_{i+1,p} \\ \vdots & & & \vdots & r^{\star}_{i+2,i+1} & \cdots & r^{\star}_{i+2,p} \\ \vdots & & & \vdots & r^{\star}_{i+3,i+1} & \cdots & r^{\star}_{i+3,p} \\ \vdots & & & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & r^{\star}_{m,i+1} & \cdots & r^{\star}_{m,p} \end{bmatrix} \tag{2.40}$$

For the construction of the new orthogonal rotation matrix, all Housholder matrices of iterations $i, i+1, i+2, ...p$ are replaced by new transformation matrices $H^{new}_{i+1} H^{new}_{i+2} \cdots H^{new}_{p}$ [8]:

$$Q_{p+1} = H_1 H_2 \cdots H_{i-1} H^{new}_{i+1} H^{new}_{i+2} \cdots H^{new}_{p} = Q_{i-1} H^{new}_{i+1} H^{new}_{i+2} \cdots H^{new}_{p} \tag{2.41}$$

Just as for the updating procedure, a Gram-Schmidt method can be used for the downdating, too.[28]

Chapter 3

Parallel Programming using MPI and ScaLAPACK

As processor speeds no longer double every few months, the number of processing units needs to double instead, in order to be able to handle bigger problems that arise with progressive scientific research. The consequence is parallel programming on lots of processes that work together to solve large problems by distributing the workload. Considering that, either shared or distributed memory programming can be used. Shared memory can only be used, in case the processes are all located on the same CPU (central processing unit), as they need to share the same memory address space. The second possibility is also called Message Passing, because the processes, that are only connected over a network, have separate address spaces and communicate by sending messages to each other.[29]

In the case of parallelizing the NNLS problem, Message Passing is used, because more processes than in shared memory programming can contribute and the running time of the program can be decreased further.

3.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a programming model for message passing, or rather a programming intersection, that defines a large number of operations and their semantic. Its advantages are its standardization, portability, availability and popularity.[29] The first version was published in 1994 and the actual version 3.1 is from the year 2015.[30] It can be used with the programming languages Fortran, C and C++ and common available open source implementations are for instance MPICH, MPICH2 or OpenMPI.[31] MPI is mostly used for computations on shared computer systems but also possible for computer

with common memory.[32]

The key fact for MPI is, that the written program is executed by all processes simultaneously. Each process has its own private memory, that cannot be accessed by the other processes. So consequently every process runs the same program, but on different data. The movement of data between address spaces can only happen via explicit data exchange and inter-processor communication.[31][29]

For this, more than 300 MPI functions are available, but mostly less than ten are sufficient. Communication can be done either by point to point communication, so for instance the sending and receiving of a message from one process to another one, or by global communication. Examples for the second case are a message that is sent from one process to all other ones (broadcast) or the other way around, a message that is sent from all processes and gathered only by one single process. In order to be able to correctly send and receive data, communicators with distinct processes are created, inside which every process is initialized with a so called 'rank'. For a successful message passing, the rank of the sender and receiver have to be declared, just as the data type, size and memory location of the corresponding data.[31]

Some of the most important MPI functions are described further in the following. For reasons of simplification concerning the use of MPI functions, there exist several wrapper libraries, as for example Deal.II or Boost.MPI.

Deal.II[33]:

- To get back the number of MPI processes that exist in a given communicator, use:
  n_mpi_processes(communicator)
- To return the rank of the present MPI process inside a given communicator, use:
  this_mpi_process(communicator)
- Every process sends an object to a root process. The root process receives a vector of objects, with size equal to the number of processes in the MPI communicator. Each entry of the vector contains the object received from the process with the corresponding rank within the communicator. All other processes receive an empty vector. For this use:
  gather(communicator, object_to_send, root_process)
- Every process sends an object to all other processes. Therefore every process receives a vector of objects, with size equal to the number of processes in the MPI communicator. Each entry of the vector contains the object received from the processor with the corresponding rank within the communicator. For this use:
  all_gather(communicator, object_to_send)

Boost.MPI[34]:

- To send a message to another process, use:
  communicator.send (rank_of_receiving_process, message_tag, object_to_send)
- To receive a message from another process, use:
  communicator.recv (rank_of_sending_process, message_tag, buffer_to_save_received _object)
- To broadcast a value from a single process to all other processes within a communicator, use:
  broadcast (communicator, element_to_broadcast, rank_of_sending_process)

3.2 ScaLAPACK

ScaLAPACK is an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK. It is a library of high-performance linear algebra LAPACK routines redesigned for distributed-memory message-passing MIMD (multiple instruction, multiple data) computers and networks of workstations, supporting MPI. The freely-available software package contains routines for solving systems of linear equations, least squares problems and eigenvalue problems and the goals of the project are efficiency, reliability, portability, flexibility and ease of use. "Most of the ScaLAPACK code is written in standard Fortran 77[.] The first [...] software was written in 1989"[35] and published in 1995, while the most recent version 2.1.0.0 was released in 2019.[36] Although, only minor changes have been realized in the context of newer versions, like small bug fixes, some additional methods, more accuracy or better support of different MPI libraries. [36]

All in all, Scalapack is obsolete after two decades and cannot be adequately retrofitted for modern, GPU-accelerated architectures. The newer software, that will serve as a replacement for ScaLAPACK and provides sufficient coverage of existing LAPACK and ScaLAPACK functionality is called SLATE (software for linear algebra targeting exascale). It is compatible with shared memory parallelization and supports hardware accelerators, which are an integral part of today's HPC hardware infrastructure.[37][38]

3.2.1 Distribution of data onto a process grid

"ScaLAPACK requires that all global data (vectors or matrices) be distributed across the processes"[35] according to a specific data decomposition scheme. For that, all processes $0, 1, ..., P-1$ are mapped to a one- or two-dimensional process grid, where P is the total

number of processes. All matrices and vectors are then partitioned into rows, columns or blocks and distributed along this grid. Each process consequently only receives a distinct number of rows, columns or blocks. In the case of a two dimensional process grid with dimensions $P_r \times P_c$, either row- or column-major order may be used to map the total number of processes $(P_r + 1) \times (P_c + 1) = P$ to the grid.[35]

The choice of the appropriate data distribution scheme depends on the characteristics of the matrix, being for instance banded, diagonal, tridiagonal or symmetric. For normal dense matrix computations, ScaLAPACK in general uses a two-dimensional block-cyclic data layout scheme, that reduces the frequency with which data must be transferred between processes. This means, that the data is split into blocks, and uniformly distributed among a two-dimensional process grid in a recurring manner [35] (figure 3.1).

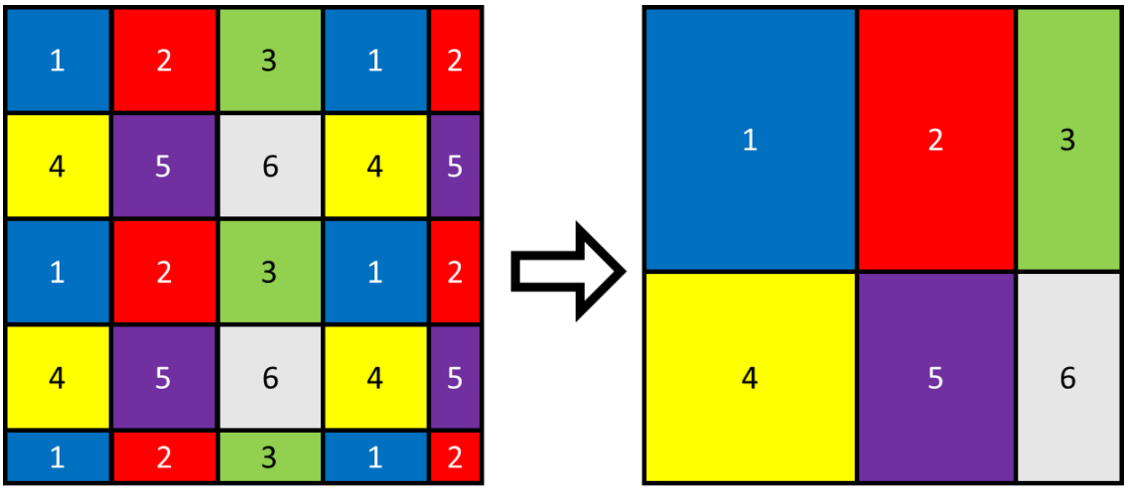

Figure 3.1: Example for a two-dimensional block-cyclic distribution of a matrix onto a process grid with $2 \times 3 = 6$ processes using column major order

When distributed onto a process, data is locally stored in conventional arrays and the information about the data layout and the local storage scheme is stored in a so called 'array descriptor'.

3.2.2 ScaLAPACK Matrix

The distribution of a matrix happens with the help of the wrapper class 'ScaLAPACKMatrix'. Its most essential private attributes are [39]:

- *grid*: used process grid

- *n_rows/ n_columns*: number of rows/ columns in the matrix
- *descriptor*: description of the ScaLAPACKMatrix
- *row_block_size/ column_block_size*: size of the blocks in row/ column direction; in general, it is recommended to use powers of 2
- *work*: workspace array

Regarding this thesis, the mostly used public member functions are [39]:

- *ScaLAPACKMat*: Constructor for a rectangular matrix with *n_rows* and *n_cols*, that is distributed onto the grid *process_grid* with a block-cyclic distribution using blocks of size *row_block_size* times *column_block_size*.
- *copy_to*: Copy the whole content of the matrix into another one. Alternatively only a submatrix may be copied to another submatrix as well.
- *add*: Adding a matrix to another one accordingly to the scheme $A = \alpha A + \beta B$, where B may may transposed as well.
- *mult*: Multiplying a matrix with another one accordingly to the scheme $C = \beta A \times B + \gamma C$. A or B may be transposed as well.
- *frobenius_norm*: Compute the frobenius norm of a matrix.
- *n/ m*: Number of rows/columns of a $n \times m$ matrix.
- *local_m/ local_n*: Number of local rows/ columns on the current MPI process.
- *global_row/ global_column*: Returns the global row/ column number for a given local row/ column on the current MPI process.
- *local_el*: Read and Write access to a local element.

3.2.3 Important ScaLAPACK Functions

The three ScaLAPACK functions, which are used in the context of this thesis, are explained in detail in the following. All routines begin with the letter 'P', followed by 'x', that denotes the data type of the matrix data, that can be replaced by either 'S' for 'single precision', 'D' for 'double precision', 'C' for 'complex' or 'Z' for 'double complex'. Behind, there appear letters to specify the properties of the matrix, as 'GE' for 'general',

'OR' for 'orthogonal' or 'TR' for 'triangular' and some final letters to design the purpose of the routine.[35]

PxGEQRF:
This Function "computes the QR factorization [of the $m \times n$ ScaLAPACKMatrix $sub(A)$]. The matrix Q is not formed explicitly, but is represented as a product of elementary reflectors"[35], also referred to as elementary Housholder matrices. A Housholder matrix $H = I - \frac{2uu^T}{u^T u}$, purposed to zero the k sub diagonal elements of a specific column with index i is only stored via the vector u. This vector contains exactly $k + 1$ non-zero elements, from which the first one is stored on the i-th position of the array tau, while the other ones are stored in the k sub diagonal elements, that are anyway supposed to take the value zero.

All associated ScaLAPACK routines are provided to work with this representation and can for instance generate Q from the elementary reflectors or just use it for multiplications.[35]

Arguments of the routine are [39]:

- M (input): number of rows to be operated on
- N (input): number of columns to be operated on
- A (input and output): On entry, this matrix contains the submatrix $sub(A)$ of dimensions $m \times n$, which shall be factorized. On exit it will be overwritten by the triangular matrix R in its upper triangular part and the elementary reflectors in its lower triangular part of $sub(A)$.
- IA (input): row index of the first row of the submatrix $sub(A)$ within A
- JA (input): column index of the first column of the submatrix $sub(A)$ within A
- DESCA (input): descriptor of the matrix A
- TAU (output): This array is tied to the matrix A and contains on exit parts of the elementary reflectors.
- WORK (output): workspace array
- LWORK (input): size of WORK
- INFO (output): contains information about the status of the routine

PxORMQR:
This routine pre- or post-multiplies a given $m \times n$ matrix $sub(C)$ by the orthogonal

matrix Q or Q^T, that is defined as the product of k elementary reflectors, which are stored in the matrix $sub(A)$.

Arguments of the routine are [39]:

- SIDE (input): contains the char 'L' or 'R' for indicating 'pre-' or 'postmultiplying'
- TRANS (input): contains the char 'N' or 'T' for indicating, if Q shall be used 'normal' or 'transposed'
- M (input): number of rows to be operated on
- N (input): number of columns to be operated on
- K (input): number of elementary reflectors, whose product defines the orthogonal matrix Q; it corresponds to the number of columns of the matrix $sub(A)$, that shall be used for Q
- A (input): matrix that contains the submatrix $sub(A)$ of dimensions $m \times k$, which contains the elementary reflectors in its lower triangular part
- IA (input): row index of the first row of the submatrix $sub(A)$ within A
- JA (input): column index of the first column of the submatrix $sub(A)$ within A
- DESCA (input): descriptor of the matrix A
- TAU (input): This array is tied to the matrix A and contains parts of the elementary reflectors.
- C (input and output): On entry, this matrix contains the submatrix $sub(C)$ of dimensions $m \times n$, which shall be multiplied by the orthogonal matrix. On exit, it will be overwritten by the solution matrix of the multiplication.
- IC (input): row index of the first row of the submatrix $sub(C)$ within C
- JC (input): column index of the first column of the submatrix $sub(C)$ within C
- DESCC (input): descriptor of the matrix C
- WORK (output): workspace array
- LWORK (input): size of WORK
- INFO (output): contains information about the status of the routine

PxTRSV:
This function solves an unconstrained linear system of equations ($Ax = b$), where the $n \times n$ matrix $sub(A)$ already has been triangularized.

Arguments of the routine are [39]:

- UPLO (input): contains the char 'U' or 'L' for indicating, if the matrix is upper or lower triangular
- TRANS (input): contains the char 'N' or 'T' for indicating, if Q shall be 'normal' or'transposed'
- DIAG (input): contains the char 'U' or 'N' for indicating, if the matrix is unit-triangular or not
- N (input): number of rows and columns to be operated on
- A (input): matrix that contains the submatrix $sub(A)$ of dimensions $n \times n$, which is used as parameter matrix for the linear system of equations
- IA (input): row index of the first row of the submatrix $sub(A)$ within A
- JA (input): column index of the first column of the submatrix $sub(A)$ within A
- DESCA (input): descriptor of the matrix A
- X (input and output): On entry, this matrix contains in the submatrix $sub(X)$ with dimensions $n \times 1$ the righthandside b of of the linear system of equations. On exit, b is overwritten by the solution vector.
- IX (input): row index of the first row of the submatrix within X
- JX (input): column index of the first column of the submatrix $sub(X)$ within X
- DESCX (input): descriptor of the matrix X
- INCX (input): global increment for the elements of X

As all routines are written in Fortran, special care has to be taken on the indices of arrays, that always start with one and not as in current programming languages like C or C++ with zero. Also the notation of parameters with a star (*) doesn't denote a pointer, but just an array.

Chapter 4

Implementation of the Lawson-Hanson Algorithm

In the report of the US Army Research Laboratory "Parallel Nonnegative Least Squares Solvers for Model Order Reduction" by James P. Collins [6], inter alia an algorithm for the parallelization of the Lawson-Hanson active set method [8] is presented. The aim of this thesis is to write an open source implementation of that parallel algorithm by following the notes of the cited US report. By finally integrating the code into the open-source library deal.II, it shall be made accessible to the scientific community.

The algorithm has to be implemented as an additional public member function 'pNNLS' of the ScaLAPACKMatrix class. For simplification and the overall view some other public member functions are implemented additionally to be used in the 'pNNLS'-routine.

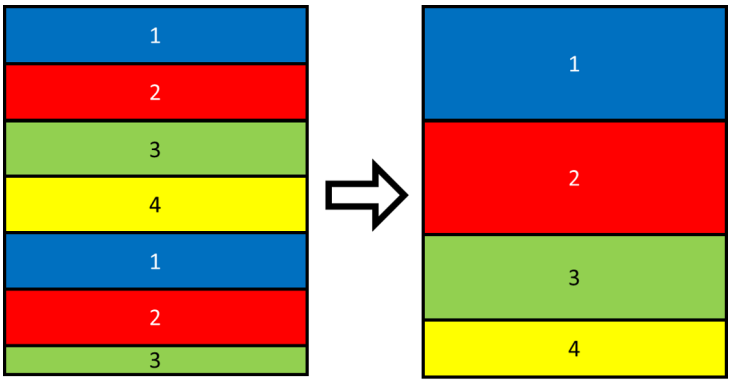

Figure 4.1: Example for a one-dimensional block-cyclic row distribution of a matrix onto a process grid with $4 \times 1 = 4$ processes

As denoted in the US report, a one-dimensional process grid with dimensions $N_{processes} \times 1$

shall be used for all matrices an vectors [6]. This coincides with a one-dimensional block-cyclic row distribution, where the processes are distributed along a column and therefore every process owns several whole rows [35] (figure 4.1).

Algorithm 1 Parallel LH (PLH) NNLS Algorithm

Input:

$\mathbf{A} \in \mathbb{R}^{M \times N}$, $\mathbf{b} \in \mathbb{R}^{M}$, τ, p^{max}, $scale$

Output:

$\mathbf{x}$

1: **Initialize:** $\mathbf{x} = \mathbf{y} = \mathbf{0}$, $\widehat{\boldsymbol{\mathcal{P}}} = \varnothing$, $p = 0$, $\mathbf{g} = \mathbf{b}$, $\mathbf{Q}_0 = \mathbf{I}$
2: **if** $scale$ **then**
3: $\quad \mathbf{A} \leftarrow \mathbf{A}\,\Lambda_s$
4: **repeat**
5: $\quad \mathbf{r} \leftarrow \mathbf{Q}_p \left[\mathbf{0}_p,\ \mathbf{g}_{(N-p)}\right]^T$
6: $\quad \mathbf{w} \leftarrow -\nabla f(\mathbf{x}) = \mathbf{A}^T \mathbf{r}$
7: $\quad w_{i_{max}} = \max \{w_i \mid i = 1, 2, \ldots N,\ i \notin \widehat{\boldsymbol{\mathcal{P}}}\}$
8: $\quad$ **if** $w_{i_{max}} > 0$ **then**
9: $\quad\quad p \leftarrow p + 1,\ \ \widehat{\boldsymbol{\mathcal{P}}}(p) = i_{max}$
10: $\quad\quad$ UpdateQR(p)
11: $\quad\quad$ Solve $\mathbf{R}_p \mathbf{y}_p = \mathbf{g}_p$
12: $\quad\quad y_{min} = \min \{y_{p_i} \mid i = 1, 2, \ldots p\}$
13: $\quad\quad$ **while** $y_{min} \leq 0$ **do**
14: $\quad\quad\quad \alpha = \min \left\{ \frac{y_i}{y_i - y_{p_i}} \mid y_{p_i} \leq 0, i \in \{1, 2, \ldots, p\} \right\}$
15: $\quad\quad\quad \mathbf{y} \leftarrow \mathbf{y} + \alpha(\bar{\mathbf{y}}_p - \mathbf{y}),$ where $\bar{\mathbf{y}}_p = \left[\mathbf{y}_p, \mathbf{0}_{(N-p)}\right]^T$
16: $\quad\quad\quad \boldsymbol{\mathcal{P}}_0 = \{\widehat{\boldsymbol{\mathcal{P}}}(i) \mid y_i = 0,\ i = 1, 2, \ldots, p\}$
17: $\quad\quad\quad q_{min} = \min i \in \boldsymbol{\mathcal{P}}_0$
18: $\quad\quad\quad \widehat{\boldsymbol{\mathcal{P}}} = \widehat{\boldsymbol{\mathcal{P}}} \setminus \boldsymbol{\mathcal{P}}_0,\ \ p \leftarrow \mid \widehat{\boldsymbol{\mathcal{P}}} \mid$
19: $\quad\quad\quad$ UpdateQR(q_{min})
20: $\quad\quad\quad$ Solve $\mathbf{R}_p \mathbf{y}_p = \mathbf{g}_p$
21: $\quad\quad\quad y_{min} = \min \{y_{p_i} \mid i = 1, 2, \ldots p\}$
22: $\quad\quad \mathbf{x} \leftarrow \P_p \left[\mathbf{y}_p, \mathbf{0}_{(N-p)}\right]^T$
23: **until** $w_i \leq 0\ \forall i \in \boldsymbol{\mathcal{Z}}$ or $p = p^{max}$ or $\|\mathbf{r}\|_2 \leq \tau \|\mathbf{b}\|_2$
24: **if** $scale$ **then**
25: $\quad \mathbf{x} \leftarrow \Lambda_s \mathbf{x}$
26: **Return** $\mathbf{x}$

Figure 4.2: Parallel non-negative least squares algorithm, that shall be implemented [6]

```
27: procedure UPDATEQR(k)
28:     for i = k → p do
29:         j = P̂(i)
30:         q_i ← a_j
31:     if k > 1 then
32:         [q_k, q_{k+1}, ... q_p] ← H_{k-1}, H_{k-2}, ... H_1 [q_k, q_{k+1}, ... q_p]
33:     [q_k, q_{k+1}, ... q_p] ← QR(k, [q_k, q_{k+1}, ... q_p])
34:     if k = p then
35:         g ← H_p g
36:     else
37:         g ← Q_p b
```

Figure 4.3: Subroutine for the QR up- and downdating [6]

Therefore all column-vectors are as well distributed along the processes by contrast to a block-cyclic column distribution. Furthermore, the specified distribution leads to very efficient matrix-vector-multiplications in the case of a $m \times n$ -matrix with $m >> n$, as whole rows of the matrix, owned by different processes, are multiplied by a vector.

The active set algorithm is implemented as described in chapter 2.4 with an outer and inner loop. For up- and downdating, Housholder transformation is used, although Givens rotation would be faster for downdating, but it is less efficient for parallelization, as only two single rows are changed and the load balancing is thus of quite poor quality.

The whole algorithm from the US army report [6] can be seen in figure 4.2 and 4.3.

4.1 Implemented Methods

Before describing the implemented methods, first have a look at the definition of all used variables [6]:

Table 4.1: Definition of all used variables in the implemented methods

variable	dimension	explanation
A	$m \times n$	parameter matrix of the NNLS problem $Ax = b$
x	$n \times 1$	solution vector of the NNLS problem $Ax = b$
b	$m \times 1$	right hand side of the NNLS problem $Ax = b$
$y = \begin{bmatrix} y_p \\ y_z \end{bmatrix}$	$n \times 1$	reordered vector x according to the sets P and Z
$A_{sub} = [A_p \ A_z]$	$m \times n$	reordered matrix A according to the sets P and Z
$g = Q_p^T b = \begin{bmatrix} g_p \\ g_z \end{bmatrix}$	$m \times 1$	right hand side vector of the reformulated NNLS problem $Rx = g$, its partition in g_p and g_z occurs by using the first p variables of g for g_p and the remaining ones for g_z
$r = Q_p \begin{bmatrix} 0_p \\ g_z \end{bmatrix}$	$m \times 1$	residual vector
$w = A^T r$	$n \times 1$	dual vector
Q_p	$m \times m$	orthogonal rotation matrix, stored in the lower triangular part of the p columns of A_p, is used for the QR-factorization of A_p
R_p	$m \times p$	upper triangular matrix, stored in the upper triangular part of the p columns of A_p, is used for the QR-factorization pf A_p
p	scalar	number of elements in the passive set

4.1.1 UpdateQR

The purpose of the subroutine 'updateQR' is the up- or downdating of the QR-factorization of A_p, after the addition or deleting of a column. As already mentioned, Housholder transformation is used for both. In the case of updating, only the just added column is treated. Whereas in the case of downdating, all columns right of the deleted one are deleted as well, prior to taking the initial columns from A and treating them again.

The arguments of the subroutine are the following, where 'NumberType' is later defined by the type of the elements of the ScaLAPACKMatrix:

Table 4.2: Definition of the arguments of the function 'updateQR'

name	type	explanation
this	ScaLAPACKMat<NumberType>	matrix, that provides the routine with the initial columns of A
Asub	std::shared_ptr<ScaLAPACKMat <NumberType>>	matrix, whose QR-decomposition is to be updated
k	const int	the index of the first column that has to be updated, or rather the index of the added/ deleted column
passive_set	const std::vector<int>	contains the indices of the passive set; its length equals p
tau	std::vector<NumberType>	contains parts of the elementary reflectors of Q_p; is necessary for the ScaLAPACK routines PxORMQR and PxGEQRF

Although, the matrix Asub is of dimension $m \times n$, only the first p columns are considered as they correspond to the matrix A_p. In the case of updating, the first $p-1$ columns are already factorized and consequently only the p-th column needs to be updated. In the case of downdating only the factorization of the columns left of the deleted one is kept and the rest of A_p has to be updated again.

At the beginning of the routine, all columns of A_p with index k to p are overwritten with the corresponding original columns of A. Then, in case $k > 1$, these new columns k to p are multiplied by the transposed of the orthogonal rotation matrix, which is stored in the lower triangular part of the columns 1 to $k-1$, by using the ScaLAPACK routine PxORMQR. The reason for this multiplication is, that the columns 1 to $k-1$ already have been multiplied by this rotation matrix and therefore all columns are treated the in same way.

Finally, a new QR-decomposition is applied to the columns k to p by using the ScaLAPACK routine PxGEQRF. Here, only the rows k to m are used, as the routine shall only factorize the lower part of the columns and the upper part remains the same.

4.1.2 UpdateG

This routine is called as a function of the ScaLAPACKMatrix *Asub* and intended to update the right hand side vector g of the reformulated NNLS problem $Rx = g$. In the case of the addition of a column ($k == p$), only one single element of g needs to be updated using only the new computed elementary reflector H_p ($g = H_p g$). Otherwise, the vector g is recomputed from the original right hand side vector b by using all available elementary reflectors ($g = Q^T b$). The arguments of the subroutine are the following:

Table 4.3: Definition of the arguments of the function 'updateG'

name	type	explanation
this	std::shared_ptr<ScaLAPACKMat <NumberType>>	matrix Asub, that contains the current A_p with its QR-decomposition
b	const std::shared_ptr<ScaLAPACKMat <NumberType>>	vector, that is used for the updating of g
g	std::shared_ptr<ScaLAPACKMat <NumberType>>	vector, that shall be updated
k	const int	the index of the first column that has to be updated, or rather the index of the added or deleted column
p	const int	number of elements in the passive set
tau	std::vector<NumberType>	contains parts of the elementary reflectors of Q_p; is necessary for the ScaLAPACK routine PxORMQR

For the multiplication the ScaLAPACK routine PxORMQR is used.

4.1.3 Min_value

The purpose of this function is to find the minimal element of a submatrix within a ScaLAPACK matrix. The return argument is a 'std::pair<NumberType,std::array<int,2>>', that contains the value of the minimal element as first object and an array with its row and column index as second object. The arguments are described in the following table:

Table 4.4: Definition of the arguments of the function 'min_value'

name	type	explanation
this	ScaLAPACKMat<NumberType>	contains the submatrix, which shall be examined to find its minimal element
row_begin	const unsigned int	indicates the first row, of the submatrix
row_end	const unsigned int	indicates the last row, of the submatrix
column_begin	const unsigned int	indicates the first column, of the submatrix
column_end	const unsigned int	indicates the last column, of the submatrix

As the elements of the matrix are distributed along the different processes, a common for-loop-iteration over all elements of the submatrix is not possible.

Instead, the routine begins with checking, which processes are indeed owning any matrix entries. The idea of parallelization is, that every process first iterates over its own local columns and rows. After analyzing, if the global row and column indices of an element lay within the desired submatrix, the minimal value of each process can be found by using the member function 'local_el' and hence comparing all values. The minimal element of each process is then stored together with its global row and column index. Subsequently, the minimal values of all processes are gathered in an array on all processes as well as their indices by using 'all_gather' from deal.II. Finally all processes search in another loop for the minimal element within the gathered array and store it together with its indices in the return variable.

4.1.4 Max_value

This routine is for finding the maximal element of a submatrix within a ScaLAPACK matrix. The return argument is a 'std::pair<NumberType,std::array<int,2>>' that contains the value of the maximal element as first object and an array with its row and column index as second object. The arguments are the following:

Table 4.5: Definition of the arguments of the function 'max_value'

name	type	explanation
this	ScaLAPACKMat<NumberType>	contains the submatrix, which shall be examined to find its maximal element
row_begin	const unsigned int	indicates the first row, of the submatrix
row_end	const unsigned int	indicates the last row, of the submatrix
column_begin	const unsigned int	indicates the first column, of the submatrix
column_end	const unsigned int	indicates the last column, of the submatrix

The function works in the same way as the routine 'min_value.

4.1.5 set_element_to_value

This function as well as the next one ('return_element') are the parallel version of the member function 'local_el', that accesses a specific element of a distributed ScaLAPACK-Matrix. While the function 'return_element' returns the value of the specified element for further use (read access), this routine modifies it to take a given value (write access). Its arguments are:

Table 4.6: Definition of the arguments of the function 'set_element_to_value'

name	type	explanation
this	ScaLAPACKMat<NumberType>	matrix, that contains the element, that shall be accessed
row_index	const unsigned int	indicates the global row index
column_index	const unsigned int	indicates the global column index
value	const NumberType	the value, which the indicated element shall be set to

While the function 'local_el' takes the global row and column indices of the element to

access it, its parallel version first needs to check, which process is indeed owning that indicated element. Therefore each process iterates over its own local rows and columns and compares the global indices with the given ones. Only the process, that is actually owning the element, subsequently sets the element to the given value, while all other processes remain passive.

4.1.6 return_element

This function as well as the previous one ('set_element_to_value') are the parallel version of the member function 'local_el', that accesses a specific element of a distributed ScaLAPACKMatrix. The function arguments are:

Table 4.7: Definition of the arguments of the function 'return_element'

name	type	explanation
this	ScaLAPACKMat<NumberType>	matrix, that contains the element, that shall be accessed
row_index	const unsigned int	indicates the global row index
column_index	const unsigned int	indicates the global column index

The routine works in the same way as the function 'set_element_to_value'. By contrast, this routine has a return variable of type 'NumberType', which shall take the value of the specified element.

As only one process owns the described element, its value has to be broadcasted to all ones, so that it can be returned by all processes for further use. A problem, that arises, is that the rank of the process owning the element is necessary for broadcasting, but only the process itself knows its rank. Therefore the rank of the process owning the element has to be communicated to the other processes first. Thus, a boolean variable is created for all processes and only that specific process sets its boolean to 'true'. Afterwards all boolean variables are gathered on all processes and the position of the single 'true' inside the array indicates the necessary process rank. Finally, the value of the specified element can be broadcasted to all processes and stored in the return variable.

4.1.7 pNNLS

The 'parallel NNLS' routine represents the parallelized active set algorithm to solve a non-negative least squares problem. The necessary arguments are the following:

Table 4.8: Definition of the arguments of the function 'pNNLS'

name	type	explanation
this	ScaLAPACKMat<NumberType>	parameter matrix A of the NNLS problem
b	const std::shared_ptr<ScaLAPACKMat <NumberType>>	rigth hand side vector of NNLS problem
x	std::shared_ptr<ScaLAPACKMat <NumberType>>	solution vector of the NNLS problem
epsilon	const double	the tolerance of the termination criterion $\|r\| < \epsilon\|b\|$
pmax	const int	the algorithm terminates, as soon as the given maximal number of passive variables is reached
max_iterations	const int	the algorithm terminates, as soon as the given maximal number of iterations is reached

The routine begins with the initialization of all ScaLAPACK vectors and matrices and the passive set as an empty 'std::vector<int>'.

Then, the outer loop begins as a do-while loop with the following termination criterion:

$$
\begin{gathered}
'p < pmax \\
\&\&\ r.frobenius\_norm() > epsilon * b->frobenius\_norm() \\
\&\&\ all\_wi\_negative == false \\
\&\&\ iteration < max\_iterations'
\end{gathered}
$$

Subsequently, the vector r is computed as $r = Q_p\,[0_p, g_z]$ with the help of the ScaLAPACK routine PxORMQR, where only the first p columns of $Asub$ are used for the computation of Q_p. Beforehand, r is set to $[0_p, g_z]$, as the ScaLAPACK routine takes as argument the vector of the multiplication and overwrites it with the solution vector. Therefore, the first p elements of r are set to zero by using the function 'set_element_to_value'

in a for-loop iteration and the remaining elements are copied from the vector g utilizing the function 'copy_to'. Only in the very first iteration this multiplication is skipped, because with $p = 0$ and the orthogonal rotation matrix just corresponds to an identity matrix.

Afterwards the vector w is computed as $w = A^T r$ with the help of a variant of the function 'mult', that multiplies a transposed ScaLAPACK matrix with a normal one. The next step is to find the largest value in w while only considering the indices not currently in the passive set. Therefore, a vector w_active with the same dimensions than w is created and all active elements are copied to it by using 'set_element_to_value', while the elements corresponding to the passive set remain zero. By means of the function 'max_value', the largest value $wmax$ with its index $imax$ is finally found within the vector w_active.

If this value $wmax$ is negative, the algorithm terminates by setting the boolean variable $all\_wi\_negative$ for the termination criterion to true, as no positive w_i can be found anymore. Otherwise, it goes on with checking the index $imax$ for being a good candidate to be added to the passive set. Thus, the column of A belonging to the index $imax$ is examined to be linear independent of the first p columns of $Asub$. Also, the index is analyzed to create a positive value $ztest$ when introduced into the solution y. For this, the QR-decomposition as well as the vector g are updated with copied variables of $Asub$ and g and $ztest$ is computed as

$$g\_copy- > return\_element(p-1,0)/Asub\_copy- > return\_element(p-1,p-1).$$

If this precomputed new solution variable $ztest$ is positive and the corresponding column is linear independent, the index $imax$ is a good candidate and is used in the further computations. Else, it is rejected as candidate by setting its corresponding variable $wmax$ to zero and a new $wmax$ has to be found.

The suitable index $imax$ is then added to the passive set via 'push_back' and its size p is augmented by one. The QR-decomposition and the vector g are updated for the new column $imax$ by utilizing the two corresponding functions 'updateQR' and 'updateG'. Subsequently, the triangular system of equations ($R_p y_p = g_p$) is solved via the ScaLAPACK routine PxTRSV using only the variables in the passive set. This means, only the first p columns of $Asub$ are used for R_p. A vector yp with same dimensions than y is created as solution vector to still keep the prior solution y for an eventually modifying. As the ScaLAPACK routine takes the right hand side vector as argument, yp is set to g_p, which consists of the first p elements of g, while the remaining ones stay at zero. To set yp equal $[g_p, 0_z]$, the functions 'copy_to' and 'set_element_to_value' are used.

After solving the triangular system for yp, the smallest value within the first p elements

of yp is found via 'min_value'. In case this value $ymin$ is smaller than or equal to zero, the inner loop of the algorithm is entered to modify the solution and keep it from containing non-positive elements.

Thus, *alpha* has to be computed according to formula (2.22) and therefore a 'std::vector<double>' *alpha_vec* is created containing all possible values for *alpha*, before the smallest value is found within it via 'min_value'. Having computed *alpha*, the vector y can be updated as interpolation between the previous feasible solution y and the new but infeasible one yp according to formula (2.21) using the function 'add'.

As some variables of yp are shifted to zero by the described update, they need to be deleted from the passive set. Therefore, a set p_0 is created as 'std::vector<int>' with all indices, whose variables y_i are non-positive. The smallest index in p_0 is saved as $qmin$, indicting the index of the most left column, that has been deleted, for the later on use in the QR-decomposition update. Then, a new empty set is generated with the indices from the passive set, excluding the indices from p_0, and finally the passive set is replaced by that new set while p is diminished by one. At last, all elements of the vector y behind the deleted one have to be shipped forward to close the gap by using the functions 'copy_to' and 'set_element_to_value'.

Owing to the deletion of one or several columns, the QR-decomposition and the vector g need to be updated again via the functions 'updateQR' and 'updateG'. Subsequently, the triangular system of equations can be solved once more for yp with the help of the ScaLAPACK routine PxTRSV using only the variables in the passive set. Therefore, yp is set beforehand to the right hand side vector $[g_p, 0_z]$ using 'copy_to' and 'set_element_to_value'.

The smallest value in the solution vector yp is found via 'min_value' and examined. In case of a nonpositive $ymin$, the algorithm runs the inner loop iteration again. Otherwise, the loop is exited with a new feasible solution vector yp. Therefore, the previous solution y is replaced by yp and the solution vector x from the initial problem is updated via reordering of the values in y according to the passive set. To be more precise, all variables y_i with index in the passive set are placed inside x at the position indicated by their index, while the remaining elements are set to zero.

Afterwards, the $if(wmax > 0)$-loop terminates, just as the outer do-while-loop iteration, if at least one of the termination criteria is fulfilled. Finally the active termination criterion, the number of passed outer-loop iterations and the residual as $norm(r)/norm(b)$ are presented on the terminal.

An overview of the whole algorithm with its loops and most important steps can be seen in figure (4.4).

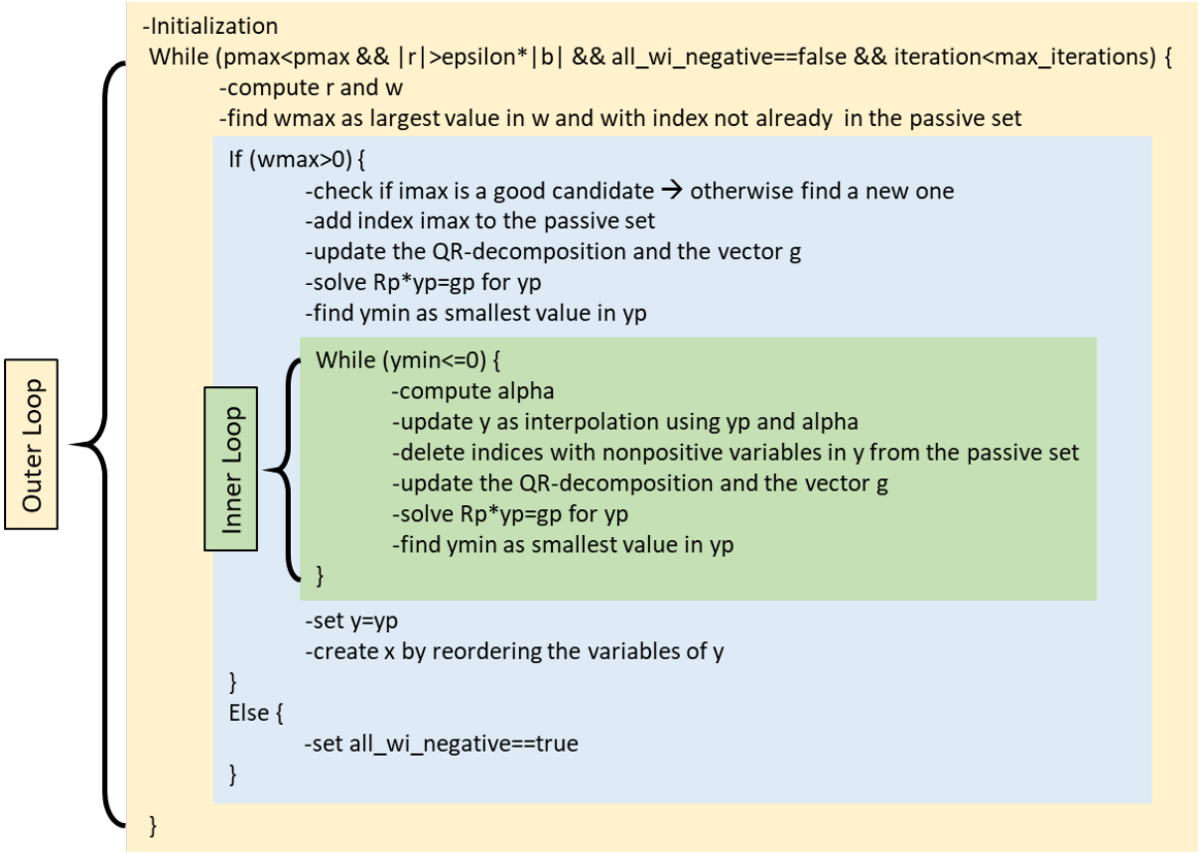

Figure 4.4: Overview over the pNNLS algorithm

The working code can be accessed via 'https://github.com/christinaschwarz/pNNLS.git'.

4.2 Problems during the implementation

In this section some of the problems, that arose during the implementation are mentioned and their solution is explained in detail.

- All new written routines in the file ScaLAPACKMat.cpp have to be defined in the corresponding header file ScaLAPACKMat.hpp as well.
- The three used ScaLAPACK routines have to be implemented as template functions in the file ScaLAPACK.templates.hpp, so that they can be used with single and double precision variables, depending on the 'NumberType' of the ScaLAPACKMatrix. The 'x' inside the routines name is then replaced according to the template type by either 'S' or 'D'.
- In particular, one need to be very careful with all indices. As the ScaLAPACK

routines are written in Fortran, they begin counting with index one, whereas for example all 'std::vector<>' and the self written functions 'min_value' and 'max_value' begin with index zero. But on the other hand side, the variable p as amount of indices in the passive set actually starts at one. Although p equals zero at the complete beginning of the algorithm, it is only used in the ScaLAPACK routines after taking the value one.
Also, one needs to be really careful to not mix up the row and column indices. In general row indices are indicated by $i \in 1, ..., m$ and column indices with $j \in 1, ..., n$.

- For debugging purpose, a 5×5 NNLS-example problem is created. Unfortunately, the algorithm only runs the outer loop and never enters the inner loop. Therefore it can only be used for examination of the outer loop. The vector w, the elementary reflectors, the triangular matrices and the solution vector are manually computed and compared to the results of the coded algorithm. But as the matrices H_i and Q are not explicitly stored, it is hard to compare them with the manually computed ones.

- The original matrix A cannot be used for the QR-decomposition, as the matrix would be overwritten by the triangular matrix R and the elementary reflectors. Therefore an additional matrix $Asub$ has to be created for the decomposition, so that the original matrix A can be kept unchanged.

- Because of rounding errors and inaccuracy of the program, some variables of y may not be exactly set to zero by the updating procedure in the inner loop. Therefore all elements of y with values in the range of $e - 10$ are manually set to zero after the procedure.

- For the multiplication of the vector g with a single elementary reflector, one need to be careful with the indices. As the routine PxORMQR always supposes the elementary reflectors being stored in the lower part of the triangular matrix, the submatrix indicated by IA and JA needs to start at the element (p, p), when multiplying with H_p. H_p is therefore of lower dimension than for example H_1, which is stored in the first column of $Asub$.
Consequently, the dimensions of the vector g need to coincide with the dimensions of H_p, so that g also has to begin at the p-th element.

- To compare the results with the sequential code, terminal outputs had to be inserted into the Fortran code, which has been quite challenging.

- To only have the first process doing all terminal outputs in the parallel code, the following can be used:

$$dealii :: ConditionalOStream\ \ pcout\ \ (std :: cout,\ \ (dealii :: Utilities :: MPI :: this\_mpi\_process\ \ (this- > grid- > mpi\_communicator)\ \ == 0\ \ )\ \ )$$

- Very challenging was also the parallelization of the code. So instead of the simple 'local_el'-function, the new functions 'set_element_to_value' and 'return_element' must have been implemented.

A huge problem during the implementation were small mistakes, only vague explanations or missing steps in the documentation of the US army report [6]:

- First of all in the documentation of the algorithm n was mixed up with m as dimension of the vector g and the transposed sign was missing in the computation $g = Q_p^T b$.

- While testing the example problem of dimensions 2700×5120, it occurred again and again, that a index was found as $imax$, but, when introduced into the solution, it generated a non-positive value. In the updating procedure (formula (2.22)) then $alpha$ takes the value zero as the corresponding y_i from the previous solution equals zero. This occurs as a consequence of y_i having one element less than yp, because it represents the solution of the previous iteration, where p was one smaller. If $alpha$ equals zero, the solution vector y is never updated but stays constant and no index is deleted from the passive set.
  To keep $alpha$ from becoming zero, the new element of the solution vector must never take a non-positive value. Therefore, the index has to be checked for being a good candidate and producing only a positive value, when introduced into the solution. This step can be contemplated in the sequential code from Lawson and Hanson [8] but it is missing in the US army report. Nevertheless it is established in the parallel algorithm to guarantee a working code.

- Another difficulty were the vague explications concerning the updating of the QR-decomposition. In the US army report only '$QR(k, [q_k, q_{k+1}, ...q_p])$' is marked as computation step concerning the factorization as well as the use of the routine 'PxGEQRF'. But unfortunately it was never explained, what this notation signifies and what for instance the letter 'k' indicates.
  For debugging purpose, a simplified approach was used for the function 'updateQR', that always recomputes the factorization completely afresh. Therefore only the original columns of A are necessary and $Asub$ needs to be restored with

these columns before decomposing it again.
After an intense study of the sequential code[8] and different methods for the QR-updating, the functioning of the update became clearer (see chapter (2.5)). So as the routine PxGEQRF always starts at the upper left element with the decomposition, only the lower part of the new columns may be used. Therefore the first entry indicated by IA and JA is the element (p, p). Otherwise the QR-decomposition would already start at the first row of the indicated columns.

- Also the definition of gp is very unclear in the US army report. At the beginning of the implementation it was assumed, that the vector g is partitioned onto g_p and g_z in the same way as the vector y according to the passive set. As for the solving of a quadratic $p \times p$ triangular system only the first p columns and rows are used, gp instead just contains the first p elements of g.

Chapter 5

Conclusion

To conclude, the implementation of a parallel version of the non-negative least square solver in C++ using ScaLAPACK was successful, as the code produces the same solution for the test problem with dimensions 2700×5120 than the sequential code [8]. When comparing the indices, that are added and deleted in the course of the algorithm, with the sequential algorithm, they are absolutely identical, as well as p, the number of elements in the passive set.

The problem was finally tested on a laptop with one CPU and four processes on a different number of processes and with a different numbers of iterations. The resulting residual an the measured running time can be seen in figure (5.1) and (5.2). For these tests the configuration mode was changed from 'Debugging' to Release'.

The block size (figure (5.1)), which is used for the partition of the ScaLAPACK matrix has only minor influence on the running time, because it is actually irrelevant, which

blocksize	number of processes			
	1	2	3	4
2	71,1	54,8	53,5	55,3
4	71,2	53,2	52,1	54,6
8	71,2	51,9	51,6	54,2
16	71,2	52,4	51,7	54,0
32	71,3	52,3	**51,5**	53,7
64	71,2	52,3	**51,4**	53,6
128	71,2	51,9	51,8	53,6
256	71,2	51,7	51,6	53,4
512	71,4	51,9	53,7	55,9

Table 5.1: Overview over the running time (in seconds) of a NNLS problem with dimensions 2700×5120 using 50 iterations and different block sizes

iterations	20	50	100	500	750	1000
1 process	27,3	71,3	156,3	780,4	1166,5	1503,9
2 processes	19,9	52,3	116,5	584,6	866,5	1109,7
3 processes	19,6	51,5	109,3	546,6	844,1	1091,7
4 processes	20,5	53,7	117,7	600,6	929,6	1145,8
residual	0,579871	0,382858	0,18319	3,83E-04	1,88E-05	1,25E-06

Table 5.2: Overview over the running time (in seconds) of a NNLS problem with dimensions 2700×5120 using a blocksize of 32 and different numbers of iterations

rows a process owns. As there is indeed no partition, if the program is executed on only one process, there is almost no change at all in the required time. The best result is acquired on three processes and a block size of 32 or 64.

In figure (5.2) it is visible, that the residual decreases in a good manner by augmenting the number of iterations. Also the necessary wall time usually decreases by augmenting the number of processes. Only on four processes, the code needs more time than expected.

When compared to the sequential code, this parallel version still runs significantly slower. For example with 500 iterations, the sequential code takes only 24.387 seconds, whereas the implemented version needs at least 546,648 seconds. This significant difference results in the fact, that the implemented version is not yet optimized regarding. During the following 'profiling' process, all functions are examined exactly regarding their efficiency and bottlenecks of the code are identified using an extra software. Also the utilization of more processes might speed up the code little bit.

As the further treatment of the code for optimization exceeds the extend of this thesis, the code is not yet integrated into the open-source library deal.II. Nevertheless this thesis is a good base for further treatment and future publication of the code.

Bibliography

[1] Luo, Y.; Duraiswami, R. *Efficient Parallel Non-Negative Least Squares on Multicore Architectures*, SIAM Journal on Scientific Computing 2011

[2] Kim, D.; Sra, S.; Dhillon, I. S. *A New Projected Quasi-Newton Approach for the Nonnegative Least Squares Problem*, Department of Computer Sciences, The University of Texas at Austin 2006

[3] Liu, H.; Li, X.; Zheng, X. *Solving non-negative matrix factorization by alternating least squares witch modified strategy*, 2012

[4] He, Z.; Xie, S. et al *Symmetric Nonnegative Matrix Factorization: Algorithms and Applications to Probabilistic Clustering*, IEEE, 2011

[5] Rajko, R.; Zheng, Y. *Distance algorithm based procedure for non-negative least squares*, Journal of Chemometrics 2014

[6] Collins, J. P. *Parallel Nonnegative Least Squares Solvers for Model Order Reduction*, US Army Research Laboratory 2016

[7] Brands, B.; Davydov, D.; Mergheim, J.; Steinmann, P. *Reduced-Order Modeling and Homogenization on Magneto-Mechanics: A Numerical Comparison of Established Hyper-Reduction Methods*, Chair of Applied Mechanics, Friedrich-Alexander Universität Erlangen-Nürnberg 2019

[8] Lawson, C. L.; Hanson, R. J. *Solving Least Squares Problems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974

[9] Bro, R.; De Jong, S. *A Fast Non-Negativity-Constrained Least Squares Algorithm*, Journal of Chemometrics 1997

[10] Myre, J. M.; Frahm, E.; Lilja, D. J.; Saar, M. S. *TNT-NN: A Fast Active Set Method for Solving Large Non-Negative Least Squares Problems*, International Conference on Computational Science, Zurich, 2017

[11] Myre, J. M.; Frahm, E.; Lilja, D. J.; Saar, M. S. *TNT: A Solver for Large Dense Least-Squares Problems That Takes Conjugate Gradient From Bad in Theory, to Good in Practice*, IEEE, Vancouver 2018

[12] Rosen, J. B. *The Gradient Projection Method for Nonlinear Programming. Part I. Linear Constraints*, J. Soc. Indust: Appl. Math. 1960

[13] Bertsekas, D. P. *Projected Newton Methods for Optimization Problems with simple Constraints*, Society for Industrial and Applied Mathematics, Siam J. Control and Optimization 1982

[14] Polyak, R. A. *Projected Gradient Method for Non-Negative Least Square*, Contemporary Mathematics 2015

[15] Ang, A. *Nonnegative Least Squares: PGD, accelerated PGD and with restarts*, Mathématique et recherche opérationelle UMOS, Belgium 2017

[16] Bellavia, S.; Macconi, M.; Morini, B. *An interior point Newton-like method for nonnegative least-squares problems with degenerate solution*, Numerical Linear Algebra with applications, Wiley InterScience 2006

[17] Portugal, L. F.; Judice, J. J.; Vicente, L. N. *A comparison of block pivoting and interior-point algorithms for linear least squares problems with nonnegative variables*, Mathematics of Computation, Volume 6, 1994

[18] Kim, J.; Park, H. *Fast Nonnegative Matrix Factorization: An Active-Set-Like Method and Comparison*, SIAM Journal on Scientific Computing 2011

[19] Franc, V.; Hlavác, V.; Navara, M. *Sequential Coordinate-Wise Algorithm for the Non-negative Least Squares Problem*, In: Gagalowicz A., Philips W. (eds) 'Computer Analysis of Images and Patterns', Springer-Verlag Berlin Heidelberg 2005

[20] Kim, H.; Park, H *Nonegative Matrix Factorization Based on Alternating Nonnegativity Constrained Least Squares and Active Set Method*, Society for Industrial and Applied Mathematics 2008

[21] Zeb, S.; Yousaf, M. *Updating QR factorization procedure for solution of linear least squares problem with equality constraints*, Journal of Inequalities and Applications 2017

[22] Hammarling, S.; Higham, N. J.; Lucas, C. *LAPACK-Style Codes for Pivoted Cholesky and QR Updating*, In: Kågström B., Elmroth E., Dongarra J., Waśniewski J. (eds) 'Applied Parallel Computing. State of the Art in Scientific Computing', Springer-Verlag Berlin Heidelberg 2007

[23] Olsson, O.; Ivarsson, T. *Using the QR Factorization to swiftly update least squares problem*, Center for Mathematical Sciences, Numerical Analysis, The Faculty of Engineering at Lund University, LTH 2014

[24] Andrew, R.; Dingle, N. *Implementing QR factorization updating algorithms on GPUs*, School of Mathematics, University of Manchester, In: 'Parallel Computing', 2014

[25] Hammarling, S.; Lucas, C. *Updating the QR Factorization and the Least Squares Problem*, 2008

[26] Richter, T.; Wick, T. *Einführung in die Numerische Mathematik*, Springer-Verlag GmbH Deutschland 2017

[27] Kail, J. *Housholder-Transformation*, 2015

[28] Yoo, K.; Park, H. *Accurate Downdating of a modified Gram-Schmidt QR Decomposition*, Department of Computer SCience, University of Minnesota, Minneapolis 1996

[29] Balaji, P.; Hoefler, T. *MPI for Dummies*, ETH Zürich 2013

[30] author unknown *MPI Forum*, [date unknown, accessed 2020 Sept 10], https://www.mpi-forum.org/docs/

[31] Hageeier, A. *Einführung in die Parallelprogrammierung*, Jülich Supercomputing Centre 2018

[32] Lanser, M. *Grundlagen zu MPI*, Mathematisches Institut, Universität zu Köln 2017

[33] author unknown *deal ii: Utilities::MPI Namespace Reference*, [date unknown, accessed 2020 Sept 10], https://www.dealii.org/current/doxygen/deal.II/namespaceUtilities_1_1MPI.html

[34] Douglas, G.; Troyer, M. *Boost.MPI*, Boost Software License 2007

[35] Dongarra, J. J. et al *ScaLAPACK Users' Guide*, Siam Society for Industrial and Applied Mathematics, Philadelphia 1997

[36] author unknown *ScaLAPACK Scalable Linear Algebra PACKage*, [date unknown, accessed 2020 Sept 10], http://www.netlib.org/scalapack/

[37] Charara, A.; Gates, M.; Kurzak, J.; YarKhan, A.; Dongarra, J. *SLATE Developers' Guide*, Innovative Computing Laboratory 2019

[38] author unknown *Software for Linear Algebra Targeting Exascale (SLATE)*, ECP Slate ICL [date unknown, accessed 2020 Sept 10], https://icl.utk.edu/slate/

[39] author unknown *dealii, ScaLAPACKMatrix*, [date unknown, accessed 2020 Sept 10], https://www.dealii.org/current/doxygen/deal.II/classScaLAPACKMatrix.html