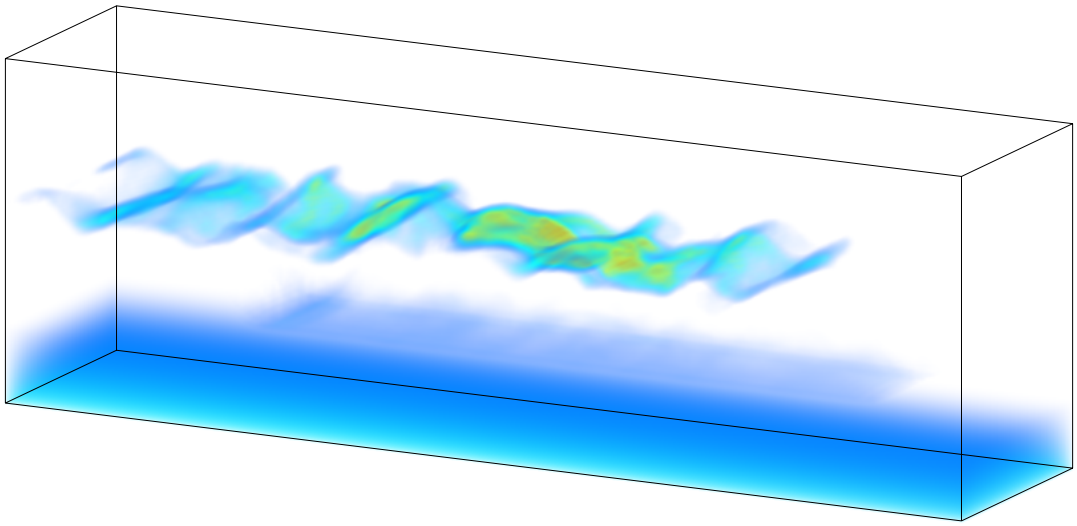
CHALMERS
UNIVERSITY OF TECHNOLOGY



invlib

A Generic Implementation of Bayesian Methods for Inverse Problems in Remote Sensing

Master's Thesis in Physics and Astronomy

SIMON PFREUNDSCHUH

Department of Earth and Space Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

invlib: A Generic Implementation of Bayesian Methods for Inverse Problems in Remote Sensing

SIMON PFREUNDSCHUH

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Earth and Space Sciences
Chalmers University of Technology
Gothenburg, Sweden 2016

invlib: A Generic Implementation of Bayesian Methods for Inverse Problems in Remote Sensing
SIMON PFREUNDSCHUH

Supervisor: Patrick Eriksson, Department of Earth and Space Sciences
Examiner: Patrick Eriksson, Department of Earth and Space Sciences

Department of Earth and Space Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Volume rendering of a three-dimensional, tomographic retrieval of a polar mesospheric cloud using simulated data from the future MATS satellite mission.

Typeset in LaTeX
Gothenburg, Sweden 2016

Abstract

Spaceborne earth observation missions constitute an essential source of information for climate science. In order to extract physically relevant quantities from the electromagnetic signals recorded by the detector onboard the satellite several data processing steps are required. This process, the so called *data retrieval*, requires a thorough mathematical formulation as well as an efficient implementation to ensure both the correctness of the retrieved data and the ability to handle large retrievals at a sufficiently high bandwidth. As part of this thesis project the invlib C++ template library has been developed, which implements Bayesian methods for inverse problems arising from the retrieval of remote sensing earth observation data. The library provides functionality for the memory-efficient calculation of maximum a posteriori estimators of Bayesian inverse problems with Gaussian priors and measurement error. This method, known in the field of remote sensing as the *optimal estimation method* (OEM), has been implemented using generic programming techniques in order to provide a maximum of flexibility and performance to the user of the library. The invlib library has been integrated into the *Atmospheric Radiative Transfer Simulator* (ARTS), which is a publicly available software package for the simulation of the propagation of electromagnetic radiation through the atmosphere. The integration of the retrieval functionality considerably simplifies the data retrieval workflow with ARTS, which previously required the use of a separate, external software package. In this thesis report the use of the invlib library for tomographic retrievals from the Odin SMR mission is demonstrated. Using the new implementation based on the invlib library, it is now possible to perform the tomographic retrievals of data from a complete half-orbit in a single computation. Furthermore, it is demonstrated how the computations can be parallelized using invlibs generic parallel matrix and vector types. Finally, also the use of the invlib library for the three-dimensional retrieval of simulated data from the MATS satellite mission is illustrated.

The principal result of this thesis is a *free software* C++ template library for remote sensing data retrieval, that emphasizes generality and performance. Furthermore, the invlib library has been integrated into the ARTS software package and the retrieval functionality made available to a large community of existing users. The capabilities of the library to perform and accelerate the solution of real world retrieval problems has been demonstrated by applying it to data from two different earth observation missions.

Acknowledgments

I would like to thank Patrick Eriksson for introducing me to the ARTS project and the opportunity to contribute to it as a developer. Furthermore, I want to thank him for his supervision and guidance during the work on this thesis project. I want to thank Ole Martin Christensen for the provision and preparation of the data sets and numerous interesting discussions on remote sensing. I would also like to acknowledge Dr. Jörn Ungermann for sharing his knowledge and experience in designing and implementing remote sensing data processing systems.

Finally, I would like to thank my family for their support, in particular Julia for being who she is.

The computations presented in this thesis were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

Contents

Introduction

It is scientific consent that the human-induced climate change and its related effects constitute a major challenge to the development of mankind (Oppenheimer et al., 2014). In order to find ways to efficiently reduce anthropological influences on the climate and mitigate the effects of global warming, a deep understanding of the climate system is indispensable. Advancing this understanding requires scientists to be able to monitor this system not only locally but on a global scale. The main source of such global observations are spaceborne earth observation missions. Already today, a large number of earth observation missions are orbiting the earth providing a constant stream of observational data containing information about its climate system. In general, however, the quantities measured by the detectors onboard these earth observation satellites are only indirectly related to physical properties of the atmosphere or the earth. The process of inferring these physical quantities from the data recorded by the detector onboard the satellite is called the *data retrieval* and is the subject of this thesis.

One such earth observation mission is the Swedish Odin satellite carrying the *Sub-Millimetre Radiometer* (SMR), which measures thermal emission from the atmosphere. The measurements recorded by the SMR are spectra of sub-millimeter radiation. The *Global environmental measurements and modeling* group at the *Department of Earth and Space Sciences* at *Chalmers University of Technology*, where this thesis project has been carried out, is responsible for the retrieval of atmospheric properties including trace gas concentrations, temperature and atmospheric pressure from this data. The previous implementation of the data retrieval used a combination of two software packages, that had to exchange large amounts of data. In addition to that, this implementation lacked the ability to exploit specific structures of the retrieval data as well as functionality to reduce its memory footprint. Thus, the primary aim of this thesis project was to simplify the data processing of the Odin SMR data by integrating all functionality required for the retrieval into a single software package as well as to extend the functionality of

the implementation to achieve improved performance.

To the best knowledge of the author, there is currently only a very limited number of publicly available implementations of maximum a posteriori methods for Bayesian inverse problems. One of them is the implementation provided by the Qpack package (Eriksson et al., 2005) developed by Patrick Eriksson. This is also the implementation that is currently used for the processing of the Odin SMR data. While these methods are commonly used for the processing of remote sensing data retrieval (Ungermann et al., 2015; Hoffmann et al., 2008), developing a general implementation of the method is difficult. This is because the retrieval requires an implementation of a so called *forward model*, which simulates the measurements performed by the satellite. Since the forward model implementation is usually highly problem dependent, the retrieval implementations are developed specifically for a given forward model.

Aim

The primary aim of this Master's thesis is the integration of retrieval functionality into the *Atmospheric Radiative Transfer Simulator* (ARTS), which is currently used as a forward model for the retrieval of the Odin SMR data. The implementation of the retrieval method should be optimized with respect to computational performance and memory footprint making it suitable also for large retrieval problems. In particular, the new implementation should yield improved performance for the retrieval of Odin SMR data and if possible also allow for the tomographic retrieval of data from a complete half-orbit of the satellite. While tomographic retrievals of Odin SMR data have been performed before, memory limitations required the splitting up of the data into chunks which then had to be processed separately. Apart from being cumbersome, this introduced considerable processing overhead due to the overlap between adjacent chunks.

As a secondary aim, the implementation should strive to be as general as possible, exploring the possibilities of developing a general-purpose library for remote sensing data retrieval which can be applied to a large number of retrieval problems without trading off performance.

Limitations

In terms of methodology, this thesis is limited to Bayesian methods for inverse problems with Gaussian priors and measurement errors. Furthermore, for all inverse problems treated here the existence of a suitable forward model is assumed. The discussion will be limited to mathematical and computational aspects of the retrieval and the physical results will be presented only briefly.

The main work of this thesis was conducted over a duration of four months, which of course required limiting its scope. During this time, the focus was put on integrating the retrieval functionality into ARTS and the parallelization of the tomographic retrievals of Odin SMR data. A treatment of the computational aspects of the calculation of measurement diagnostics has not been possible within the limited amount of time available for this project.

Method

The invlib library implements *maximum a posteriori estimators* (MAP) for Bayesian inverse problems with Gaussian prior and Gaussian measurement errors. In the field of atmospheric remote sensing this method is also known as the *optimal estimation method* (OEM) as presented in Rodgers (2000). The library is implemented in C++ and uses template programming techniques to avoid code duplication and simplify the interaction with the forward model implementation.

Results

The main result of this thesis is the invlib free software library, which provides a generic implementation of Bayesian methods for inverse problems with Gaussian prior and Gaussian measurement error. The code is freely accessible online (Pfreundschuh, 2016) and distributed under the MIT License (MIT, 2016). The invlib library has been integrated into the ARTS software package, which allows ARTS users to perform remote sensing data retrievals directly, without the need for additional software.

The new ARTS retrieval functionality has been used to perform tomographic

retrievals of Odin SMR data. By performing the retrieval on a dedicated compute cluster, it was possible to retrieve the data from a complete half-orbit of the satellite. Furthermore, the computations have been parallelized using MPI and distributed over multiple compute nodes, which significantly reduced the processing time required for the retrieval.

As a secondary result, it is demonstrated how the invlib library can be used for the retrieval of remote sensing data outside of ARTS using simulated data from the future MATS satellite mission. Also for this application, a considerable reduction in processing time could be achieved by parallelizing the computations.

Outline

Following this introduction, this thesis report contains four additional chapters. Chapter 1 introduces the mathematical theory of inverse problems and presents the numerical methods required to solve them efficiently. Chapter 2 contains a description of the design of the invlib library together with a general usage example. Chapter 3 describes the retrieval setup for tomographic measurements of the upper mesosphere based on Odin SMR data, which serves as a verification and benchmark case for the implementation. The results of this thesis project as well future work are discussed in Chapter 4.

Appendix A contains the algorithms required for the solution of inverse problems. Appendix B contains implementation details accompanying the invlib usage example from Chapter 2.

Chapter 1

Theory

This chapter introduces the mathematical theory and computational methods required for computing the solution of a retrieval problem. The first section introduces the mathematical theory of *inverse problems*, based on which a general solution of the retrieval problem can be formulated. Following this, the numerical methods required for the computation of such a solution are presented. The chapter closes with a discussion of general performance characteristics of the previously introduced methods.

1.1 Inverse Problem Theory

Although the problem of retrieving atmospheric parameters from satellite measurements may seem like a very specific one, the underlying mathematical problem is much more general. So general in fact that a whole mathematical theory, the theory of *inverse problems* or *inverse problem theory*, has been developed to treat problems of this kind. This section briefly introduces the main concepts of inverse problem theory relevant to the processing of remote sensing data. For a more in-depth treatment of the subject, the reader is referred to the excellent text by Tarantola (Tarantola, 2005).

The general setting in inverse problem theory is as follows: Given a system in an unknown state, the problem is to gain information on the state of this system. This state, however, can not be measured directly. Instead, the only way to obtain

information about it is through *indirect* measurements that depend in a known and predictable way on the state of the system.

This is analogous to the situation in atmospheric remote sensing. While it is not possible to measure the global state of the atmosphere directly, electromagnetic radiation propagating through the system can be measured by a suitable detector. Since the theory of radiative transfer is well understood, most situations allow an efficient simulation of the signals recorded by such a detector for given atmospheric states.

The theory of inverse problems provides a theoretical framework for the extraction of information about the state of the system from these indirect measurements as well as tools to investigate the accuracy and precision of this information.

1.1.1 The Forward Problem

As the name suggests, the actual problem in inverse problem theory is the inversion of yet another problem. The problem to be inverted is referred to as the *forward problem*. The fundamental assumption of inverse problem theory is that this problem can be solved efficiently by a so called *forward model*. This may be any model that allows to predict the measurement from a given state of the system. In the case of the Odin SMR data retrieval, this is a numerical simulation of radiation passing through the atmosphere. In mathematical terms, the forward model $\mathbf{F}$ is a mapping from the set of possible states of the system, the *state space* $\mathcal{S}$, into the space of possible measurements, the *measurement space* $\mathcal{M}$:

$$\mathbf{F} : \mathcal{S} \to \mathcal{M} \tag{1.1.1}$$

In what follows, it is assumed that there exist suitable mappings that allow the representation of elements of the state and measurement spaces by real vectors $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, respectively. The forward model $\mathbf{F}$ thus takes the form of a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$, mapping an n-dimensional state vector $\mathbf{x}$ onto an m-dimensional measurement vector $\mathbf{y}$. The respective dimensions n and m of the state and measurement space will later be important characteristics of the inverse problem at hand. In the case of the Odin SMR retrieval, the state space is the set of all possible parameter vectors that describe the model atmosphere. These may consist for example of the temperature or water vapor concentrations at different vertical and horizontal positions in the atmosphere. The measurement space is the set of all possible signals that can be recorded by the detector onboard the satellite.

1.1.2 The Inverse Problem Solution

In an ideal setting, a solution of the inverse problem given by the measurement vector $\mathbf{y}$ and the forward model $\mathbf{F}$ would be the *exact solution*, i.e. the vector $\mathbf{x}$ satisfying

$$\mathbf{y} = \mathbf{F}(\mathbf{x}). \tag{1.1.2}$$

Unfortunately, as will be seen below, such a vector $\mathbf{x}$ is unlikely to exist for general inverse problems. An important point to note is that the forward and the inverse problem are generally not symmetric. The forward problem is a process, which allows a unique prediction $\mathbf{F}(\mathbf{x})$ (up to some prediction uncertainty) of its outcome based only on the parameter vector $\mathbf{x}$. Moreover, a small deviation from $\mathbf{x}$ will result in a small deviation in the prediction $\mathbf{F}(\mathbf{x})$. A problem that satisfies these three properties, namely the *existence*, *uniqueness* and *continuity* of the solution $\mathbf{F}(\mathbf{x})$ in $\mathbf{x}$ is called a *well posed* problem (Hadamard, 1907). The inverse problem on the other hand is generally *ill posed*. This means that for the inverse problem any of the following conditions are met:

An exact solution to the problem does not exist. Due statistical errors in the measurement process and systematic errors in the forward model, a state vector $\mathbf{x}$ satisfying $\mathbf{F}(\mathbf{x}) = \mathbf{y}$ may not exist.

The solution is not unique. Even in the absence of systematic or statistic errors, there may be two state vectors $\mathbf{x}_1, \mathbf{x}_2$ that yield the same observed measurement $\mathbf{y}$. Without additional information, it is thus impossible to know which of the two vectors to choose.

The solution is *ill-conditioned*. An inverse problem is ill-conditioned if a small deviation in the measurement vector $\mathbf{y}$ may lead to a completely different solution $\mathbf{x}$.

Because of this, one can generally not expect to find an exact solution to a given inverse problem. Nonetheless, the vector $\mathbf{y}$ contains information about the state $\mathbf{x}$ of system. In order to extract this information from the measurement $\mathbf{y}$, it is thus necessary to reconsider the definition of a solution of the inverse problem. A very general and powerful formulation of inverse problems can be obtained from a Bayesian approach to the problem.

1.1.3 The Bayesian Formulation

The fundamental idea of the Bayesian approach to inverse problems is to formulate the problem in terms of available knowledge about the state of the system, rather than trying to find an exact solution for a particular measurement. A general way to represent knowledge about a system is by specifying a probability distribution $P(\mathbf{x})$ over the state space. In the Bayesian framework, there are two types knowledge about the state $\mathbf{x}$: The *a priori* and the *a posteriori* knowledge. The a priori knowledge is any knowledge about the state of the system that is available before the measurement. It is given by the *a priori probability* $P(\mathbf{x})$, also called the *prior*. The a posteriori knowledge is the combination of a priori knowledge and information gained through the measurement. It is given by the conditional probability distribution $P(\mathbf{x}|\mathbf{y})$ over the the state space for the given measurement vector $\mathbf{y}$. This distribution is called the *a posteriori distribution* or *posterior*.

In the Bayesian formulation the *solution of the inverse problem* is just the knowledge about the state $\mathbf{x}$ after the measurement, i.e. the posterior distribution $P(\mathbf{x}|\mathbf{y})$. Using the definition of the conditional probability $P(\mathbf{x}|\mathbf{y})$ of $\mathbf{x}$ given $\mathbf{y}$

$$P(\mathbf{x}|\mathbf{y}) = \frac{P(\mathbf{x},\mathbf{y})}{P(\mathbf{y})}, \tag{1.1.3}$$

the posterior distribution of $\mathbf{x}$ can be obtained from

$$P(\mathbf{x}|\mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{x})P(\mathbf{x})}{P(\mathbf{y})} \tag{1.1.4}$$

$$= \frac{P(\mathbf{y}|\mathbf{x})P(\mathbf{x})}{\int_{\mathbf{x}} P(\mathbf{x},\mathbf{y})\, d\mathbf{x}}. \tag{1.1.5}$$

This result is known as *Bayes' theorem*. Equation (1.1.4) is the general Bayesian solution of the inverse problem. The computation of a concrete solution of the inverse problem requires the definition of a suitable prior distribution $P(\mathbf{x})$ as well as a conditional probability $P(\mathbf{y}|\mathbf{x})$. The approach taken here is to assume a *Gaussian prior* on the state space as well as *Gaussian-distributed errors* in measurement space. That means the prior distribution is assumed to be of the form

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|\mathbf{S}_a|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mathbf{x}_a)^T\mathbf{S}_a^{-1}(\mathbf{x}-\mathbf{x}_a)\right\} \tag{1.1.6}$$

for given *a priori state* $\mathbf{x}_a$ and *a priori covariance matrix* $\mathbf{S}_a$. For the conditional probability $P(\mathbf{y}|\mathbf{x})$, it is assumed that the forward model $\mathbf{F}(\mathbf{x})$ is a physically

exact model of the measurement process and that the only error entering the measurement is multi-variate Gaussian- distributed zero-mean noise:

$$P(\mathbf{y}|\mathbf{x}) = \frac{1}{(2\pi)^{\frac{m}{2}}|\mathbf{S}_\epsilon|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(\mathbf{y}-\mathbf{F}(\mathbf{x}))^T\mathbf{S}_\epsilon^{-1}(\mathbf{y}-\mathbf{F}(\mathbf{x}))\right\} \tag{1.1.7}$$

Inserting the formulas for the prior and conditional distribution into (1.1.4) gives, up to a normalization factor, the following solution of the inverse problem:

$$P(\mathbf{x}\,|\,\mathbf{y}) \propto \exp\left\{-\frac{1}{2}\left((\mathbf{y}-\mathbf{F}(\mathbf{x}))^T\mathbf{S}_\epsilon^{-1}(\mathbf{y}-\mathbf{F}(\mathbf{x})) + (\mathbf{x}-\mathbf{x}_a)^T\mathbf{S}_a^{-1}(\mathbf{x}-\mathbf{x}_a)\right)\right\} \tag{1.1.8}$$

Even though this is the general solution to the problem, for further data analysis it is desirable to represent the solution using a single state vector $\mathbf{x}$ and a suitable measure of uncertainty. A common way of doing this is by choosing the state $\mathbf{x}$ that maximizes the posterior probability $P(\mathbf{x}\,|\,\mathbf{y})$. This technique is known as the *Optimal Estimation Method (OEM)* as described in Rodgers (2000) or alternatively the *Maximum A Posteriori (MAP) Method* as described in Tarantola (2005).

Consider first the case of a linear forward model:

$$\mathbf{F}(\mathbf{x}) = \mathbf{K}\mathbf{x} \tag{1.1.9}$$

For a linear forward model, the a posteriori distribution (1.1.8) is just a Gaussian distribution with mean and covariance given by:

$$\mathbf{x}_\mu = \mathbf{x}_a - \left(\mathbf{K}^T\mathbf{S}_\epsilon^{-1}\mathbf{K} + \mathbf{S}_a^{-1}\right)^{-1}\mathbf{K}^T\mathbf{S}_\epsilon^{-1}(\mathbf{F}(\mathbf{x}_a)-\mathbf{y}) \tag{1.1.10}$$

$$\mathbf{S} = \left(\mathbf{K}^T\mathbf{S}_\epsilon^{-1}\mathbf{K} + \mathbf{S}_a^{-1}\right)^{-1} \tag{1.1.11}$$

Due to the symmetry of the Gaussian distribution, the mean $\mathbf{x}_\mu$ is at the same time the maximum of the posterior distribution and thus (1.1.10) is also the MAP estimator of the linear inverse problem.

Unfortunately, for a general forward model no such closed form of the maximum of the posterior distribution is available. In this case the MAP estimator can still be obtained by numerically minimizing the negative log-likelihood

$$-\mathcal{L}(\mathbf{x}) = \frac{1}{2}\left((\mathbf{F}(\mathbf{x})-\mathbf{y})\mathbf{S}_\epsilon^{-1}(\mathbf{F}(\mathbf{x})-\mathbf{y}) + (\mathbf{x}-\mathbf{x}_a)\mathbf{S}_a^{-1}(\mathbf{x}-\mathbf{x}_a)\right), \tag{1.1.12}$$

where an additive constant arising from the normalization factors of the distribution has been ignored.

In order to obtain an uncertainty measure for the MAP estimator $\hat{\mathbf{x}}$ obtained from minimizing (1.1.12) one generally assumes that the forward model $\mathbf{F}$ is approximately linear around the maximum a posteriori state $\hat{\mathbf{x}}$. In this case the posterior distribution in the vicinity of $\hat{\mathbf{x}}$ will be approximately Gaussian and its covariance can be obtained from (1.1.11) with $\mathbf{K} = \mathbf{K}_{\hat{\mathbf{x}}}$, the Jacobian of the forward model evaluated at the maximum a posteriori state $\hat{\mathbf{x}}$:

$$(\mathbf{K}_{\hat{\mathbf{x}}})_{i,j} = \frac{d(F_i(\hat{\mathbf{x}} + \mathbf{x}))}{dx_j} \tag{1.1.13}$$

Note that formulas (1.1.10) and (1.1.11) for a linear forward model arise as special cases from the more general formulation of the minimization of the negative log-likelihood (1.1.12).

1.1.4 Error Analysis

Equally as important as a method to compute the solution of an inverse problem are diagnostic quantities that characterize its accuracy and precision. This requires a more detailed analysis of the sources of error in the inverse problem formulation, which is provided in this section. The discussion is based on the presentation in Chapter 4 in Rodgers (2000).

The computation of the solution of the inverse problem for a given measurement $\mathbf{y}$ may be viewed as the application of a retrieval operator $\mathbf{R}$ to $\mathbf{y}$:

$$\hat{\mathbf{x}} = \mathbf{R}(\mathbf{y}, \mathbf{x}_a) \tag{1.1.14}$$

The retrieval operator $\mathbf{R}$ represents the method used to compute the inverse problem solution $\hat{\mathbf{x}}$. Except for the measurement vector $\mathbf{y}$, the result of this method also depends on the a priori state $\mathbf{x}_a$. The measurement vector $\mathbf{y}$ is the result of a physical process that is described by the forward model $\mathbf{F}$. In a real world application, this model will be inexact and describe the physical processes involved only up to a modeling error $\epsilon_{\mathbf{F}}$. In addition to that, the measurement process is subject to noise which requires the incorporation of an additional error term ϵ_n:

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) + \epsilon_{\mathbf{F}} + \epsilon_n \tag{1.1.15}$$

Let $\mathbf{x}_t$ designate the true state of the system. Linearizing (1.1.15) about the a priori state $\mathbf{x}_a$ yields:

$$\mathbf{y} = \mathbf{F}(\mathbf{x}_a) + \mathbf{K}_{\mathbf{x}_a}(\mathbf{x}_t - \mathbf{x}_a) + \epsilon_{\mathbf{F}} + \epsilon_n \tag{1.1.16}$$

where $\mathbf{K}_{\mathbf{x}_a}$ is the Jacobian of the forward model $\mathbf{F}$ evaluated at $\mathbf{x}_a$. Plugging this into (1.1.14) and linearizing with respect to $\mathbf{y}$ about $\mathbf{F}(\mathbf{x}_a)$ gives

$$\hat{\mathbf{x}} = \mathbf{R}(\mathbf{F}(\mathbf{x}_a), \mathbf{x}_a) + \mathbf{G}\mathbf{K}_{\mathbf{x}_a}(\mathbf{x} - \mathbf{x}_a) + \mathbf{G}(\epsilon_{\mathbf{F}} + \epsilon_n) \tag{1.1.17}$$

where $\mathbf{G}$ is the Jacobian of the retrieval operator $\mathbf{R}$ evaluated at $\mathbf{F}(\mathbf{x}_a)$. Next, consider the difference $\mathbf{x}_t - \mathbf{x}_a$ of the true state $\mathbf{x}_t$ and the a priori state $\mathbf{x}_a$. This term represents the ideal knowledge that can be gained from the measurements. Inserting this into the equation above leads to an interesting characterization of the retrieval process:

$$\hat{\mathbf{x}} - \mathbf{x}_a = \underbrace{\mathbf{R}(\mathbf{F}(\mathbf{x}_a), \mathbf{x}_a) - \mathbf{x}_a}_{\text{Bias}} + \underbrace{\mathbf{A}(\mathbf{x}_t - \mathbf{x}_a)}_{\text{Smoothing}} + \underbrace{\mathbf{G}(\epsilon_{\mathbf{F}} + \epsilon_n)}_{\text{Measurement Error}} \tag{1.1.18}$$

The result of the measurement process may thus be viewed as the sum of three separate terms: A bias term $\mathbf{R}(\mathbf{F}(\mathbf{x}_a), \mathbf{x}_a) - \mathbf{x}_a$, a smoothing term $\mathbf{A}(\mathbf{x}_t - \mathbf{x}_a)$ and the measurement error term $\mathbf{G}(\epsilon_{\mathbf{F}} + \epsilon_n)$.

The bias term represents the bias introduced by the retrieval of a simulated measurement of the system in a priori state. For all retrieval methods treated here this term will be zero, so it can be ignored.

Equation (1.1.18) implies that the information retrieved from a measurement is the sum of a smoothed version of the *true information* $\mathbf{A}(\mathbf{x}_t - \mathbf{x}_a)$ and an error term $\mathbf{G}(\epsilon_{\mathbf{F}} + \epsilon_n)$. The matrix $\mathbf{A} = \mathbf{G}\mathbf{K}$ is called *averaging kernel matrix*. For an ideal measurement, the averaging kernel matrix is the identity matrix and the error term in (1.1.18) equal to zero. Since in general the averaging kernel matrix $\mathbf{A}$ will have non-zero off-diagonal elements, it has a smoothing effect on the information about the true state of the system $\mathbf{x}_t - \mathbf{x}_a$ on the information $\hat{\mathbf{x}} - \mathbf{x}_a$ obtained from the measurement. The columns $\mathbf{a}_i$ of $\mathbf{A}$ represent the response in the retrieved vector to a unit perturbation in the true state $\mathbf{x}_t$ of the system. For a good measurement the columns should be peaked around the index of the corresponding component of the state vector and the sum of all elements close to 1.

The Jacobian of the retrieval operator $\mathbf{G}$, also called the *gain matrix*, determines the effect of measurement noise and modeling errors on the inverse problem solution. A detailed analysis of the contribution of this error source to the retrieved state $\hat{\mathbf{x}}$ requires knowledge of the errors $\epsilon_{\mathbf{F}}, \epsilon_n$, which depend on the specific problem setting.

For very large problems, the computation of $\mathbf{A}$ or $\mathbf{G}$ may computationally not be feasible or desirable. In such cases the error analysis can be performed by

examining specific columns of the averaging kernel and gain matrices corresponding to certain elements of the retrieval vector.

1.2 Numerical Optimization

As presented in Section 1.1.3, the solution of an inverse problem can be reduced to minimizing a cost function of the form

$$J(\mathbf{x}) = \frac{1}{2}\left((\mathbf{F}(\mathbf{x}) - \mathbf{y})\mathbf{S}_\epsilon^{-1}(\mathbf{F}(\mathbf{x}) - \mathbf{y}) + (\mathbf{x} - \mathbf{x}_a)\mathbf{S}_a^{-1}(\mathbf{x} - \mathbf{x}_a)\right). \tag{1.2.1}$$

In general, no closed-form solution of this minimization problem exists and therefore numerical methods are required to find a minimum of (1.2.1).

The numerical optimization algorithms discussed in this section are all iterative methods that, starting from a start vector $\mathbf{x}_0$, generate a sequence of iterates $\{\mathbf{x}\}_{i=0}^{\infty}$ that under certain conditions are guaranteed to converge to a minimum of the cost function. At each given step i, such a numerical optimization method uses local properties of the cost function at the current iterate $\mathbf{x}_i$ and possibly also a limited number of additional points to determine the next iterate $\mathbf{x}_{i+1}$. Important local properties of the cost function are its value $J(\mathbf{x})$, the Jacobian $\nabla_{\mathbf{x}} J$ and the Hessian $\nabla_{\mathbf{x}}^2 J$. For the cost function given in (1.2.1), these depend directly on the forward model $\mathbf{F}(\mathbf{x})$. In what follows, it will be assumed that besides evaluating the forward model for a given state vector $\mathbf{x}$ also its Jacobian $\mathbf{K}_{\mathbf{x}}$ at $\mathbf{x}$ can be computed efficiently. Furthermore it is assumed that the Hessian of the forward model can be neglected. For a sufficiently linear forward model, this will be a reasonable assumption. The Gradient and Hessian of the cost function (1.2.1) are given by

$$\nabla_{\mathbf{x}} J(\mathbf{x}) = \mathbf{K}_{\mathbf{x}}^T \mathbf{S}_\epsilon^{-1}(\mathbf{F}(\mathbf{x}) - \mathbf{y}) + \mathbf{S}_a^{-1}(\mathbf{x} - \mathbf{x}_a) \tag{1.2.2}$$

$$\nabla_{\mathbf{x}}^2 J = \mathbf{K}_{\mathbf{x}}^T \mathbf{S}_\epsilon^{-1} \mathbf{K}_{\mathbf{x}} + \mathbf{S}_a^{-1} \tag{1.2.3}$$

1.2.1 Gauss-Newton Method

Newton-methods are a family of optimization algorithms that choose the direction for the current minimization step $\Delta\mathbf{x}$ by solving the linear system

$$\nabla^2_{\mathbf{x}} J\, \Delta\mathbf{x} = -\nabla_{\mathbf{x}} J. \tag{1.2.4}$$

The advantage of Newton methods over gradient-based methods is that they incorporate information about the curvature of the function J in form of the Hessian $\nabla^2_{\mathbf{x}} J$ into the choice of $\Delta\mathbf{x}$. This allows them to achieve superlinear convergence (cf. Chapter 11 in Nocedal and Wright (2006)) sufficiently close to a solution of the minimization problem, which considerably reduces the time required to find a minimum of J.

The Gauss-Newton method is a modified Newton method for least-squares problems. Least-sqares problems give rise to cost functions of the form

$$J(\mathbf{x}) = (f(\mathbf{x}) - \mathbf{y})^T (f(\mathbf{x}) - \mathbf{y}) \tag{1.2.5}$$

for some vector-valued function $f(\mathbf{x})$. For a cost function of this form the direction of the Gauss-Newton step is computed using

$$(\boldsymbol{\nabla}_{\mathbf{x}} f)^T \boldsymbol{\nabla}_{\mathbf{x}} f \boldsymbol{\Delta}\mathbf{x} = -\boldsymbol{\nabla}_{\mathbf{x}} f (f(\mathbf{x}) - \mathbf{y}). \tag{1.2.6}$$

The Hessian of the cost function is here approximated by the matrix product $(\boldsymbol{\nabla}_{\mathbf{x}} f)^T \boldsymbol{\nabla}_{\mathbf{x}} f$. This approximation is valid when the partial second derivatives of f are negligible. While the above method applies to least squares problems, it can easily be generalized to the cost function (1.2.1) by replacing the approximate Hessian on the left hand side of (1.2.6) by (1.2.3) and the gradient on the right hand side by (1.2.2). The resulting optimization method is given as pseudocode in Algorithm 1 in Appendix A. In general, equation (1.2.4) is only used to determine the direction of the current step $\Delta\mathbf{x}$ and further criteria may be applied to determine the length of the minimization step. The invlib implementation currently uses $\Delta\mathbf{x}$ directly without optimizing the step length. The final form of the Gauss-Newton method for the Bayesian formulation of inverse problems presented above can be written as follows:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left(\mathbf{K}_{\mathbf{x}}^T \mathbf{S}_\epsilon^{-1} \mathbf{K}_{\mathbf{x}} + \mathbf{S}_a^{-1}\right)^{-1} \left(\mathbf{K}_{\mathbf{x}}^T \mathbf{S}_\epsilon^{-1} (\mathbf{F}(\mathbf{x}) - \mathbf{y}) + \mathbf{S}_a^{-1} (\mathbf{x} - \mathbf{x}_a)\right) \tag{1.2.7}$$

As shown in Rodgers (2000), using fundamental matrix algebra, (1.2.7) can also be written in two additional ways:

$$\mathbf{x}_{i+1} = \mathbf{x}_a - \left(\mathbf{S}_a^{-1} + \mathbf{K}_{\mathbf{x}_i}^T \mathbf{S}_\epsilon^{-1} \mathbf{K}_{\mathbf{x}_i}\right)^{-1} \left(\mathbf{K}_{\mathbf{x}_i} \mathbf{S}_\epsilon^{-1} (\mathbf{y} - \mathbf{F}(\mathbf{x}_i)) + \mathbf{S}_a^{-1} (\mathbf{x}_i - \mathbf{x}_a)\right) \quad (1.2.8)$$

$$\mathbf{x}_{i+1} = \mathbf{x}_a - \mathbf{S}_a \mathbf{K}_{\mathbf{x}_i}^T \left(\mathbf{S}_\epsilon + \mathbf{K}_{\mathbf{x}_i} \mathbf{S}_a \mathbf{K}_{\mathbf{x}_i}^T\right)^{-1} (\mathbf{y} - \mathbf{F}(\mathbf{x}_i) + \mathbf{K}_{\mathbf{x}_i} (\mathbf{x}_i - \mathbf{x}_a)) \quad (1.2.9)$$

Equations (1.2.7), (1.2.8), (1.2.9) will be referred to as the *standard*, *n-* and *m-form*, respectively. The main difference between the standard and the n-form on the one hand and the m-form on the other hand is the size of the linear system that needs to be solved in each minimization step. For the standard and n-form this linear system has n equations and n unknowns, while for the m-form it has m equations and m unknowns. For problems where the values of m and n differ strongly, it may thus be computationally advantageous to choose the formulation accordingly.

1.2.2 Levenberg-Marquardt Method

The Levenberg-Marquardt method may be viewed as a trust region version of the Gauss-Newton method described above. This means that the next iterate $\mathbf{x}_{i+1}$ is required to lie within a certain distance of the current iterate $\mathbf{x}_i$. As shown in section 10.3 in Nocedal and Wright (2006), the constrained optimization problem of finding the step $\Delta\mathbf{x}$ that leads to the largest reduction in the norm of the gradient in a given region around the current iterate $\mathbf{x}_i$

$$\underset{\Delta\mathbf{x}}{\text{minimize}} \qquad ||\nabla_{\mathbf{x}_i} J(\mathbf{x}_i) + \nabla^2_{\mathbf{x}_i} J \, \Delta\mathbf{x}||_2^2 \quad (1.2.10)$$

$$\text{subject to} \qquad ||\Delta\mathbf{x}|| < d \quad (1.2.11)$$

is solved by the solution of the linear system of equations

$$-\left(\nabla^2_{\mathbf{x}} J + \lambda \mathbf{D}\right) \Delta\mathbf{x} = \nabla_{\mathbf{x}} J \quad (1.2.12)$$

for some scalar λ that satisfies

$$\lambda(d - ||\Delta x||) = 0. \quad (1.2.13)$$

The matrix $\mathbf{D}$ must be positive definite and defines the distance measure $||\mathbf{x}||^2 = x^T \mathbf{D} x$ that is used to constrain the size of the trust region.

In practice, a heuristic is used to set the value of λ and the step $\Delta\mathbf{x}$ for the current iteration is determined by solving (1.2.12). While this provides no direct

way to control the size of the trust region, Equation (1.2.13) indicates that if λ is non-zero the size of the trust region d is equal to the length of the current step $\Delta\mathbf{x}$. Since increasing λ will decrease the length of $\Delta\mathbf{x}$, this provides an indirect way to influence the size of the trust region.

The value of λ is adapted in each iteration depending on whether the predicted value obtained from a second order approximation to the cost function agrees with the actual cost function value at the new iterate $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}$. The degree to which the second order model agrees with the true cost function can be quantified by computing the ratio of the actual and the expected cost function reduction:

$$c = \frac{J(\mathbf{x}_{i+1}) - J(\mathbf{x}_i)}{\frac{1}{2}\Delta\mathbf{x}^T \nabla_{\mathbf{x}_i} J} \tag{1.2.14}$$

The heuristic used to set the value of λ is the following: If the value of c is larger than 0.75, the quadratic model fits the true cost function well and considerable progress can be made by incorporating curvature information contained in the Hessian into the computation of the step direction. The iterate is thus accepted and λ decreased, i.e. the trust region radius enlarged. If c is positive but less than 0.75, the computed step leads to a reduction of the cost function, but the quadratic model does not fit the data very well. In this case the iterate $\mathbf{x}_{i+1}$ is still accepted but λ is kept constant. If $c < 0.2$, λ is increased but the step still accepted. If c is negative, i.e. no reduction in the cost function is achieved, the step is not accepted and $\Delta\mathbf{x}$ is recomputed with a higher value for λ. Note that for large enough λ, solving (1.2.12) will eventually lead to a reduction in the cost function since $\nabla^2_{\mathbf{x}_i} J + \lambda\mathbf{D}$ will be positive definite for sufficiently large λ and $\Delta\mathbf{x}$ thus point into a descent direction.

The increasing and decreasing of λ is performed by scaling the value with predefined factors. In addition to that, a minimum and maximum value for λ are defined. If λ falls below the minimum value, it is set to zero in order to benefit from the good convergence properties of the Gauss-Newton method. The maximum value for λ, if chosen sufficiently large, ensures that the trust region size is not decreased indefinitely. For completeness, the above method is given as pseudocode in Algorithm 2 in Appendix B.

1.2.3 Solving Linear Systems

Both of the iterative methods described above require the solution of a linear system of equations of the form

$$\underbrace{\left(\mathbf{K}_{\mathbf{x}}^{T}\mathbf{S}_{\epsilon}\mathbf{K}_{\mathbf{x}}+\mathbf{S}_{a}\right)}_{\mathbf{M}}\Delta\mathbf{x}=-\nabla_{\mathbf{x}}J \tag{1.2.15}$$

or

$$\underbrace{\left(\mathbf{K}_{\mathbf{x}}^{T}\mathbf{S}_{\epsilon}\mathbf{K}_{\mathbf{x}}+(1+\lambda)\mathbf{S}_{a}\right)}_{\mathbf{M}}\Delta\mathbf{x}=-\nabla_{\mathbf{x}}J \tag{1.2.16}$$

to compute the step $\Delta\mathbf{x}$. This linear system is also referred to as the *linear subproblem* of the minimization method. To simplify notation, the shorthand $\mathbf{M}$ will be used to refer to the matrix defining the respective linear system. From a computational point of view, there are two fundamentally different ways of solving a linear system: Using a *direct solver* based on a decomposition technique that computes an exact solution of the linear system or using an *indirect solver* that iteratively computes an approximate solution of the linear system.

The most common decomposition technique for general dense matrices is the LU-decomposition. For symmetric, positive definite matrices also the Cholesky decomposition is available, which has the advantage of improved performance of about a factor of two (Ungermann, 2011). Solving dense linear systems is a very common problem in computational science and numerical implementations are readily available. A detailed discussion of these methods is therefore omitted here and the reader is referred to one of the many references in literature, such as for example Chapter 2 in Press et al. (2007). The disadvantage of direct solvers is that they require the explicit computation of the matrix $\mathbf{M}$ representing the linear system. For the linear system arising in each minimization step during the computation of the solution of an inverse problem, this involves two matrix multiplications as well as storing at least two additional matrices. For large m and n the computational time required to set up the linear system may even exceed the time that is required to solve it. Furthermore, the memory required to store the system may become prohibitively large and limit the size of the inverse problems that can be solved.

An alternative to the decomposition techniques discussed above is the conjugate gradient (CG) method, which iteratively computes a solution of the linear system. Its advantage is that the method requires only the computation of matrix-vector products of the matrix $\mathbf{M}$ representing the linear system and an arbitrary vector $\mathbf{x}$. While the computational performance of the conjugate gradient method for small problems is often inferior to that of decomposition techniques, the method achieves comparable performance for medium- and large-sized problems. Furthermore,

since the method does not require explicitly computing the matrix $\mathbf{M}$, its memory requirements are significantly reduced. This makes it possible to solve even very large problems which cannot be handled by decomposition techniques due to memory limitations. On the other hand, since it is an iterative method, the CG method requires specification of suitable starting vector and convergence criterion. Since the derivation of the method is quite involved and does not provide particular insight into its workings, it is omitted here. A formulation conjugate gradient method in pseudo code is given in Algorithm 3 in Appendix A. A detailed discussion of the algorithm and its properties can be found in Nocedal and Wright (2006).

1.3 Performance Considerations

For large problems with high-dimensional state and measurement spaces, computing the solution of an inverse problem quickly becomes computationally demanding both with respect to computation time as well as memory required to hold the data in main memory. In addition to that, non-linear problems require the repeated evaluation of the forward model and computation of its gradients. In many cases, this is even more expensive than the matrix operations involved in the computation of the MAP estimator. Since the dimensions of state and measurement space as well as the complexity of the forward model and the resulting structures of the Jacobian and correlation matrices vary from problem to problem, it is not possible to specify a single, optimal way of solving an inverse problem. Nevertheless, there are some general performance characteristics of the computational methods involved that should be considered when choosing a concrete solution scheme for a given inverse problem. In order to illustrate these characteristics, a number of numerical experiments have been performed, which will be presented in this section.

The timings presented in this section have been obtained on a laptop machine equipped with an Intel®Core™i5-3320M CPU and 8 GB of main memory. While this kind of architecture is unlikely to be used for large retrieval problems, it should be sufficient to demonstrate the general scaling properties of the numerical routines considered. For dense matrix and vector arithmetic the standard Ubuntu implementation of the BLAS/LAPACK (Netlib, 2016) library is used. Sparse arithmetic is implemented using the Eigen 3 (Eigen, 2016) library.

To illustrate the general performance characteristics of computing the MAP estimator of an inverse problem, a linear toy problem with randomly chosen Jacobian and Covariance Matrices is considered. Since solving a linear problem is equivalent

to computing one iteration of a non-linear problem, all considerations for the linear problem extend to the case of a non-linear problem. Care has been taken, that the precision matrices involved are positive definite. In order to illustrate the advantages of using sparse matrix algebra, the Jacobian as well as the covariance matrices were chosen to be symmetric, diagonal band matrices with a bandwidth of 10. While this is a somewhat idealized assumption, it will serve to demonstrate the potential performance gains that can be achieved through the use of sparse arithmetic.

1.3.1 Critical Operations

The arithmetic operations that are critical for the performance of the computation of a solution of an inverse problem are matrix-matrix and matrix-vector multiplication as well as solving linear systems of equations. While additional operations such as matrix and vector addition or the computation of the vector norm need to be performed, their contribution to the total time required to compute the solution of an inverse problem can generally be neglected.

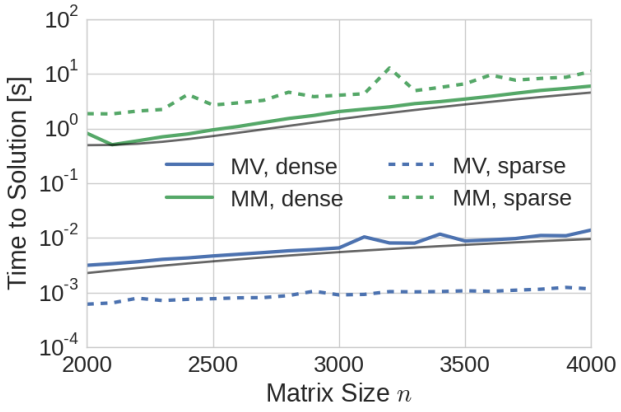


Figure 1.3.1: Time to solution for matrix-matrix (MM) and matrix-vector (MV) multiplication using dense and sparse matrix representations. The gray lines mark a polynomial fit to the computation time for the dense operations with degree 3 for matrix-matrix multiplication and 2 for matrix-vector multiplication.

The scaling of matrix-matrix and matrix-vector multiplication for a square matrix of size $n \times n$ is displayed in Figure 1.3.1. For the matrix-matrix multiplication (blue) and the matrix-vector multiplication (green) a third-degree and a second-degree

polynomial have been fitted to the computational times and are plotted in gray. Since the purpose of the fit is to illustrate the scaling of the computation time, they have both been offset slightly so that they are not covered by the curve they have been fitted to. For the product of a $m \times k$ matrix with a $k \times n$ matrix using a standard BLAS implementation, the computation time can be expected to scale linearly in the dimensions m, k and n. The multiplication of a $m \times n$ matrix and a n-dimensional vector scales linearly in the dimensions m and n. For an inverse problem with n-dimensional state and m dimensional measurement space, the involved matrix products are thus expected to scale proportionally to n^2m or m^2n, depending on the chosen formulation. Due to their complexity, matrix products are critical for the performance of the solution method and should be avoided if the product is not required explicitly. Also displayed in Figure 1.3.1 are the timings for the corresponding sparse operations. While sparse matrix-matrix multiplication performs worse than the dense version, the improved scaling of the sparse matrix-vector product is clearly visible from the graph. Another important advantage of using sparse arithmetic is that it can significantly reduce the memory required to store the matrices, which otherwise would scale linearly in both dimensions of the matrix.

The time required for the computation of a solution of a linear system is displayed in Figure 1.3.2. The matrix representing the linear system was chosen to be a random, symmetric, positive-definite, banded matrix. For a direct solver the expected scaling of the solution of the linear system is quadratic in the number of unknowns of the linear system. This is confirmed by the quadratic fit to the time to solution of the LU decomposition method, which is displayed in gray. Again, the fits have been offset slightly to avoid them being covered by the curve they have been fitted to. For this benchmark setting, solving the linear system using the CG method with sparse arithmetic performs significantly better than LU decomposition. Also given in the plot is the computational time required to fully invert the linear system. Inverting a linear system is very expensive and should be avoided. Therefore the mathematical expression $\mathbf{A}^{-1}\mathbf{x}$ should always be computed by solving the linear system given by the matrix $\mathbf{A}$ and the right hand side $\mathbf{x}$, instead of inverting the matrix and then multiplying the inverse $\mathbf{A}^{-1}$ by $\mathbf{x}$.

1.3.2 The Linear Test Problem

In order to obtain an impression of the contribution of the different operations to the overall performance, profiles of the solution of a randomly generated linear test problem have been generated. Three different problem sizes have been analyzed:

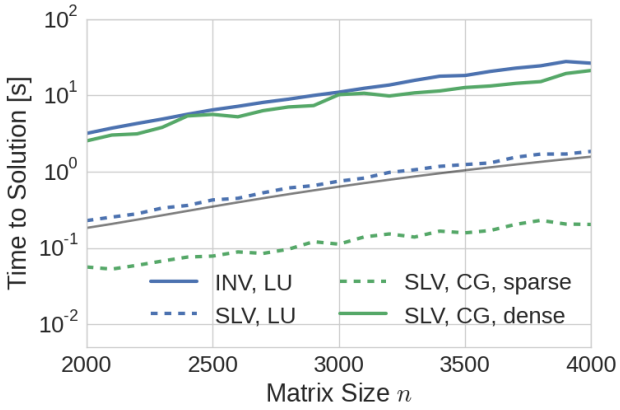


Figure 1.3.2: Time to solution for computing the inverse (INV) of a matrix and computing a single solution (SLV) of the corresponding linear system. Three different methods have been used for computing a single solution of the system: LU decomposition (blue, dashed), conjugate gradient using dense arithmetic (green, solid), conjugate gradient using sparse arithmetic (green, dashed).

$m = n = 10^3$, $m = 10^3$ and $n = 10^4$, as well as $n = 10^4$ and $m = 10^4$. The standard formulation (1.2.7) is used to compute the inverse problem solution. For each problem size, the inverse problem is solved using three different combinations of matrix representations and methods for solving the linear system:

1. Dense arithmetic, direct solution of the linear system

2. Dense arithmetic, conjugate gradient method

3. Sparse arithmetic, conjugate gradient method

For each of these configurations, the total time to solution as well as time spent computing matrix-matrix, matrix-vector products and solving the linear system has been recorded. The results displayed in Figure 1.3.3 support the claim made above, that the total time to solution for solving the linear inverse problem is mainly determined by the computation of matrix-matrix products, matrix-vector products and solution of the linear system. Furthermore, for the dense case it can be seen that when n and m are of approximately the same size, the time required to perform matrix-matrix multiplication, i.e. to set up the linear system, actually takes more time then solving the system itself. The standard formulation of the OEM method that was used for the benchmark requires the solution of a linear

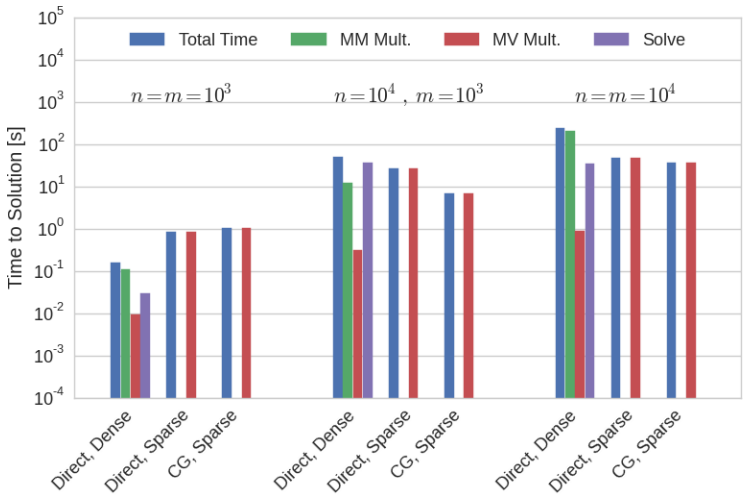


Figure 1.3.3: Linear OEM Benchmark. Three problem sizes are considered: $n = m = 10^3$, $n = 10^4$ and $m = 10^3$, $n = m = 10^4$. For each total time to solution as well as time spent in the critical arithmetic routines for matrix-matrix multiplication (MM Mult.), matrix-vector multiplication (MV Mult.) and solution of the linear system (Solve) are displayed.

system with n unknowns. For the case with $n = 10^4, m = 10^3$ it would of course be advantageous to use the m-form formulation of the OEM, which only requires the solution of a linear system of size m. Unfortunately, the applicability of the m-form is limited to some extent by the fact that it can not be used with the Levenberg-Marquardt method and the requirement of both, the covariance matrices and their inverses, for the solution of the inverse problem. Comparing the LU decomposition method to the conjugate gradient method, one can see that the conjugate gradient method provides superior performance only for larger problems. For small problems, using LU decomposition to solve the linear system is almost an order of magnitude faster.

1.3.3 Memory

Performing one step of an OEM computation using the standard formulation involves the matrices $\mathbf{K}_\mathbf{x}$, i.e. the Jacobian of the forward model evaluated at $\mathbf{x}$, and the inverses of the covariance matrices $\mathbf{S}_\epsilon^{-1}$, $\mathbf{S}_a^{-1}$. These matrices are of size $m \times n$, $m \times m$ and $n \times n$, respectively. If decomposition techniques are used to solve the linear subproblem of the chosen optimization method, a number of

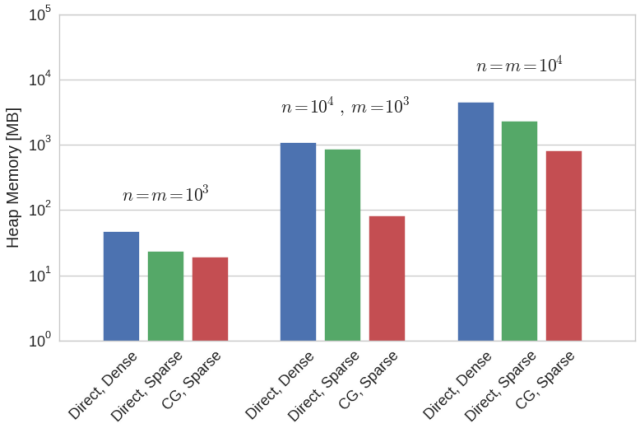


Figure 1.3.4: Peak heap memory allocated for the linear OEM Benchmark problem. As above, three problem sizes are considered: $n = m = 10^3$, $n = 10^4$ and $m = 10^3$, $n = m = 10^4$. For each solution configuration the peak value of allocated heap memory is given as determined using the Valgrind tool Massif.

additional, temporary matrices are required to set up, hold and solve the linear system. If these matrices are stored in dense representation, the memory required to hold these matrices in main memory quickly becomes too large to perform the computations on an ordinary desktop machine. For example, an inverse problem with $m = n = 10^5$ requires roughly 80GB of memory just to hold the dense Jacobian in memory. Fortunately, for large problems, the covariance matrices $\mathbf{S}_a$ and $\mathbf{S}_\epsilon$, respectively the precision matrices $\mathbf{S}_a^{-1}$ and $\mathbf{S}_\epsilon^{-1}$, as well as the Jacobian $\mathbf{K}_\mathbf{x}$ tend (or can be chosen in a way) to be sparse. Using sparse storage schemes for the matrices involved can greatly reduce the memory required to store the matrices. To illustrate the memory requirements of the different configurations, heap memory profiles of the linear benchmark case described above have been recorded. To this end, the Valgrind tool Massif (Massif, 2016) has been used to measure the allocated heap storage during execution of the program. The resulting peak values for the three solution schemes described above are displayed in Figure 1.3.4. The plot clearly displays the large memory requirements of the computations when a dense matrix representation is used and especially when the linear system is solved directly. Memory-wise the best performance is achieved by using the conjugate gradient method together with sparse matrix representation.

Chapter 2

The invlib Library

In this chapter the general design and implementation of the invlib library is presented, which has been developed as a part of this Master's thesis project. After a general description of the modeling of inverse problems and the structure of the library, its usage is illustrated by using invlib to retrieve simulated measurements from the MATS (Gumbel et al., 2014) satellite mission.

The invlib library provides functions to solve inverse problems based on the minimization of a cost function of the form

$$J(\mathbf{x}) = (\mathbf{y} - \mathbf{F}(\mathbf{x}))\, \mathbf{S}_\epsilon^{-1}\, (\mathbf{y} - \mathbf{F}(\mathbf{x})) + (\mathbf{x} - \mathbf{x}_a)\, \mathbf{S}_a^{-1}\, (\mathbf{x} - \mathbf{x}_a))\,. \tag{2.0.1}$$

Such problems arise from the Bayesian formulation of inverse problems, which has been presented in the previous section, as well as the more general framework of Tikhonov regularization (c.f. Chapter 5 in Aster et al. (2005)).

2.1 Design Goals

The invlib library was designed with two primary goals:

Generality: Just as the mathematical formulation of inverse problems, the library should be applicable to a wide range of problems. In addition to that, it should be extensible, allowing for simple adding and testing of new features or adapting the library to specific problems.

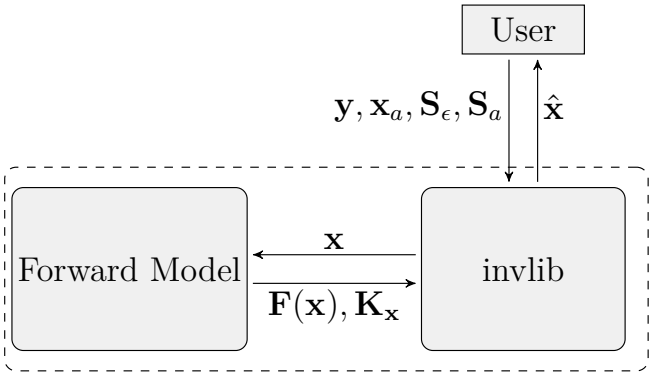


Figure 2.2.1: Data flow diagram for the solution of an inverse problem using invlib.

Performance: The library should be efficient both with respect to computation time as well as memory requirements. This is necessary for the handling of large retrieval problems.

In order to reach these goals invlib has been implemented in C++ , which provides both fined grained control over performance critical processes as well as powerful abstraction mechanisms required to handle the complexity of the problem. The code relies heavily on template programming as well as features introduced in the C++11 standard (ISO C++, 2012). Resulting from the underlying formulation of inverse problems, the invlib library has to interact with a forward model. Through the use of template programming, the library can be used on top of the foward model code and directly reuse its underlying datastructures. The invlib library therefore does not provide an implementation of data types for vectors and matrices, but relies on them being provided by the forward model or an external library.

2.2 General Usage

The data flow of the general usage of the invlib library to compute the solution of an inverse problem is illustrated in Figure 2.2.1. The forward problem is represented by a forward model type, which provides functions to evaluate the measurement vector $\mathbf{y} = \mathbf{F}(\mathbf{x})$ and compute its Jacobian $\mathbf{K}_\mathbf{x}$ for a given vector $\mathbf{x}$. In addition to that, the user is required to provide the a priori state vector $\mathbf{x}_a$, the covariance matrices $\mathbf{S}_\epsilon, \mathbf{S}_a$, directly or in the form of precision matrices $\mathbf{S}_\epsilon^{-1}, \mathbf{S}_a^{-1}$, as well as the measurement vector $\mathbf{y}$. The invlib library then computes the maximum a posteriori (MAP) state vector $\hat{\mathbf{x}}$ of the inverse problem.

2.2.1 User Interface

All invlib classes and functions live in the the `invlib` namespace. The main interface of the invlib library is provided by the `invlib::MAP` class template. The template parameters of the `MAP` class template specify the type of the forward model (`ForwardModel`), the types implementing the matrix and vector algebra represented by the matrix type to be used in the calculations (`MatrixType`), the type used to represent the a priori covariance matrix $\mathbf{S}_a$ (`SaType`), the type used to represent the measurement space covariance matrix $\mathbf{S}_\epsilon$ (`SeType`) and finally which formulation should be used.

```
template
<
typename ForwardModel,
typename MatrixType,
typename SaType  = MatrixType,
typename SeType  = MatrixType,
Formulation Form = Formulation::STANDARD
>
class MAP;
```

The class structure used to model the computation of MAP estimators for a given inverse problem is presented in Figure 2.2.2. The inverse problem is described by a forward model object, an a priori vector as well as a priori and measurement space covariance matrices. The corresponding objects have to be provided to the constructor of the given instantiated `MAP` template class:

```
MAP(ForwardModel        &F_,
    const VectorType    &xa_,
    const SaType        &Sa_,
    const SeType        &Se_ );
```

The actual computation is defined by which `Formulation` enum was used to instantiate the `MAP` class as well as which optimization method is used to minimize the cost function (2.0.1). The actual computation of the MAP estimator is executed by the `compute` member function of the `MAP` object. The optimization method that is used for the computation is defined by the `Minimizer` object that is passed to the `compute` function. In addition to the `Minimizer` type, the `compute` member function takes an additional `Log` template argument which is used to specify the way log output is printed to the user.

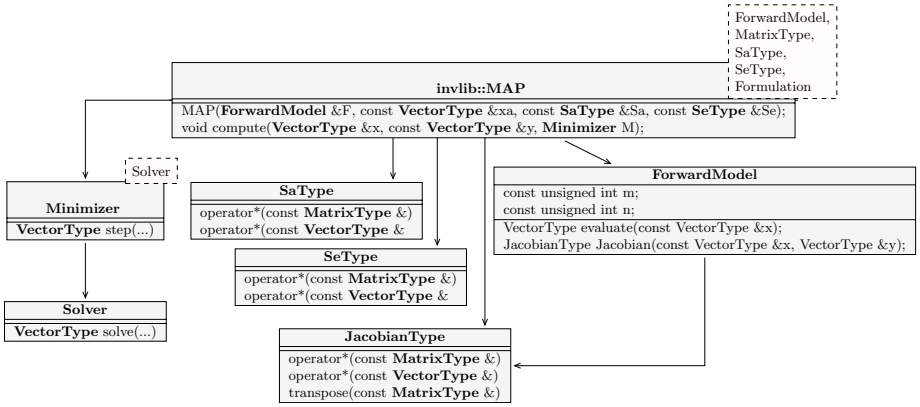


Figure 2.2.2: Class diagram displaying the abstraction levels used to model the solution of the inverse problem.

```
template<typename Minimizer, template <LogType> class Log = StandardLog>
void compute(VectorType         &x,
             const VectorType   &y,
             Minimizer           M,
             int verbosity = 0);
```

The setting up of a given inverse problem is performed in two steps. First, the `MAP` class template and all other required class templates for the forward model, optimizers, solvers and matrix types are instantiated. This specifies the solution scheme for the inverse problem, i.e. *which* calculations are performed to compute the MAP estimator. In the next step, concrete objects of the `MAP` type and the instantiated types for solver and optimization method are created and the computation parameters are set. This defines *how* the chosen computations are performed.

2.2.2 Matrix Algebra

The invlib library does not provide its own implementation of the linear algebra data types and routines required for solving an inverse problem but relies on them being provided either by the forward model or an interface to an external linear algebra library. The motivation for this design choice is to avoid type conversions from the forward model types to the invlib types and allow the use of matrix and vector representations that are specific to the concrete problem at hand.

In order to separate the mathematical formulation from implementation details

of the linear algebra types and routines, invlib provides abstract, generic matrix and vector types, that serve as wrapper for the user provided types. The resulting generic matrix algebra implements a number of high level optimizations such as avoiding matrix-matrix products and matrix inversions as well as minimizing the creation of temporary objects required to store intermediate results. The matrix algebra is implemented using mathematical operator notation (`*`, `+`, `-`) to make the code more readable and also easier to develop. As an example, take a product of two matrices **AB** multiplied from the right by a vector **x**:

$$\mathbf{y} = \mathbf{ABx} \tag{2.2.1}$$

Computationally this should be computed in two steps: First compute the vector $\mathbf{t} = \mathbf{Bx}$ and then $\mathbf{y} = \mathbf{At}$, which requires two matrix-vector multiplications and the allocation of one temporary vector. On the other hand, first computing the matrix product $\mathbf{T} = \mathbf{AB}$ and then $\mathbf{y} = \mathbf{Tx}$, requires the computation of one matrix-matrix product, one matrix-vector product and the allocation of storage to hold the intermediate result matrix **T**. Unfortunately, C++ operator precedence rules lead to the product `A * B` to be evaluated before `B * x` in the expression `A * B * x`. The invlib library uses proxy-types that represent algebraic expressions to implement those optimizations automatically at compile time. Furthermore, the invlib matrix algebra replaces inverses of matrices by solving a linear system where possible and minimizes the number of temporary matrix or vector objects. The code below results in one matrix-vector product and the solution of one linear system instead of the inversion of a matrix and a matrix-matrix product, as one might suspect at first glance:

```
auto C = inv(A) * B;
VectorType w = C * w;
```

invlibs generic matrix algebra is implemented by the `invlib::Matrix` and `invlib::Vector` class templates, which take as type arguments the user provided matrix and vector types. An example of how to instantiate the generic matrix and vector classes is given in Appendix B. The user provided matrix and vector types must implement the interface required by the `invlib::Matrix` and `invlib::Vector` class templates. However, since for class templates only functions that are actually called in the application code are instantiated by the C++ compiler, the interfaces that the user provided matrix and vector types must implement depend on the actual computations performed. For example, if the inverse problem is solved using the conjugate gradient method as solver for the linear subproblem, the user-provided matrix type is required only to provide functions for the multiplication and transposed multiplication of the matrix with a vector but not for matrix-matrix multiplication. Full

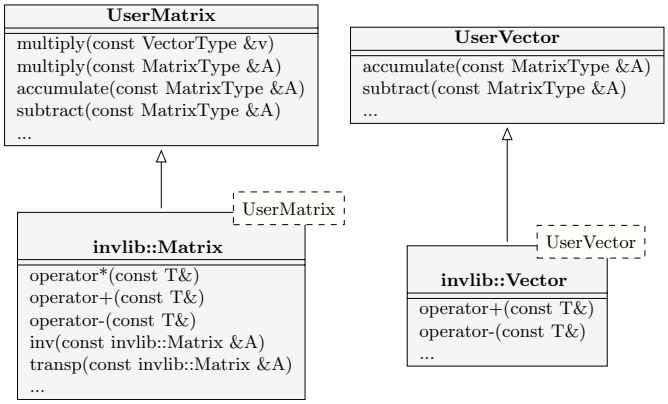


Figure 2.2.3

interfaces for the matrix and vector types, that provide the functions required for all computation schemes are given by the matrix and vector archetypes distributed with the invlib code. The class structure of the invlib matrix and vector types with respect to user provided types are displayed in Figure 2.2.3.

2.2.3 Covariance Matrices

The `MAP` class template takes as optional template arguments types for the covariance matrices $\mathbf{S}_a$ and $\mathbf{S}_\epsilon$. This allows the user to freely choose and combine different matrix representations for the covariance matrices. In addition to that, this mechanism can be used to specify the covariance matrices either directly or using their inverse, i.e. the so called precision matrices. To this end, invlib provides the `PrecisionMatrix` wrapper class, which takes any matrix type and signals to invlib that the provided matrix is the inverse of a covariance matrix.

```
template
<
typename Matrix
>
class PrecisionMatrix;
```

2.2.4 Formulation

The invlib library aims to provide high flexibility also with respect to how the MAP estimator is computed. To this end the user can use the `Formulation` template argument of the `MAP` class template to specify which formulation of the MAP estimator should be used. As described in the previous chapter, three different formulations can be derived to compute the MAP estimator of an inverse problem solution: The standard, n- and m-form (cf. Section 1.2.1). Which form should be used for the computation of the MAP estimator is specified by the `Formulation` enum used to instantiate the MAP class.

```
enum class Formulation {STANDARD = 0, NFORM = 1, MFORM = 2};
```

2.2.5 Optimization Methods

Currently invlib implements three optimization methods that can be used to perform the actual minimization of the cost function (2.0.1). Each optimization method is implemented by a class that provides a member function `step`

```
VectorType step(const VectorType &x,
                const VectorType &g,
                const MatrixType &H,
                CostFunction & J);
```

which takes the current iteration vector $\mathbf{x}$, the gradient $\mathbf{g}$ and (approximate) Hessian $\mathbf{H}$ of the cost function as well as a `CostFunction` object that can be used to evaluate the cost function. The function returns a minimization step $\Delta\mathbf{x}$ computed according to the chosen optimization method.

The optimization method object is used to specify convergence criterion and maximum number of steps for the optimization loop via the member functions `get_tolerance()` and `get_maximum_iterations()`, respectively. Those values can be set using either the constructor or the corresponding setter functions. The `GaussNewton` and `LevenbergMarquardt` class templates take, in addition to the type used to represent floating point numbers, a `Solver` type argument. The `Solver` argument specifies the method used to solve the linear system arising in each step

of the optimization method. Furthermore, the `LevenbergMarquardt` class template method takes an additional type argument which specifies the type of the positive definite matrix $\mathbf{D}$ that is used to define the distance measure used for the trust region of the Levenberg-Marquardt method via

$$||x||^2 = \mathbf{x}^T\mathbf{D}\mathbf{x}. \tag{2.2.2}$$

```
template
<
typename RealType
>
class GradientDescent;

template
<
typename RealType,
typename Solver = Standard
>
class GaussNewton;

template
<
typename RealType,
typename DampingMatrix,
typename Solver = Standard
>
class LevenbergMarquardt;
```

2.2.6 Solver Types

The solver type argument of the `GaussNewton` or `LevenbergMarquardt` class template defines how the linear system arising in each minimization step is solved. Each solver object must provide a `solve` member function of the form

```
    VectorType solve(const MatrixType&A,
                     const VectorType& v);
```

Note that the `MatrixType` here is not fixed and may for example be a proxy type representing an algebraic expression such as `A + B` for two matrix objects `A` and `B`.

invlib currently provides two basic solver types. The `Standard` solver simply forwards the call to `solve` to the `solve(const VectorType &)` member function which must be implemented by the underlying matrix type provided by the user. Note that for general algebraic expressions this will require the evaluation of the expression which may involve other expensive operations such as matrix-matrix multiplication or matrix inversions.

The `ConjugateGradient` solver implements the conjugate gradient method (c.f. 1.2.3). Since the conjugate gradient method only requires the computation of matrix-vector products, using a `ConjugateGradient` solver to solve a linear system given by an arbitrary algebraic expression will not trigger the evaluation of the corresponding matrix but rather propagate the vector successively through the expression.

2.3 Usage Example

In this section an example is provided that demonstrates the usage of the invlib library to retrieve simulated measurements from the MATS satellite mission.

The *Mesospheric Airglow/Aerosol Tomography and Spectroscopy* (Gumbel et al., 2014) satellite mission uses optical measurement techniques to study the mesosphere. The corresponding foward model is linear and thus fully described by a constant Jacobian matrix. While invlib was designed to build upon an already existing forward model implementation, it can also handle the case where a linear forward model is given only by its Jacobian matrix. All that is required here is an interface to an external library for the fundamental matrix and vector types and a wrapper class for the Jacobian that implements the invlib forward model interface.

In this example the Eigen 3 (Eigen, 2016) library is used as an external library to provide the underlying matrix algebra implementation. While both the Eigen 3 interface as well as an interface class for the linear forward model are distributed with the invlib library, a documented implementation of both of them is given in Appendix B.

2.3.1 Solving An Inverse Problem

The code below demonstrates the general usage of the invlib library when concrete matrix and vector types as well as a forward model implementing the invlib forward model interface are available. The general usage is the same independent of which type of problem one wants to solve.

```
int main()
{

    using MatrixType = invlib::Matrix<EigenSparse>;
    using VectorType = invlib::Vector<EigenVector>;

    using SolverType          = invlib::ConjugateGradient;
    using MinimizerType       = invlib::GaussNewton<double, SolverType>;
    using PrecisionMatrixType = invlib::PrecisionMatrix<MatrixType>;
    using MAPType             = invlib::MAP<LinearModel,
                                            MatrixType,
                                            PrecisionMatrixType,
                                            PrecisionMatrixType>;
```

While not compulsory, the readability of the code can be greatly improved by introducing type aliases (or typedefs) for the instances of the template classes that will be used for the computation. This also makes sense semantically, since the types define which representations, methods and formulations are used to solve the inverse problem. In the code above, first the concrete matrix and vector types are defined using the `invlib::Matrix` and `invlib::Vector` wrapper instantiated with the `EigenSparse` and `EigenVector` Eigen 3 interface types. To solve the linear inverse problem the Gauss-Newton method is used, since it converges after only one step. Due to the size of the system, it is necessary to use the conjugate gradient method, which is specified by using the `invlib::ConjugateGradient` class as solver type argument for the `invlib::GaussNewton` class template.

Since the MATS retrieval uses Tikhonov regularization, the (pseudo) covariance matrices are given in the form of precision matrices. The `invlib::MAP` class template is thus instantiated here with the `LinearModel` class, that represents the linear forward model, the `MatrixType` which specifies the types used for linear algebra, and the `PrecisionMatrixType` for both a priori and state covariance matrices. Since the retrievals are performed using the standard formulation of the OEM together with the standard output method, the corresponding template arguments are left unchanged from their default values.

```
// Load the data.
MatrixType K     = read_sparse_matrix("data/K.sparse");
MatrixType SaInv = read_sparse_matrix("data/SaInv.sparse");
MatrixType SeInv = read_sparse_matrix("data/SeInv.sparse");
PrecisionMatrixType Pa(SaInv);
PrecisionMatrixType Pe(SeInv);

VectorType y     = read_vector("data/y.vec");
VectorType xa    = read_vector("data/xa.vec");
```

The next step is loading the data and creating the Jacobian and covariance matrices as well as the measurement and a priori vector. In addition to that, the precision matrix wrappers are created from the loaded inverses of the covariance matrices.

```
SolverType    cg(1e-6, 1);
MinimizerType gn(1e-6, 1, cg);
LinearModel   F(K, xa);
MAPType       oem(F, xa, Pa, Pe);

VectorType x;
oem.compute(x, y, gn, 0);
```

Finally, the computation parameters are set by instantiating the `SolverType`, `MinimizerType`, `LinearModel` and `MAPType` types, with the corresponding matrices, vectors and convergence criteria. The call to `cg(1e-6, 1)` requires the size of the residual normalized by the right-hand side vector for the conjugate gradient method to be less than 10^{-6} for convergence. The second argument activates log output to standard out. The call to `gn(1e-6, 1, cg)` creates the Gauss-Newton minimizer with a convergence criterion of 10^{-6}, the maximum number of iterations equal to 1, and the previously created conjugate gradient solver as the solver for the linear subproblem. The actual MAP computation is then started by calling the `compute` member function of the `MAPType` object and the result is returned in the vector `x` passed as the first argument to the method.

The MATS satellite measures optical emissions from the upper mesosphere. The atmospheric state is represented by the values of the scattering coefficient on a three-dimensional grid along the satellite orbit. The measurements used for the retrieval are simulated data of an atmospheric volume containing a polar mesospheric cloud. Solving the linear system up to an accuracy of 10^{-6} requires about 7000 steps of the standard conjugate gradient solver. A volume redering of the of the a priori state of the scattering coefficient and the retrieved state of the calculation are displayed

in Figure 2.3.1a and Figure 2.3.1b, respectively. The a priori state contains only the Rayleigh background, whereas in the retrieved state the structure of the cloud is clearly visible.

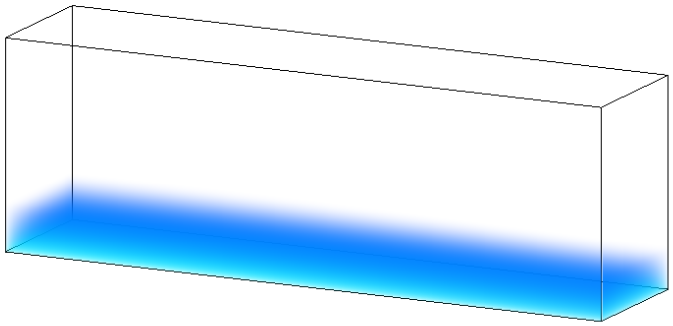
2.3.2 Parallelizing the Computation

One disadvantage of using the Eigen 3 sparse matrix implementation is that sparse matrix-vector multiplication is not parallelized. The invlib library provides a generic matrix class that parallelizes matrix and vector computations using MPI (MPI, 1994). This is helpful because it allows large problems to be run on distributed-memory systems and thus relaxes the memory requirements, but can also be used in cases such as this where the underlying matrix and vector operations are not parallelized. Using invlib's parallel matrix and vector types requires only a few changes to the code presented above and are mostly related to setting up the MPI environment and distributing the data over the different processes. However, in order to enable invlib to distribute the Eigen 3 matrices over MPI processes, additional functions in the matrix and vector interfaces are required. The required changes are described in Appendix B.

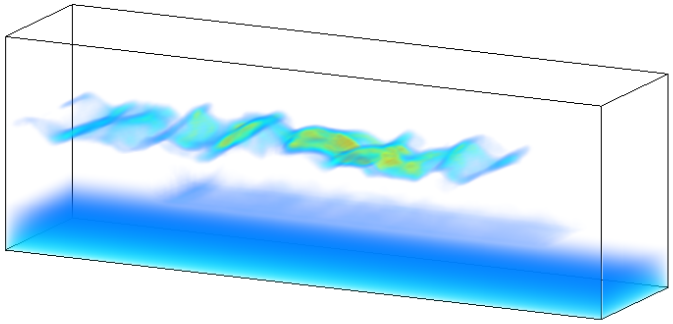
In order to use the MPI-parallel types in the actual computation, it is necessary to redefine the matrix and vector types accordingly.

```
using SolverType      = invlib::ConjugateGradient;
using MinimizerType   = invlib::GaussNewton<double, SolverType>;
using PrecisionMatrix = invlib::PrecisionMatrix<MPIMatrixType>;
using MAPType         = invlib::MAP<LinearModel,
                                    MPIMatrixType,
                                    PrecisionMatrix,
                                    PrecisionMatrix>;
```

invlibs parallel matrix and vector types parallelize computations by splitting up matrices and vectors row-wise over processes. Here, each process reads the full Jacobian and precision matrices as well as a priori and measurement vectors. The splitting up is performed by using the corresponding types' static `split` functions, which return the distributed object corresponding to the provided local, full matrix or vector.



(a) Volume rendering of the a priori state used for retrieval containing only the Rayleigh background.



(b) Retrieved scattering coefficient for the retrieval of simulated data from the MATS satellite mission. In addition to the Rayleigh background the structure of the polar mesospheric cloud is clearly visible.

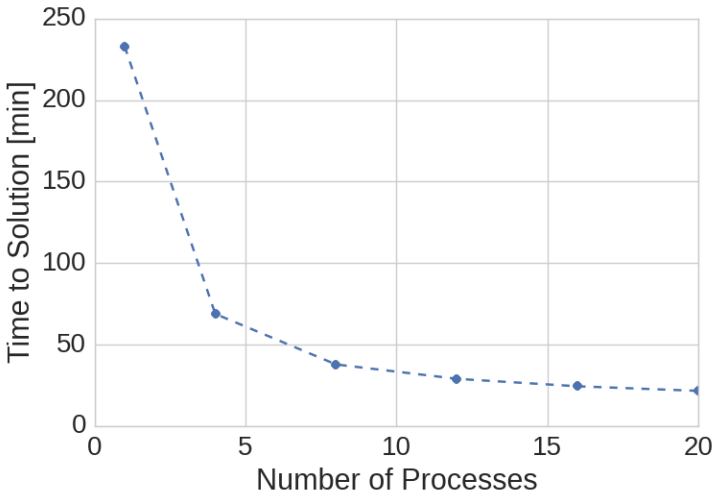
Figure 2.3.1: Volume rendering of the a priori and retrieved state of simulated data from the MATS satellite mission.

```
MatrixType K     = read_sparse_matrix("data/K.sparse");
MatrixType SaInv = read_sparse_matrix("data/SaInv.sparse");
MatrixType SeInv = read_sparse_matrix("data/SeInv.sparse");

MPIMatrixType K_mpi     = MPIMatrixType::split_matrix(K);
MPIMatrixType SaInv_mpi = MPIMatrixType::split_matrix(SaInv);
MPIMatrixType SeInv_mpi = MPIMatrixType::split_matrix(SeInv);

PrecisionMatrix Pa(SaInv_mpi);
PrecisionMatrix Pe(SeInv_mpi);

VectorType y     = read_vector("data/y.vec");
VectorType xa    = read_vector("data/xa.vec");
```

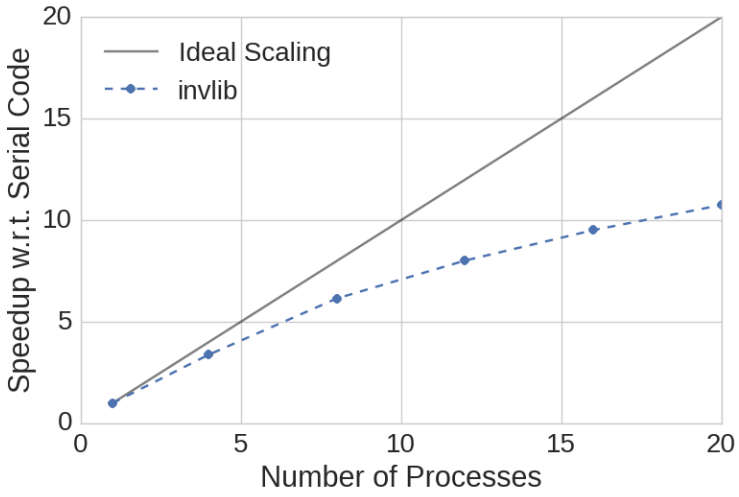
The differences in the calls to the other invlib functions are mostly due to the different naming of the distributed variables. The call to the `compute` function of the `MAP` object is slightly modified, since a specialized logging method is used which ensures that only one process prints messages to the standard output.

```
SolverType    cg(1e-6, 1);
MinimizerType gn(1e-6, 1, cg);
LinearModel   F(K_mpi, xa_mpi);
MAPType       oem(F, xa_mpi, Pa, Pe);

// Run OEM.
MPIVectorType x_mpi{};
oem.compute<MinimizerType, invlib::MPILog>(x_mpi, y_mpi, gn, 1);

MPI_Finalize();
```

Figure 2.3.2a displays the performance gains achieved using the the parallelized matrix and vector types that could be achieved by distributing the calculations over two nodes of the Hebbe compute cluster. Considerable performance can be gained by scaling the application up to ten cores, i.e. a single node, while for more cores not much additional performance can be gained. While the scaling is not ideal, considerable speed up of the calculation can be obtained with a very low effort from the user.



(a) Time to solution in with respect to the number of cores of the parallelized MATS retrieval performed above.



(b) Scaling of the parallelized MATS retrievals.

Figure 2.3.2: Performance of the MPI-parallelized implementation of the MATS retrievals.

Chapter 3

Tomographic Retrievals of Odin-SMR Data

While the discussion of inverse problems so far has been kept general, this chapter describes the application of the developed theory to tomographic retrievals of satellite remote sensing data from the Odin-SMR imager. In the field of remote sensing, tomography refers to an imaging technique which combines multiple measurements from overlapping imaging regions in order to improve the spatial reconstruction and increase the accuracy of the retrieval. These computations were first performed by Christensen, Eriksson, Urban and Murtagh in (Christensen et al., 2015). However, due to limitations of the implementation of the retrieval method, the retrievals had to be performed in relatively small batches, combining a reduced number of spectra along the half-orbit over which the tomographic measurements were performed. The batch sizes had to be chosen with sufficient overlap to guarantee consistency of the retrieved data, which unnecessarily increased the computational cost of the retrievals. As part of this thesis work, data from one half orbit has been retrieved using the newly developed OEM implementation in ARTS, that is based on the invlib library. In this chapter, the setup of the computations and the results of the retrieval are presented. In addition to the computations using the standard OEM implementation in ARTS, it is also demonstrated how the computation time can be reduced using invlibs support for arbitrary matrix representations and generic distributed matrices.

3.1 Retrieval Setup

From a mathematical point of view, the retrieval process is fully described by the formulas presented in Chapter 1. Nonetheless, finding a solution for a concrete inverse problem requires a consistent setup of the problem. For satellite remote sensing retrieval, this requires an accurate modeling of the atmosphere as well as the measurement device itself. The purpose of this section is to provide an overview of physical processes involved in the measurement and the technicalities of incorporating them into the problem setup. The discussion is based on the article by Christensen, Eriksson, Urban and Murtagh Christensen et al. (2015).

3.1.1 The Odin Satellite

The Odin satellite (Murtagh et al., 2002) is a Swedish-led project in cooperation with Canada, France and Finland and was launched into a sun-synchronous orbit in 2001. The satellite carries two instruments, the *optical spectrograph and infrared imaging system* (OSIRIS) and the *sub-millimetre radiometer* (SMR).

The Odin SMR is a limb sounding instrument, which means it measures submillimeter radiation looking backwards along its orbit through the atmosphere with cold space as background. The SMR sweeps over a range of vertical viewing angles so that measurements at different tangential altitudes of the line of sight (LOS) are obtained. The SMR provides a special mode for tomographic measurements, in which the vertical scanning range is reduced in order to shorten the horizontal distance between overlapping measurements. The resulting measured spectra from overlapping lines of sight can then be combined using tomographic retrieval techniques to increase the resolution of the retrievals. The viewing geometry is illustrated in Figure 3.1.1. Note that refractive effects of the atmosphere are ignored here, since they are negligible for measurements that are limited to the mesosphere. The vertical resolution of the SMRs antenna is about 1.8 km at the tangent point due to its continuous, vertical scanning movement.

The vertical range scanned in tomographic mode is limited to altitudes between 75 km and 90 km. The available tomographic measurements were started when the satellite crossed the equator and continued over half its orbit. The region of the atmosphere under consideration is thus a two-dimensional band stretching over the northern hemisphere below the orbit of the satellite.

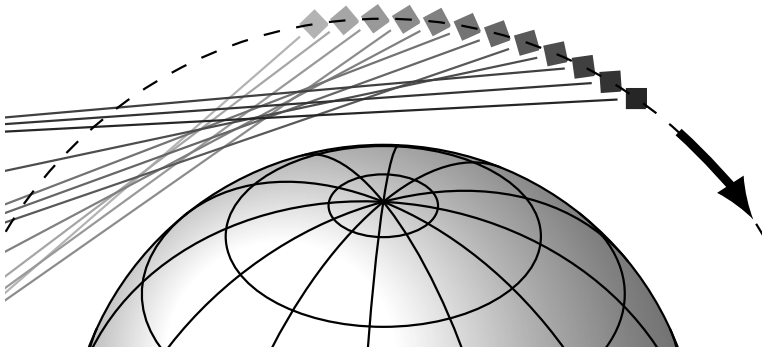


Figure 3.1.1: Illustration of the Odin SMR tomography mode in which spectra are recorded with increased frequency so that there is considerable overlap between consecutive measurements of the same altitude.

The spectra used for the retrieval cover the frequency range from 556.90 GHz to 557.01 GHz with a channel separation of 1 MHz and an effective resolution of 2 GHz. The frequency range is chosen so that it covers the H_2O spectral line at 556.9 GHz, which constitutes the main source of radiation in this frequency range. Spectra measured during one sweep over the vertical scanning range are displayed in Figure 3.1.2.

3.1.2 The Model Atmosphere

The forward model for the retrievals is a numeric simulation of the propagation of the radiation measured by the SMR through the atmosphere. This simulation is based on a suitable model of the atmosphere which includes the physical quantities that influence the propagation of radiation in the considered frequency range. The atmospheric model is implemented using ARTS. Here, a two-dimensional model is used, which represents an atmospheric state by a number of two-dimensional, discrete scalar fields holding the values of the considered physical quantities at given grid points. ARTS uses the pressure as vertical grid dimension, while the height is treated as an atmospheric quantity described by a corresponding scalar field. In addition to the atmospheric processes, also the imaging properties of the SMR must be taken into account to properly simulate the measurements. This functionality is also provided by the ARTS package. A complete description of the

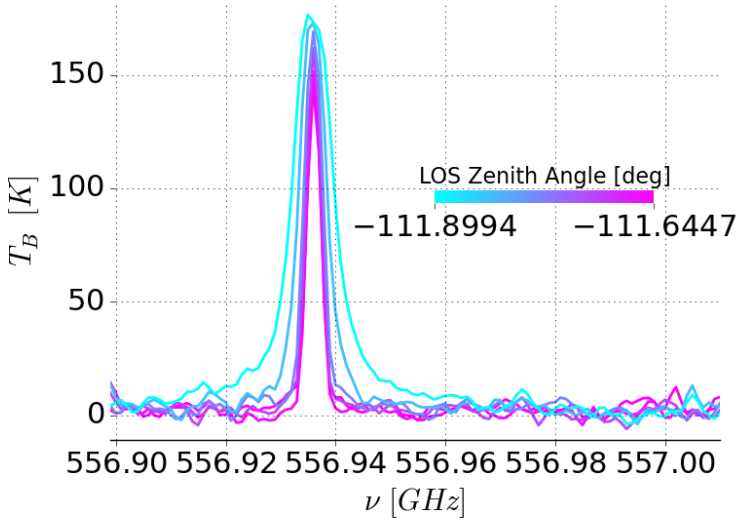


Figure 3.1.2: Odin SMR spectra recorded during one sweep over the vertical scanning range. Line colors indicate the zenith angle of the line of sight (LOS) of the measurement.

atmospheric model is out of the scope of this work, so in the following only a brief description of the properties specific to this retrieval problem is provided.

The two-dimensional atmospheric model represents the atmosphere below the orbit of the satellite. It extends over pressures from about 4.2×10^{-4} to 1.3×10^{3} Pa corresponding to altitudes from about 30 to 161 km. Horizontally, the atmosphere model extends from 20° to 201.5° with 0° designating the point where the satellite crosses the equator from south to north.

The State Space

The choice of the state space defines which quantities can be retrieved from the measurements as well as the degrees of freedom available to the model to fit the measurement vector. In the frequency range around the H_2O spectral line at 556.9 GHz all significant emission is due to water vapor. The measured spectra can thus be assumed to depend only on the pressure, temperature and concentration of water vapor in the atmosphere. The retrieval grid consists of 79 grid points along the vertical pressure dimension and 195 points along the satellite orbit. To represent the atmospheric state as a single state vector, the discrete scalar fields

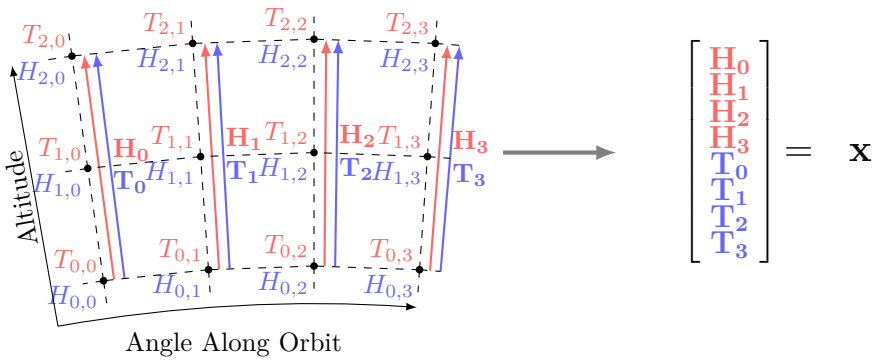


Figure 3.1.3: A discrete representation of the atmospheric state is obtained by representing it by the values of the physical quantities under consideration on each point of the retrieval grid. The grid is then transformed into a single vector by concatenating the fixed-latitude vectors into one large vector $\mathbf{x}$ as illustrated in figure.

holding the water vapor concentrations and the temperature at the atmospheric grid points are concatenated into a single large vector first along the pressure dimension and then along the latitudes. This is illustrated in Figure 3.1.3. In addition to the water vapor concentrations and the atmospheric temperatures also a constant correction to the pointing and a baseline for the recorded spectra are fitted to the data. This is done in order to allow the retrieval to correct for errors in the estimates of these parameters and reduce their effects on the retrieved quantities. The resulting dimension n of the state space is $n = 31279$.

The Measurement Space

The measurements used for the retrievals consists of the combined spectra recorded by the SMR as the Odin satellite moves along its orbit. A single spectrum consists of 112 intensity values spanning the frequency range from 556.89 GHz to 557.01 GHz. The measurement for the complete half orbit consist of 468 spectra. The resulting dimension m of the measurement space is $m = 52416$.

Atmospheric Radiative Transfer

The physical processes that determine the propagation of radiation through the atmosphere are described by the theory of radiative transfer. While a general theory of radiative transfer is very complex, for this specific problem setting a number of simplifying assumptions can be made, that considerably simplify the calculations. First of all, for the frequency regions measured by the Odin SMR scattering can be neglected. Furthermore, the radiation is assumed to be unpolarized and absorbing species to be in local thermal equilibrium. In this case the transmission of the radiation trough the atmosphere along a given direction is described by the following form of the scalar radiative transfer equation:

$$\frac{dI(\nu)}{dl} = k(l,\nu)\left(B(l,\nu) - I(\nu)\right) \tag{3.1.1}$$

Here, I is the spectral radiance, $\frac{dI}{dl}$ its derivative along a given path through the atmosphere, k the absorption coefficient and $B(\nu, \mathbf{r})$ the intensity of the black body radiation at the given point in the atmosphere. The absorption coefficient models the attenuation of radiation through gaseous line and continuum absorption. The black body term represents the emission of black body radiation at a given point in the atmosphere. ARTS solves (3.1.1) by first determining the propagation paths of the monochromatic pencil beams entering the detector and then integrating (3.1.1) along these paths.

The Sensor Model

The sensor model describes how the radiation entering the sensor is converted to the digital data that constitute the measurements obtained from the satellite. As described in Eriksson and Buehler (2016), these effects can be described by a linear model of the form

$$\mathbf{y} = \mathbf{H}\mathbf{i} + \epsilon_s \tag{3.1.2}$$

where $\mathbf{H}$ is the so called sensor response matrix, $\mathbf{i}$ is the radiance vector of the

monochromatic pencil beams entering the sensor and ϵ_s is the noise entering the measurement.

3.1.3 A Priori State and Covariance Matrices

The mathematical formulation of inverse problems also requires the specification of an a priori state $\mathbf{x}_a$ as well as covariance matrices for the a priori vector and the measurement vector.

The a priori vector and the a priori covariance matrix describe the available knowledge about the region of the atmosphere under investigation before the actual measurements are made. It should be noted here, that the Bayesian interpretation of those quantities differs from the maybe more common frequentist interpretation. All probabilistic quantities treated here should be interpreted as representing states of knowledge as opposed to frequencies in a fictive sampling experiment. This allows specification of the a priori vector and covariance matrices even if those quantities would be unknown in a frequentist interpretation. This, of course, needs to be taken into account in the interpretation of the results.

For the a priori vector $\mathbf{x}_a$, values of all retrieval quantities must be specified for each point on the atmosphere grid. For water vapor, those values were obtained from measurements from another satellite mission. Of these measurements the mean was taken over all latitudes over 60° and over the three months in which the tomographic measurements took place. The resulting a priori water vapor concentrations vary only with altitude, so that horizontal structures can be more easily detected in the retrievals. The a priori data for the temperature grid was obtained from the MSISE-90 model.

The a priori covariance matrix specifies the uncertainty of the a priori knowledge. Here it is assumed that different atmospheric variables are uncorrelated. The a priori matrix thus takes the form of a block diagonal matrix with one block for each retrieval species ($\mathbf{S}_a^H, \mathbf{S}_a^T$) and an additional block for the variables specifying the pointing and baseline fits ($\mathbf{S}_a^M$).

$$\mathbf{S}_a = \begin{bmatrix} \mathbf{S}_a^H & & \\ & \mathbf{S}_a^T & \\ & & \mathbf{S}_a^M \end{bmatrix} \tag{3.1.3}$$

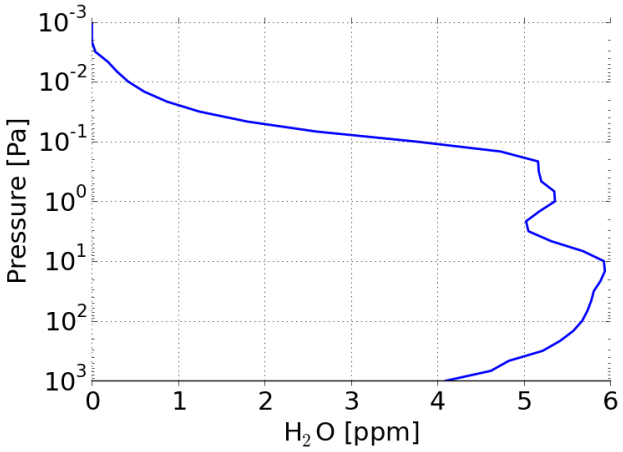


Figure 3.1.4: The one dimensional water vapor concentration profile used in the a priori vector.

The water vapor concentrations and the temperature are assumed to be locally correlated with an exponentially decaying covariance. For two variables located at grid points i and j the covariance takes the form

$$\sigma_{i,j} = \sigma_s \exp\left\{-\sqrt{\left(\frac{\Delta x}{l_x}\right)^2 + \left(\frac{\Delta y}{l_y}\right)^2}\right\} \tag{3.1.4}$$

where σ_s is the variance of the retrieval species s, the variable $\Delta x, \Delta y$ denote the horizontal and vertical distances between the two grid points, and l_x, l_y the horizontal and vertical correlation lengths. The chosen values are given in Table 3.1.1. A plot of the shape of the correlation function along the horizontal grid dimension is given in Figure 3.1.5. The retrieved parameters for the pointing correction and the baseline fit are assumed to be uncorrelated and the corresponding block in the covariance matrix is thus diagonal.

For the measurement state covariance matrix, it is assumed that the forward model accurately predicts the measurement vector $\mathbf{y}$. The only source of uncertainty is the noise in the signal detected by the Odin SMR sensor. Noise in different channels and from different measurements is uncorrelated and $\mathbf{S}_\epsilon$ is thus a diagonal matrix. The noise level for each channel can be obtained from the noise temperature of the receiver and the integration time used for the measurements.

Quantity	σ_s	l_x	l_y
H_2O	3×10^5 ppm	5°	8 km
T	7 K	5°	8 km

Table 3.1.1: Retrieval quantity correlation lengths used for the a priori covariance matrix.

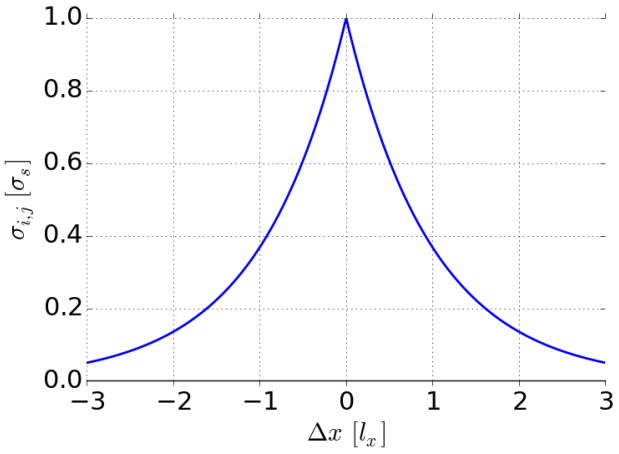


Figure 3.1.5: Shape of the correlation function along the horizontal grid dimension.

3.2 Retrieval Results

The retrieved concentrations of water vapor and temperature together with the respective a priori states are displayed in Figure 3.2.1a and Figure 3.2.1b. The retrieval region displays the upper region of the mesosphere limited from above by the mesopause, which is the temperature minimum between between the mesosphere and thermosphere above. The current understanding of the dynamics of the summer mesosphere is that water vapor is brought up and then removed by photodissociation as it reaches the mesopause. This knowledge is represented in the a priori water vapor concentrations which show a strong gradient at a height of about 90 km. While this general structure is reproduced in the retrievals, they also reveal a much finer structure of this transition region. The most salient of these structures are the peaks around 80° and 110° along orbit. Those are related to the formation of *polar mesospheric clouds* and are usually found beneath or between them. Also clearly visible is an increased concentration of water vapor in the upper mesosphere around the pole. The retrieved temperature exhibits less deviation from the a priori state, of which the strongest are the temperature minima in the region around

the pole. This observation is also in agreement with current understanding of the mesosphere. For a more detailed discussion of these results the reader is referred to the article by Christensen et al.Christensen et al. (2015).

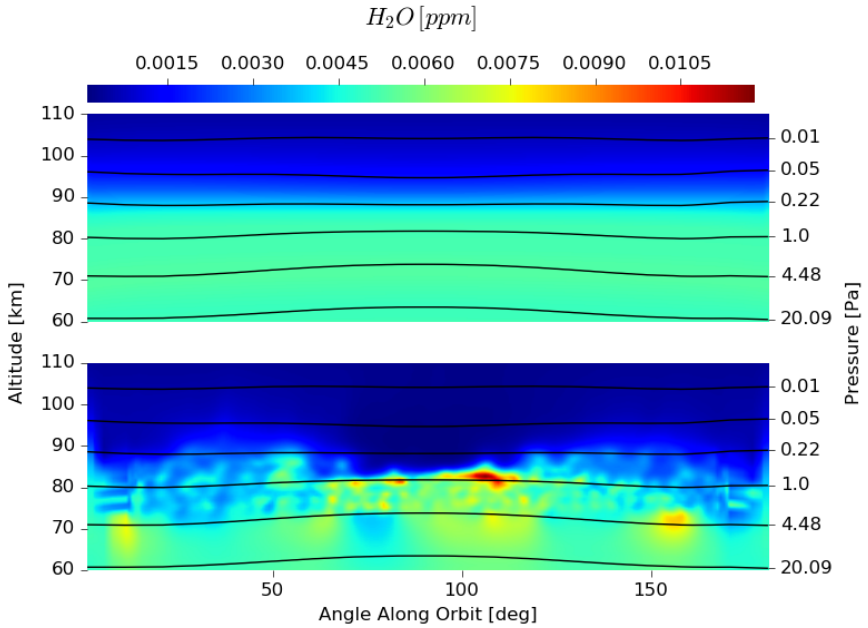
3.3 Computational Performance

The tomographic retrievals of the Odin SMR data are computationally demanding both numerically as well as with respect to memory requirements. The main limitation of the calculations performed by Christensen, Eriksson and Murtagh in Christensen et al. (2015) were that they were performed on a desktop machine whose main memory limited the maximum problem size and thus required the splitting up of the orbit into several batches. These limitations can be overcome by performing the calculations on a dedicated compute cluster. In addition to that, the newly developed OEM implementation in ARTS provides the conjugate gradient method as a solver for the linear subproblem of each minimization step, which can further reduce the memory footprint of the calculations. In this section, the general performance characteristics of the two solution methods are presented. Furthermore, two possible performance optimizations are investigated: Representing the Jacobian as a sparse matrix and distributing the computations over multiple compute nodes using invlibs generic distributed matrix types.

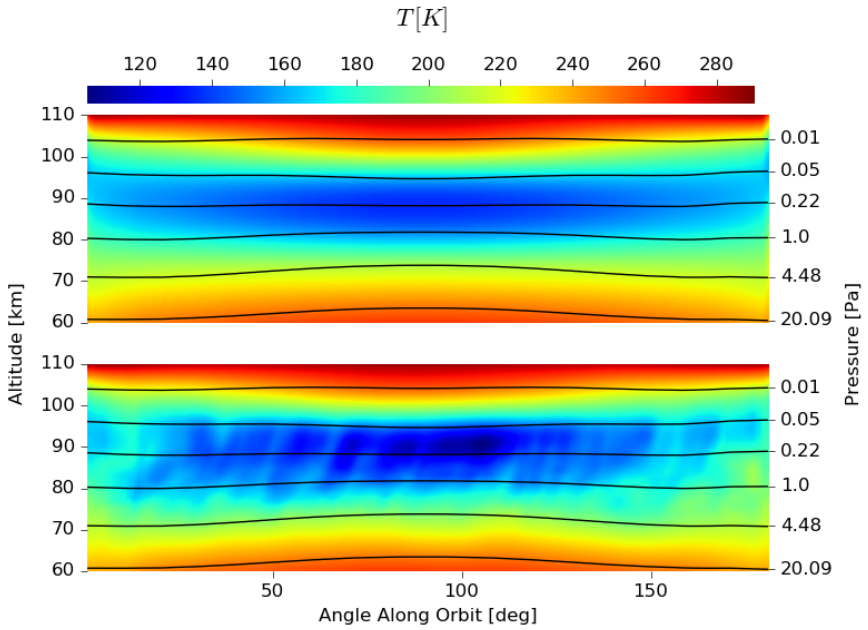
The computations presented here have been performed on 1 to 8 nodes of the Hebbe compute cluster at Chalmers University of Technology. Each node is equipped with a 10-core Intel Xeon E5-2650 and 64 GB of main memory. The optimization method used for the retrieval is the Levenberg-Marquardt method with the convergence criterion chosen to be 10^{-3}. The start value for the λ parameter in the Levenberg-Marquardt method was set to 10 with a decrease factor of 10 and an increase factor of 2. The convergence criterion for the conjugate gradient method was set to 10^{-4}.

3.3.1 Single-Node Calculations

The increased amount of main memory available on a node of the Hebbe compute cluster made it possible to perform the retrieval using a conjugate gradient solver as well as a direct solver using LU decomposition. These calculations have been performed with the current development version of ARTS. In addition to that, possible runtime improvements that can be obtained by exploiting the sparse



(a) A priori and maximum a posteriori water vapor concentrations as obtained from the Odin SMR tomographic measurements.



(b) A priori and maximum a posteriori temperature as obtained from the Odin SMR tomographic measurements.

Figure 3.2.1: Results of the tomographic retrievals of the Odin SMR data.

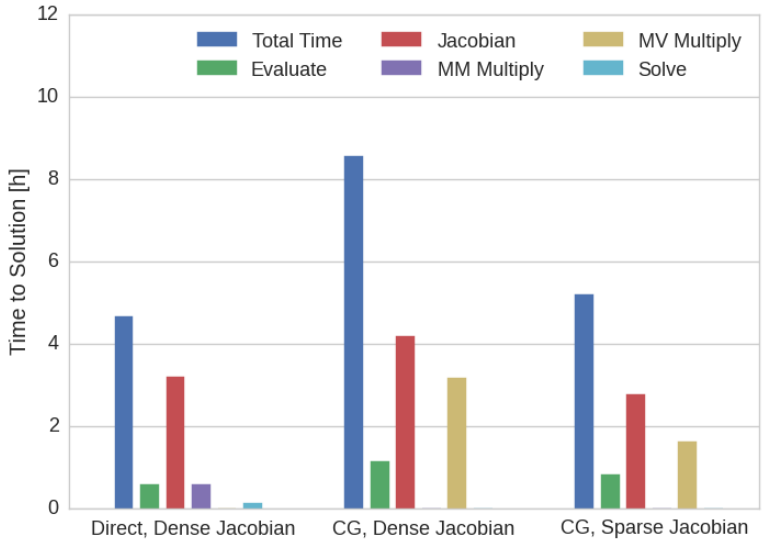
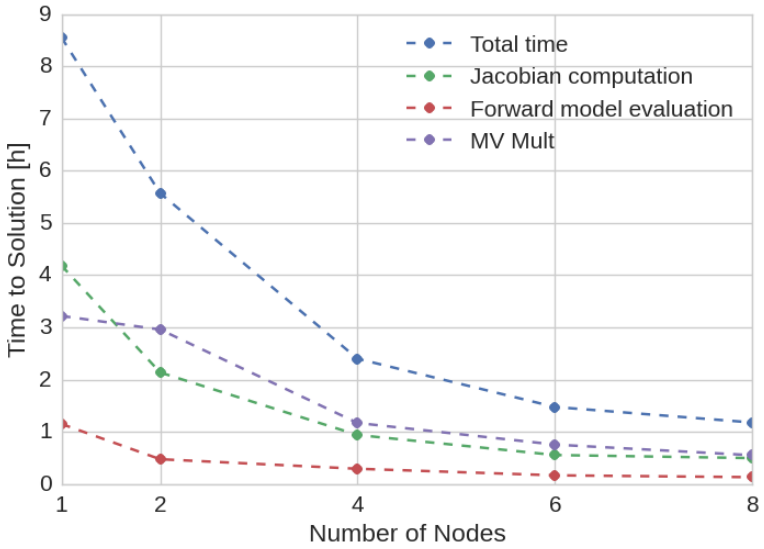


Figure 3.3.1: Computational time required for the retrieval of a full half orbit as well as the time spent in the main subroutines for the three retrieval configurations.

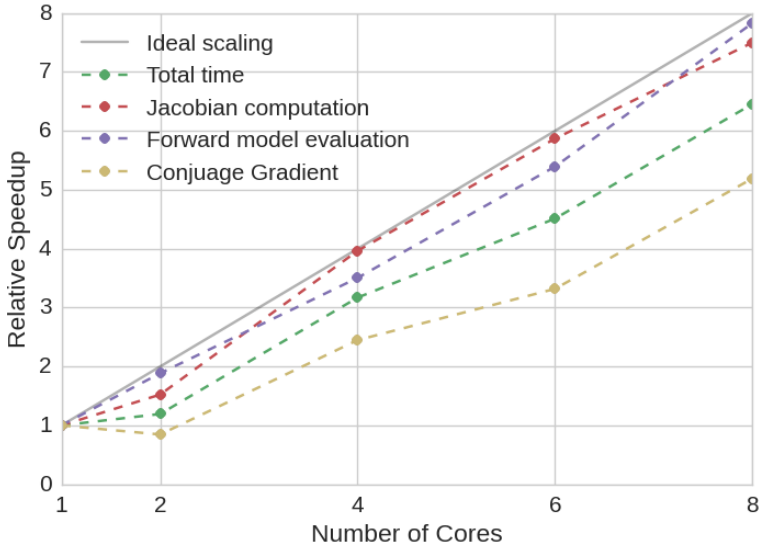
structure of the Jacobian in the calculations have been investigated by adding the corresponding functionality to ARTS. This development is currently experimental and not published as an official version of ARTS. The required time for the retrieval calculations together with the time spent in the principal compute routines are displayed in Figure 3.3.1. As can be seen from the plot, the best performance is obtained using the direct solver, while the calculations using the conjugate gradient method take almost twice as much time. The performance of the conjugate gradient method can be improved by choosing a sparse representation for the Jacobian, which reduces the computation time to about the time required using the direct solver. What can be further seen from the plots is that the largest part of the computation time is spent evaluating and computing the Jacobian of the forward model. For the direct solver this amounts to about 80 % of the total computation time, whereas for the conjugate gradient solver without sparse Jacobian this fraction is reduced to 60 %. This also shows that the conjugate gradient solver not only requires more time to solve the linear subproblem of the minimization method, but also prolongs the computation by requiring more minimization steps to reach convergence.

3.3.2 Distributed Calculations

Due to the complexity of the forward model, a large part of the computational time required for the retrieval calculations is spent evaluating and computing the Jacobian of the forward model. However, since the tomographic retrievals combine observations from different positions of the satellite, which are computed completely independently of each other, these calculations can be very easily parallelized. This parallelism is exploited in ARTS using OpenMP (OpenMP, 2008). Since OpenMP only supports shared memory systems, the calculations are restricted to execution on a single node of the compute cluster, which of course limits the achievable performance gains through parallelism. Nonetheless, the complete independence of the calculations for the different satellite positions can be used to easily extend ARTS to distribute calculations over multiple compute nodes using MPI. Together with invlibs support for generic distributed matrices, this extended version of ARTS can be used to distribute the retrieval calculations over multiple compute nodes. In order to investigate the performance of the resulting distributed implementation of the retrieval, the Odin-SMR tomographic retrievals have been performed on 1,2,4,6 and 8 compute nodes of the Hebbe cluster. The corresponding timing results as well as the time spent in the main computational subroutines are displayed in Figure 3.3.2a. As expected, the computation time can be decreased considerably by distributing the calculations over multiple compute nodes. The overall speed-up as well as the speed-ups obtained for the main subroutines are displayed in Figure 3.3.2b. From the plot it can be seen that the overall computation time scales acceptably well with the number of compute nodes. In particular, the computation of the Jacobian as well as the evaluation of the forward model scale very well with the number of compute nodes, but these performance gains are neutralized to some extent by the sub-optimal scaling of the conjugate gradient method.



(a) Time to solution of the MPI-parallelized retrieval.



(b) Scaling of the MPI-parallelized retrieval.

Figure 3.3.2: Performance of the distributed, tomographic retrievals of Odin-SMR data.

Chapter 4

Conclusions and Future Work

In this chapter, the main results of this thesis work are summarized and discussed. Finally, the main part of the report is concluded with a future outlook and ideas for further developments and improvements of the invlib library and the implementation of the OEM in ARTS.

4.1 Results

The main result of this thesis work is the invlib library, which provides a free software implementation of Bayesian methods for inverse problems with Gaussian a priori knowledge and measurement error. The library has been integrated into the ARTS software package for the simulation of atmospheric radiative transfer. ARTS can now be used as a stand-alone package for the retrieval of atmospheric remote sensing data. The retrieval functionality in ARTS provides the Gauss-Newton and Levenberg-Marquardt method as minimization methods, which both can be combined with a direct or an indirect solver for the linear system arising in each minimization step.

This newly developed OEM implementation has been used to perform tomographic retrievals of Odin-SMR data. By performing the computations on the Hebbe compute cluster, the calculations for a complete half orbit could be computed in a single computation, which has not been possible previously. Furthermore, experimental extensions of the ARTS code have been developed that allow for a

sparse representation of the Jacobian as well as the distribution of the calculations over multiple compute nodes using MPI. As an additional result, it has been demonstrated that the invlib library is general enough to handle retrieval problems outside of ARTS. In particular, invlib has been used to perform the retrieval of simulated data from the MATS satellite mission. Also here, the use of invlibs distributed data types to parallelize the calculations could considerably speed up the retrievals.

4.2 Discussion

From a mathematical point of view, the treatment of inverse problems as they arise in remote sensing is straight forward. Nevertheless, the implementation of those methods in a numerical software library is still challenging. This is due to the great flexibility and generality required to handle arbitrary problems efficiently. Different problems may, for example, require different representations for the matrices involved in the computation in order to achieve optimal performance. Moreover, large problems require an implementation that is efficient both computationally as well as with respect to memory usage. The approach taken here was to make extensive use of template programming. While this may make the code more difficult to use for users unexperienced with this technique, the successful application of the library to two large-scale retrieval problems demonstrates that this approach is capable of achieving both the generality and performance required for the retrieval of remote sensing data.

Based on the invlib library, functionality for performing retrievals using the optimal estimator method has been integrated into ARTS. This implementation has been used to perform tomographic retrievals of Odin-SMR data and to investigate the performance characteristics of different retrieval configurations. Interestingly, the single-node retrievals that used a direct solver for the linear subproblem achieved the best performance. Nevertheless, similar performance could be obtained using a sparse representation for the Jacobian. One of the reasons for the inferior performance of the conjugate gradient method is likely to be the structure of the a priori covariance matrix $\mathbf{S}_a$, which has non-zero values in about 48% of its elements and thus not really exhibits a sparse structure. This is in accordance with the results from Section 1.3, which indicate that the conjugate gradient method yields improvements in computational time only for sufficiently sparse matrices.

Furthermore, this thesis explored the parallelization of the computation of MAP

estimators. On the invlib side, the parallelization is achieved through generic, distributed matrix and vector types. As was shown in the application to the retrieval of MATS data, the implementation achieves acceptable scaling to a small number of nodes. It should be noted here that the implementation uses a single, generic implementation of the MAP estimators, the minimization method and the conjugate gradient method for the serial as well as for the parallel version. This has the advantage of reducing the code base and thus simplifies ensuring correctness and maintaining the code. While it is very likely that better performance can be achieved by further optimizing communication patterns, the current code should be regarded as a proof of concept which can be further optimized if required by the application. Finally, also a simple parallelization of the ARTS forward model implementation was developed. Since the ARTS code is currently not designed for distributed computations, one aim of the parallelization was to require as little changes to the existing ARTS code as possible. Still, as the numerical results in the previous chapter show, this distributed version of ARTS achieves good scaling to a small number of nodes. All in all, using the distributed retrieval implementation, the compute time for the tomographic retrieval of the Odin-SMR data could be reduced to about 4 hours.

While a number of different retrieval configurations has been investigated in this work with respect to their performance, there still remains a large number of different configurations and parameters whose influence could not be investigated here due to time constraints. These configurations include the convergence criterion for the conjugate gradient method, preconditioners for the conjugate gradient method, the parameter settings of the Levenberg-Marquardt method, optimization of the step size of the Gauss-Newton and Levenberg-Marquardt method as well as different OEM formulations.

4.3 Future Work

While the main aim of this thesis, namely adding an implementation of the optimal estimation method to ARTS, has been achieved, there still exist a number of interesting extensions of the invlib functionality. One of them would be to implement preconditioners for the conjugate gradient method, which may accelerate the convergence of the conjugate gradient solver and thus improve the run time of retrievals using the indirect solver. Furthermore, for linear retrievals, it would also be interesting to investigate potential speed ups that can be obtained by performing the calculations on an hardware accelerator device such as a GPU. Finally, it would

also be interesting to apply the invlib library to solve other inverse problems in order to obtain further insight into the capabilities and requirements on the library.

Appendices

Appendix A

Algorithms

The Gauss-Newton method iteratively minimizes a cost function J. In each step, the function J is approximated by a quadratic function using its gradient $\nabla_{\mathbf{x}} J$ at the current iteration vector $\mathbf{x}$ and the corresponding Hessian $\nabla^2_{\mathbf{x}} J$. The minimum of this second-order fit is then used as the next step in the iteration.

A.1 Gauss-Newton Method

Algorithm 1 Gauss-Newton Method

1: $\mathbf{x} \leftarrow \mathbf{x}_a$
2: **repeat**
3: $\quad \Delta\mathbf{x} \leftarrow -\left(\nabla^2_{\mathbf{x}} J\right)^{-1} \nabla_{\mathbf{x}} J$
4: $\quad \mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$
5: **until** converged

A.2 Levenberg-Marquardt Method

The Levenberg-Marquardt method extends the Gauss-Newton method with a heuristically controlled trust region around the current iteration step. The size of the trust region is adapted at each step of the iteration depending on the cost reduction obtained in the current step.

Algorithm 2 Levenberg-Marquardt Method

$\mathbf{x} \leftarrow \mathbf{x}_a$
$\lambda \leftarrow \lambda_{\text{start}}$
repeat
    repeat
        $\Delta\mathbf{x} \leftarrow -\left(\nabla^2_{\mathbf{x}} J + \lambda \mathbf{D}\right)^{-1} \nabla_{\mathbf{x}} J$
        $\mathbf{x}_n \leftarrow \mathbf{x} + \Delta\mathbf{x}$
        $c \leftarrow \frac{J(\mathbf{x}_n) - J(\mathbf{x})}{\nabla_{\mathbf{x}} J^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \nabla^2_{\mathbf{x}} J \Delta\mathbf{x}}$
        if c > 0.75 **then**
            if $\lambda > 0$ **then**
                $\lambda = \lambda_{dec} \cdot \lambda$
                if $\lambda \geq \lambda_{\max}$ **then**
                    Error
                end if
            else
                $\lambda = \lambda_{\min}$
            end if
        else if c < 0.2 **then**
            $\lambda = \lambda_{inc} \cdot \lambda$
            if $\lambda < \lambda_{\min}$ **then**
                $\lambda = 0$
            end if
        end if
    until $c \geq 0$
until converged

A.3 Conjugate Gradient Method

The conjugate gradient method iteratively solves a linear system $\mathbf{Ax} = \mathbf{b}$.

Algorithm 3 Conjugate Gradient Method

$\mathbf{x} \leftarrow \mathbf{x}_0$
$\mathbf{r}_0 \leftarrow \mathbf{A}\,\mathbf{x}_0$
$\mathbf{p}_0 \leftarrow \mathbf{A}\,\mathbf{x} - \mathbf{r}_0$
$k \leftarrow 0$
while $\frac{||\mathbf{r}_k||}{||\mathbf{x}||} < \text{tol}$ **do**
    $\alpha \leftarrow \frac{r_k^T \mathbf{A}\, p_k}{p_k^T \mathbf{A}\, p_k}$
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha\, \mathbf{p}_k$
    $\mathbf{r}_{k+1} \leftarrow \mathbf{A}\,\mathbf{x}_{k+1} - \mathbf{b}$
    $\beta \leftarrow \frac{r_{k+1}^T r_k}{r_k^T r_k}$
    $\mathbf{r}_{k+1} \leftarrow -\mathbf{r}_k + \beta\, \mathbf{p}_k$
    $k \leftarrow k + 1$
end while

Appendix B

Code Examples

This appendix contains the documented implementation of the matrix, vector and forward model interfaces for the usage example from Section 2.

One pecularity arising through the use of template programming in the implementation of the invlib library is that the interfaces for the underlying matrix and vector types depend on the configuration of the retrieval calculations that are performed. Indeed the matrix and vector types need only implement the operations that are finally called when the code is compiled. For example, if the conjugate gradient method is used as a solver for the linear subproblem, the matrix type needs only provide funtions for matrix-vector and transposed matrix-vector multiplication, but no functions for matrix-matrix multiplication. Since invlib also allows the user to specify different types for the covariance matrices as well as the Jacobian, also the required interfaces for these types may differ. Used with the conjugate gradient method, for example, the covariance matrices are not required to provide an implementation of transposed matrix-vector multiplication due to their symmetry. A complete implementation of the interfaces for the matrix and vector types, that is compatible with all computations that can be performed with invlib is provided by the matrix and vector archetypes that are distributed with the invlib code (Pfreundschuh, 2016).

B.1 The Eigen3 Interface

The Eigen3 library is used for the underlying implementation of the matrix and vector types for the usage example. The interfaces for the vector and matrix types are implemented by the `EigenVector` and the `EigenSparse` classes, respectively. To make the code more concise, we begin by introducing type aliases for the Eigen3 types used. We also forward declare the `EigenSparse` class, since it is required by the `EigenVector` class.

```
using EigenSparseBase   = Eigen::SparseMatrix<double, Eigen::RowMajor>;
using EigenVectorBase   = Eigen::VectorXd;

class EigenSparse;
```

B.1.1 The `EigenVector` class

All types that are to be used in an arithmetic expression in invlib are required to declare public types that specify the associated types implementing the matrix algebra. Those are:

`BaseType` : Interfaces in invlib are implemented using inheritance and the type that is being interface is available through the BaseType type alias.

`RealType` : The floating point type used to represent scalars.

`VectorType` : The type used to represent vectors.

`MatrixType` : The type used to represent matrices.

`ResultType` : The result type of a product with an object of this type as right operand.

```
class EigenVector : public EigenVectorBase
{

public:

    using BaseType   = EigenVectorBase;
    using RealType   = double;
    using VectorType = EigenVector;
    using MatrixType = EigenSparse;
    using ResultType = EigenVector;
```

Next, the constructors for the interface class are implemented. For the default constructor `EigenVector()` the compiler-generated default implementation is used, which simply calls the default constructor of the base class. In addition to that, a one-argument constructor that uses perfect forwarding to call the constructor of the base class is defined. This is done for efficiency reasons. Since Eigen3 calculations produce proxy objects that represent the results of matrix and vector computations, these proxy objects can be directly forwarded to the constructor of the `EigenVectorBase` object which avoids unnecessary copy operations.

```
    EigenVector() = default;

    template <typename T>
    EigenVector(T &&t)
        : EigenVectorBase(std::forward<T>(t))
    {
        // Nothing to do here.
    }
```

In addition to that, invlib requires a function to resize a vector to a given size. Since the `EigenVector` is derived of the `EigenVectorBase` class, an implementation of this function is already available from the base class. For completeness the inheritance relation was declared `protected`, which is why a wrapper for the resize function must be reimplemented here:

```
    void resize(unsigned int n)
    {
        this->EigenVectorBase::resize((int) n);
    }
```

Now all that is left to do is implement the mathematical operations required to be

performed on vectors. Those are:

`accumulate` : Add another vector to the given vector

`subtract` : Subtract another vector from the given vector

`scale` : Scale the given vector by a scalar.

`norm` : Compute the norm of the given vector.

`dot` : Compute the dot product of two vectors.

These functions can be implemented in a straight-forward way by simply forwarding them to the corresponding call of arithmetic functions provided by Eigen3 base types:

```
    void accumulate(const EigenVector& v)
    {
        *this += v;
    }

    void subtract(const EigenVector& v)
    {
        *this -= v;
    }

    void scale(RealType c)
    {
        *this *= c;
    }

    RealType norm() const
    {
        return this->EigenVectorBase::norm();
    }
};

double dot(const EigenVector &v, const EigenVector &w)
{
    return v.dot(w);
}
```

B.1.2 The `EigenSparse` class

The interface for the Eigen3 sparse matrix class requires even less functions. As above, we begin with declaring the associated types of the matrix algebra as well as a perfect forwarding constructor.

```
class EigenSparse : protected EigenSparseBase
{

public:

    using BaseType   = EigenSparseBase;
    using RealType   = double;
    using VectorType = EigenVector;
    using MatrixType = EigenSparse;
    using ResultType = EigenSparse;

    template <typename T>
    EigenSparse(T &&t)
        : EigenSparseBase(std::forward<T>(t))
    {
        // Nothing to do here.
    }
```

In addition to that, for the matrix class `rows()` and `cols()` functions returning the number of rows and columns of the matix are required. Again, those could be directly inherited, if the inheritance relation would have been declared `public`.

```
    unsigned int rows() const
    {
        return this->EigenSparseBase::rows();
    }

    unsigned int cols() const
    {
        return this->EigenSparseBase::cols();
    }
```

The arithmetic operations that are required by invlib are matrix-vector multiplication, which is implemented by the `multiply` member function, and transposed matrix-vector multiplication, implemented by the `transpose_multiply` function.

```
    VectorType multiply(const VectorType &v) const
    {
        VectorType w = *this * v;
        return w;
    }

    VectorType transpose_multiply(const VectorType &v) const
    {
        VectorType w = this->transpose() * v;
        return w;
    }

};
```

This concludes the implementation of the interface for the necessary matrix and vector operations. This code can be found in the `MATS` folder in the example directory tree distributed with invlib.

B.1.3 The Linear Forward Model

Now that the matrix arithmetic is in place, the next step is to implement a forward model class that represents the linear model here given only by the Jacobian $\mathbf{K}$:

$$\mathbf{y} = \mathbf{K}(\mathbf{x} - \mathbf{x}_a) \tag{B.1.1}$$

The invlib forward model interface consists of two functions `evaluate` and `Jacobian`. The `evaluate` function takes a state vector $\mathbf{x}$ as argument and returns the corresponding measurement vector $\mathbf{y}$ as predicted by the forward model. The `Jacobian` function takes as arguments a state vector $\mathbf{x}$ and a measurement vector $\mathbf{y}$, returns the Jacobian corresponding to the measurement vector $\mathbf{x}$ and sets the provided vector $\mathbf{y}$ to the current value of the forward model. Since the forward model is passed as a template argument to the `MAP` class, the user is free to choose the exact argument and return types of the two functions. This, for example, allows the user to choose a different matrix representation for the Jacobian than for the other covariance matrices or the general matrix type used for the calculations. The only requirement is that this type can be multiplied from the right with the vector type used for the calculations. Furthermore, the foward model class must provide public

member variables `m` and `n`, which specify the dimensions of the measurement and state spaces, respectively.

Again, we begin by introducing type aliases for conciseness.

```
using MatrixType = invlib::Matrix<EigenSparse>;
using VectorType = invlib::Vector<EigenVector>;
```

The implementation of the `LinearFoward` model class is straight forward. The class simply holds references to a given Jacobian and the a priori vector.

```
class LinearModel
{
public:

    LinearModel(const MatrixType &K_, const VectorType &xa_)
        : K(K_), xa(xa_), m(K_.rows()), n(K_.cols())
    {
        // Nothing to do here.
    }
```

To evaluate the forward model, we simply compute $\mathbf{K}(\mathbf{x} - \mathbf{x}_a)$. Since we are using the `invlib` matrices and vectors, these expressions can be written using operator notation.

```
    VectorType evaluate(const VectorType &x)
    {
        return K * (x - xa);
    }

    const MatrixType & Jacobian(const VectorType &x, VectorType &y)
    {
        y = K * (x - xa);
        return K;
    }
```

Finally, the required public `m`, and `n` members, that contain the dimensions of the inverse problem are added to the class.

```
    const unsigned int m, n;

private:

    const MatrixType &K;
    const VectorType &xa;

};
```

B.2 The MPI Interface

In order to use the Eigen3 matrices and vectors with invlibs generic distributed matrix and vector types, additional functions in the matrix and vector interfaces are required. Those additions to the interfaces are described in this section.

B.2.1 The `EigenVector` class

The invlib distributed vector and matrix types split up vectors and matrices row-wise over processes. The library thus needs to access blocks of rows. For vectors, the corresponding required function is `get_block`:

```
    EigenVector get_block(unsigned int start, unsigned int extent) const
    {
        return this->block((int) start, 0, (int) extent, 1);
    }
```

For the communication between processes using MPI, invlib requires access to the raw data arrays of the vector type. This is provided by the `data_pointer()` member functions for both, mutable as well as immutable access.

```
template <typename Real>
auto MatrixArchetype<Real>::data_pointer()
    -> Real *
{
    return data.get();
}

template <typename Real>
auto MatrixArchetype<Real>::data_pointer() const
    -> const Real *
{
    return data.get();
}
```

This concludes the necessary additions to the `EigenVector` interface class.

B.2.2 The `EigenSparse` class

For this example only one additional function for row-block access must be added to the matrix interface. This function is required to restrict the local matrices of each process to the corresponding row ranges. The `get_block` function is used by invlibs generic `split` function, which takes a full matrix loaded by each process and returns the row block obtained by splitting the matrix evenly over the total number of processes.

```
    EigenSparse get_block(unsigned int row_start,
                          unsigned int col_start,
                          unsigned int row_extent,
                          unsigned int col_extent) const
    {
        return this->block((int) row_start,
                           (int) col_start,
                           (int) row_extent,
                           (int) col_extent);
    }
```

Bibliography

R. C. Aster, B. Borchers, and C. H. Thurber. *Parameter estimation and inverse problems*. Elsevier, New York, 2005. ISBN 0-12-065604-3.

O. M. Christensen, P. Eriksson, J. Urban, D. P. Murtagh, K. Hultgren, and J. Gumbel. Tomographic retrieval of water vapour and temperature around polar mesospheric clouds using odin-smr. *Atmospheric Measurement Techniques*, 8: 1981–1999, 2015. ISSN 1867-8548.

Eigen. Eigen 3. `http://eigen.tuxfamily.org/`, 2016. Accessed: 2016-05-17.

P. Eriksson and S. Buehler. *ARTS Theory*. 2016.

P. Eriksson, C. Jiménez, and S. Buehler. Qpack, a tool for instrument simulation and retrieval work. *J. Quant. Spectrosc. Radiat. Transfer*, 91:47–64, 2005.

J. Gumbel, N. Ahlgren, N. Larsson, and F. v. Schéele. Mats mission definition phase report. *J. Phys. Theor. Appl.*, 6(1):202–241, 2014.

J. Hadamard. Les problèmes aux limites dans la théorie des équations aux dérivées partielles. *J. Phys. Theor. Appl.*, 6(1):202–241, 1907.

L. Hoffmann, M. Kaufmann, R. Spang, R. Müller, J. J. Remedios, D. P. Moore, C. M. Volk, T. von Clarmann, and M. Riese. Envisat mipas measurements of cfc-11: retrieval, validation, and climatology. *Atmospheric Chemistry and Physics*, 8(13):3671–3688, 2008. doi: 10.5194/acp-8-3671-2008. URL `http://www.atmos-chem-phys.net/8/3671/2008/`.

ISO C++. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2012.

Massif. Massif: a heap profiler. `http://valgrind.org/docs/manual/ms-manual.html`, 2016. Accessed: 2016-06-17.

MIT. Mit software license. https://opensource.org/licenses/MIT, 2016. Accessed: 2016-07-16.

MPI. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

D. Murtagh, U. Frisk, F. Merino, M. Ridal, A. Jonsson, J. Stegman, G. Witt, P. Eriksson, C. Jiménez, G. Megie, J. de la Noë, J. Ricaud, P. Baron, J. R. Pardo, A. Hauchcorne, E. J. Llewellyn, D. A. Degenstein, R. L. Gattinger, N. L. Lloyd, W. F. Evans, I. C. McDade, C. S. Haley, C. Sioris, S. von Savigny, B. H. Solheim, J. C. McConnell, K. Strong, E. H. Richardson, G. W. Leppelmeier, E. Kyrölä, H. Auvinen, and L. Oikarinen. An overview of the odin atmospheric mission. *Canadian Journal of Physics*, 80(4):309–319, 2002.

Netlib. Blas (basic linear algebra subprograms). http://www.netlib.org/blas/, 2016. Accessed: 2016-06-17.

J. Nocedal and S. J. Wright. *Numerical optimization*. Springer Series in Operations Research and Financial Engineering. Springer, Berlin, 2006. ISBN 978-0387-30303-1.

OpenMP. OpenMP application program interface version 3.0, May 2008. URL http://www.openmp.org/mp-documents/spec30.pdf.

M. Oppenheimer, M. Campos, R. Warren, G. Birkmann J., Luber, B. O'Neill, and K. Takahashi. Emergent risks and key vulnerabilities. in: Climate change 2014: Impacts, adaptation, and vulnerability. part a: Global and sectoral aspects. contribution of working group ii to the fifth assessment report of the intergovernmental panel on climate change [field, c.b., v.r. barros, d.j. dokken, k.j. mach, m.d. mastrandrea, t.e. bilir, m. chatterjee, k.l. ebi, y.o. estrada, r.c. genova, b. girma, e.s. kissel, a.n. levy, s. maccracken, p.r. mastrandrea, and l.l. white (eds.)]. pages 1039–1099, 2014.

S. Pfreundschuh. invlib, a generic implementation of bayesian methods for inverse problems in remote sensing. www.github.com/simonpf/invlib, 2016. Accessed: 2016-09-26.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

C. D. Rodgers. *Inverse Methods for Atmospheric Sounding: Theory and Practice*. Series on Atmospheric, Oceanic and Planetary Physics. World Scientific, 2000. ISBN 9789814498685.

A. Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, Philadelphia, 2005. ISBN 0898715725;9780898715729;.

J. Ungermann. *Tomographic Reconstruction of Atmospheric Volumes from Infrared Limb-Imager Measurements*. Schriften des Forschungszentrums Jülich, 1th edition, 2011. ISBN 978-3-89336-708-5.

J. Ungermann, J. Blank, M. Dick, A. Ebersoldt, F. Friedl-Vallon, A. Giez, T. Guggenmoser, M. Höpfner, T. Jurkat, M. Kaufmann, S. Kaufmann, A. Kleinert, M. Krämer, T. Latzko, H. Oelhaf, F. Olchewski, P. Preusse, C. Rolf, J. Schillings, O. Suminska-Ebersoldt, V. Tan, N. Thomas, C. Voigt, A. Zahn, M. Zöger, and M. Riese. Level 2 processing for the imaging fourier transform spectrometer gloria: derivation and validation of temperature and trace gas volume mixing ratios from calibrated dynamics mode spectra. *Atmospheric Measurement Techniques*, 8(6):2473–2489, 2015. doi: 10.5194/amt-8-2473-2015. URL `http://www.atmos-meas-tech.net/8/2473/2015/`.