Universität Stuttgart

Geodätisches Institut

Variance-covariance matrix estimation with LSQR in a parallel programming environment

Diplomarbeit im Studiengang

Geodäsie und Geoinformatik

an der Universität Stuttgart

Ronggang Guo

Stuttgart, September 2008

Betreuer:

Dr.-Ing. Oliver Baur
Universität Stuttgart

Prof. Dr.-Ing. Nico Sneeuw
Universität Stuttgart

Acknowledgment

It is a pleasure to acknowledge Dr.-Ing. Oliver Baur and Prof. Dr.-Ing. Nico Sneeuw (Institute of Geodesy, University Stuttgart), for their guidance, discussion, advice and also encouragement. Without their support I could not have finished this thesis in time. Special thanks go to Dr.-Ing. Oliver Baur for his help to tackle various difficulties and for spending time on discussions. In addition, I want to address thanks to Prof. Dr.-Ing. Nico Sneeuw for his serious checking of this thesis.
Furthermore, I highly acknowledge the help from my friends and colleagues, for their advices and ideas. Finally, I would like to dedicate this thesis to my parents, and my family. With their support, I could concentrate on my studies here in Stuttgart.

Selbstständigkeitserklärung

Hiermit versichere ich, Ronggang Guo, die vorliegende Arbeit

Variance-covariance matrix estimation with LSQR in a parallel programming environment

selbständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Datum, Ort

Unterschrift

Abstract

Knowledge about the gravity field allows an insight into the structure and dynamics of the earth. It provides the geoid as the most important physical reference surface in geodesy and oceanography. Since 2000, the CHAMP (CHallenging Mini-satellite Payload) mission detects the structure of the global gravity field, followed by the launch of GRACE (Gravity Recovery And Climate Experiment) in 2002. In 2008, finally, the GOCE (Gravity field and steady-state Ocean Circulation Explorer) satellite is supposed to be set in orbit. These missions demonstrate satellite-based gravity field recovery to be at the center of geo-scientific interest.
Interpretation and evaluation of satellite observations are difficult, especially the determination of the unknown gravity field parameters from a huge amount of measurements. Because of the immense demand for memory and computing time, the occurring systems of equations pose a real numerical challenge. Therefore, High-Performance Computing (HPC) is commonly adopted to overcome computational problems. Basically, parallel programming with MPI and OpenMP routines allows to speed up the solution process considerably.
In this thesis, firstly global gravity field modelling by means of satellite observations is reviewed. Secondly, the LSQR method (Least-Squares using QR factorization) is introduced in detail in order to solve the resulting least-squares problems. Because the LSQR method is an iterative solver, it basically can not provide the variance-covariance information of the parameter estimate. To investigate the approximate computation of the variance-covariance matrix, two methods are introduced. The first one is based on the generalized inverse of the design matrix. The second approach applies Monte-Carlo integration techniques. Because parallel programming is very helpful to implement such iterative methods, it is necessary to introduce some basic principles and concepts about HPC.

Keywords: GOCE, gravity field determination, parallel programming, high performance computing, MPI, OpenMP, LSQR, covariance estimation, Monte Carlo simulation.

Zusammenfassung

Das Wissen über das Erdschwerefeld ermöglicht einen Einblick in die Struktur und Dynamik der Erde. Es liefert das Geoid als die wichtigste physikalische Bezugsoberfläche in der Geodäsie und Ozeanographie. Seit 2000 erfasst die CHAMP (CHallenging Minisatellite Payload) Mission die Struktur des globalen Schwerefeldes, gefolgt vom Start der GRACE (Gravity Recovery And Climate Experiment) Mission im Jahr 2002. Im Jahr 2008 ist der Start des GOCE (Gravity field and steady-state Ocean Circulation Explorer) Satelliten vorgesehen. Diese Missionen zeigen, dass die satellitengestützte Schwerefeldbestimmung im Zentrum des geowissenschaftlichen Interesses liegt.
Die Interpretation und Auswertung von Satellitenbeobachtungen sind schwierig, vor allem die Bestimmung der unbekannten Parameter des Schwerefeldes aus einer großen Anzahl von Messungen. Aufgrund der immensen Nachfrage nach Speicher und Rechenzeit, stellen die auftretenden Gleichungssysteme eine große numerische Herausforderung dar. Zur Behebung der rechentechnischen Schwierigkeiten kommt deshalb üblicherweise High Performance Computing (HPC) zum Einsatz. Grundsätzlich kann die parallele Programmierung mit MPI und OpenMP den Lösungsprozess deutlich beschleunigen.
In dieser Arbeit wird zunächst ein kurzer überblick über die satellitengestützte Schwerefeldbestimmung gegeben. Danach wird die LSQR Methode ausführlich eingeführt um die resultierenden least-squares Probleme zu lösen. Da die LSQR Methode ein iterativer Löser ist, kann das Verfahren die Varianz-Kovarianz Informationen der Parameterschätzung grundsätzlich nicht liefern. Um die approximierte Berechnung der Varianz-Kovarianz Matrix zu untersuchen, werden zwei Methoden eingeführt. Die erste basiert auf der generalisierten Inverse des Designmatrix. Der zweite Ansatz gründet auf Monte-Carlo Integration. Da parallele Programmierung zweckdienlich bei der Umsetzung solcher iterativen Methoden ist, ist es notwendig, einige grundlegende Prinzipien und Konzepte zum Thema HPC einzuführen.

Schlüsselwörter: GOCE, Schwerefeldbestimmung, parallele Programmierung, Hochleistungsrechnen, MPI, OpenMP, LSQR, Varianz-Kovarianz Matrix, Monte Carlo Simulation.

Contents

Chapter 1

Global gravity field recovery using the LSQR algorithm

1.1 Determination of the global gravity field

Nowadays, gravity field determination is typically based on satellite data, since only by satellite observations the global coverage with largely homogeneous data can be obtained. Theoretically, it is possible to perform terrestrial gravity measurements on a global scale, but this is not feasible due to both political and logistical reasons.
A satellite model is referred to as a gravity field model derived from satellite observations only. The disadvantage of such a model is its reduced sensitivity to local gravity field features with increasing orbit altitude. Observations on the surface of the earth can overcome regional deficiencies. A gravity field model based on both satellite and surface data is referred to as combination model.

1.2 Global gravity field model

A global gravity field model is represented as a set of spherical harmonic coefficients $\{\overline{c}_{l,m}, \overline{s}_{l,m}\}$, The degree l and order m specify the resolution of the spherical harmonic expansion, the dash indicates that the coefficients refer to the fully normalized spherical harmonics expansion, which is used frequently in satellite geodesy. The potential spherical harmonic series for the external space of the earth is (Heiskanen Moritz, 1967):

$$V = \frac{GM}{r} + \frac{GM}{R} \sum_{l=2}^{L} \sum_{m=0}^{l} \left(\frac{R}{r}\right)^{l+1} \overline{P}_{l,m}(\sin\varphi)(\overline{c}_{l,m}\cos(m\lambda) + \overline{s}_{l,m}\sin(m\lambda)). \qquad (1.1)$$

GM denotes the geocentric constant, R the earth radius and r the distance from the geocenter to the evaluation point. λ and φ are the longitude and latitude of the point, $\overline{c}_{l,m}$ and $\overline{s}_{l,m}$ the gravity field coefficients (cf. figure 1.1), and $\overline{P}_{l,m}(\sin\varphi)$ the fully nor-

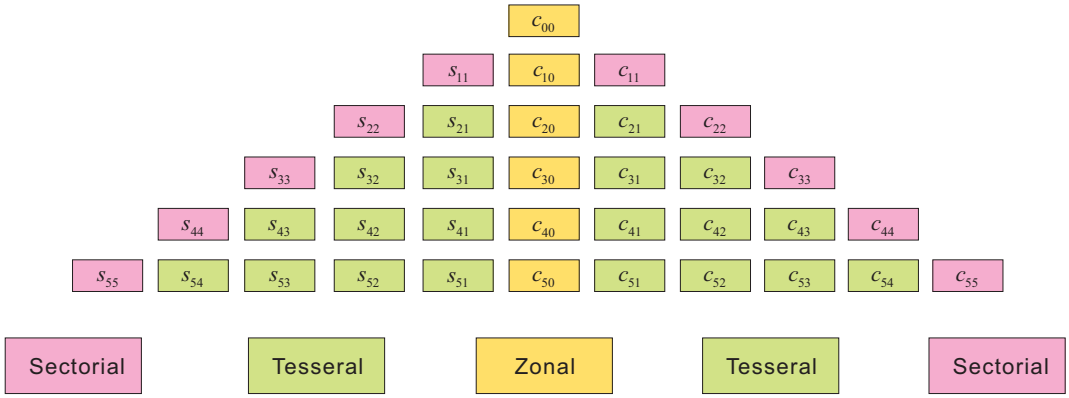

Figure 1.1: Spherical harmonic coefficients

malized associated legendre functions of the first kind. The higher the series resolution L, the better is the approximation to the real gravitational potential.

1.3 Observation equation

Satellite orbit tracking provides the satellite positions $\mathbf{x}(\lambda, \varphi, r)$. Satellite accelerations $\ddot{\mathbf{x}}(\lambda, \varphi, r)$ are derived by second-order numerical differentiation (Reubelt, Austen, Grafarend, 2003). As a matter of fact, the acceleration is just the gradient of the gravitation potential (1.1). Thus, the observation equation becomes:

$$\ddot{\mathbf{x}} = \mathrm{grad}V, \tag{1.2}$$

and the gradient operator in spherical coordinates is defined as

$$\mathrm{grad} = \mathbf{e}_r \frac{\partial}{\partial r} + \mathbf{e}_\lambda \frac{1}{r\cos\lambda}\frac{\partial}{\partial \lambda} + \mathbf{e}_\varphi \frac{1}{r}\frac{\partial}{\partial \varphi}. \tag{1.3}$$

Therefore, from equations (1.1) and (1.3), the gradient of gravitational potential yields:

$$\begin{aligned}\mathrm{grad}V = \frac{GM}{R^2}\sum_{l=0}^{L}\sum_{m=0}^{l}\left(\frac{R}{r}\right)^{l+2} &[-(l+1)\bar{P}_{l,m}(\sin\varphi)(\bar{c}_{l,m}\cos(m\lambda) + \bar{s}_{l,m}\sin(m\lambda))\mathbf{e}_r \\ &+ \frac{1}{\cos\varphi}\bar{P}_{l,m}(\sin\varphi)(-\bar{c}_{l,m}m\sin(m\lambda) + \bar{s}_{l,m}m\cos(m\lambda))\mathbf{e}_\lambda \\ &+ \frac{\partial\bar{P}_{l,m}(\sin\varphi)}{\partial\varphi}(\bar{c}_{l,m}\cos m\lambda + \bar{s}_{l,m}\sin(m\lambda))\mathbf{e}_\varphi]\end{aligned} \tag{1.4}$$

Because the two components of the surface gradient contain the same information as the radial derivative, I only consider the first term on the right hand side in (1.4), hence:

$$\frac{\partial V}{\partial r} = \frac{GM}{R^2} \sum_{l=0}^{L} \sum_{m=0}^{l} \left(\frac{R}{r}\right)^{l+2} [-(l+1)\bar{P}_{l,m}(\sin\varphi)(\bar{c}_{l,m}\cos(m\lambda) + \bar{s}_{l,m}\sin(m\lambda))]\mathbf{e}_r. \tag{1.5}$$

Equation (1.5) constitutes a linear system of equations

$$\mathbf{y} = \mathbf{A}\mathbf{x}, \tag{1.6}$$

where $\mathbf{y} \in \mathbb{R}^{n\times 1}$, $\mathbf{A} \in \mathbb{R}^{n\times u}$, and $\mathbf{x} \in \mathbb{R}^{u\times 1}$ (u denotes the number of unknowns and n refers to the number of observations).
The vector of unknown parameters becomes

$$\mathbf{x} = [\bar{c}_{00}, \ldots, \bar{c}_{L0}, \bar{c}_{11}, \ldots, \bar{c}_{L1}, \bar{c}_{22}, \ldots, \bar{c}_{L2}, \ldots, \bar{c}_{LL}, \bar{s}_{11}, \ldots, \bar{s}_{L1}, \bar{s}_{22}, \ldots, \bar{s}_{L2}, \ldots, \bar{s}_{LL}], \tag{1.7}$$

As the coordinate system origin coincides with the earth's center of mass it follows

$$\bar{c}_{10} = \bar{c}_{11} = \bar{s}_{10} = 0.$$

Therefore, expression (1.7) reduces to

$$\mathbf{x} = [\bar{c}_{00}, \bar{c}_{20}, \ldots, \bar{c}_{L0}, \bar{c}_{21}, \ldots, \bar{c}_{L1}, \bar{c}_{22}, \ldots, \bar{c}_{L2}, \ldots, \bar{c}_{LL}, \bar{s}_{21}, \ldots, \bar{s}_{L1}, \bar{s}_{22}, \ldots, \bar{s}_{L2}, \ldots, \bar{s}_{LL}]. \tag{1.8}$$

Thus, the number of unknowns amounts to

$$u = l_{\max}^2 + 2 \cdot l_{\max} - 2. \tag{1.9}$$

The substitutions

$$KC_{l,m} = \frac{GM}{R^2} \left(\frac{R}{r}\right)^{l+2} \left(-(l+1)\bar{P}_{l,m}(\sin\varphi)\cos(m\lambda)\right)$$

$$KS_{l,m} = \frac{GM}{R^2} \left(\frac{R}{r}\right)^{l+2} \left(-(l+1)\bar{P}_{l,m}(\sin\varphi)\cos(m\lambda)\right),$$

yield

$$\begin{aligned}\mathbf{A}_i(\lambda_i, \varphi_i, r_i) = [&KC_{00}, KC_{20}, \ldots, KC_{L0}, KC_{21}, \ldots, KC_{L1}, \ldots, KC_{LL},\\ &KS_{21}, \ldots, KS_{L1}, KS_{22}, \ldots, KS_{L2}, \ldots, KS_{LL}],\end{aligned}$$

where $\mathbf{A}_i(\lambda_i, \varphi_i, r_i)$ denotes the i-th row of the design matrix $\mathbf{A}$, related to position $(\lambda_i, \varphi_i, r_i)$.
The total design matrix $\mathbf{A}$ can be expressed as follows

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1(\lambda_1, \varphi_1, r_1) \\ \mathbf{A}_2(\lambda_2, \varphi_2, r_2) \\ \mathbf{A}_3(\lambda_3, \varphi_3, r_3) \\ \cdots \end{bmatrix},$$

and the observation vector as

$$\mathbf{y} = \begin{bmatrix} \ddot{\mathbf{x}}_r(\lambda_1, \varphi_1, r_1) \\ \ddot{\mathbf{x}}_r(\lambda_2, \varphi_2, r_2) \\ \ddot{\mathbf{x}}_r(\lambda_3, \varphi_3, r_3) \\ \cdots \end{bmatrix}$$

1.4 The LSQR method

1.4.1 Motivation

In order to solve the linear system of equations (1.6), Least-Squares (LS) methods are usually adopted (Pail & Plank, 2002; Ditmar et al., 2003; Schuh, 1996). Minimization of squared residuals subject to $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$ yields the best linear unbiased estimate (Koch, 1999)

$$\hat{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y} = \mathbf{N}^{-1}\mathbf{A}^T\mathbf{y}. \tag{1.10}$$

With increasing number of satellite observations, the design matrix $\mathbf{A}$ and normal matrix $\mathbf{N}$ enlarge accordingly. This comes along with huge memory and runtime requirements. Hence, brute-force evaluation of (1.10) may not be feasible from the computational point of view. Opposed to normal equation system inversion, the LSQR (Least-Squares using QR decomposition) method solves the minimization problem

$$\min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$$

iteratively. With increasing iterations k, the estimate $\hat{\mathbf{x}}_k$ approaches more and more the exact least-squares solution $\hat{\mathbf{x}}$. The latter is reached after $k = u$ iterations. For LSQR neither the design matrix $\mathbf{A}$ nor the normal equation matrix $\mathbf{N}$ needs to be stored. Moreover, matrix-vector products can be examined row by row. The row-wise processing scheme causes memory requirements to be reduced dramatically. The LSQR solver was published in detail in Paige & Saunders (1982a) and Paige & Saunders (1982b). Its application to geodetic research is carried out in e.g. Kusche & Mayer-Gürr (2001), Baur & Austen (2005) and Baur et al. (2007).

1.4.2 LSQR algorithm

The LSQR method treats LS problems by means of bidiagonalization and QR decomposition (see figures 1.2 and 1.3), in order to solve the linear system of equations (1.6)

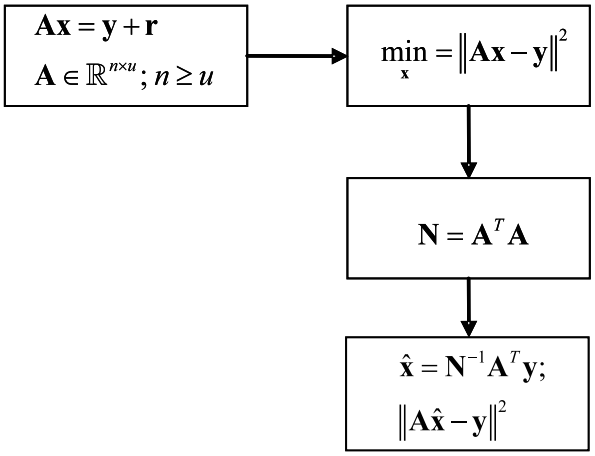

Figure 1.2: Brute-force approach to solve LS problems

iteratively. According to Lanczos lower bidiagonalization (Paige & Saunders, 1982a), the design matrix $\mathbf{A}$ can be split into three matrices subject to

$$\mathbf{A}_k = \mathbf{U}_{k+1}\mathbf{B}_k\mathbf{V}_k^T \qquad k = 1, \ldots, u, \tag{1.11}$$

where $\mathbf{A}_k$ denotes the approximation to the design matrix $\mathbf{A}$ in the k-th iteration, and

$$\mathbf{U}_{k+1} = (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k, \mathbf{u}_{k+1}) \qquad \mathbf{V}_k = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k) \tag{1.12}$$

as well as

$$\mathbf{B}_k = \begin{pmatrix} \alpha_1 & & & & \\ \beta_2 & & & & \\ & \cdots & & & \\ & & \cdots & & \\ & & & \cdots & \\ & & & & \alpha_k \\ & & & & \beta_{k+1} \end{pmatrix} \tag{1.13}$$

hold true, with $\|\mathbf{u}_k\| = \|\mathbf{v}_k\| = 1$ and

$$\mathbf{U}_{k+1}^T\mathbf{U}_{k+1} = \mathbf{I}, \qquad \mathbf{V}_k^T\mathbf{V}_k = \mathbf{I}.$$

With the initial relations

$$\beta_1\mathbf{u}_1 = \mathbf{y}, \qquad \alpha_1\mathbf{v}_1 = \mathbf{A}^T\mathbf{u}_1, \tag{1.14}$$

the bidiagonalization process becomes

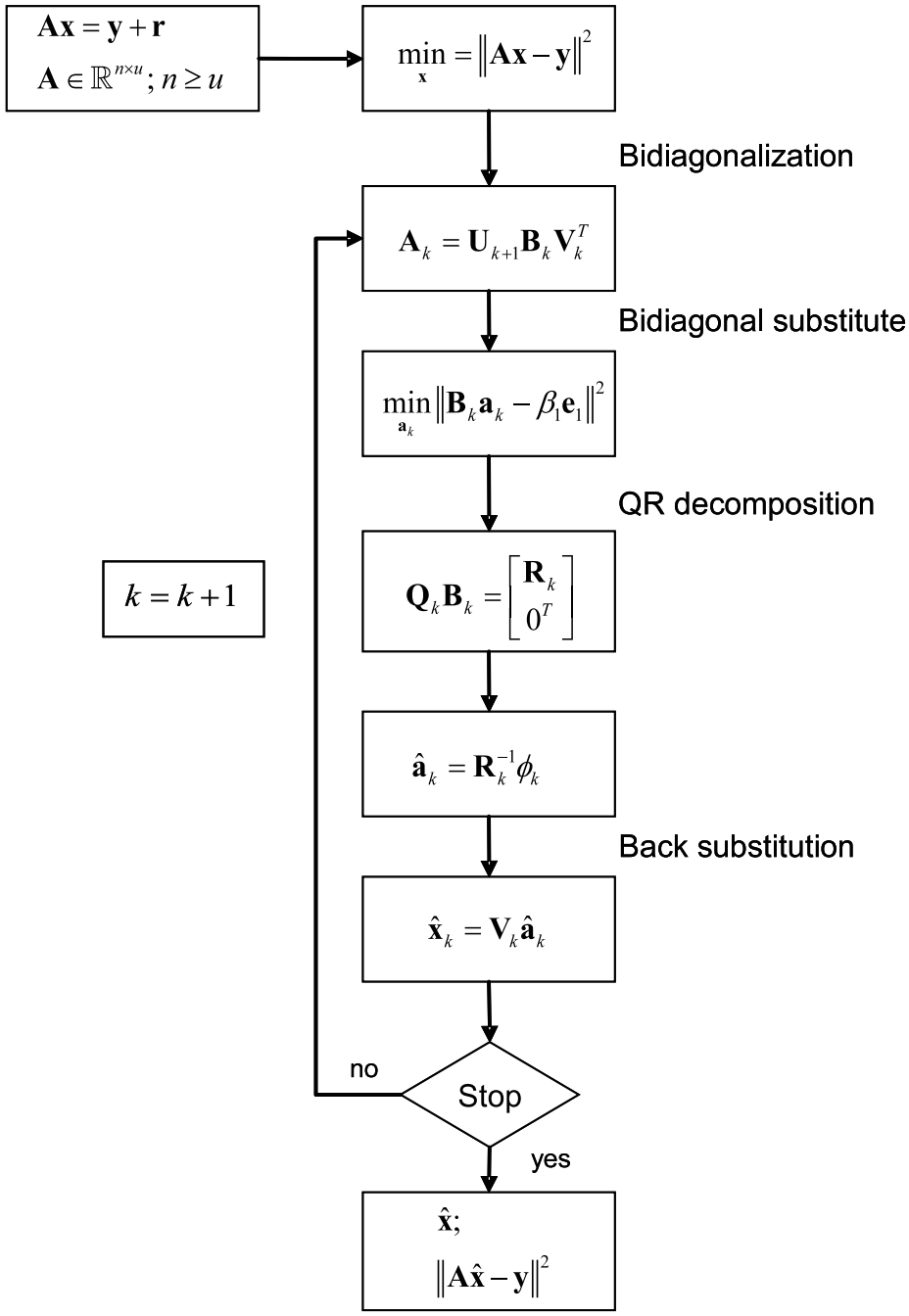

Figure 1.3: LSQR method

$$\beta_{i+1}\mathbf{u}_{i+1} = \mathbf{A}\mathbf{v}_i - \alpha_i\mathbf{u}_i, \qquad \alpha_{i+1}\mathbf{v}_{i+1} = \mathbf{A}^T\mathbf{u}_{i+1} - \beta_{i+1}\mathbf{v}_i, \tag{1.15}$$

where α_k and β_k are positive numbers. $\beta_1\mathbf{u}_1 = \mathbf{y}$ can be rewritten as

$$\mathbf{U}_{k+1}(\beta_1\mathbf{e}_1) = \mathbf{y}, \tag{1.16}$$

where $\mathbf{e}_1$ corresponds to the unit vector $(1, 0, \cdots, 0)^T$. Inserting equations (1.11) and (1.16) in $\min_x \|\mathbf{A}\mathbf{x}_k - \mathbf{y}\|^2$ yields the bidiagonal minimization problem

$$\min_{\mathbf{a}_k} \|\mathbf{B}_k\mathbf{a}_k - \beta_1\mathbf{e}_1\|^2 = \min_{\mathbf{a}_k} \left\| \begin{pmatrix} \alpha_1 & & & & & \\ \beta_2 & \ddots & & & & \\ & & \cdots & & & \\ & & & \cdots & & \\ & & & & \alpha_k & \\ & & & & \beta_{k+1} & \end{pmatrix} \cdot \begin{pmatrix} \hat{a}_1 \\ \vdots \\ \vdots \\ \hat{a}_k \end{pmatrix} - \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \right\|^2. \tag{1.17}$$

In order to solve this problem efficiently, the matrix $\mathbf{B}_k$ is decomposed in $\mathbf{Q}_k$ and $\mathbf{R}_k$ using QR decomposition, where $\mathbf{Q}_k$ is an orthogonal matrix and $\mathbf{R}_k$ is an upper triangular matrix, i.e.,

$$\mathbf{Q}_k\mathbf{B}_k = \begin{pmatrix} \mathbf{R}_k \\ 0^T \end{pmatrix}. \tag{1.18}$$

The QR decomposition is performed by a set of Givens rotations. The Givens rotation matrix is represented as

$$\mathbf{G}_{ij}(\upsilon) = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & & s & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & -s & & & & c & \\ & & & & & & & & 1 \\ & & & & & & & & & \ddots \end{pmatrix}$$

with $c = \cos(\upsilon)$ and $s = \sin(\upsilon)$.
Thereby the cos and sin terms are located at the positions $(i,i), (i,j), (j,i), (j,j)$. The successive multiplication of Givens matrices to the bidiagonal matrix from the left-hand side causes its diagonalization. The first Givens rotation, for example, becomes

$$c_1 = \frac{\alpha_1}{\rho_1} \qquad s_1 = \frac{\beta_2}{\rho_1},$$

with

$$\rho_1 = \sqrt{\alpha_1^2 + \beta_2^2} \neq 0.$$

Thus,

$$\mathbf{G}_{1,2} \begin{bmatrix} \alpha_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \rho_1 \\ 0 \end{bmatrix}.$$

Therefore, $\mathbf{Q}_k$ and $\mathbf{R}_k$ can be computed by

$$\mathbf{Q}_k = \mathbf{G}_{k,k+1}, \mathbf{G}_{k-1,k} \cdots \mathbf{G}_{1,2} \tag{1.19}$$

and

$$\mathbf{R}_k = \begin{pmatrix} \rho_1 & \theta_1 & & & \\ & \rho_2 & \theta_2 & & \\ & & \ddots & \ddots & \\ & & & \rho_{k-1} & \theta_{k-1} \\ & & & & \rho_k \end{pmatrix}, \tag{1.20}$$

with $\theta_k = s_{k-1}\alpha_k$ and $\rho_k = \sqrt{(c_{k-1}\alpha_k)^2 + \beta_{k+1}^2}$. Rewriting (1.18) in

$$\mathbf{Q}_k \mathbf{B}_k = \begin{bmatrix} \mathbf{Q}_{k1} \\ \mathbf{Q}_{k2} \end{bmatrix} \mathbf{B}_k = \begin{bmatrix} \mathbf{R}_k \\ 0^T \end{bmatrix} \tag{1.21}$$

yields the product $\mathbf{Q}_k^T \beta_1 \mathbf{e}_1$ to become

$$\mathbf{Q}_k^T \beta_1 \mathbf{e}_1 = \begin{bmatrix} \mathbf{Q}_{k1}^T \\ \mathbf{Q}_{k2}^T \end{bmatrix} \beta_1 \mathbf{e}_1 = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \\ \phi_k \\ \bar{\phi}_{k+1} \end{bmatrix}. \tag{1.22}$$

Fox example:

$$\mathbf{G}_{1,2} \beta_1 \mathbf{e}_1 = \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ 0 \end{bmatrix} = \begin{bmatrix} \phi_1 = c_1 \beta_1 \\ \bar{\phi}_2 = -s_1 \beta_1 \end{bmatrix}.$$

Therefore, $\hat{\mathbf{a}}_k$ can be calculated by

$$\hat{\mathbf{a}}_k = \mathbf{B}_k^{-1}(\beta_1 \mathbf{e}_1) = \mathbf{R}_k^{-1}\mathbf{Q}_{k1}^T(\beta_1 \mathbf{e}_1) = \mathbf{R}_k^{-1} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \\ \phi_k \end{bmatrix}. \tag{1.23}$$

$\hat{\mathbf{x}}_k$ is embedded in the Krylov space $\kappa_k(\mathbf{A}^T\mathbf{A}, \mathbf{A}^T\mathbf{y})$. The vectors $(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k)$ and $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$ are the orthonormal bases of the Krylov spaces

$$\kappa_k(\mathbf{A}\mathbf{A}^T, \mathbf{u}_1), \quad \kappa_k(\mathbf{A}^T\mathbf{A}, \mathbf{A}^T\mathbf{u}_1),$$

respectively (Baur, 2007). Therefore, $\hat{\mathbf{x}}_k$ can be represented via the orthonormal base $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$, i.e.,

$$\hat{\mathbf{x}}_k \in \kappa_k(\mathbf{A}^T\mathbf{A}, \mathbf{A}^T\mathbf{u}_1)$$

$$\hat{\mathbf{x}}_k = \mathbf{V}_k\hat{\mathbf{a}}_k = \hat{a}_1\mathbf{v}_1 + \hat{a}_2\mathbf{v}_2 + \cdots + \hat{a}_k\mathbf{v}_k, \tag{1.24}$$

Hence, the solution of the original minimization problem can be represented as

$$\hat{\mathbf{x}}_k = \mathbf{V}_k\mathbf{R}_k^{-1} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \\ \phi_k \end{bmatrix}. \tag{1.25}$$

The implementation of the LSQR solver is represented in Tables 1.1 and 1.2 (Baur, 2007). Briefly summarized, the advantages of the LSQR method are as follows: (i) simple recursions to update the successive approximations of the model parameters imply a minimal CPU memory requirement, (ii) Lanczos bidiagonalization suits the numerical calculation by involving sparse matrices very well, and (iii) a good solution in the Krylov subspace can often be obtained with a small number of iterations, making the LSQR method very efficient (Yao et al., 1999).

1.4.3 Preconditioning the normal equation matrix

The convergence of the iterative solution of a system of equations is dominated by the condition number of the design matrix, the normal equation matrix respectively. Tailored preconditioning can improve the condition number significantly, and speed up the iterative process (Benbow, 1999). In linear algebra and numerical analysis, a preconditioner $\mathbf{P}$ of a matrix $\mathbf{A}$ is a matrix such that $\mathbf{P}^{-1}\mathbf{A}$ has a smaller condition number than $\mathbf{A}$, therefore, preconditioners are very useful when using an iterative method to solve a large, sparse linear system, for example,

$$\mathbf{A}\mathbf{x} = \mathbf{y}. \tag{1.26}$$

LSQR method for solving $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$

Initialization

1. $\beta_1 \mathbf{u}_1 = \mathbf{y}$
2. $\alpha_1 \mathbf{v}_1 = \mathbf{A}^T \mathbf{u}_1$
3. $\alpha_1 = \frac{\alpha_1}{\beta_1}$
4. $\bar{\phi_1} = \beta_1$
5. $\bar{\rho_1} = \alpha_1$

1. Iteration: $k = 1$

6. $\beta_2 \mathbf{u}_2 = \mathbf{A}\mathbf{v}_1 - \alpha_1 \mathbf{u}_1$
7. $\mathbf{h}_1 = \mathbf{A}^T \beta_2 \mathbf{u}_2$
8. $[c_1, s_1, \rho_1] = givrot(\bar{\rho}_1, \beta_2)$
9. $\Phi_1 = c_1 \bar{\Phi}_1$
10. $\bar{\Phi}_2 = -s_1 \bar{\Phi}_1$
11. $\mathbf{q}_1 = \frac{1}{\rho_1} \mathbf{v}_1$
12. $\mathbf{x}_1 = \Phi_1 \mathbf{q}_1$
13. $\alpha_2 \mathbf{v}_2 = \mathbf{h}_1 - \beta_2^2 \mathbf{v}_1$
14. $\alpha_2 = \frac{\alpha_2}{\beta_2}$

Further iterations: for $k = 2 : u$

15. $\beta_{k+1} \mathbf{u}_{k+1} = \mathbf{A}\mathbf{v}_k - \alpha_k \mathbf{u}_k$
16. $\mathbf{h}_k = \mathbf{A}^T \beta_{k+1} \mathbf{u}_{k+1}$
17. $\theta_{k-1} = s_{k-1} \alpha_k$
18. $\bar{\rho_k} = c_{k-1} \alpha_k$
19. $[c_k, s_k, \rho_k] = givrot(\bar{\rho}_k, \beta_{k+1})$
20. $\phi_k = c_k \bar{\phi_k}$
21. $\bar{\Phi}_{k+1} = -s_k \bar{\Phi}_k$
22. $\mathbf{q}_k = \frac{1}{\rho_k}(\mathbf{v}_k - \theta_{k-1} \mathbf{q}_{k-1})$
23. $\mathbf{x}_k = \mathbf{x}_{k-1} + \Phi_k \mathbf{q}_k$
24. $\alpha_{k+1} \mathbf{v}_{k+1} = \mathbf{h}_k - \beta_{k+1}^2 \mathbf{v}_k$
25. $\alpha_{k+1} = \frac{\alpha_{k+1}}{\beta_{k+1}}$

Table 1.1: LSQR method

Givens rotation $[c, s, \rho] = givrot(\bar{\rho}, \beta)$
if $\beta = 0.0$ a. $c = 1.0, s = 0.0, \rho = \bar{\rho}$
else if $\|\beta\| > \|\bar{\rho}\|$ a. $t = \bar{\rho}/\beta, q = \sqrt{1.0 + t^2}$ b. $s = 1.0/q, c = ts, \rho = q\beta$
else a. $t = \beta/\bar{\rho}, q = \sqrt{1.0 + t^2}$ b. $c = 1.0/q, s = tc, \rho = q\bar{\rho}$

Table 1.2: Givens rotation

In order to reduce the condition number, equation (1.26) can be rewritten in

$$(\mathbf{P}^{-1}\mathbf{A})\mathbf{x} = \mathbf{P}^{-1}\mathbf{y}. \tag{1.27}$$

Because $\mathbf{P}^{-1}\mathbf{A}$ has a smaller condition number than $\mathbf{A}$, equation (1.27) converges faster than equation (1.26), thus, selecting an appropriate preconditioner is very important before solving linear systems of equations with LSQR. According to Baur (2007) an appropriate preconditioner of the design matrix $\mathbf{A}$ can be obtained by Cholesky decomposition of the symmetric, positive definite block diagonal approximation $\mathbf{N}_{\mathrm{bd}}$ of the normal equation matrix $\mathbf{N}$ subject to

$$\mathbf{N}\mathbf{N}_{\mathrm{bd}}^{-1} \approx \mathbf{I} \tag{1.28}$$

and the Cholesky decomposition

$$\mathbf{N}_{\mathrm{bd}} = \mathbf{L}_{\mathrm{N}}^{T}\mathbf{L}_{\mathrm{N}}. \tag{1.29}$$

The structure of $\mathbf{N}_{\mathrm{bd}}$ is

$$\begin{pmatrix} {}_c\mathbf{H}_0^{L-1} & & & & & & & & \\ & {}_c\mathbf{H}_1^{L-1} & & & & & & & \\ & & {}_c\mathbf{H}_2^{L-1} & & & & & & \\ & & & {}_c\mathbf{H}_3^{L-2} & & & & & \\ & & & & \ddots & & & & \\ & & & & & {}_c\mathbf{H}_L^1 & & & \\ & & & & & & {}_s\mathbf{H}_1^{L-1} & & \\ & & & & & & & \ddots & \\ & & & & & & & & {}_s\mathbf{H}_L^1 \end{pmatrix}, \tag{1.30}$$

where $L = l_{\max}$ and ${}_{c,s}\mathbf{H}_l^p (\in \mathbb{R}^{p\times p}, p < L)$ are blocks along the diagonal. The blocks of $\mathbf{N}_{\mathrm{bd}}$ are consisted of the elements of the series expansion under the same order, thus most elements of the resulting matrix are zero. Opposed to the full matrix, the memory requirement is reduced considerably, as well as the requirement of computing time, $\mathbf{L}_{\mathrm{N}}$ and its inverse $\mathbf{L}_{\mathrm{N}}^{-1}$ can be obtained by block-wise decomposition and inversion. With (1.29), the equation

$$\mathbf{A}\mathbf{x} = \mathbf{y} + \mathbf{r} \tag{1.31}$$

becomes

$$\mathbf{A}\mathbf{L}_{\mathrm{N}}^{-1}\mathbf{L}_{\mathrm{N}}\mathbf{x} = \mathbf{y} + \mathbf{r}, \tag{1.32}$$

with $\mathbf{L}_{\mathrm{N}}^{-1}\mathbf{L}_{\mathrm{N}} = \mathbf{I}$. Through the substitutions $\mathbf{A}_{\mathrm{N}} = \mathbf{A}\mathbf{L}_{\mathrm{N}}^{-1}$ and $\mathbf{z} = \mathbf{L}_{\mathrm{N}}\mathbf{x}$, equation (1.32) can be rewritten as

$$\mathbf{A}_{\mathrm{N}}\mathbf{z} = \mathbf{y} + \mathbf{r}. \tag{1.33}$$

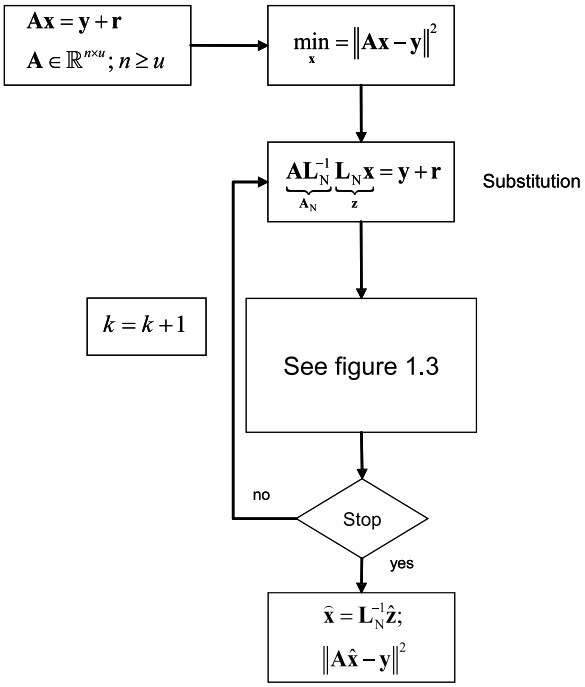

Figure 1.4: PCA-LSQR

Obviously, the solution of equation (1.31) is

$$\hat{\mathbf{x}} = \mathbf{L}_{\mathrm{N}}^{-1}\hat{\mathbf{z}}. \tag{1.34}$$

The preconditioned LSQR algorithm is referred to as PCA-LSQR (see figure 1.4), and for its implementation, the iterative steps

$$\mathbf{v}_1 = \mathbf{A}_{\mathrm{N}}^T\mathbf{u}_1 \tag{1.35}$$
$$\beta_{k+1}\mathbf{u}_{k+1} = \mathbf{A}_{\mathrm{N}}\mathbf{v}_k - \alpha_k\mathbf{u}_k \tag{1.36}$$
$$\alpha_{k+1}\mathbf{v}_{k+1} = \mathbf{A}_{\mathrm{N}}^T\mathbf{u}_{k+1} - \beta_{k+1}\mathbf{v}_k, \tag{1.37}$$

must be rewritten as

$$\mathbf{v}_1 = (\mathbf{L}_{\mathrm{N}}^{-1})^T\mathbf{A}^T\mathbf{u}_1 \tag{1.38}$$
$$\beta_{k+1}\mathbf{u}_{k+1} = \mathbf{A}\mathbf{L}_{\mathrm{N}}^{-1}\mathbf{v}_k - \alpha_k\mathbf{u}_k, \tag{1.39}$$
$$\alpha_{k+1}\mathbf{v}_{k+1} = (\mathbf{L}_{\mathrm{N}}^{-1})^T\mathbf{A}^T\mathbf{u}_{k+1} - \beta_{k+1}\mathbf{v}_k. \tag{1.40}$$

The whole processes are represented in Table 1.3, and because equation (1.33) has a faster convergence than (1.31), a satisfying solution can be obtained with only a few iterations. The main computational costs are

PCA-LSQR method: supplement to Table 1.1 Solution of $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$	
Initialization	
1a. compute $\mathbf{N}_{\mathrm{bd}} = \mathbf{L}_{\mathrm{N}}^T \mathbf{L}_{\mathrm{N}}$ and the inverse Cholesky factor $\mathbf{L}_{\mathrm{N}}^{-1}$	
2a. $\mathbf{v}_1 = (\mathbf{L}_{\mathrm{N}}^{-1})^T \mathbf{v}_1$	
5a. $\mathbf{v}_1 = \mathbf{L}_{\mathrm{N}}^{-1} \mathbf{v}_1$	
1. Iteration: $k = 1$	Further iterations: for $k = 2 : u$
7a. $\mathbf{h}_1 = (\mathbf{L}_{\mathrm{N}}^{-1})^T \mathbf{h}_1$	16a. $\mathbf{h}_k = (\mathbf{L}_{\mathrm{N}}^{-1})^T \mathbf{h}_k$
12. $\mathbf{z}_1 = \Phi_1 \mathbf{q}_1$	23. $\mathbf{z}_k = \mathbf{z}_{k-1} + \Phi_k \mathbf{q}_k$
12a. $\mathbf{x}_1 = \mathbf{L}_{\mathrm{N}}^{-1} \mathbf{z}_1$	23a. $\mathbf{x}_k = \mathbf{L}_{\mathrm{N}}^{-1} \mathbf{z}_k$
14a. $\mathbf{v}_2 = \mathbf{L}_{\mathrm{N}}^{-1} \mathbf{v}_2$	25a. $\mathbf{v}_{k+1} = \mathbf{L}_{\mathrm{N}}^{-1} \mathbf{v}_{k+1}$

Table 1.3: PCA-LSQR method

- the construction of $\mathbf{N}_{\mathrm{bd}}$ as well as
- the additional calculations of the matrix-vector products $\mathbf{L}_{\mathrm{N}}^{-1} \mathbf{v}_k$, $(\mathbf{L}_{\mathrm{N}}^{-1})^T \mathbf{h}_k$ and $\mathbf{L}_{\mathrm{N}}^{-1} \mathbf{z}_k$.

In general, the costs are comparatively low and all calculations can be implemented easily and quickly using BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage), which will be introduced in chapter 3.

Chapter 2

Solutions of the LSQR variance-covariance matrix ploblem

2.1 Motivation

In addition to the estimated vector $\hat{\mathbf{x}}$, the quality of the LS result is also important, which is typically expressed in terms of the variance-covariance matrix $D(\hat{\mathbf{x}})$. Generally, it can be calculated via the generalized inverse $\mathbf{A}^{-g}$ of the design matrix $\mathbf{A}$ (Menke, 1984) subject to

$$\mathbf{A}^{-g} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \tag{2.1}$$

$$\hat{\mathbf{x}} = \mathbf{A}^{-g}\mathbf{y} \tag{2.2}$$

$$D(\hat{\mathbf{x}}) = \mathbf{A}^{-g}(\mathbf{A}^{-g})^T. \tag{2.3}$$

As LSQR is an iterative method, the generalized inverse $\mathbf{A}^{-g}$ can not be computed explicitly, hence the quality of the parameter estimate can not be provided by the straightforward evaluation of (2.3). Thus, searching a suitable method to approximate the variance-covariance information is necessary. Yao et al. (1999) and Schuh & Alkhatib (2007) recommended two independent approaches for the approximate computation of $D(\hat{\mathbf{x}})$. The method according to Yao et al. (1999) is here referred to as the LSQR internal method, whereas Schuh & Alkhatib (2007) use an approach based on Monte Carlo simulation. In the following sections, the two methods will be reviewed.

2.2 LSQR internal method

The LSQR internal method is based on the explicit calculation of the generalized inverse $\mathbf{A}^{-g}$. According to

$$\min_{\mathbf{a}_k} \|\mathbf{B}_k\mathbf{a}_k - \beta_1\mathbf{e}_1\|^2, \tag{2.4}$$

it yields

$$\mathbf{B}_k\hat{\mathbf{a}}_k = \beta_1\mathbf{e}_1 \tag{2.5}$$
$$\mathbf{B}_k^T\mathbf{B}_k\hat{\mathbf{a}}_k = \mathbf{B}_k^T\beta_1\mathbf{e}_1. \tag{2.6}$$

From above, we get

$$\hat{\mathbf{a}}_k = (\mathbf{B}_k^T\mathbf{B}_k)^{-1}\mathbf{B}_k^T\beta_1\mathbf{e}_1 = \mathbf{B}_k^{-g}\beta_1\mathbf{e}_1. \tag{2.7}$$

Moreover, according to the relation (1.16)

$$\mathbf{U}_{k+1}\beta_1\mathbf{e}_1 = \mathbf{y} \Longleftrightarrow \beta_1\mathbf{e}_1 = \mathbf{U}_{k+1}^T\mathbf{y}, \tag{2.8}$$

the parameter estimate becomes

$$\hat{\mathbf{x}}_k = \mathbf{V}_k\hat{\mathbf{a}}_k = \mathbf{V}_k\mathbf{B}_k^{-g}\mathbf{U}_{k+1}^T\mathbf{y}. \tag{2.9}$$

The comparison between equation (2.9) with equation

$$\hat{\mathbf{x}}_k = \mathbf{A}_k^{-g}\mathbf{y} \tag{2.10}$$

yields

$$\mathbf{A}_k^{-g} = \mathbf{V}_k\mathbf{B}_k^{-g}\mathbf{U}_{k+1}^T. \tag{2.11}$$

According to equation (2.3), one obtains with the generalized inverse of an overdetermined LS problem (2.1)

$$\begin{aligned} D(\hat{\mathbf{x}}) &= \mathbf{A}^{-g}(\mathbf{A}^{-g})^T \\ &= \mathbf{A}^{-g}(\mathbf{A}^T)^{-g} \\ &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A} \\ &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T(\mathbf{A}^T)^{-1}\mathbf{A}^{-1}\mathbf{A} \\ &= (\mathbf{A}^T\mathbf{A})^{-1} \\ &= \mathbf{N}^{-1}. \end{aligned} \tag{2.12}$$

Hence, from equation (2.11), the approximation of the variance-covariance matrix $\mathbf{N}^{-1}$ after k iterations can be written as

$$\begin{aligned} \mathbf{N}_k^{-1} &= \mathbf{A}_k^{-g}(\mathbf{A}_k^{-g})^T \\ &= \mathbf{V}_k\mathbf{B}_k^{-g}\mathbf{U}_{k+1}^T(\mathbf{V}_k\mathbf{B}_k^{-g}\mathbf{U}_{k+1}^T)^T \\ &= \mathbf{V}_k\mathbf{B}_k^{-g}\mathbf{U}_{k+1}^T\mathbf{U}_{k+1}(\mathbf{B}_k^{-g})^T\mathbf{V}_k^T, \end{aligned} \tag{2.13}$$

and with the assumption that $\mathbf{U}_{k+1}$ keeps always good orthogonality, i.e., $\mathbf{U}_{k+1}^T\mathbf{U}_{k+1} = \mathbf{I}$, the above equation (2.13) can be consequently reduced to

Solution of $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{Ax} - \mathbf{y}\|^2$ and approximation of $\mathbf{N}^{-1}$	
Reorthogonalization: supplement to Table 1.1	
Initialization	
1. Iteration: $k = 1$	Further iterations: for $k = 2 : u$
	15. $\mathbf{u}^*_{k+1} = \mathbf{A}\mathbf{v}_k - \alpha_k \mathbf{u}_k$
	15a. $\beta_{k+1}\mathbf{u}_{k+1} = \mathbf{u}^*_{k+1} - \mathbf{U}_k(\mathbf{U}_k^T \mathbf{u}^*_{k+1})$
	24. $\mathbf{v}^*_{k+1} = \mathbf{h}_k - \beta^2_{k+1}\mathbf{v}_k$
	24a. $\alpha_{k+1}\mathbf{v}_{k+1} = \mathbf{v}^*_{k+1} - \mathbf{V}_k(\mathbf{V}_k^T \mathbf{v}^*_{k+1})$
Computation of the VCM: supplement to Table 1.1	
After k iterations, $\mathbf{B}_k$ and $\mathbf{V}_k$ are kept in memory. So,	
26. $\mathbf{N}_k^{-1} = \mathbf{V}_k(\mathbf{B}_k^T \mathbf{B}_k)^{-1}\mathbf{V}_k^T$	

Table 2.1: LSQR internal method for VCM computation

$$\mathbf{N}_k^{-1} = \mathbf{V}_k(\mathbf{B}_k^{-g}(\mathbf{B}_k^{-g})^T)\mathbf{V}_k^T = \mathbf{V}_k(\mathbf{B}_k^T\mathbf{B}_k)^{-1}\mathbf{V}_k^T. \tag{2.14}$$

Therefore, in order to calculate the variance-covariance matrix $\mathbf{N}_k^{-1}$, the matrices $\mathbf{V}_k$ and $\mathbf{B}_k$ must be kept in the memory, requiring additional storage space. From the implementation point of view, only small changes are required to the original LSQR code. It is important to note that the quality of the variance-covariance matrix is heavily dependent on the number of iterations k, and commonly the convergence of $\mathbf{N}_k^{-1}$ is slower than the convergence of the parameter estimate $\hat{\mathbf{x}}_k$ (Yao et al., 1999).

Reorthogonalization

In practice, orthogonality of the matrix $\mathbf{U}_{k+1}$ will be destroyed after only a few iterations, i.e., $\mathbf{U}_{k+1}^T\mathbf{U}_{k+1} \neq \mathbf{I}$ holds true, falsifying the results considerably. Therefore, reorthogonalization of the matrices $\mathbf{U}_{k+1}$ and $\mathbf{V}_k$ is necessary. Referring to Yao et al. (1999), in terms of reorthogonalization, (1.15) can be rewritten as follows

$$\mathbf{u}^*_{k+1} = \mathbf{A}\mathbf{v}_k - \alpha_k\mathbf{u}_k \tag{2.15}$$

$$\text{Reorthogonalization:}\quad \beta_{k+1}\mathbf{u}_{k+1} = \mathbf{u}^*_{k+1} - \mathbf{U}_k(\mathbf{U}_k^T\mathbf{u}^*_{k+1}) \tag{2.16}$$

$$\mathbf{v}^*_{k+1} = \mathbf{A}^T\mathbf{u}_{k+1} - \beta_{k+1}\mathbf{v}_k \tag{2.17}$$

$$\text{Reorthogonalization:}\quad \alpha_{k+1}\mathbf{v}_{k+1} = \mathbf{v}^*_{k+1} - \mathbf{V}_k(\mathbf{V}_k^T\mathbf{v}^*_{k+1}). \tag{2.18}$$

In addition to the matrices $\mathbf{V}_k$ and $\mathbf{B}_k$, matrix $\mathbf{U}_{k+1}$ must be also kept in memory. Due to the increasing storage requirements, memory allocation and reallocation must be carefully performed. The steps to implement both reorthogonalization and the approximate variance-covariance matrix (VCM) are summarized in Table 2.1 as well as in figure 2.1.

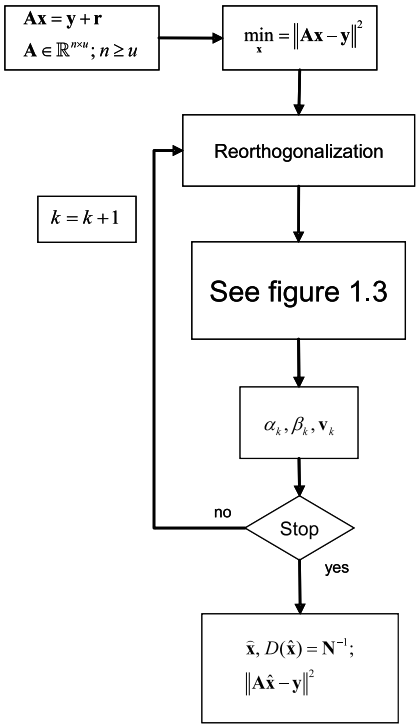

Figure 2.1: LSQR internal method for VCM computation

2.3 Monte Carlo method

2.3.1 Introduction

The Monte Carlo (MC) method was developed in the 1940s, when it was used at first in the simulation of random processes for the construction of the atomic bomb, in order to theoretically predict the interaction between neutrons and matter. The name is an allusion to the famous gambling city. Today Monte Carlo methods belong to the most important numerical (and non-numeric) procedures, which was applied to many scientific, technical and medical problems with great success [1].

2.3.2 Monte Carlo algorithm

The MC method allows to approximate the VCM ($\mathbf{N}^{-1}$) by means of random samples $\mathbf{s}_l^{(i)}, \ldots, \mathbf{s}_l^{(M)}$. For the sake of simplicity, the observations are assumed to be uncorrelated and equal in accuracy. Hence, $\mathbf{s}_l \sim N(0, \mathbf{I})$ holds true. The transformation

$$\mathbf{s}_x^{(i)} = \mathbf{B}\mathbf{s}_l^{(i)}, i = 1, \ldots, M \tag{2.19}$$

yields the samples $\mathbf{s}_x^{(i)}, \ldots, \mathbf{s}_x^{(M)}$ to be distributed according to

[1] http://www.exp.univie.ac.at

Input:	linear system $\mathbf{y} = \mathbf{A}\mathbf{x}$ $\mathbf{A}\ldots$ design matrix
Output:	$\mathbf{N}^{-1}$
1.	Generate samples $\mathbf{s}_l^{(i)}$, $i = 1, \ldots, M$ with $\mathbf{s}_l \sim N(0, \mathbf{I})$.
2.	Transform the sample vectors $\mathbf{s}_l^{(i)}$ into $\mathbf{s}_x^{(i)}$ by solving the linear systems of equations $\mathbf{A}\mathbf{s}_x^{(i)} = \mathbf{s}_l^{(i)} + \mathbf{r}^{(i)}$ with LSQR.
3.	Estimate the VCM by $\mathbf{N}^{-1} = \hat{D}(\hat{\mathbf{x}}) = \frac{1}{M}\sum_{i=1}^{M} \hat{\mathbf{s}}_x^{(i)}\hat{\mathbf{s}}_x^{(i)T}, i = 1, \ldots, M.$

Table 2.2: Monte Carlo algorithm for VCM estimation

$$\mathbf{s}_x \sim N(0, \mathbf{N}^{-1}). \tag{2.20}$$

Consequently, for the VCM holds (Schuh & Alkhatib, 2007):

$$D(\hat{\mathbf{x}}) = E\left\{\Big(\mathbf{s}_x - E\{\mathbf{s}_x\}\Big)\Big(\mathbf{s}_x - E\{\mathbf{s}_x\}\Big)^T\right\} \tag{2.21}$$

$$= E\left\{\mathbf{s}_x\mathbf{s}_x^T\right\} \tag{2.22}$$

$$\approx \frac{1}{M}\sum_{i=1}^{M} \mathbf{s}_x^{(i)}\mathbf{s}_x^{(i)T}, i = 1, \ldots, M. \tag{2.23}$$

Hence, for $M < \infty$ random samples, the VCM can be approximated by

$$\hat{D}(\hat{\mathbf{x}}) = \frac{1}{M}\sum_{i=1}^{M} \mathbf{s}_x^{(i)}\mathbf{s}_x^{(i)T}, i = 1, \ldots, M. \tag{2.24}$$

Each individual sample $\mathbf{s}_x^{(i)}, i = 1 \cdots M$ is derived from the solution of the linear system

$$\mathbf{A}\mathbf{s}_x^{(i)} = \mathbf{s}_l^{(i)} + \mathbf{r}^{(i)},$$

i.e., $\mathbf{B} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T = \mathbf{A}^{-g}$ holds true and LSQR has to be applied to $M + 1$ right hand sides, namely $\mathbf{y}, \mathbf{s}_l^{(1)}, \ldots, \mathbf{s}_l^{(M)}$, in order to compute $M+1$ estimates of the unknown vectors $\mathbf{x}, \mathbf{s}_x^{(1)}, \ldots, \mathbf{s}_x^{(M)}$. The algorithm is summarized in Table 2.2.
The augmented vectors $\mathbf{X} = [\mathbf{x}, \mathbf{s}_x^{(1)}, \mathbf{s}_x^{(2)}, \ldots, \mathbf{s}_x^{(M)}]$ and $\mathbf{Y} = [\mathbf{y}, \mathbf{s}_l^{(1)}, \mathbf{s}_l^{(2)}, \ldots, \mathbf{s}_l^{(M)}]$ have dimensions $u \cdot (M+1)$ and $n \cdot (M+1)$, respectively. Therefore, the memory requirement is driven by the number of samples M. From figure 2.2 it can be seen that the MC method is very well suited for parallel computing. The original LSQR process (see Table 1.1) can be supplemented by the Monte Carlo method as outlined in Table 2.3.

Solution of $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$ and approximation of $\mathbf{N}^{-1}$	
Monte Carlo method: supplement to Table 1.1	
Initialization	
Create a set of random samples $\mathbf{s}_l^{(i)}$ with normal distribution, $i = 1, \ldots, M$.	
1. $\mathbf{Y} = [\mathbf{y}, \mathbf{s}_l^{(1)}, \mathbf{s}_l^{(2)}, \ldots, \mathbf{s}_l^{(M)}]$	
1a. $\beta_1 \mathbf{u}_1 = \mathbf{Y}$	
1. Iteration: $k = 1$	Further iterations: for $k = 2 : u$
12. $\mathbf{X}_1 = \Phi_1 \mathbf{q}_1$	23. $\mathbf{X}_k = \mathbf{X}_{k-1} + \Phi_k \mathbf{q}_k$
12a. $\mathbf{X}_1 = [\mathbf{x}_1, \mathbf{s}_{x1}^{(1)}, \mathbf{s}_{x1}^{(2)}, \ldots, \mathbf{s}_{x1}^{(M)}]$	23a. $\mathbf{X}_k = [\mathbf{x}_k, \mathbf{s}_{xk}^{(1)}, \mathbf{s}_{xk}^{(2)}, \ldots, \mathbf{s}_{xk}^{(M)}]$
Compute the VCM: supplement to Table 1.1	
26. $\mathbf{N}_k^{-1} = \frac{1}{M} \sum_{i=1}^{M} \mathbf{s}_{xk}^{(i)} \mathbf{s}_{xk}^{(i)T}$	

Table 2.3: Monte Carlo method for VCM estimation

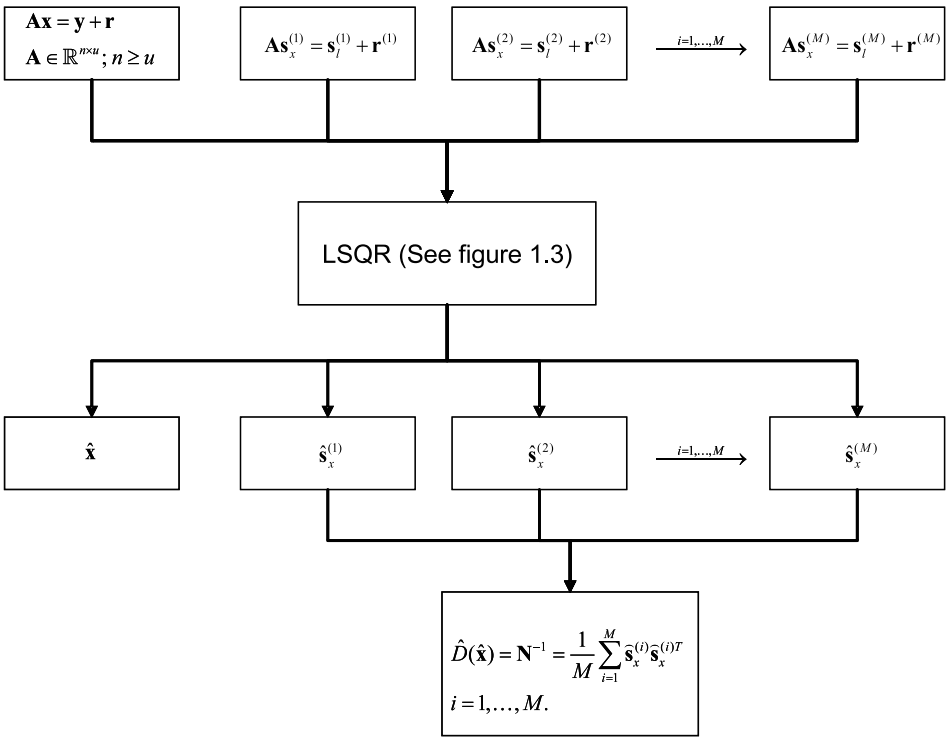

Figure 2.2: Flowchart of the Monte Carlo method

Referring to Schuh & Alkhatib (2007), the accuracy of the MC method primarily depends on the number of samples M. As pointed out there, the convergence to the exact VCM by increasing the number of samples is very slow. An alternative way to improve the accuracy of the method is given by stepwise estimation by conditioning (Schuh & Alkhatib, 2007), as pointed out next.

2.3.3 Stepwise estimation by conditioning

First, the normal equation matrix $\mathbf{N}$ is divided into $p \times p$ blocks subject to

$$\begin{pmatrix} \mathbf{N}_{11} & \dots & \mathbf{N}_{i1} \\ \vdots & \ddots & \vdots \\ \mathbf{N}_{1j} & \dots & \mathbf{N}_{ij} \end{pmatrix} \quad i,j = 1,\dots,p. \tag{2.25}$$

According to (1.30), $\mathbf{N}$ can be approximated by a symmetric, positive definite block diagonal matrix $\mathbf{N}_{\mathrm{bd}}$ and it can be inverted block-wise as follows

$$\begin{pmatrix} \mathbf{N}_{\mathrm{bd}_1}^{-1} & & & & \\ & \ddots & & & \\ & & \mathbf{N}_{\mathrm{bd}_k}^{-1} & & \\ & & & \ddots & \\ & & & & \mathbf{N}_{\mathrm{bd}_p}^{-1} \end{pmatrix} \quad k = 1,\dots,p. \tag{2.26}$$

New vectors $(\hat{\mathbf{s}}_{\bar{x}}^{(i)})_k$ result from the vectors $(\hat{\mathbf{s}}_{x}^{(i)})_k$ with the transformation

$$\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k = \left(\mathbf{s}_{x}^{(i)}\right)_k - \mathbf{N}_{\mathrm{bd}_k}^{-1}\left(\mathbf{A}^T\mathbf{s}_{l}^{(-1)}\right)_k, \tag{2.27}$$

where subscript k denotes the kth part of a vector or the kth block of a diagonal block matrix. After the $(\hat{\mathbf{s}}_{\bar{x}}^{(i)})_k$ are calculated completely, the diagonal blocks of the VCM can be obtained by using the equation

$$D(\hat{\mathbf{x}})_{kk} = \mathbf{N}_{\mathrm{bd}_k}^{-1} + \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k^T \tag{2.28}$$

and in order to compute the remaining blocks off the diagonal, the equation

$$D(\hat{\mathbf{x}})_{kj} = \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_{x}^{(i)}\right)_j^T \quad j = k+1,\dots,p. \tag{2.29}$$

is applied. Because the matrix $D(\hat{\mathbf{x}})$ is a symmetric matrix, the VCM becomes

$$\mathbf{N}^{-1} = D(\hat{\mathbf{x}}) = D(\hat{\mathbf{x}})_{kk} + D(\hat{\mathbf{x}})_{kj} + D(\hat{\mathbf{x}})_{kj}^T \quad k = 1,\dots,p; \quad j = k+1,\dots,p \tag{2.30}$$

The algorithm for conditioning is presented in Table 2.4. Table 2.5 outlines the MC method including conditioning.

Input:	$\mathbf{N}_{\mathrm{bd}_k}$ ... blocks of the normal equations
	$(\mathbf{s}_x^{(i)})_k$ ... subvectors of the samples
Output:	$\mathbf{N}^{-1}$

$k = 1, \ldots, p$

1. Invert the diagonal block $\mathbf{N}_{\mathrm{bd}_k}$
   $\mathbf{N}_{\mathrm{bd}_k}^{-1} = \mathrm{INV}(\mathbf{N}_{\mathrm{bd}_k})$.
2. Transform the vectors $\left(\mathbf{s}_x^{(i)}\right)_k$ in $\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k$ with
   $\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k = \left(\mathbf{s}_x^{(i)}\right)_k - \mathbf{N}_{\mathrm{bd}_k}^{-1}\left(\mathbf{A}^T\mathbf{s}_l^{(-1)}\right)_k$.
3. Compute the diagonal blocks of the VCM
   $\mathbf{N}_{kk}^{-1} = \hat{D}(\hat{\mathbf{x}})_{kk} = \mathbf{N}_{\mathrm{bd}_k}^{-1} + \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k^T$.
4. Compute the off diagonal blocks of the VCM
   $\mathbf{N}_{kj}^{-1} = \hat{D}(\hat{\mathbf{x}})_{kj} = \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_x^{(i)}\right)_j^T$.
   $j = k+1, \ldots, p$.

END k

Table 2.4: Conditioning

Solution of $\min \|\mathbf{r}\|^2 = \min_x \|\mathbf{A}\mathbf{x} - \mathbf{y}\|^2$ and approximation of $\mathbf{N}^{-1}$

Monte Carlo method with conditioning: supplement to Table 1.1

Initialization

Create a set of random samples $\mathbf{s}_l^{(i)}$ with normal distribution, $i = 1, \ldots, M$.

1. $\mathbf{Y} = [\mathbf{y}, \mathbf{s}_l^{(1)}, \mathbf{s}_l^{(2)}, \ldots, \mathbf{s}_l^{(M)}]$

1a. $\beta_1\mathbf{u}_1 = \mathbf{Y}$

2a. $\mathbf{M} = \mathbf{A}^T\mathbf{u}_1$

1. Iteration: $k = 1$

12. $\mathbf{X}_1 = \Phi_1\mathbf{q}_1$

12a. $\mathbf{X}_1 = [\mathbf{x}_1, \mathbf{s}_{x1}^{(1)}, \mathbf{s}_{x1}^{(2)}, \ldots, \mathbf{s}_{x1}^{(M)}]$

Further iterations: for $k = 2 : u$

23. $\mathbf{X}_k = \mathbf{X}_{k-1} + \Phi_k\mathbf{q}_k$

23a. $\mathbf{X}_k = [\mathbf{x}_k, \mathbf{s}_{xk}^{(1)}, \mathbf{s}_{xk}^{(2)}, \ldots, \mathbf{s}_{xk}^{(M)}]$

Compute the VCM: supplement to Table 1.1

26. Calculate the blocks $\mathbf{N}_{\mathrm{bd}_k}$ and inverses $\mathbf{N}_{\mathrm{bd}_k}^{-1}$, $\quad k = 1, \ldots, p$

27. $\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k = \left(\mathbf{s}_x^{(i)}\right)_k - \mathbf{N}_{\mathrm{bd}_k}^{-1}\left(\mathbf{M}\right)_k$

28. $\mathbf{N}_{kk}^{-1} = \mathbf{N}_{\mathrm{bd}_k}^{-1} + \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k^T$

29. $\mathbf{N}_{kj}^{-1} = \frac{1}{M}\sum_{i=1}^{M}\left(\mathbf{s}_{\bar{x}}^{(i)}\right)_k\left(\mathbf{s}_x^{(i)}\right)_j^T, \quad j = k+1, \ldots, p$

30. $\mathbf{N}^{-1} = \mathbf{N}_{kk}^{-1} + \mathbf{N}_{kj}^{-1} + (\mathbf{N}_{kj}^{-1})^T$

Table 2.5: Monte Carlo method with conditioning

Chapter 3

High performance computing (HPC): Parallel programming

3.1 High performance computing and computers

High performance computing is increasingly important in scientific computing as a tool for calculating, modeling and simulation of complex systems as well as for processing huge amounts of data. Such applications can be found today in all areas of natural and technical sciences; typical application areas are meteorology and climatology, astronomers and particle physics, systems biology, genetics, flow and quantum mechanics. Even in commercial computing, there are applications of the high performance computing. Many of them are scientific origin (such as weather forecasting, crash test simulation, fluid dynamics in the aircraft), but there are also applications without scientific character, for example, in the production of animation films.
High performance computers are computer systems that are suitable for tasks of the high performance computing. On high performance computers, different scientific application programs, math libraries, compilers and programming tools are installed. A typical characteristic of a modern high performance computer is its large number of processors, common peripherals and a partially shared memory.
High performance computers are based on different processor architectures. Vector processors can complete a calculation simultaneously on many data (in a vector or array). Clusters consist of a large number of (mostly cheap) individual central processing units (CPUs), assembled to a large computer networks. Compared to vector computers, the cluster nodes have own peripherals and exclusively use their own local memory. High performance computers are programmed generally in 64-bit, and programming languages are Fortran and C. The fastest high performance computers are listed in the top-500 list[1] according to their performance.

[1] http://www.top500.org

3.1.1 System architectures, Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data (Flynn (1972)).

SISD (Single instruction, single data) SISD computers are understood as traditional single computers, which do their tasks sequentially. SISD computers are built according to the von Neumann or Harvard architecture (e.g. Personal computers (PCs) or workstations). In this case, a single processor, an uniprocessor, executes a single instruction stream, to operate on data stored in a single memory (see figure 3.1).

SIMD (Single instruction, multiple data) SIMD computers, also known as array processors or vector processor, serve for the rapid execution of similar calculations of available input data streams and are used mainly in the processing of video, audio and video data (see figure 3.2).
Many modern microprocessors (such as PowerPC and x86) have now SIMD Extensions, which means additional specific instruction sets. With one command call several similar processes can be implemented simultaneously.

MISD (Multiple instruction, single data) The assignment of systems to this class is difficult, so it is controversial. Many people are of the opinion that such systems should not be existent (see figure 3.3).

MIMD (Multiple instruction, multiple data) MIMD computer implement various operations at the same time and deal with different kind of input data streams. Moreover, the distribution of tasks to the available resources is managed mostly by one or more processors. Each processor has access to the data of other processors (see figure 3.4).
There are closely coupled systems and loosely coupled systems. Closely coupled systems are multi-processor system, while loosely coupled systems multicomputer systems.
Multi-processor systems share the available memory and are therefore also a shared memory system. These shared-memory systems can be divided further in UMA (uniform memory access), NUMA (non-uniform memory access) and COMA (cache-only memory access) .
The principal idea using MIMD systems is to split a huge problem in several smaller parts. Doing so, the different parts of the problem must be synchronized with each other.

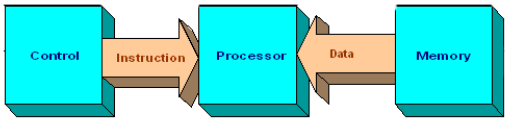

Figure 3.1: SISD

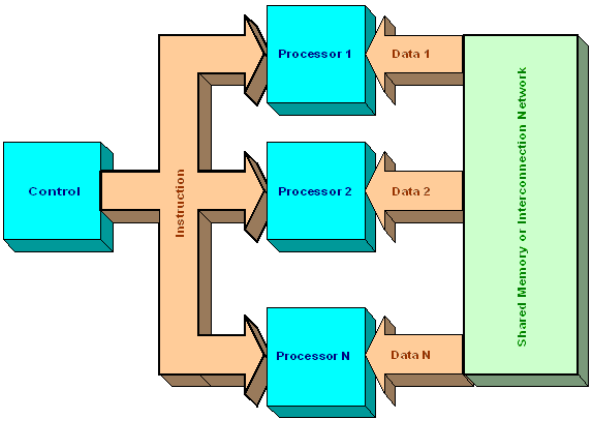

Figure 3.2: SIMD

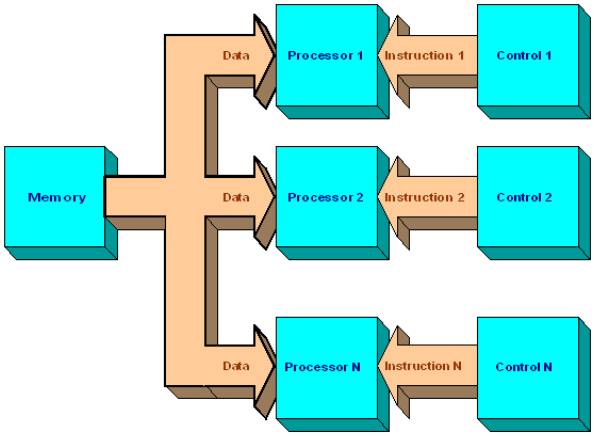

Figure 3.3: MISD

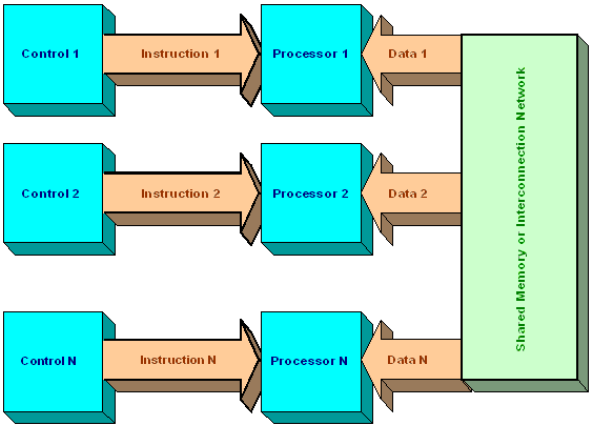

Figure 3.4: MIMD

3.1.2 Parallel computers

A parallel computer is a computer, in which operations will be distributed simultaneously on multiple CPUs to advance the working speed. A massively parallel computer is a single computer with many networked processors. They have usually far more than 100 processors. In a massively parallel computer, each CPU contains its own memory and copy of the operating system and application. Application examples can be found in computational physics or in weather forecast. For the efficient performance of a parallel computer, the computations must be distributed on the CPUs. In principle, this is a logistical problem. The scarce resources - computer time, memory accesses, date buses - must be exploited effectively. Always, the sequential program overhead should be minimal (Amdahl's Law).

Amdahl's Law Amdahl's Law (named after Gene Amdahl) is a model in computer science for the acceleration of program by parallel execution. According to Amdahl the speed increase depends especially on the sequential parts of the parallel problem (Amdahl (1967)). If P is the number of parallel processors and a is the sequential (non-parallelized) part of the program, then the maximum speedup s is:

$$s = \frac{1}{a + o(P) + \frac{1-a}{P}} \leq \frac{1}{a}$$

It should be noted that the acceleration with increasing number of processor depends on both the sequential part of the algorithm and the processor communication.
The larger the number of CPUs, the faster the communication costs increase in a parallel computing environment. The relationship is not linear.

3.1.3 Memory access principles

Shared Memory Shared memory denotes a certain kind of interprocess communication (IPC). In this case, two or more processes can use a certain part of the communal memory. For all involved processes, this shared memory area are located just in the address space, it can be used and changed with normal memory access operations. Most of this are implemented by paging mechanisms, in which both processes use the same page descriptors. Most modern operating systems provide mechanisms to use shared memory.

Distributed Memory In computer science, distributed memory refers to a multiple-processor computer system in which each processor has its own private memory. This requires computational tasks to be distributed to the different processors for processing. Afterward the data must be reassembled. Various network topologies are used to connect the multiple processors in distributed memory systems, including ring, mesh, tree, etc.

Distributed Shared Memory Distributed shared memory (DSM) is an intermediate form well-known distributed memory and shared memory architectures. In general, a DSM is considered as a virtual shared memory, it means that the user has the sight of a shared memory architecture. However, the real memory is distributed on a number of different, separate and independent physical memory. So, a DSM system provide a mediation layer between users and hardware.

3.1.4 Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between

basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common[2].

Multicore computing

A multicore processor is a processor that includes multiple execution units. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); by contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. Each core in a multicore processor can potentially be superscalar as well, that is, on every cycle, each core can issue multiple instructions from one instruction stream.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors.

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

Cluster computing

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. The vast majority of the TOP500 supercomputers are clusters.

Massive parallel processing

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but they are usually larger, typically having "far more" than 100 processors. In an MPP, each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect.
Blue Gene/L, the fastest supercomputer in the world according to the TOP500 ranking (August 2008), is an MPP.

[2] http://en.wikipedia.org/wiki/Parallel_computing

Grid computing

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the internet, grid computing typically deals only with embarrassingly parallel problems.

3.2 Parallel programming (MPI, OpenMP)

3.2.1 MPI

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers[3].
Message Passing Interface (MPI) is a standard, which describes the message exchange in parallel computing on distributed computer systems. It defines a collection of operations and semantics, i.e. a programming interface, but there is no specific protocol and no implementation.
An MPI application usually consists of several processes communicating with each other, which starts at the beginning of the parallel program execution. Then all these processes work together on a problem and use the messages to exchange data, which is sent explicitly from one to the other process.
One advantage of this principle is that the message exchange works ignoring computer boundaries. MPI parallel programs are executable both on PC clusters[4], as well as on dedicated parallel computers[5].
The aim of the message passing interface can be simply explained to design a widely used standard for writing message-passing programs. As such an interface it should be a practical, portable, efficient and flexible standard for message passing.

3.2.2 Some useful MPI routines

Header file

```
#include <mpi.h>
```

[3] http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
[4] in this case the exchange of messages works e.g. via TCP
[5] The message exchange works here e.g. via the common main memory

Initializing MPI

```
int MPI_Init(int *argc, char ***argv)
```

```
#include<mpi.h>
int main(int argc, char **argv)
{
 MPI_Init(&argc, &argv);
 ...
```

Starting the MPI program

```
mpirun -np number_of_processes ./executable
```

Communicator MPI_COMM_WORLD

All processes of one MPI program are combined in the communicator

`MPI_COMM_WORLD`.

It is a predefined handle in mpi.h. Each process has its own rank in a communicator.

Rank

The rank identifies different processes and is the basis for any work and data distribution.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Size

It denotes how many processes are contained within a communicator.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Exiting MPI

`MPI_Finalize`

finalizes the MPI programming environment. Further MPI-calls are forbidden and especially reinitialization with

`MPI_Init`

is forbidden.

3.2.3 OpenMP

OpenMP (Open Multi Processing) is a programming interface, which is developed jointly by various hardware manufacturers and compiler since 1997. This standard is used for shared memory programming on multi-processor computers. It is portable across any shared-memory architectures and allows incremental parallelization[6].
The OpenMP standard defines special compiler directives, e.g. the processing of a for-loop is distributed on several threads or processors. However, there are also library functions and environment variables for OpenMP programming.
In modern super computers, OpenMP and MPI are often used together. Then OpenMP works on shared memory nodes, which exchange with MPI.

3.2.4 Some useful OpenMP routines

OpenMP parallel region construct

```
#pragma omp parallel
structured block
/*omp end parallel*/
```

OpenMP parallel region construct syntax

```
#pragma omp parallel [clause [[,] clause] ...] new-line
 structured-block
```

clause can be one of the following:

- *private*(list)[7]
- *shared*(list)[8]

OpenMP directive format

```
#pragma omp directive_name [clause[[,] clause ]...] new-line
```

OpenMP runtime library

Include file for library routines:

```
#include <omp.h>
```

The function

[6] http://de.wikipedia.org/wiki/OpenMP
[7] declares the variables in list to be private to each thread in a team
[8] makes variables that appear in list shared among all the threads in a team

```
int omp_get_num_threads(void)
```

returns the number of threads currently in the team executing the parallel region.

The function

```
int omp_get_thread_num(void)
```

returns the thread number. The master thread of the team is thread 0. Wall clock time is measured with

```
MPI_WTIME
```

A simple OpenMP example is given next:

```
#pragma omp parallel private(f)
{
 f = 7;
 #pragma omp for
  for (i = 0; i < 20; i++)
   a[i] = b[i] + f*(i+1);
}
/*omp end parallel*/
```

3.3 Parallel libraries

BLAS - Basic Linear Algebra Subroutines:

Basic Linear Algebra Subprograms (BLAS) is a defacto application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication[9]. They were first published in 1979, and are used to build larger packages such as LAPACK. Heavily used in high-performance computing, highly optimized implementations of the BLAS interface have been developed by hardware vendors such as by Intel as well as by other authors (e.g. ATLAS is a portable self-optimizing BLAS). The LINPACK benchmark relies heavily on DGEMM, a BLAS subroutine, for its performance.

BLAS-1

- vector × vector
- data transfer 3*n
- operations 2*n-1

[9] http://www.netlib.org/blas/

BLAS-2

- matrix × vector
- data transfer n*n+2*n
- operations n*(2*n-1)

BLAS-3

- matrix × matrix
- data transfer 3*n*n
- operations n*n*(2*n-1)

LAPACK - Linear Algebra PACKage

LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems[10]. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.
LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). LAPACK was designed at the outset to exploit the Level 3 BLAS and the solution of triangular systems with multiple right-hand sides. Because of the coarse granularity of the Level 3 BLAS operations, their use promotes high efficiency on many high-performance computers, particularly if specially coded implementations are provided by the manufacturer.

[10] http://www.netlib.org/lapack/

Chapter 4

Case studies

4.1 NEC Xeon EM64T cluster (cacau)

In order to study the VCM estimation methods, which were introduced in chapter 2, the NEC Xeon EM64T cluster supported by the high performance computing center Stuttgart (HLRS) was chosen as computing platform (see figure 4.1). This platform consists of one front node for interactive access (cacau.hww.de) and 200 nodes for execution of parallel programs. Each of the 200 nodes consists of two 3.2 GHz Xeon EM64T CPUs with either 1GB or 2GB memory on each node. Table 4.1 lists some of the architecture parameters.

Figure 4.1: NEC Xeon EM64T cluster

Peak Performance:	2.5 TFlops
Processors:	400 Intel Xeon EM64T CPUs (3.2GHz)
Memory:	160 nodes * 1 GB + 40 nodes * 2 GB
Disk:	1.2 TB distributed scratch, 1 TB shared HOME
Number of Nodes:	200 compute, 1 frontend
Node-node interconnect:	Infiniband 1000 MB/s

Table 4.1: NEC Xeon EM64T cluster

4.2 Operating procedure

Access to the platform is provided by secure shell:

```
ssh -X cacau.hww.de,
```

where, "cacau.hww.de" is the frontend node of the NEC cluster. Workspace has to be allocated individually. For example, the command

```
ws_allocate myWorkspace 30
```

allocates a new workspace entitled "myWorkspace" and with a lifetime of 30 days. Because the new allocated space will be released after 30 days, it is very important to back up the output data in time and upload the input data once more. The command

```
ws_list
```

lists all allocated workspaces with their names, paths as well as remaining lifetime. To release the workspace "myWorkspace" the command

```
ws_release myWorkspace
```

can be used. Before submitting a job to the batch system, the environment variables have to be set correctly. The following modules are needed:

```
mpi:openmpi-1.2.4
compiler:intel9.0
mkl:8.0
```

They are selected via the 'switcher' command subject to

```
switcher mpi = openmpi-1.2.4
switcher mkl = 8.0
switcher compiler = intel9.0.
```

Finally, a batch job gets started with

```
qsub <options>.
```

For example: to submit a job running for two hours on 5 cluster nodes with 2G memory, the following command has to be used:

```
qsub -I -l nodes=5:mem2gb,walltime=2:00:00.
```

The flag "-I" denotes that the job will be executed interactively, and "mem2gb" defines the node memory (see Table 4.2). "walltime" defines the runtime. If the true runtime exceed this limit, the job will be stopped automatically. The higher the runtime limit, the lower the job priority. It is important to mention that the nodes of the cluster are exclusive, i.e., one node can be used only by one user at the same time.

Feature	Nodes
mem1gb	1GB per node available
mem2gb	2GB per node available

Table 4.2: Feature of nodes

Compilation on the fronted node is carried out with the commands

```
make clean;
make all.
```

After submitting a job successfully, the required nodes are ready and the program can be executed by

```
mpirun -np <number of nodes> <program name>.
```

4.3 Numerical results

4.3.1 A simple example

Before applying the program to larger systems of equations, a simple example is examined first. The test data set consists of 10 000 satellite observations of type SST (Satellite to Satellite Tracking, see figure 4.2). It is based on a simulated GOCE-like orbit with orbit parameters according to Table 4.3.
The data is generated using the EGM96 model up to degree and order 10. Hence, the objective of data analysis is to resolve the spherical harmonic coefficients up to degree $l_{\max} = 10$ and to evaluate the appropriate VCM. According to equation (1.9), the number of unknowns is

$$u = l_{\max}^2 + 2 \cdot l_{\max} - 2 = 10^2 + 2 \cdot 10 - 2 = 118. \tag{4.1}$$

Orbit element	Initial value
Semi-major axis a	6628 000 m
Eccentricity e	0.001
Inclination I	96.6°
Right ascension of the ascending node Ω	0°
Argument of perigee ω	0°
Mean anomaly M	0°

Table 4.3: Initial values for orbit simulation

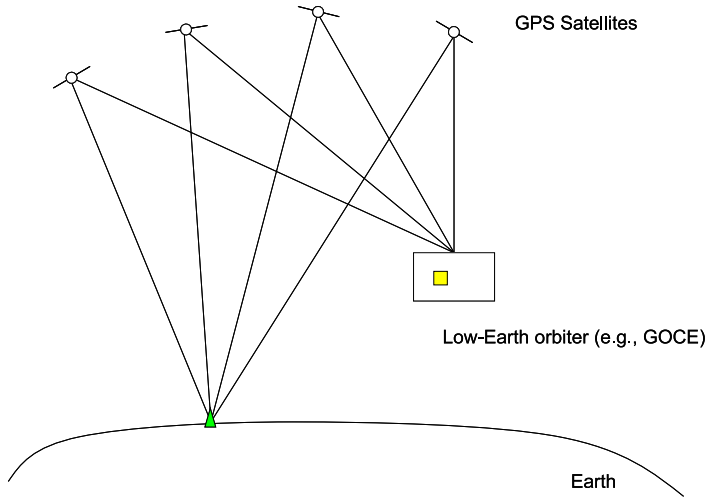

Figure 4.2: SST observation principle

Preconditioning

According to section 1.4.3, preconditioning is an ideal method to improve the convergence of the LSQR solver and thus to reduce the number of iterations. Figures 4.3 and 4.4 show the parameter estimation results after 30, 60, 90, 110 and 118 iterations with and without preconditioning.

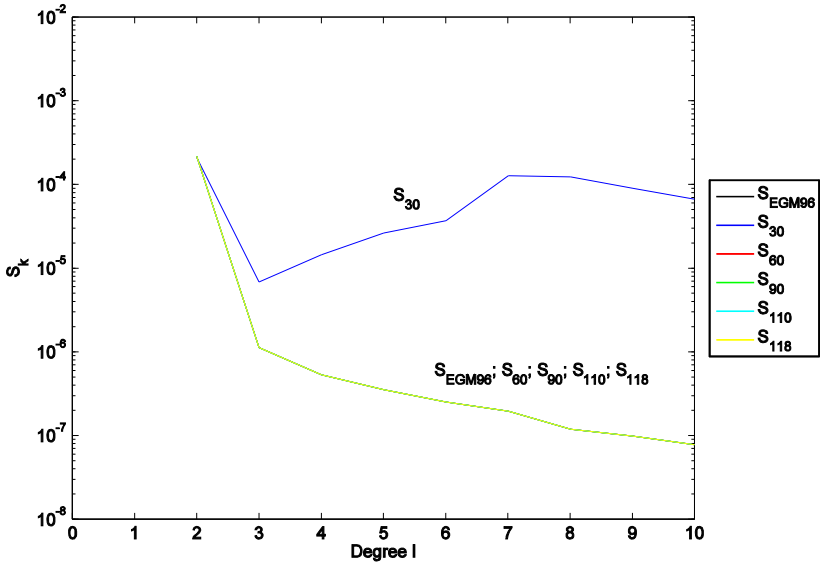

Figure 4.3: Power spectrum with preconditioning

In the figures, S_k^l denotes the RMS[1] value of the spherical harmonic coefficients with degree l, i.e.,

$$S_k^l = \sqrt{\frac{1}{2l+1}\sum_{m=0}^{l}(c_{l,m}^2 + s_{l,m}^2)}.$$

From the figures, it is obvious that without preconditioning the solution converges much more slowly towards the EGM96 model than with preconditioning. After 110 iterations the solution without preconditioning starts covering the EGM96 curve, while the solution with preconditioning covers the EGM96 model very well after only 60 iterations. To show the performance more clearly, figures 4.5 and 4.6 show up the differences ΔS_k^l between the LS results and the EGM96 model, according to

$$\Delta S_k^l = \sqrt{\frac{1}{2l+1}\sum_{m=0}^{l}((c_{l,m} - c_{l,m;\mathrm{EGM96}})^2 + (s_{l,m} - s_{l,m;\mathrm{EGM96}})^2)}.$$

[1]Root Mean Square

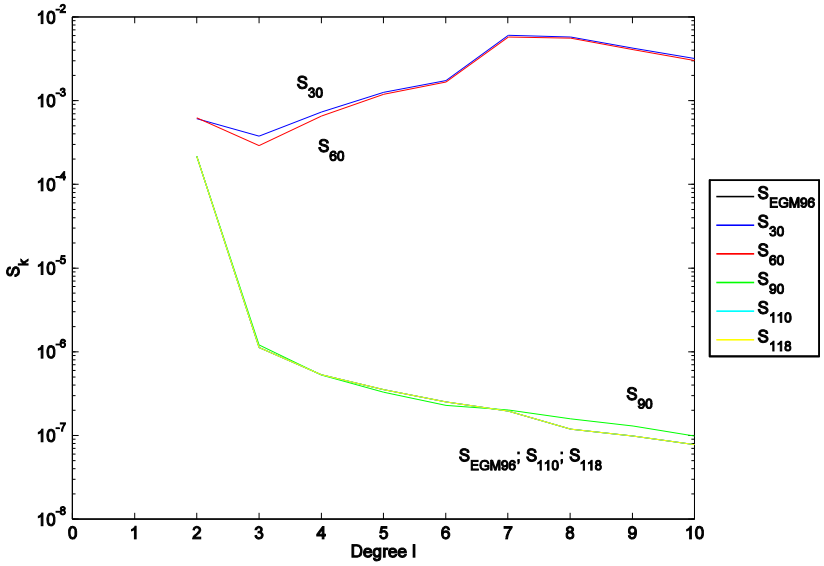

Figure 4.4: Power spectrum without preconditioning

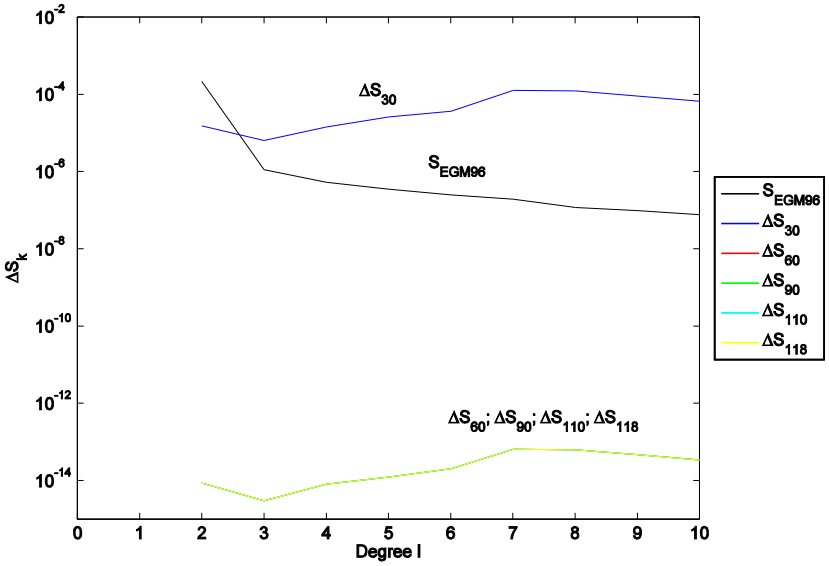

Figure 4.5: Degree variances with preconditioning

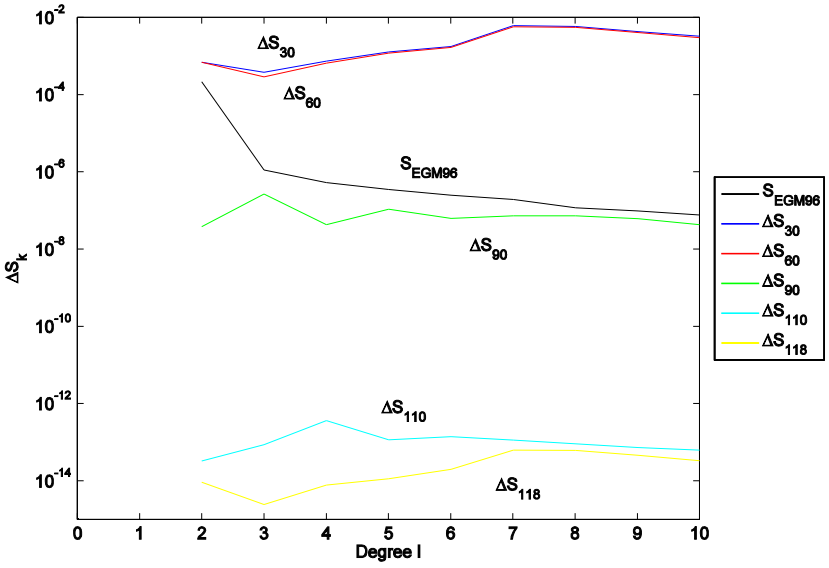

Figure 4.6: Degree variances without preconditioning

From these figures, it can be realized that the differences ΔS_k^l with preconditioning are smaller than 10^{-13} after 60 iterations, while the differences ΔS_k^l without preconditioning are smaller than 10^{-13} only after 118 iterations. The differences ΔS_k^l decrease very slowly from ΔS_{30} to ΔS_{90} without preconditioning, therefore it can be confirmed that the convergence with preconditioning is obviously faster than without. This means also that with preconditioning runtime can be reduced.

Reorthogonalization

According to section 2.2, it is known that the LSQR internal method for VCM computation requires reorthogonalization. To verify that, figures 4.7 and 4.8 illustrate the absolute differences $\Delta \mathbf{N}_k^{-1}$ according to

$$\Delta \mathbf{N}_k^{-1} = \mathbf{N}_k^{-1} - \mathbf{N}_{\text{ref}}^{-1} \tag{4.2}$$

after $k = 118$ iterations. $\mathbf{N}_{\text{ref}}^{-1}$ is the exact VCM obtained by brute-force inversion of the normal equation matrix.
The absolute differences $\Delta \mathbf{N}_k^{-1}$ with reorthogoanlization are smaller than there ones without reorthogonalization, by approximately ten orders of magnitude. Additionally, figures 4.9 and 4.10 show the relative relative differences $\delta \mathbf{N}_k^{-1}$ according to

$$(\delta \mathbf{N}_k^{-1})_{i,j} = \frac{(\Delta \mathbf{N}_k^{-1})_{i,j}}{(\mathbf{N}_{\text{ref}}^{-1})_{i,j}} \quad i,j = 1, \cdots, 118, \tag{4.3}$$

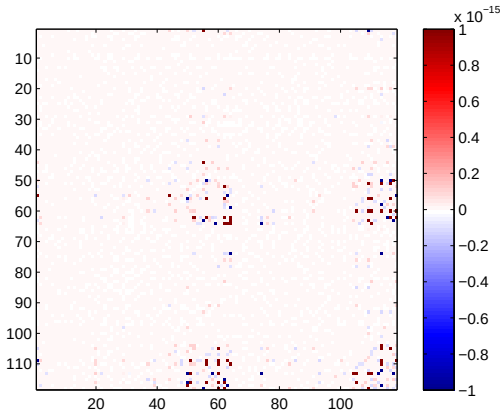

Figure 4.7: $\Delta \mathbf{N}_{118}^{-1}$ with reorthogonalization

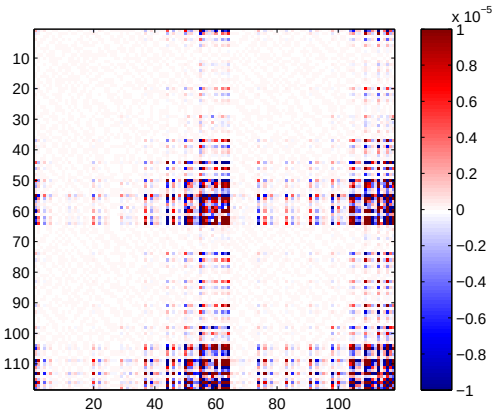

Figure 4.8: $\Delta \mathbf{N}_{118}^{-1}$ without reorthogonalization

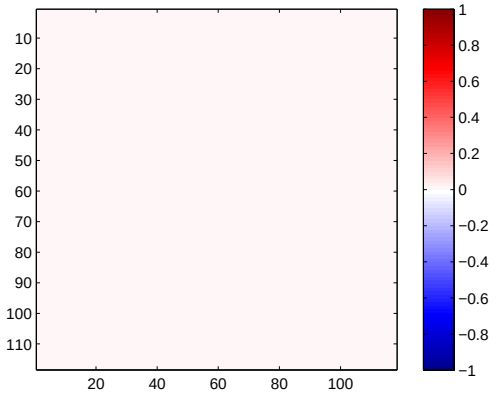

Figure 4.9: $\delta \mathbf{N}_{118}^{-1}$ with reorthogonalization

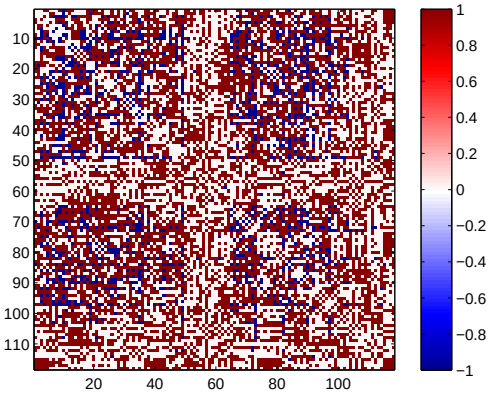

Figure 4.10: $\delta \mathbf{N}_{118}^{-1}$ without reorthogonalization

again after $k = 118$ iterations.
The relative differences $\delta\mathbf{N}_{118}^{-1}$ improve considerably with reorthogonalization. $\delta\mathbf{N}_{118}^{-1}$ with reorthogonalization has 99.7% values smaller than 10^{-10}, while there are only 42.3% values smaller than 1 without reorthogonalization. The numerical comparisons are represented in Table 4.4.

		Diagonal only		Full matrix	
Reorthogonalization		with	without	with	without
Percentage $(\delta\mathbf{N}_k^{-1})_{i,j}$	< 1	100%	79.7%	100%	42.3%
	< 0.1	100%	11.9%	100%	1.03%
	$< 10^{-10}$	100%	11.9%	99.7%	1.03%

Table 4.4: Impact of reorthogonalization

Conditioning

According to section 2.3.3, the MC method for VCM computation can be improved by conditioning. In order to verify its feasibility, I set the number of random samples to $M = 800$ and plot the absolute difference $\Delta\mathbf{N}_k^{-1}$ as well as the relative differences $\delta\mathbf{N}_k^{-1}$ after $k = 118$ iterations, cf. figures 4.11 to 4.14.

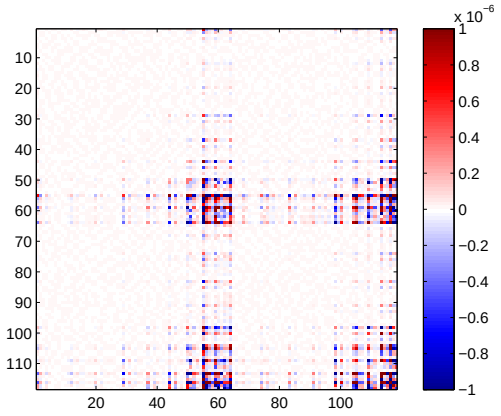

Figure 4.11: $\Delta\mathbf{N}_{118}^{-1}$ with conditioning

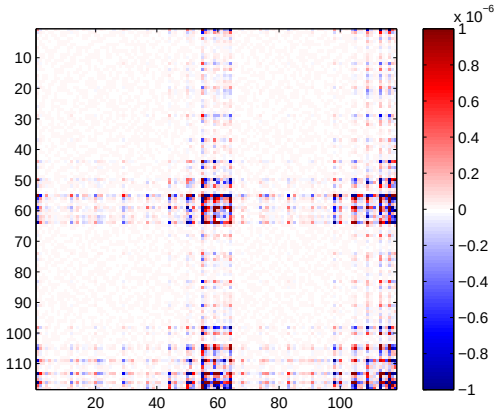

Figure 4.12: $\Delta\mathbf{N}_{118}^{-1}$ without conditioning

In figure 4.11 there are more values near to zero than in figure 4.12, i.e., the absolute differences $\Delta\mathbf{N}_{118}^{-1}$ with conditioning are better than without. From figures 4.13 and 4.14, it is clear that $\delta\mathbf{N}_{118}^{-1}$ is improved by conditioning with success. The relative differences $\delta\mathbf{N}_{118}^{-1}$ with conditioning come up with 88.4% of the values smaller than 1, while there are only 74.5% of the values smaller than 1 without conditioning. At the same time 49.9% of the values with conditioning are smaller than 0.1, opposed to only 33.3% of the values without conditioning. The numerical comparisons are represented

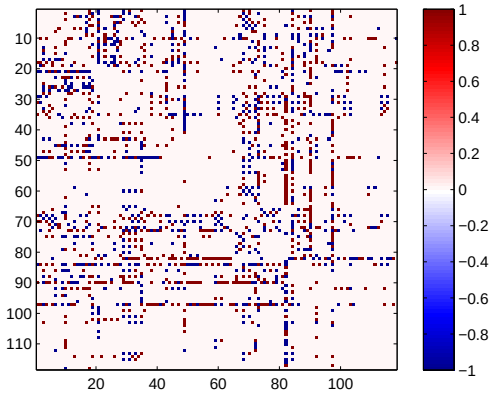

Figure 4.13: $\delta \mathbf{N}_{118}^{-1}$ with conditioning

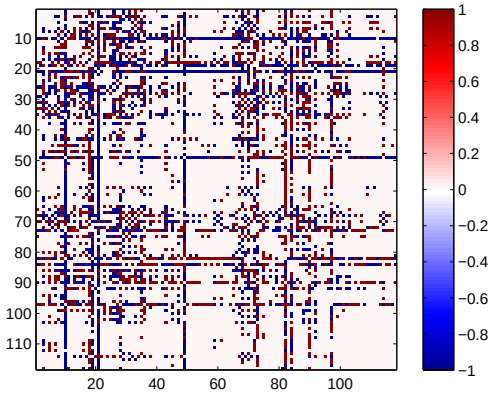

Figure 4.14: $\delta \mathbf{N}_{118}^{-1}$ without conditioning

in Table 4.5. It is important to note that 7.5% of the values are smaller than 0.01 with conditioning, but only 4.3% of the values without conditioning. On the diagonal, however, 61.5% of the values are smaller than 0.01 with conditioning and only 15.3% of the values without conditioning. This means that the results on the diagonal are improved by conditioning better than off the diagonal.

		Diagonal only		Full matrix	
Conditioning		with	without	with	without
Percentage $(\delta \mathbf{N}_k^{-1})_{i,j}$	< 1	100%	100%	88.4%	74.5%
	< 0.1	100%	98.3%	49.9%	33.3%
	< 0.01	61.0%	15.3%	7.5%	4.3%

Table 4.5: Impact of conditioning

Comparisons between different numbers of iterations

Figures 4.15 to 4.19 illustrate the relative differences $\delta \mathbf{N}_k^{-1}$ using the LSQR internal method for VCM computation after k=30, 60, 90, 110, 118 iterations, respectively.
After 110 iterations 85.6% of the values are smaller than 1, For 90 and 60 iterations the number drops to 79.6%, 80% respectively, cf. Table 4.6. Hence, the LSQR internal method has a rather poor convergence behavior. Therefore, the LSQR internal method is not suitable to be applied to large systems of equations, which need to be resolved only after a few iterations. The results become even worse without adapting reorthogonalization, cf. figures 4.20 to 4.24.
Compared to the figures with reorthogonalization, it can be seen that after 30, 60 iterations the results are almost the same as without reorthogonalization. This is because the orthogonality of $\mathbf{U}_{k+1}$ is not destroyed yet. But after 90, 110 and 118 iterations the

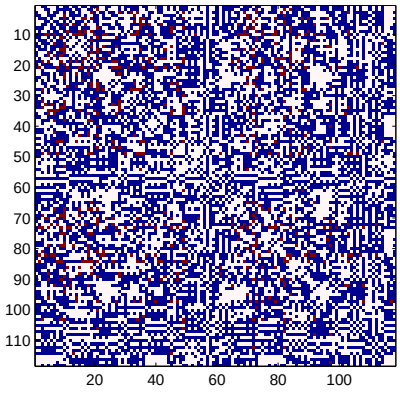

Figure 4.15: $\delta \mathbf{N}_{30}^{-1}$ with reorthogonalization

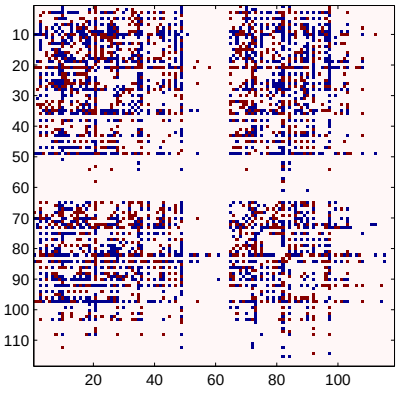

Figure 4.16: $\delta \mathbf{N}_{60}^{-1}$ with reorthogonalization

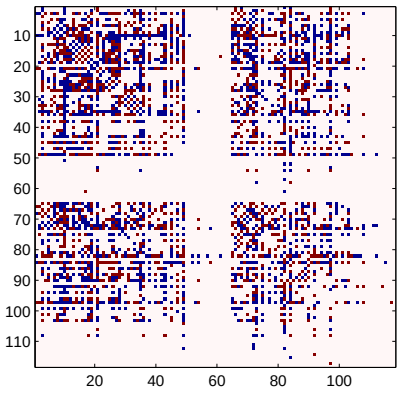

Figure 4.17: $\delta \mathbf{N}_{90}^{-1}$ with reorthogonalization

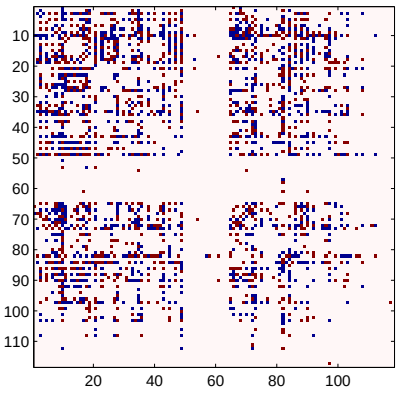

Figure 4.18: $\delta \mathbf{N}_{110}^{-1}$ with reorthogonalization

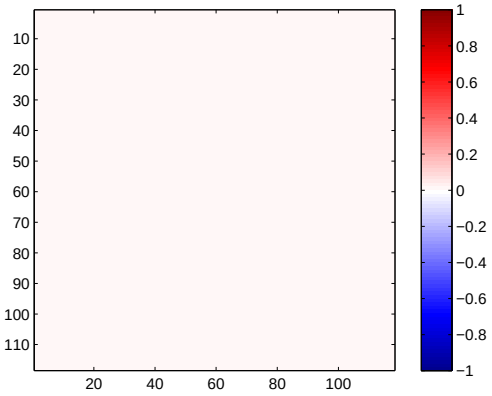

Figure 4.19: $\delta \mathbf{N}_{118}^{-1}$ with reorthogonalization

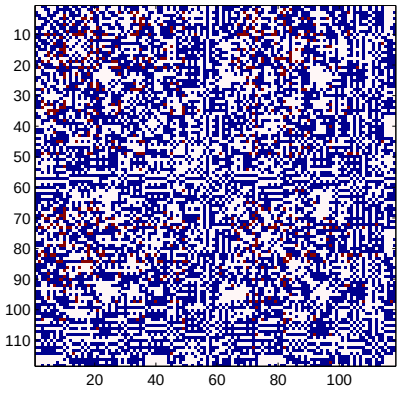

Figure 4.20: $\delta\mathbf{N}_{30}^{-1}$ without reorthogonalization

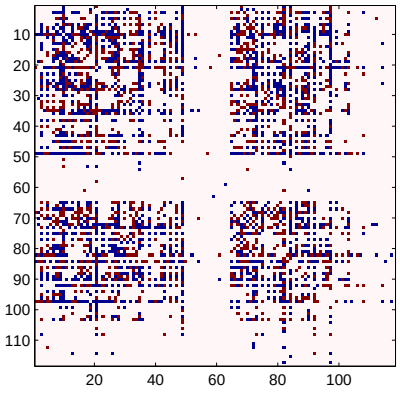

Figure 4.21: $\delta\mathbf{N}_{60}^{-1}$ without reorthogonalization

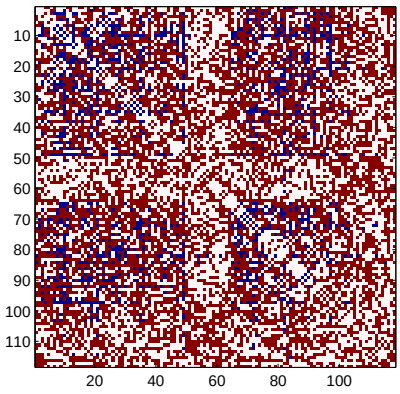

Figure 4.22: $\delta\mathbf{N}_{90}^{-1}$ without reorthogonalization

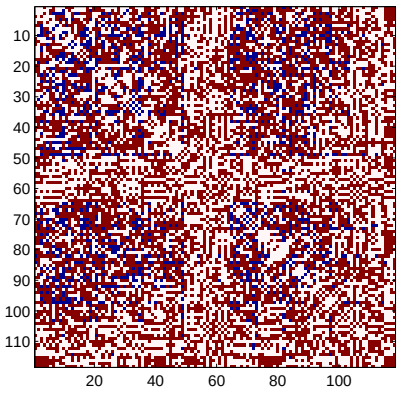

Figure 4.23: $\delta\mathbf{N}_{110}^{-1}$ without reorthogonalization

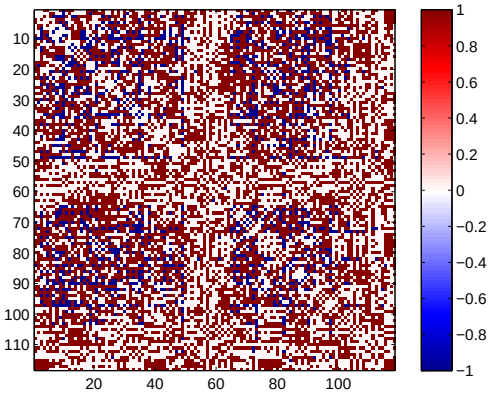

Figure 4.24: $\delta\mathbf{N}_{118}^{-1}$ without reorthogonalization

	#iterations	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$
Diagonal only	30	100%	0%
	60	100%	31.3%
	90	100%	37.3%
	110	100%	78.8%
	118	100%	100%
Full matrix	30	45.5%	1.2%
	60	80.0%	56.1%
	90	79.6%	56.4%
	110	85.6%	62.5%
	118	100%	100%

Table 4.6: Quality of VCM using the LSQR internal method with reorthogonalization

results become much worse without reorthogonalization. The orthogonality of $\mathbf{U}_{k+1}$ is destroyed at that stage, cf. Table 4.7.

	#iterations	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$
Diagonal only	30	100%	0%
	60	100%	31.4%
	90	100%	5.1%
	110	80.5%	6.8%
	118	79.7%	11.9%
Full matrix	30	45.5%	1.1%
	60	80.4%	55.6%
	90	41.7%	1.1%
	110	42.7%	1.1%
	118	42.3%	1.0%

Table 4.7: Quality of VCM using the LSQR internal method without reorthogonalization

According to the investigations above, figures 4.25 to 4.29 depict the relative differences $\delta\mathbf{N}_k^{-1}$ using the MC method for VCM estimation after k=30, 60, 90, 110 and 118 iterations based on 800 random samples. Conditioning is applied.
From the figures, it can be seen that the results after 60, 90 and 110 iterations are similar to the result after 118 iterations. In Table 4.8, 88.4% of the values are smaller than 1 after 60, 90, 110 and 118 iterations. This means that the MC method converges after only a few iterations.
Finally, figures 4.30 to 4.34 show the relative differences $\delta\mathbf{N}_k^{-1}$ without conditioning.
According to Table 4.9, the result without conditioning after 60 iterations has only

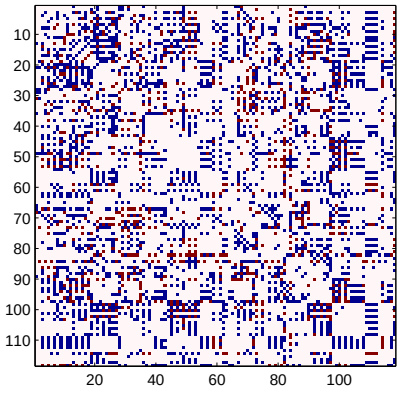

Figure 4.25: $\delta \mathbf{N}_{30}^{-1}$ with conditioning

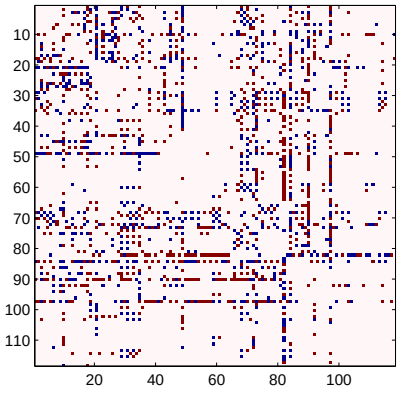

Figure 4.26: $\delta \mathbf{N}_{60}^{-1}$ with conditioning

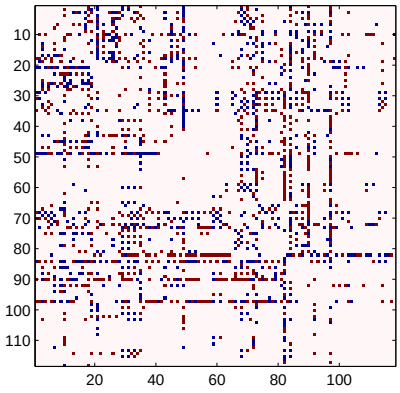

Figure 4.27: $\delta \mathbf{N}_{90}^{-1}$ with conditioning

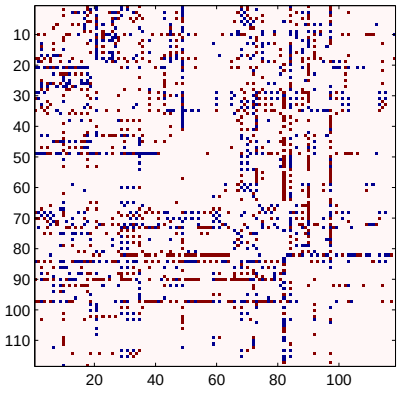

Figure 4.28: $\delta \mathbf{N}_{110}^{-1}$ with conditioning

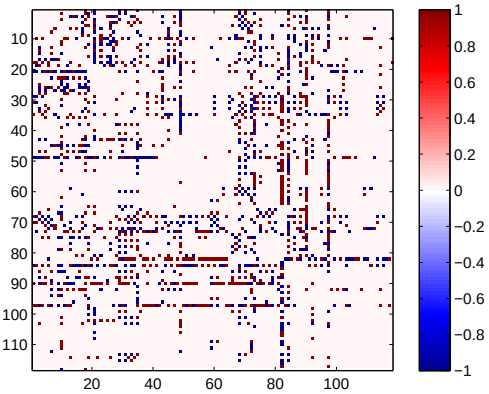

Figure 4.29: $\delta \mathbf{N}_{118}^{-1}$ with conditioning

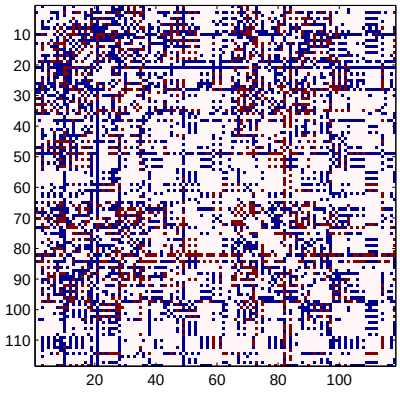

Figure 4.30: $\delta \mathbf{N}_{30}^{-1}$ without conditioning

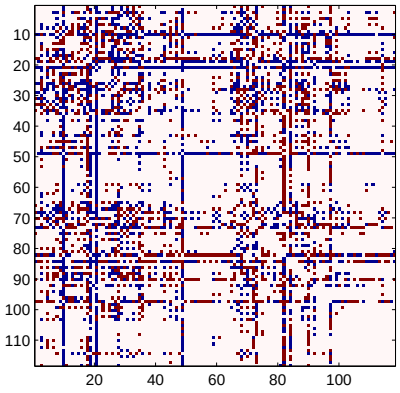

Figure 4.31: $\delta \mathbf{N}_{60}^{-1}$ without conditioning

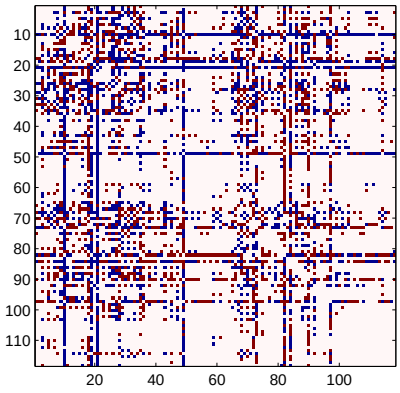

Figure 4.32: $\delta \mathbf{N}_{90}^{-1}$ without conditioning

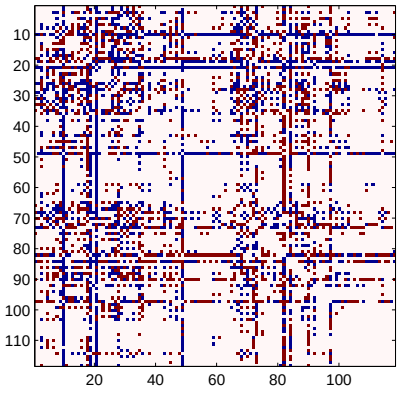

Figure 4.33: $\delta \mathbf{N}_{110}^{-1}$ without conditioning

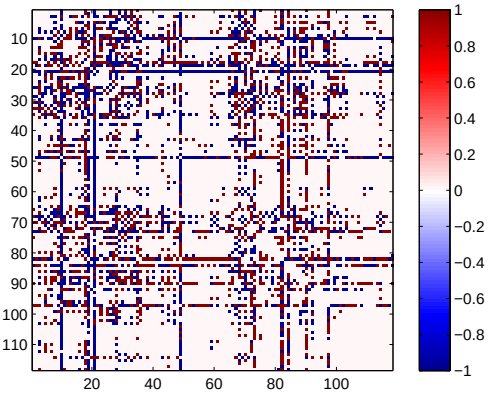

Figure 4.34: $\delta \mathbf{N}_{118}^{-1}$ without conditioning

	#iterations	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$
Diagonal only	30	100%	49.1%
	60	100%	100%
	90	100%	100%
	110	100%	100%
	118	100%	100%
Full matrix	30	73.9%	5.3%
	60	88.4%	49.9%
	90	88.4%	49.9%
	110	88.4%	49.9%
	118	88.4%	49.9%

Table 4.8: Quality of VCM using the MC method with conditioning

74.5% of the values smaller than 1, while there are 88.4% of the values smaller than 1 with conditioning, cf. Table 4.8. It can be confirmed that the results without conditioning are worse than with conditioning. In general, the MC method for VCM estimation has a better convergence than the LSQR internal approach.

	#iterations	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$
Diagonal only	30	100%	45.8%
	60	100%	98.3%
	90	100%	98.3%
	110	100%	98.3%
	118	100%	98.3%
Full matrix	30	64.9%	3.4%
	60	74.5%	33.3%
	90	74.5%	33.3%
	110	74.5%	33.3%
	118	74.5%	33.3%

Table 4.9: Quality of VCM using the MC method without conditioning

Comparisons between the LSQR internal method and the Monte Carlo method

The comparison between Tables 4.6 and 4.8 shows that the results from the LSQR internal method become very accurate after 118 iterations. 100% of the values are smaller than 0.1, opposed to only 49.9% using the MC method. To sum up, if the LSQR internal method has full iterations, the exact VCM can be computed. However, if the number

of iterations is small, the results become poor. The quality of the MC method mainly depends on the number of random samples M. Theoretically M should be expanded to infinity. Because the size of the multiple solution vector $\mathbf{X} = [\mathbf{x}, \mathbf{s}_x^{(1)}, \mathbf{s}_x^{(2)}, \ldots, \mathbf{s}_x^{(M)}]$ is $u \cdot (M+1)$, and the multiple observation vector $\mathbf{Y} = [\mathbf{y}, \mathbf{s}_l^{(1)}, \mathbf{s}_l^{(2)}, \ldots, \mathbf{s}_l^{(M)}]$ needs $n \cdot (M+1) \cdot 8$[2] byte storage space, M has to be kept rather small to avoid memory runoff.

4.3.2 A more advanced example

This section deals with the problem of solving a system of equations with $n = 256\,000$ unknowns and a spectral resolution of $l_{\max} = 50$, i.e., the number of unknowns is

$$u = l_{\max}^2 + 2 \cdot l_{\max} - 2 = 50^2 + 2 \cdot 50 - 2 = 2598. \tag{4.4}$$

Therefore, the augmented vector $\mathbf{X} = [\mathbf{x}, \mathbf{s}_x^{(1)}, \mathbf{s}_x^{(2)}, \ldots, \mathbf{s}_x^{(M)}]$ needs $u \cdot (M+1) \cdot 8 = 2598 \cdot 801 \cdot 8 = 16\,647\,984$ byte ≈ 15.9 MB and the augmented observation vector $\mathbf{Y} = [\mathbf{y}, \mathbf{s}_l^{(1)}, \mathbf{s}_l^{(2)}, \ldots, \mathbf{s}_l^{(M)}]$ needs $n \cdot (M+1) \cdot 8 = 256\,000 \cdot 801 \cdot 8 = 1640\,400\,000$ byte ≈ 1.5 GB. The simulated SST data is based on the EGM96 model up to degree and order 300.

Parallel computing

In order to test the performance of the parallel implementation, 1, 2, 4, 8, 16 and 32 nodes are used to compute the spherical harmonic coefficients and the differences ΔS_k between the estimate after 50 iterations and the reference EGM96 model. The results using MPI, OpenMP respectively, are summarized in the figures 4.35 and 4.36.
In the figures, the results with 1, 2, 4, 8, 16 and 32 nodes are identical, i.e., $\Delta S_{\text{MPI=1}} = \Delta S_{\text{MPI=2}} = \Delta S_{\text{MPI=4}} = \Delta S_{\text{MPI=8}} = \Delta S_{\text{MPI=16}} = \Delta S_{\text{MPI=32}}$ and $\Delta S_{\text{OpenMP=1}} = \Delta S_{\text{OpenMP=2}} = \Delta S_{\text{OpenMP=4}} = \Delta S_{\text{OpenMP=8}} = \Delta S_{\text{OpenMP=16}} = \Delta S_{\text{OpenMP=32}}$.
Moreover, figure 4.37 proves the consistency between the MPI and OpenMP implementation. The results are identical, i.e., $\Delta S_{\text{MPI}} = \Delta S_{\text{OpenMP}}$ holds true. Hence, the program works correctly.
Figure 4.38 shows the differences ΔS_k between the estimated spherical harmonic coefficients and the EGM96 model after 10, 20, 30, 40 and 50 iterations.
All results are smaller than 10^{-8} and $\Delta S_{10} = \Delta S_{20} = \Delta S_{30} = \Delta S_{40} = \Delta S_{50}$ holds true, i.e., the convergence is very good. The final spherical harmonic coefficients are obtained with less than 10 iterations.

Runtime results

The command

```
t=MPI_Wtime()
```

[2] the size of a double value is 8 byte

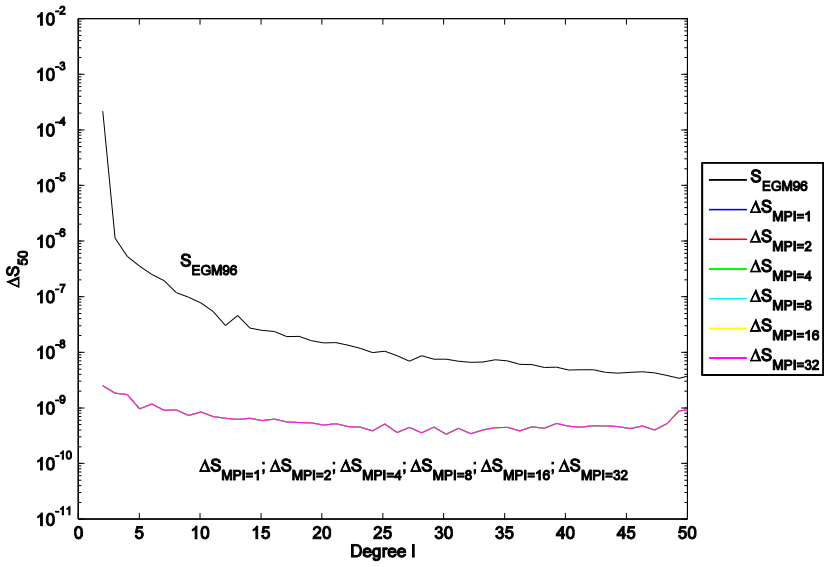

Figure 4.35: Degree variances ΔS_{50} using MPI for parallelization

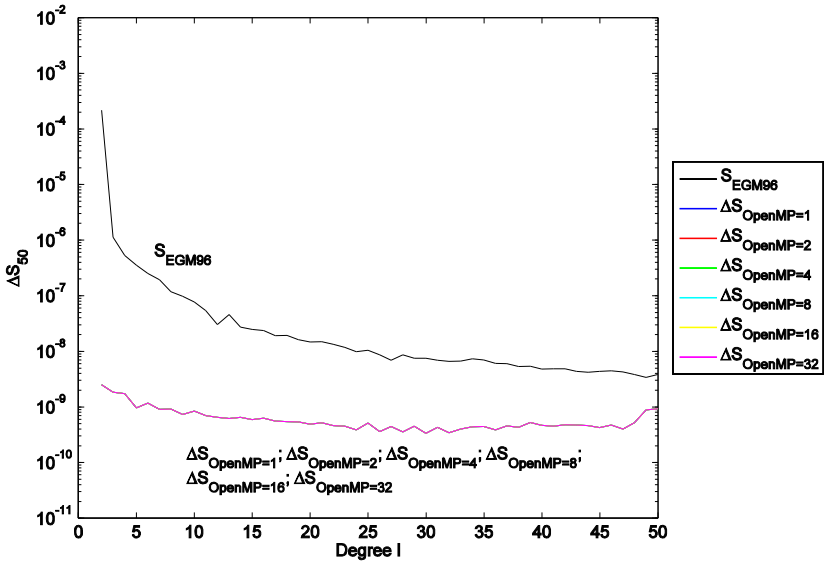

Figure 4.36: Degree variances ΔS_{50} using OpenMP for parallelization

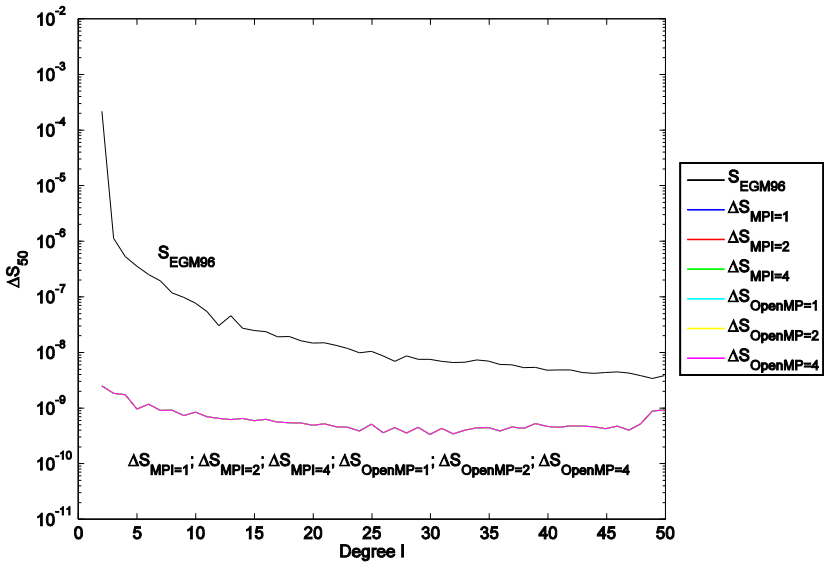

Figure 4.37: Degree variances ΔS_{50} using either MPI or OpenMP

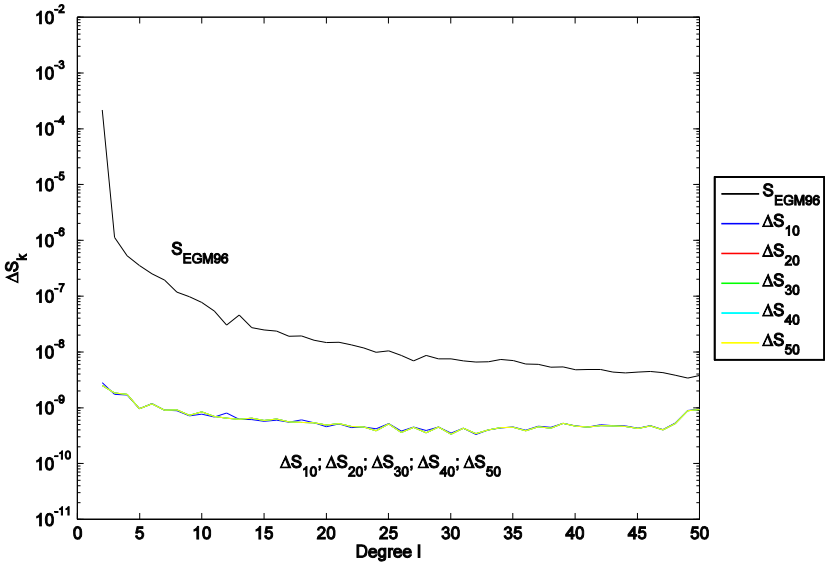

Figure 4.38: Degree variances ΔS_k

is used to get the runtime results. The total runtime becomes

$$t_{\text{total}} = t_{\text{end}} - t_{\text{start}},$$

where t_{start} denotes the starting time and t_{end} denotes the ending time. The total runtime results of the LSQR internal method for VCM estimation are presented in Table 4.10, the achievements using the Monte Carlo method in Table 4.11.

#nodes	32	16	8	4	2	1
Runtime MPI [s]	865	1264	2016	2064	3638	6826
Runtime OpenMP [s]	488	656	1027	1044	1838	3414

Table 4.10: Total runtime results of the LSQR internal method for VCM estimation (50 iterations)

#nodes	32	16	8	4	2	1
Runtime MPI [s]	2909	4750	8560	8894	16 601	31 158
Runtime OpenMP [s]	2386	4059	7155	7939	15 732	27 020

Table 4.11: Total runtime results of the Monte Carlo method for VCM estimation (50 iterations)

For both implementations satisfying scaling results could be achieved (except for the step from 4 to 8 CPUs). Basically, runtime for the MPI implementation is roughly twice as high than for OpenMP. This is because OpenMP uses 2 CPUs per node opposed to 1 CPU for MPI. The additional runtime requirement for VCM estimation compared to the original LSQR method (without VCM estimation) is examined as follows:

$$t^{\text{LSQR internal method}} = t_{\text{total}}^{\text{LSQR internal method}} - t_{\text{total}}^{\text{LSQR method}},$$
$$t^{\text{MC method}} = t_{\text{total}}^{\text{MC method}} - t_{\text{total}}^{\text{LSQR method}}.$$

For example, the total runtime of the LSQR method with MPI on 1 CPU is 6774 s, i.e., $t_{\text{total}}^{\text{LSQR method}} = 6774\,\text{s}$. From Tables 4.10 and 4.11 it is known that

$$t_{\text{total}}^{\text{LSQR internal method}} = 6826\,\text{s}$$
$$t_{\text{total}}^{\text{MC method}} = 31\,158\,\text{s}.$$

Therefore, one gets

$$t^{\text{LSQR internal method}} = 6826\,\text{s} - 6774\,\text{s} = 52\,\text{s}$$

$$t^{\text{MC method}} = 31\,158\,\text{s} - 6774\,\text{s} = 24\,384\,\text{s}$$

Obviously, the MC method needs much more runtime than the LSQR internal method. The major reason lies in the additional effort to treat the 800 random samples (i.e., solving the system of equations for 800+1 observation vectors).

VCM comparisons

In this section, the VCM ($\mathbf{N}_k^{-1}$) of both methods presented before will be compared by analyzing their relative differences $\delta\mathbf{N}_k^{-1}$. In order to verify whether the exact VCM ($\mathbf{N}^{-1}$) can be obtained after a few iterations, the results after 10, 20, 30, 40 and 50 iterations are investigated. Figure 4.39 to 4.43 present the relative differences $\delta\mathbf{N}_k^{-1}$ results from the LSQR internal method.
From the figures, it can be stated that the achieved accuracy is rather poor. To get a deeper inside in the performance of the method, the relative differences $\delta\mathbf{N}_k^{-1}$ are numerically presented in Table 4.12.

		Percentage	
	#iterations	$(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	$(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$
Diagonal only	10	100%	0%
	20	100%	0%
	30	100%	0%
	40	100%	0%
	50	100%	0%
Full matrix	10	39.3%	1.6%
	20	31.2%	1.9%
	30	25.3%	1.8%
	40	21.2%	1.6%
	50	18.5%	1.5%

Table 4.12: LSQR internal method with reorthogonalization

Only 39.3% of the values are smaller than 1 after 10 iterations. The results after 20, 30, 40 and 50 iterations become even worse. Hence, it is not possible to get the accurate VCM information with the LSQR internal method after only a few iterations.
The results using the MC method (with 800 random samples) are outlined in figures 4.44 to 4.48, in Table 4.13 respectively. The results are identical for any number of iterations performed.
Table 4.13 confirms that only 7% of the total values are smaller than 1, while 100% on the diagonal. In fact, all values on the diagonal are even smaller than 0.01, i.e., the results on the diagonal are much better than off the diagonal. The results will not change, unless the number of samples increase, but, according to Schuh & Alkhatib (2007), the accuracy of the MC method is proportional to $\frac{1}{\sqrt{M}}$. The square root decreases very slowly, therefore the results from the MC method converge very slowly to the exact values by increasing the number of samples. On the other hand, if the number of samples M increases, memory requirement increase accordingly.

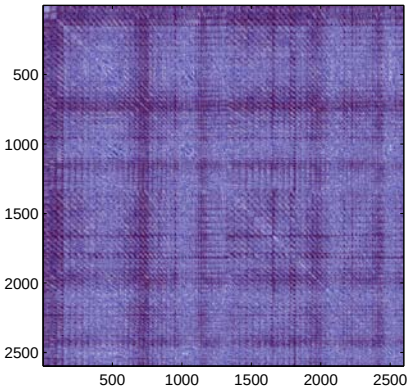

Figure 4.39: $\delta\mathbf{N}_{10}^{-1}$ with reorthogonalization

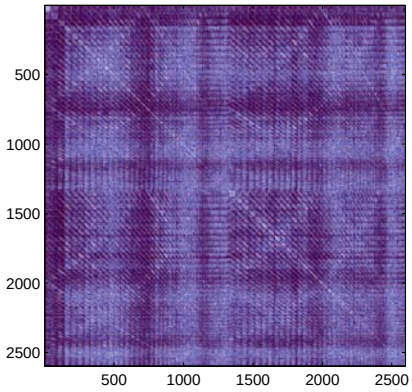

Figure 4.40: $\delta\mathbf{N}_{20}^{-1}$ with reorthogonalization

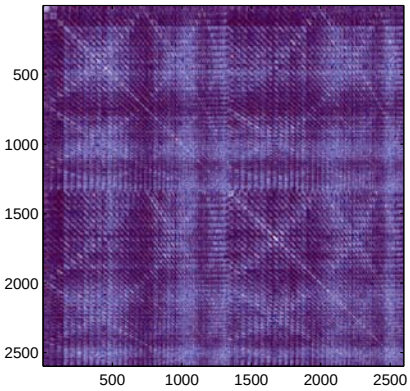

Figure 4.41: $\delta\mathbf{N}_{30}^{-1}$ with reorthogonalization

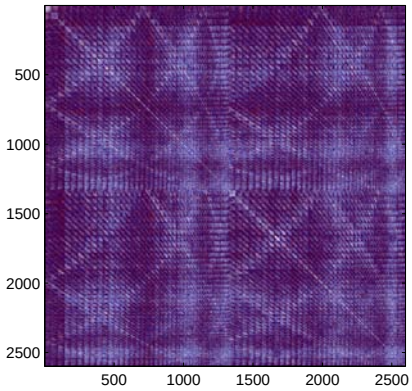

Figure 4.42: $\delta\mathbf{N}_{40}^{-1}$ with reorthogonalization

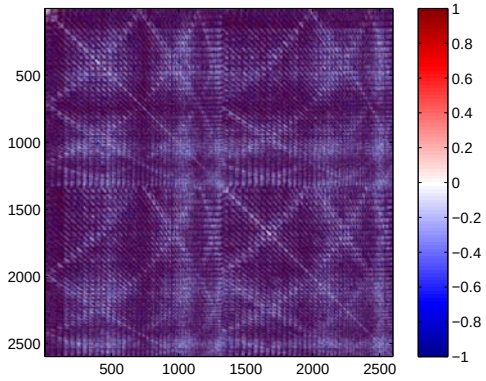

Figure 4.43: $\delta\mathbf{N}_{50}^{-1}$ with reorthogonalization

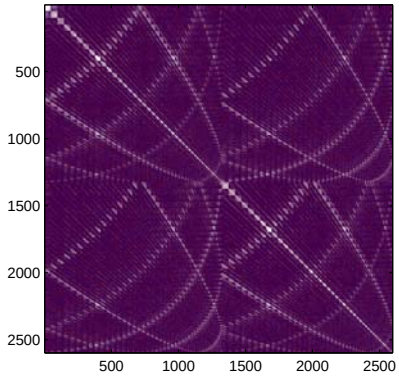

Figure 4.44: $\delta\mathbf{N}_{10}^{-1}$ with conditioning

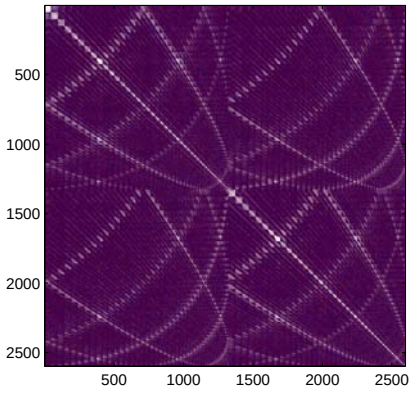

Figure 4.45: $\delta\mathbf{N}_{20}^{-1}$ with conditioning

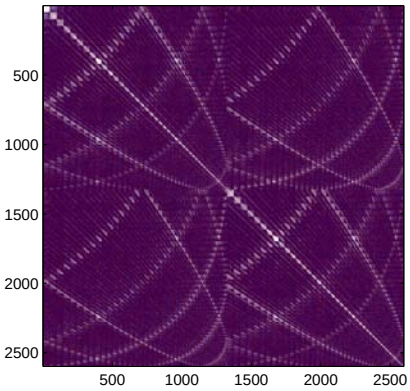

Figure 4.46: $\delta\mathbf{N}_{30}^{-1}$ with conditioning

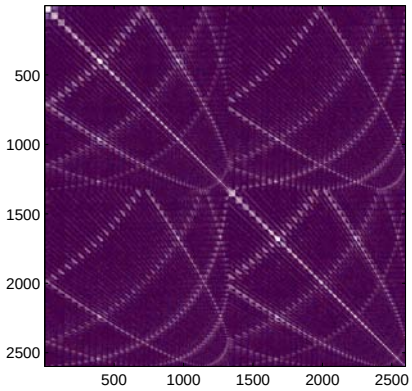

Figure 4.47: $\delta\mathbf{N}_{40}^{-1}$ with conditioning

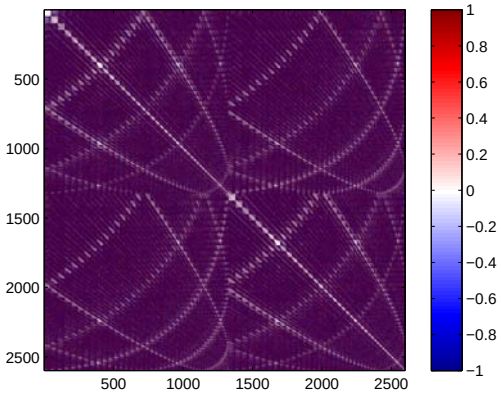

Figure 4.48: $\delta\mathbf{N}_{50}^{-1}$ with conditioning

	#iterations	Percentage $(\delta\mathbf{N}_k^{-1})_{i,j} < 1$	$(\delta\mathbf{N}_k^{-1})_{i,j} < 0.1$	$(\delta\mathbf{N}_k^{-1})_{i,j} < 0.01$
Diagonal only	10	100%	100%	100%
	20	100%	100%	100%
	30	100%	100%	100%
	40	100%	100%	100%
	50	100%	100%	100%
Full matrix	10	7.0%	1.2%	0.2%
	20	7.0%	1.2%	0.2%
	30	7.0%	1.2%	0.2%
	40	7.0%	1.2%	0.2%
	50	7.0%	1.2%	0.2%

Table 4.13: Monte Carlo method with conditioning

Impact of reorthogonalization and conditioning

Here, the impact of reorthogonalization and conditioning is addressed once more. Figures 4.49 and 4.50 show the corresponding results without these performance tools after 50 iterations.

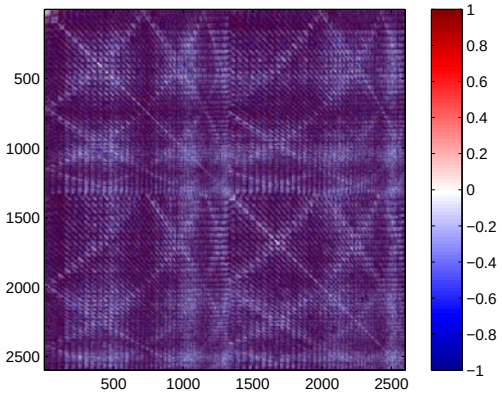

Figure 4.49: $\delta\mathbf{N}_{50}^{-1}$ without reorthogonalization

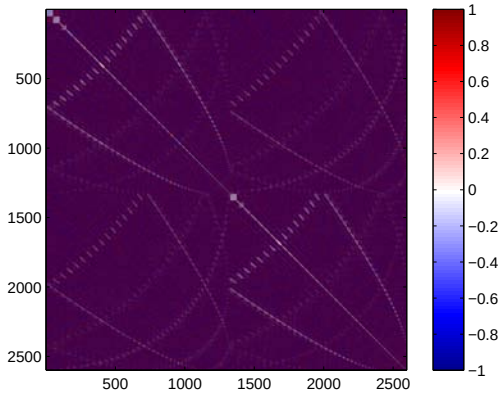

Figure 4.50: $\delta\mathbf{N}_{50}^{-1}$ without conditioning

The comparison of the figures 4.49 and 4.43 identifies no large differences. This is because the orthogonality of $\mathbf{U}_{k+1}$ is not destroyed yet after 50 iterations. The numerical values in Table 4.14 confirm the findings.
In contrast, the MC method performs much better with conditioning than without, cf. figure 4.50 opposite to figure 4.48, Table 4.15 respectively. There are more values smaller than 1 with conditioning than without. Especially on the diagonal, all values are smaller than 0.01 with conditioning, but only 16.2% of the values without conditioning.

Reorthogonalization		Diagonal only with	Diagonal only without	Full matrix with	Full matrix without
Percentage $(\delta \mathbf{N}_k^{-1})_{i,j}$	< 1	100%	100%	18.5%	18.5%
	< 0.1	0%	0%	1.5%	1.5%

Table 4.14: Impact of reorthogonalization

Conditioning		Diagonal only with	Diagonal only without	Full matrix with	Full matrix without
Percentage $(\delta \mathbf{N}_k^{-1})_{i,j}$	< 1	100%	100%	7.0%	1.8%
	< 0.1	100%	95.8%	1.2%	0.3%
	< 0.01	100%	16.2%	0.2%	0.1%

Table 4.15: Impact of conditioning

The runtime requirements for reorthogonalization and conditioning are shown in the Tables 4.16 and 4.17.

#iterations (16 nodes)	10	20	30	40	50
Reorthogonalization [s]	0.181	0.561	1.136	1.908	2.888

Table 4.16: Runtime requirement for reorthogonalization

#iterations (16 nodes)	10	20	30	40	50
Conditioning [s]	553.892	551.779	559.155	554.629	555.809

Table 4.17: Runtime requirement for conditioning

Table 4.16 shows that if the number of iteration increases, the runtime requirement for reorthogonalization will also increase. This is due to the extension of the matrices $\mathbf{U}_k$ and $\mathbf{V}_k$. In Table 4.17, the runtime requirements for conditioning are identical. They are independent on the number of iteration, cf. Table 2.5.

LSQR internal method versus MC method

Inspecting Tables 4.12 and 4.13 it turns out that the LSQR internal method outperforms the MC method as long as the whole VCM is concerned. On the diagonal only, however, the MC methods clearly provides better results. On the other hand, runtime requirements for the MC method are much higher than for the LSQR internal method. Hence, for small problems the LSQR internal method computes the VCM efficiently. However, large LS problems should be treated with the MC method.

Chapter 5

Discussion

Modern satellite missions (such as GOCE) are very effective for the determination of the global gravity field. However, due to the large number of observations it is a challenging task to solve the resulting systems of equations. In order to solve such problems effectively, the LSQR method is used. The iterative LS solver is characterized by a good convergence and it is suited well for parallel computing. In addition to the spherical harmonic coefficients themselves, their quality is also important. Therefore it is necessary to approximate the variance-covariance matrix of the parameter estimate ($\mathbf{N}^{-1}$). Two methods are introduced to solve the VCM problem. First, the LSQR internal method which is directly deduced from the LSQR method. It has a bad convergence behavior. Its results become accurate only after the full set of iterations. Contrarily, the Monte Carlo method has a better convergence. The VCM is computed by a series of random samples.
To sum up, both VCM evaluation methods have their advantages and disadvantages. The LSQR internal method requires only small memory availability and has a very good accuracy after the full set of iterations. Moreover, it is easy to implement. On the other hand, however, it converges very slowly. Above all, the VCM estimates show a completely different convergence behavior compared to the gravity field parameters (spherical harmonic coefficients). Opposed to the LSQR internal method, the Monte Carlo approach is more appropriate for VCM estimation concerning convergence issues. The number of iterations has small influence on the results, but the method works on the premise that the approximation to the exact VCM is good enough. As the achieved accuracy depends on the number of normal distributed random samples, it is very costly in terms of runtime, and a huge memory is required to store those random samples. It turns out that the variances can be approximated with better accuracy than the covariances. The implementation of the Monte Carlo method is complex.

Bibliography

Amdahl G. (1967) Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, AFIPS Conference Proceedings, 30, pp. 483-485

Baur O. (2007) Die Invariantendarstellung in der Satellitengradiometrie: Theoretische Betrachtungen und numerische Realisierung anhand der Fallstudie GOCE, DGK, Series C, 609, Munich

Baur O., Austen G. (2005) A parallel iterative algorithm for large-scale problems of type potential field recovery from satellite data, Proceedings Joint champ/grace Science Meeting, GeoForschungsZentrum Potsdam, online publication (www.gfz-potsdam.de/pb1/JCG)

Baur O., Austen G., Kusche J. (2007) Efficient GOCE satellite gravity field recovery based on least-squares using QR decomposition, J. Geod., 82, pp. 207-221

Benbow S. J. (1999) Solving generalized least squares problems with LSQR, SIAM J. Matrix Anal. Appl., Vol. 21, No. 1, pp. 166-177

Ditmar P., Klees R., Kostenko F. (2003) Fast and accurate computation of spherical harmonic coefficients from satellite gravity gradiometry data, J. Geod., 76, pp. 690-750

Flynn M. (1972) Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948

Heiskanen W. A., Moritz H. (1967) Physical Geodesy. New York: W H Freeman

Koch K. (1999) Parameter estimation and hypothesis testing in linear models. Springer, Berlin Heidelberg New York

Kusche J., Mayer-Gürr T. (2001) Iterative Solution of ill-Conditioned Normal Equations by Lanczos Methods, Proc. IAG Scientific Assembly, Budapest, Hungary

Kusche J. (2002) On fast multigrid iteration techniques for the solution of normal equations in satellite gravity recovery, J. Geodyn., 33, pp. 173-186

Menke W. (1984) Geophysical data analysis: Discrete inverse theory, Academic Press San Diego

Paige C. C., Saunders M. A. (1982a) LSQR: An algorithm for sparse linear equations and sparse least squares, ACM T. Math. Software, 8, pp. 43-71

Paige C. C., Saunders M. A. (1982b) LSQR: Sparse linear equations and least squares problems, ACM T. Math. Software, 8, pp. 195-209

Pail R., Plank G. (2002) Assessment of three numerical solution strategies for gravity field recovery from GOCE satellite gravity gradiometry implemented on a parallel platform, J. Geod., 76, pp. 462-474

Reubelt T., Austen G., Grafarend E. (2003) Harmonic Analysis of the Earth's Gravitational Field by Means of Semi-Continuous Ephemerides of a Low Earth Orbiting GPS-Tracked Satellite. Case Study: CHAMP, J. Geod., 77, pp. 257-278

Schuh W. D. (1996) Tailored numerical solution strategies for the global determination of the Earth's gravity field. Mitteilungen der Universität Graz., 81

Schuh W. D., Alkhatib H. (2007) Integration of the Monte Carlo covariance estimation strategy into tailored solution procedures for large-scale least squares problems, ITG, university of Bonn, J. Geod. 81, pp. 53-66

Yao Z. S., Roberts R. G., Tryggvason A. (1999) Calculating resolution and covariance matrices for seismic tomography with the LSQR method, Geophys. J. Int., 138, pp. 886-894