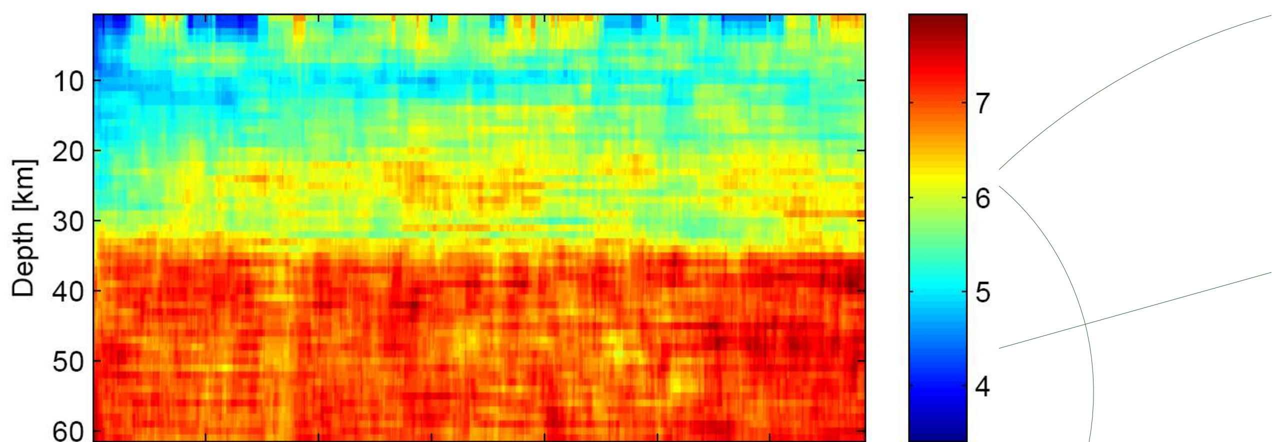
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master thesis in geophysics 2008, *December 15, 2008*

Receiver function modeling

Modeling local subsurface velocity structures using multiple diverse algorithms



Thomas R. N. Jansson
Niels Bohr Institute, University of Copenhagen

SUPERVISORS:
KLAUS MOSEGAARD AND TRINE DAHL-JENSEN

Abstract

Receiver function analysis is a seismic method utilizing that fact that seismic P-waves impinging on a subsurface boundary will result in refracted and reflected P and S-waves. The generated waves are only the result of the local structure and by deconvolution the local effect can be isolated in a data set called the receiver function. Reconstructing the local velocity-structure by comparing the observed receiver functions with synthetically generated receiver functions is usually regarded as a highly non-linear inverse problem. In this thesis I will apply four different algorithms to recreate the subsurface velocity structure from a synthetic data set as well as data collected in Greenland. The algorithms are: Uniform search, the Metropolis algorithm, Neighborhood search and the Levenberg-Marquardt algorithm. Overall, the algorithms were successfully in reconstructing the synthetic data. In the Greenland data set the algorithms generated models indicating a low velocity layer which is most likely an artifact of the difference between the wavelet used in the forward algorithm and the wavelet embedded in the data. This problem affected all the algorithms. For the Greenland data only the Metropolis and the Levenberg-Marquardt algorithm found reasonable models. It was also concluded that the Uniform search and the Neighborhood algorithm do not perform well in high dimensional problems.

Resumé

Receiver function analyse er en seismisk metode, der udnytter, at seismiske P-bølger, der rammer en laggrænse i undergrunden, vil resultere i reflekterede og refrakterede P og S-bølger. Nogle af disse bølger er kun resultatet af en lokal laggrænse i undergrunden og ved affoldning kan information fra den lokale struktur isoleres i et datasæt kaldet en receiver function. Rekonstruktionen af den lokale seismiske hastighedsstruktur ved at sammenligne det observerede datasæt med et datasæt genereret ved modellering er normalt anset for at være et stærkt ikke linært inverst problem. I dette speciale vil jeg anvende fire forskellige algoritmer for at rekonstruere hastighedsstrukturen fra både syntetiske og virkelige data indsamlet i Grønland. Algoritmerne jeg vil bruge er: Uniform search, Metropolis algoritmen, Neighborhood search og Levenberg-Marquardt algoritmen. Overordnet set havde algoritmerne alle succes i at genfinde hastighedsstrukturen i de syntetiske data. I datasættet fra Grønland fandt alle algoritmerne et lavhastighedslag som højst sandsynligt var en effekt af en forkert wavelet i forward modellen. Dette problem påvirkede alle algoritmerne. I det Grønlandske datasæt var det kun Metropolis og Levenberg-Marquardt algoritmerne som gav fornuftige resultater. Endvidere kunne det også konkluderes at Uniform og Neighborhood search algoritmerne var uegnet til høj dimensionale problemer.

Acknowledgment

I would like to thank Klaus Mosegaard, Niels Bohr Institute, University of Copenhagen and Trine Dahl Jensen, GEUS, for their guidance, advice and great help throughout the process. Also, I would like to thank Bo Holm Jacobsen, University of Aarhus for providing me with a forward model to generate the receiver functions. This forward model has formed the basis of all my numerical work. I would also like to thank Kåre Hartvig Jensen and Stine Kildegaard Poulsen for proofreading the manuscript and giving me many helpful suggestions. Additionally, I would like thank the people at the office for many fruitful discussions and good company. Finally I wish to thank my family and especially Pernille for supporting me throughout the process.

Thomas R. N. Jansson
Niels Bohr Institute, December 2008

Preface

It should be stated that I have no geological background or education in interpreting whether or not the results I obtain are geological sound. In this light I have chosen to focus on the numerical and theoretical work related to receiver function modeling.

I have decided to illustrate as many of the subjects I have covered in the theoretical chapters as this may serve to make the text easier to read and in certain cases gives an invaluable intuitive understanding of the subject. In the result chapter the density of information in the figures is quite high but I decided on this packing of information so that relevant figures remained grouped such that I would not end up with half-filled pages. The figures in the result section are all vector-graphics makeing it possible to zoom to any detail level in the digital version.

It is my objective in this thesis to investigate how the different algorithms perform facing the complicated problem of fitting synthetic data to observed data. A receiver function analysis of Greenland has already been done previously using other methods and I hope that this work will provide some insight into the advantages and disadvantages of the four algorithms on receiver function modeling.

Originally, I was using two data sets from Greenland but since I discovered a problem in the last few days of the thesis work I did not have the time to process both data sets with the rewritten algorithms.

Contents

Frontmatter **2**
Abstract . . . 2
Acknowledgment . . . 3
Preface . . . 3

1 Introduction **6**

2 Receiver functions **8**
2.1 Basic seismic . . . 8
2.1.1 Rotation of data . . . 10
2.2 What is a receiver function? . . . 11
2.3 Calculating the receiver function . . . 12
2.3.1 An example with three arrivals . . . 13
2.3.2 Deconvolution . . . 15
2.4 The wavelet . . . 16

3 Greenland data **18**
3.1 Retrieving the receiver function from observed data . . . 18
3.1.1 Stacking . . . 19
3.2 Data from Greenland . . . 20

4 Inverse problems **22**
4.1 Inverse problem theory . . . 22
4.1.1 The forward problem . . . 23
4.1.2 The inverse problem . . . 25
4.1.3 Probabilistic formulation . . . 25

5 Algorithms **28**
5.1 The Monte Carlo Family . . . 28
5.2 Uniform sampling . . . 30
5.3 The Metropolis algorithm . . . 31
5.4 The Neighborhood search algorithm . . . 33
5.4.1 Voronoi cells . . . 34
5.4.2 The behavior of the Neighborhood search algorithm . . . 34
5.4.3 Exploring the Voronoi cell . . . 35
5.4.4 Sampling the cell using Qhull . . . 36
5.4.5 Sampling the cell using discretized axis . . . 36
5.4.6 Sampling the cell using exact intersections . . . 38
5.4.7 Changing the misfit function . . . 39
5.5 The Levenberg-Marquardt algorithm . . . 40
5.5.1 Finding the minimum of S . . . 41
5.5.2 Levenberg's contribution . . . 42

5.5.3 Marquardt's contribution . . . 42

6 Algorithm and plotting considerations 44
6.1 Plotting the results . . . 44
6.2 An insight into the complexity of the problem . . . 46
6.3 General description of the algorithms used on data . . . 47
6.3.1 Uniform search . . . 47
6.3.2 Metropolis . . . 47
6.3.3 Neighborhood search . . . 49
6.3.4 Levenberg-Marquardt . . . 49

7 Results 50
7.1 Description of the plots . . . 50
7.2 Synthetic data . . . 50
7.2.1 Uniform search . . . 50
7.2.2 Metropolis . . . 53
7.2.3 Neighborhood search . . . 53
7.2.4 Levenberg-Marquardt . . . 53
7.3 Data from station Nord . . . 53
7.3.1 Uniform search . . . 53
7.3.2 Metropolis . . . 56
7.3.3 Neighborhood search . . . 56
7.3.4 Levenberg-Marquardt . . . 56

8 Discussion 57
8.1 Comparison of algorithm performance on synthetic data . . . 57
8.2 The wavelet problem . . . 57
8.3 The importance of N_s in the Neighborhood search algorithm . . . 59
8.4 A non-linear problem . . . 61
8.5 Removing the P-pulse from the receiver function . . . 62

9 Conclusions 63
9.1 Further work . . . 64

Bibliography 65

A Mathematics ii
A.1 Convolution theorem . . . ii
A.2 Approximate Hessian . . . ii
A.3 The distance from a line to a point in 5 dimensions . . . iii
A.4 The distance from a line to a point in N_{dim} dimensions . . . iv

B Code vi
B.1 Uniform Search . . . vi
B.2 Metropolis . . . vii
B.3 Neighborhood search – Discretizing the axis . . . xi
B.4 Neighborhood search – Exact intersection . . . xvi
B.5 Levenberg-Marquardt . . . xix

Chapter 1

Introduction

Conventional geological mapping of subsurface structures is done by studying reflected waves from artificially or naturally generated disturbances such as dynamite or earthquakes. This is a costly affair since large arrays of geophones need to be deployed and later collected in order to obtain the data. In remote or harsh areas such as Greenland it is very expensive and a logistical challenge to deploy a large array of geophones needed for reflection seismology. Such arrays do not exist but seismic data are being recorded in local seismic stations placed strategically along the coast and on the ice shelf itself.

In this thesis I will present a method utilizing earthquake data recorded at a single station to recover the subsurface velocity structure. The method is called receiver function analysis. When an earthquake is recorded at a seismic station the seismic data contain information on the source structure, the propagation through the earth's mantle and the local structure beneath the seismic station. Receiver function analysis is a method to remove the information in the seismic data regarding the source structure and the propagation through the mantle in order that the final data set, the receiver function, only contain information about the local structure beneath the seismic station.

Seismic data contain both noise both from the measurements and from effects not incorporated into standard geological models. The structure beneath a seismic station is most likely inhomogeneous and contains a continuously varying stratification with different rock types. A theoretical model of the earth can only approximate the structure to a certain degree and must do this using a finite amount of model variables. The process of finding the model parameters fitting the observed data is known as an inverse problem.

In this thesis I will present four numerical approaches to reconstruct the observed waveform. The algorithms are the Uniform search algorithm, the Metropolis algorithm, the Neighborhood algorithm and the Levenberg-Marquardt algorithm. The first three of these algorithms are members of the Monte Carlo family of algorithms which uses random numbers to generate models. The Levenberg-Marquardt algorithm on the other hand evaluates gradients to navigate between the models. To test the algorithms I started out by trying to recreate synthetic data and later when the algorithms were reproducing the synthetic data convincingly, I endeavored into the realm of real-world data sets

collected in Greenland.

The thesis will start off with a theoretical introduction to receiver functions and the four algorithms. This is followed by a description of the synthetic and real data, the implementation of the algorithms and the presentation of the results. In the end I will discuss some of the troubles relevant to this work and finally try to summarize the results I have obtained and compare them to previous studies.

Chapter 2

Receiver functions

In this chapter I will describe the most basic seismic terms relevant to receiver functions and after that introduce the concept of receiver functions.

2.1 Basic seismic

In order to understand what a receiver function is some basic seismic terms have to be introduced. The structure of the earth has been studied for over the last 200 years and the overall structure is known from analyzing recorded seismic signals. The interior structure of the earth is stratified as illustrated in figure 2.1 and the layers are defined by their chemical or rheological properties. The earth has a mean radius of 6371 km and the three major layers are the crust, mantle and core. These layers can be subdivided many times, but most important to this thesis is the boundary from the crust to the mantle. This boundary was found in 1909 by a Croatian seismologist, see (Mohorovičić, 1909), who discover an abrupt increase in seismic velocity in a depth estimated to be 54 km by analyzing seismic data from an earthquake recorded in Croatia. This boundary is now known as the Mohorovičić discontinuity or Moho. It is typically found in depths around 30-60 km beneath the continents and 5-9 km beneath the oceans. Later I will analyze data collected in Greenland where a recent study investigated the depth to the Moho using receiver functions, see (Dahl-Jensen et al., 2003).

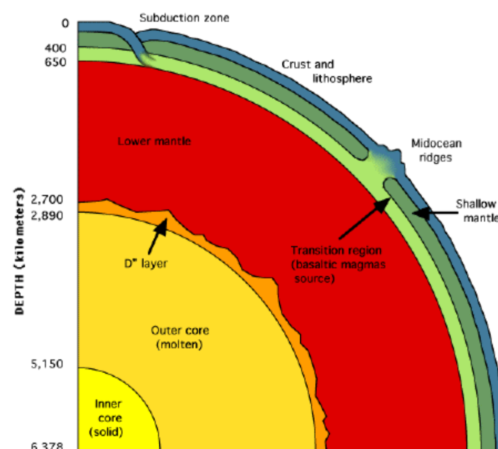
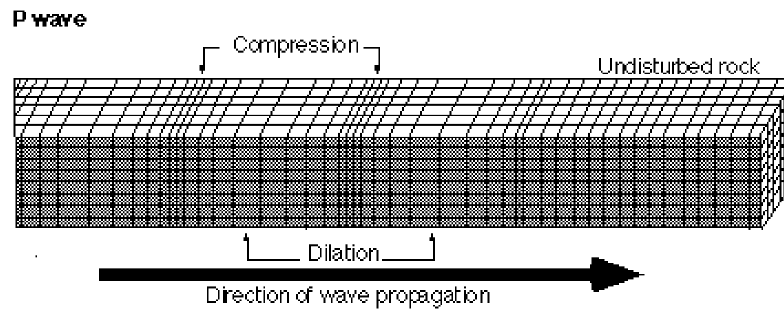
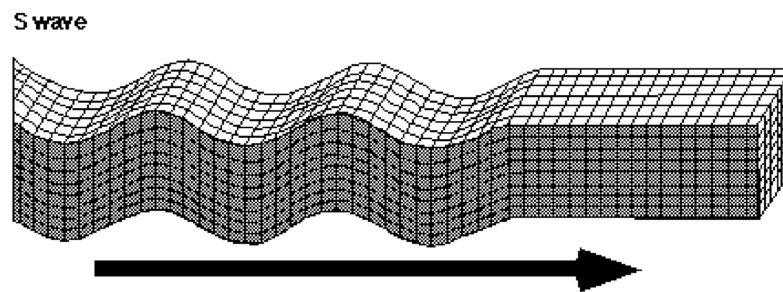


Figure 2.1: *A simplified structure model of the Earth. The Moho is usually found in a depth of 30-50 km.*

When an earthquake occurs the ground is suddenly shifted inside the earth. The coordinates of the point-like earthquake is called a hypocenter and the



(a) P-wave



(b) S-wave

Figure 2.2: *Diagrams of P and S-waves. The P-wave is a pressure wave and the fastest seismic wave. The particle motion is in the direction of the wave. The S-wave or shear wave is slower and the particle motion is perpendicular to the direction of the wave propagation. Typical waves velocities are in the order of km/s.*

projection on to the surface is called a epicenter. When an earthquake occurs movements at the hypocenter create waves in the ground. There are two basic types of waves namely P-waves and S-waves. The P-wave or primary wave, see figure 2.2(a), is a compressional pressure wave with volumetric disturbances much like sounds waves. The P-wave is the fastest wave-type and hence the first to arrive at the seismic station. The S-waves, see figure 2.2(b), is called a shear wave, since it contains only shearing deformation without change in volume. The particle motion in an S-wave is perpendicular to the direction of the propagation of the wave and S-waves cannot exist in fluids. Furthermore S-waves are the second fastest wave type and for the same reason called secondary waves.

Seismic data are recorded as a time series of the movement of the earth. Such a time series is called a seismogram and usually the movement in three directions, vertical Z, east-west EW, and north south NS is recorded. Figure 2.3 shows one component of typical seismogram and illustrates a third type of waves arriving later than both the P and S-waves. These waves are a complex combination of both P and S-waves and are called surface waves since they

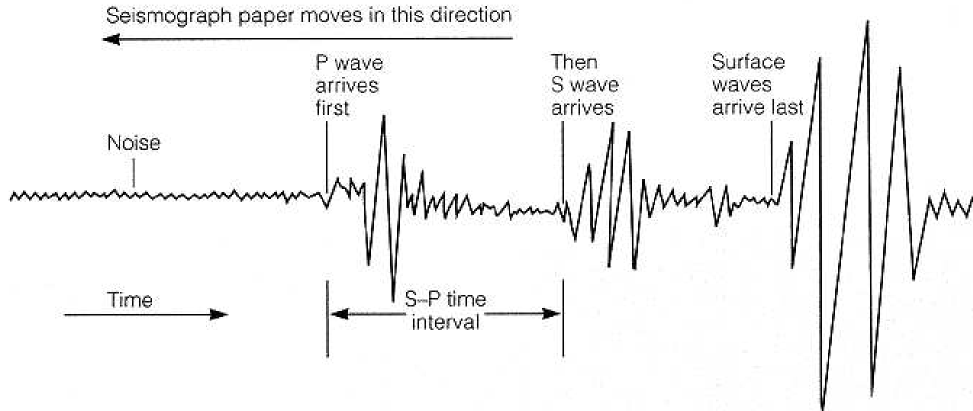


Figure 2.3: *A typical seismogram recorded on paper moving at a constant speed. Today all seismograms are recorded digitally. A seismogram is a record of the ground movement recorded by a seismometer. The first arrival is the P-wave due to the large speed followed by the slower S-waves and finally the large but much slower surface waves.*

are trapped to the surface. The surface waves travel slower than both P and S-waves and are not important when working with receiver functions.

Earthquakes that occur further away than 1000 km from the seismic station are termed teleseismic events. The wavefront from a teleseismic wave can be approximated as a plane wave at the seismic station due to the large distance from the hypocenter. When dealing with plane waves a ray description of the waves is often useful. A ray represents the normal to the wavefront and is pointing in direction of the propagation. For a plane wave the rays will be parallel.

2.1.1 Rotation of data

The seismometer records data along the vertical Z, North-South N, and East-West E directions; the ZNE rotation system. However, the raw three-component data are not aligned in the axis of the earthquake and the energy in form of various wave types will be found in each of the recorded components.

There are two rotation systems commonly used. A 2D rotation system called ZRT and 3D rotation system called LQT. The ZRT rotation is a 2D rotation where the Z component is still pointing in same direction as in the original ZNE recording, see figure 2.4. The two horizontal components N and E are rotated into the radial R and tangential T components in the following way

$$\begin{bmatrix} R \\ T \\ Z \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} E \\ N \\ Z \end{bmatrix}$$

where $\theta = 3\pi/2 - \xi$ and ξ is the back azimuth. The back azimuth describes the angle between the vector pointing from the seismic station to the earthquake and the vector pointing from the seismic station to the north. The general

rotation matrix for both ZRT and LQT is described in (Upadhyay, 2004, p. 197). The ZRT rotation is used in the forward model.

The 3D rotation LQT is related to the ZRT system. Beside being a 2D rotation the coordinate system is also tilted such that the radial component accounts for angle of incidence. In other words, the LQT is given such that L is in the direction of the direct P-wave and contains mainly P-wave energy. The Q component is perpendicular to both L and T and contains mainly shearing motion, SV energy and finally the T component is perpendicular to both L and Q and contains mainly SH energy. This rotation should isolate the energy from the different wave types more clearly.

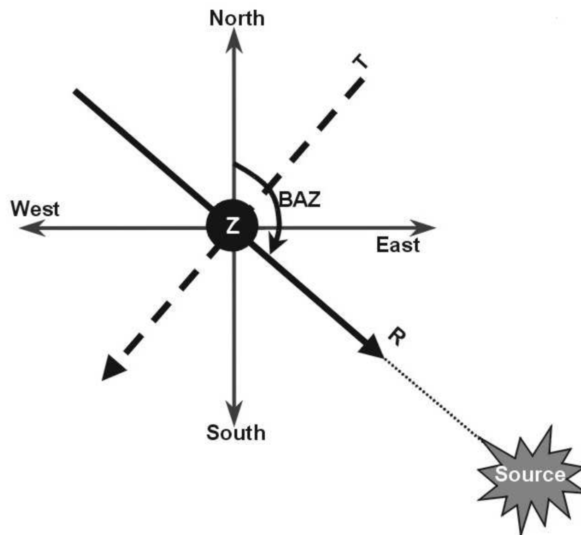


Figure 2.4: *The rotation from ZNE to ZRT. Modified from (Ekrem, 2002).*

2.2 What is a receiver function?

When a teleseismic wave arrives at a seismic station it contain information from the different parts of the path traveled. The recorded seismic data contains information on the seismic source structure, the propagation through the mantle and the local structure beneath the seismic station. Receiver function analysis is a method to remove information from seismic data, regarding source structure and the propagation through the mantle in order that the final data set, the receiver function, only contains information about the local structure beneath the seismic station.

One of the first articles dealing with receiver functions is (Phinney, 1964) concerning P-waveform modeling on spectral responses from the crust. Phinney did his work in the frequency domain but later in the year of 1979 Langston introduced the receiver function modeling in the time domain, see (Langston, 1979). One of the most cited articles on receiver functions is (Ammon, 1991).

In the following below I will describe receiver function analysis. However this description does not explain how synthetic and real receiver functions used in the thesis is created. The creation of synthetic receiver functions are described in section 4.1.1 and the receiver functions from the real world are described in chapter 3.

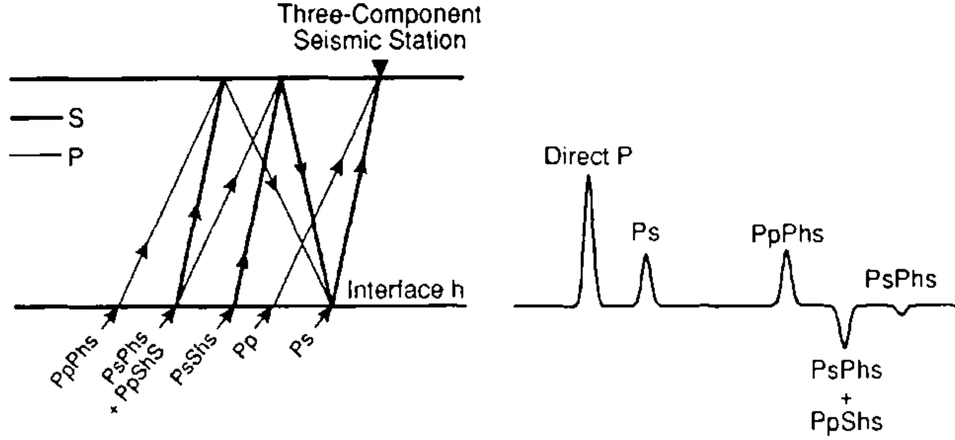


Figure 2.5: *left: Simplified schematics of a teleseismic earthquake arriving at a seismic station. Only the direct P-wave is shown and waves that end as S-waves at the seismic station since the other P-waves are removed when creating the receiver function. Right: The corresponding simplified receiver function showing the converted P-waves at layer over a half space. Upper case letters indicate upgoing motion and lower case letters indicate downward motion. h indicates the reflection at the surface h. E.g. PpPhs means an upgoing P-wave that is reflected in surface and bounce downward to the interface where it is reflected as a S-wave. Modified after (Ammon, 1991).*

2.3 Calculating the receiver function

The underlying theory for creating synthetic seismograms is the linear filter theory described in (Lay and Wallace, 1995). In this theory a seismogram can be seen as the output from a sequence of linear filters on the signal from a seismic source. These filters represent different processes such as the propagation through the earth or the recording processes at the seismic station.

The output or filter response to an instantaneous impulse, a delta function, is termed $f(t)$ and the Fourier transform $F(\omega)$. The response of the filter to an arbitrary input $x(t)$ is called $y(t)$ and can be written as

$$y(t) = f(t) * x(t) \quad \Rightarrow \quad Y(\omega) = F(\omega)X(\omega)$$

where $Y(\omega)$ is given in the frequency domain by the convolution theorem, see appendix A.1. Here $*$ is the convolution operator, $Y(\omega)$ is the Fourier transform of $y(t)$ and $X(\omega)$ is the Fourier transform of $x(t)$. If the signal encounters a series of filters $f_1(t), \ldots, f_n(t)$ the Fourier transform of the output is

$$Y(\omega) = F_1(\omega) \cdots F_n(\omega)X(\omega).$$

That is, the product of filters in frequency domain multiplied with the input signal.

With this in place we can formally describe the receiver function following (Ammon, 1991). The earth's response to an incoming wave on a one-dimensional velocity structure, illustrated in figure 2.5, can be written as two

components of motion. The vertical motion $Z(t)$, and the radial motion $R(t)$ can be written as

$$\begin{aligned} R(t) &= \sum_{k=0}^{n} r_k s(t - t_k) \\ Z(t) &= \sum_{k=0}^{n} z_k s(t - t_k) \end{aligned}$$

where $s(t)$ is the source signal time function, z_k and r_k are the amplitudes of k'th ray on each component, t_k is the arrival time of the k'th ray at the surface and ω is the radial frequency. The sums are over n rays with $k = 0$ being the direct P-wave.

Assuming that $s(t - t_k)$ is a delta function the Fourier transforms can be written as

$$\begin{aligned} R(\omega) &= r_0 \sum_{k=0}^{n} \hat{r}_k e^{-i\omega t_k} \\ Z(\omega) &= z_0 \sum_{k=0}^{n} \hat{z}_k e^{-i\omega t_k} \end{aligned}$$

where $\hat{z}_k = \frac{z_k}{z_0}$ is the amplitude of the k'th ray normalized by the amplitude of the first ray z_0 which is the direct P-wave and similarly for $\hat{r}_k$. The deconvolution of the vertical motion from the radial can be written as

$$H(\omega) = \frac{S(\omega)R(\omega)}{S(\omega)Z(\omega)} = \frac{R(\omega)}{Z(\omega)} \quad (2.1)$$

where $S(\omega)$ is the Fourier transform of source signal $S(\omega)$ and $Z(\omega)$ are assumed non-zero. This assumption does not work for real data and the solution to the problem is described in section 2.3.2. $H(\omega)$ is the Fourier transform of the radial receiver function $h(t)$. It also possible to construct a tangential receiver function by performing the division in the frequency domain between tangential and vertical components instead. In this thesis I will only focus on the radial receiver function and all references to the receiver function will be to the radial receiver function.

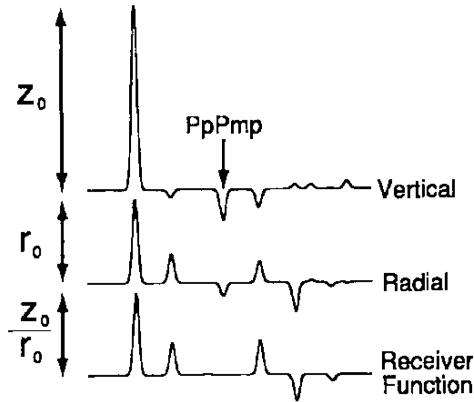


Figure 2.6: *The vertical and radial response to an incoming signal and the resulting receiver function. Notice that the PpPmp wave is not present in the receiver function. From (Ammon, 1991).*

2.3.1 An example with three arrivals

Consider a teleseismic wave arriving at the seismic station above a single layer. Since the wave is teleseismic it can be seen as a plane wave. Following (Ammon, 1991) I will present a example where a plane wave arrives underneath a seismic station. Considering only the first three distinct arrivals on the seismic station.

The direct P-wave, a P-wave reflected at the interface and finally a P-wave that have been converted into a S-wave at the interface. When a P-wave impinges on a boundary to a layer with a different seismic velocity the wave will be reflected and refracted in the boundary. If the layer is not fluid a secondary S-wave will be generated, see (Lay and Wallace, 1995, p. 97). For teleseismic waves the converted S-wave is much stronger in the horizontal component than in the vertical component of the recorded data. This difference in strength of signal in the horizontal and the vertical components form the core of the method.

The Fourier transform of the motion in the radial and vertical direction from the three arrivals can be written as

$$R(\omega) = r_0 \left(1 + \hat{r}_p e^{-i\omega t_p} + \hat{r}_s e^{-i\omega t_s}\right) \tag{2.2}$$

$$Z(\omega) = z_0 \left(1 + \hat{z}_p e^{-i\omega t_p} + \hat{z}_s e^{-i\omega t_s}\right). \tag{2.3}$$

Here the first term inside the parenthesis corresponds to the direct P-wave, the second term corresponds to the reflected P-wave and the final term corresponds to the S-wave. The main part of S-wave energy is given in the horizontal component hence it is clear that, $\hat{z}_s$, the amplitude of the S-wave relative to the amplitude of the direct P-wave, must be much less than one $\hat{z}_s \ll 1$. Using this together with equation 2.1, 2.2 and 2.3 yields

$$H(\omega) = \frac{r_0}{z_0} \frac{1 + \hat{r}_p e^{-i\omega t_p} + \hat{r}_s e^{-i\omega t_s}}{1 + \hat{z}_p e^{-i\omega t_p}}$$

For a plane wave arriving at a horizontal interface the rays are parallel and $\hat{r}_p = \hat{z}_p$

$$H(\omega) = \frac{r_0}{z_0} \frac{1 + \hat{z}_p e^{-i\omega t_p} + \hat{r}_s e^{-i\omega t_s}}{1 + \hat{z}_p e^{-i\omega t_p}} \tag{2.4}$$

Remembering that $\hat{z}_p$ is the relative amplitude of the reflected P-wave to the direct P-wave and is usually also small. By calculating the Taylor expansion of the denominator

$$\frac{1}{1 + \hat{z}_p e^{-i\omega t_p}} = 1 - (\hat{z}_p e^{-i\omega t_p}) + (\hat{z}_p e^{-i\omega t_p})^2 - (\hat{z}_p e^{-i\omega t_p})^3 + \ldots$$

and inserting only the terms with a order less than one in equation 2.4

$$H(\omega) = \frac{r_0}{z_0} \left(1 + \hat{r}_s e^{-i\omega t_s} + \hat{z}_p e^{-i\omega t_p}\right)\left(1 - \hat{z}_p e^{-i\omega t_p}\right)$$

ignoring the terms of order greater than zero in this multiplication yields

$$H(\omega) = \frac{r_0}{z_0} \left(1 + \hat{r}_s e^{-i\omega t_s}\right)$$

translated back into the time domain

$$h(t) = \frac{r_0}{z_0} \left(\delta(t) + \hat{r}_s \delta(t - t_k)\right)$$

From this equation it is clear that only the direct P-wave and the converted S-wave are present in the receiver function. This was done for a single layer but can be extended to any number of layers, (Ammon, 1991).

2.3.2 Deconvolution

The deconvolution used in equation 2.1 can be troublesome in cases where the denominator is very small or zero. This can be avoided by using a method known as the water-level deconvolution (Clayton and Wiggins, 1976). In the water-level method the small values in the denominator of equation 2.1 are replaced with a fraction of the maximum value in the denominator, see figure 2.7. Replacing small values by larger ones permits the avoidance of division with small or zero numbers which reduces the effect of small frequency elements on the receiver function and makes the process numerical stable.

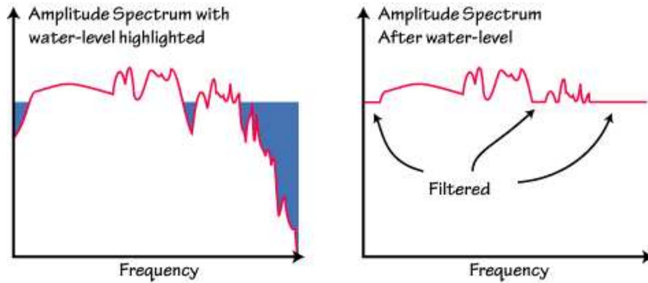


Figure 2.7: *The water level deconvolution method. From (Ammon, 1997).*

Remembering that the inverse of a complex number can be written as $\frac{1}{Z} = \frac{Z^*}{ZZ^*}$ where $*$ indicates the complex conjugated the receiver function can be rewritten to

$$H(\omega) = \frac{R(\omega)Z^*(\omega)}{Z(\omega)Z^*(\omega)}.$$

The water-level method substitutes this with

$$H(\omega) = \frac{R(\omega)Z^*(\omega)}{\phi(\omega)}G(\omega)$$

where

$$\phi(\omega) = \max\left[Z(\omega)Z^*(\omega), c\max\{Z(\omega)Z^*(\omega)\}\right]$$

and

$$G(\omega) = \xi \exp\left(\frac{-\omega^2}{4a^2}\right). \tag{2.5}$$

Here c sets the water level or minimum amplitude allowed in the denominator , ξ is normalization constant and $G(\omega)$ is a Gaussian filter with a width of a and chosen such that the width of the Gaussian match the width of the direct P-wave as in (Langston, 1979, p. 4754).

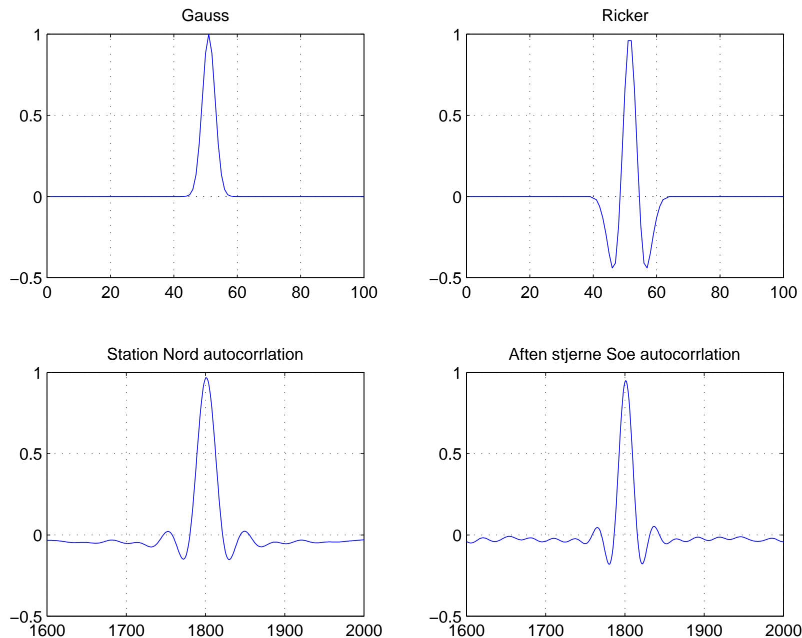


Figure 2.8: *Top left: A synthetic Gaussian wavelet. Top right : A synthetic Ricker wavelet. Bottom left: The autocorrelation of the L component of the data from Station Nord. Bottom right: The autocorrelation of the L component of the data from another Greenland station called Aften Stjerne Sø.*

2.4 The wavelet

Another important aspect of receiver functions are wavelets. Recorded seismograms can be seen as the impulse response of the earth convoluted with a seismic wavelet representing the initial movement at the source. As mentioned, the earth's impulse response is the signal that would have been recorded if the seismic wavelet was a delta function. This can formally be written as

$$R(t) = w(t) * e(t) + n(t).$$

where $R(t)$ in this example is the radially recorded seismogram, $w(t)$ is the wavelet, $e(t)$ is the impulse response of the earth, $n(t)$ represents the noise and $*$ means convolution. Examples of wavelets can be seen in figure 2.8 and the convolution of the impulse response with a wavelet is illustrated in figure 2.9.

The cross correlation of two signals is a measure of how well they resemble one another. The autocorrelation is the cross correlation of a signal with itself and by finding the autocorrelation of one of the three components of the recorded seismic time series and assuming that the impulse response and noise are not correlated it is possible to isolate the wavelet embedded in the seismogram. This is done in figure 2.8 where the wavelet in the data from station Nord and Aften Stjerne Sø are shown.

The general theory of convolution and wavelets is covered and richly illustrated in (Yilmaz and Doherty, 1987). In figure 2.8 four different wavelets are

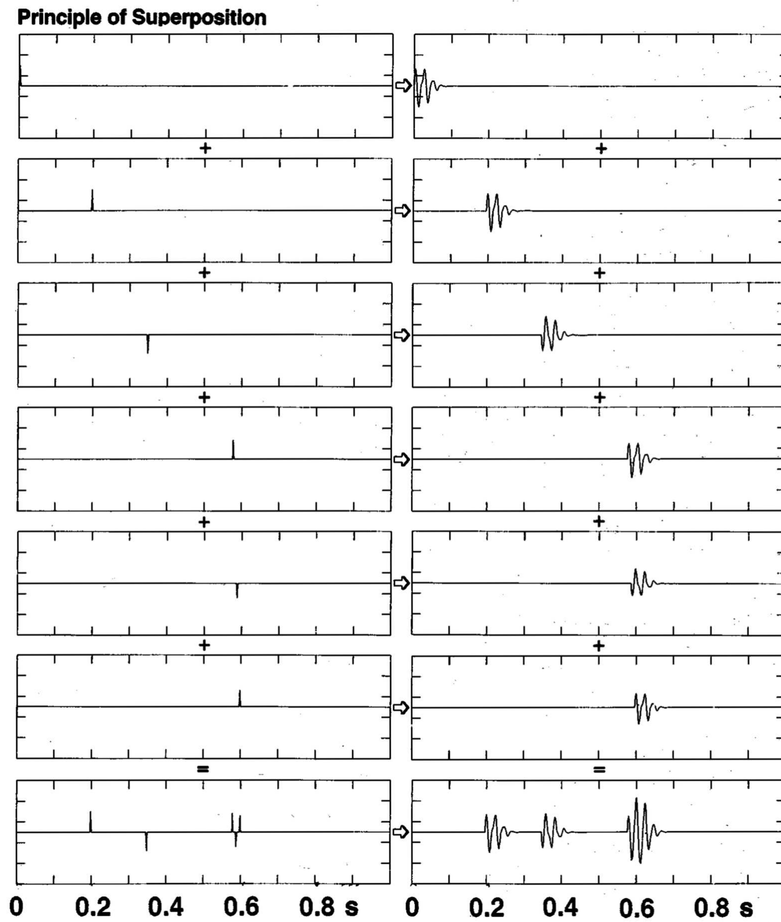


Figure 2.9: *A wavelet traveling through the earth will encounter several reflectors and each of the these will sum up to the seismogram seen in the bottom right corner. This also called the principle of superposition. From (Yilmaz and Doherty, 1987, p. 91). On the left side the earth impulse function is displayed and on the right side the wavelet is shown.*

shown. There is two synthetically generated wavelets namely the Ricker and the Gaussian wavelets beside to the wavelets embedded in the two Greenland data sets from station Nord and Aften Stjerne Sø. Later in this thesis the difference between the wavelets in the Greenland data sets and the Gaussian wavelet will prove important when interpreting the results.

Chapter 3

Greenland data

In this chapter I will introduce the data used in this thesis. The synthetic data is generated with the forward model provided by (Jacobsen, 2008). The real world data are based on data from from station Nord from Northern Greenland which was provided to me by Trine Dahl-Jensen, GEUS.

3.1 Retrieving the receiver function from observed data

At a seismic measuring station the movement of the earth is recorded in three components: vertical Z, North-South NS, and East-West EW. For a given earthquake the location of the hypocenter can be determined by comparing the difference in P and S-wave arrivals at several stations around the world. Using this difference the related travel time can be used to triangulate the epicenter. As mentioned earlier earthquakes with distance further than 1000 km away are termed teleseismic. Teleseismic waves can be approximated by plane waves arriving almost vertically beneath the seismic station.

Once data have been collected it is common to filter the data such that only data with a frequency in a certain domain are allowed. The filtering is done to exclude noise and only detect the earthquake data. As an example the data from Station Nord, described below, is within a $2 - 50Hz$ frequency range.

Once three component data are recorded at the seismic station a sequence of operations on the data is needed to get a receiver function. The data should be rotated in to the axes related to the energy directions of the earthquake since this will ease the isolation of the energy related to the secondary waves, see section 2.1.1. Furthermore the radial and vertical components will be deconvoluted to remove information from the path through the earth up to the subsurface layer and instrumentation information as described in chapter 2. In the following I assume that the underlying structure is laterally homogeneous since inclined layers would complicate the problem further.

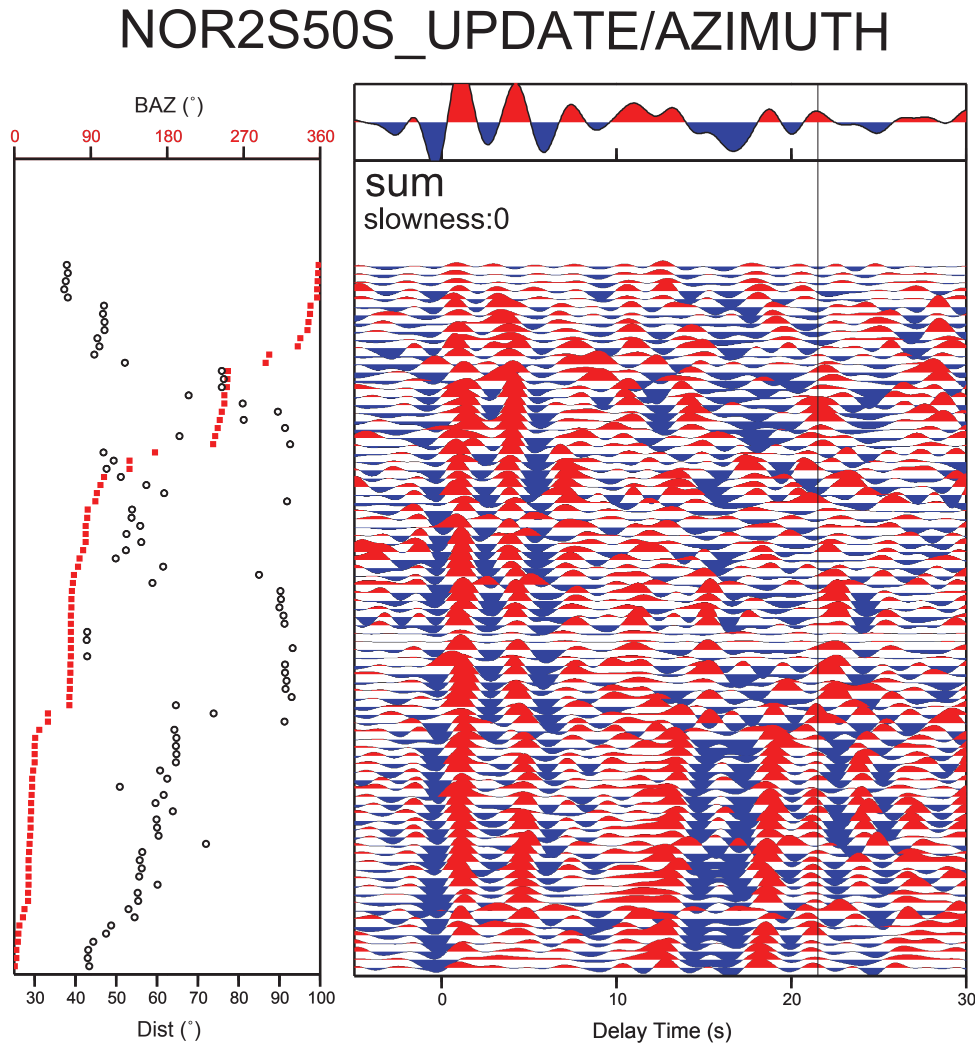


Figure 3.1: *The stacked receiver function for Station Nord. The left figure shows the angular distance to the earthquake in degrees and the back azimuth angle which provides the longitudinal transverse directions of the incoming earthquake seen as a ray. The right figure shows the related earthquake receiver functions.*

3.1.1 Stacking

During the operation of a seismometer at a seismic station several earthquakes will be recorded. Once teleseismic events have been identified the receiver functions have to be calculated for the selected seismograms. For different teleseismic events the seismograms will be stacked as in figure 3.1. By stacking the teleseismic events common elements will be enhanced through constructive interference and random noise canceled out through destructive interferences such that the signal to noise ratio will be enhanced.

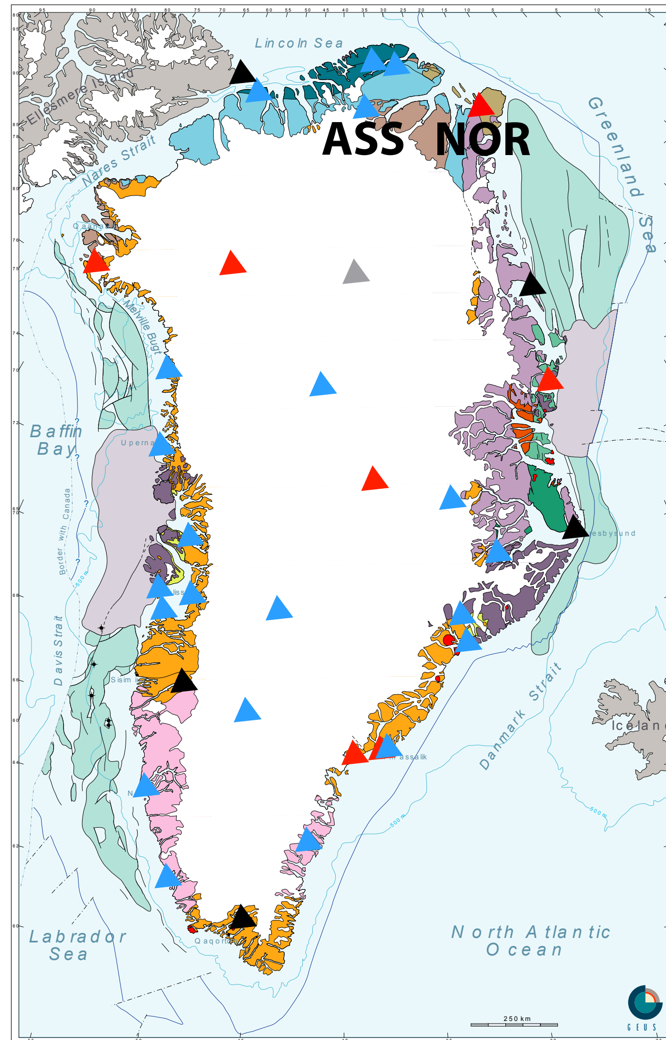


Figure 3.2: *Simplified map of Greenland. Station NORD (NOR) is located in northernmost part of Greenland.*

3.2 Data from Greenland

I have tested my algorithms with data collected in the Northern Greenland at Station Nord (NOR), see figure 3.2. The seismic stations in Greenland are placed in areas with the least background noise away from ice flow and ice edges, see (Dahl-Jensen et al., 2003).

To make the synthetic and real data resemble each other the best solution would be to generate synthetic data using the wavelet embedded in the recorded data. However the forward model used did not support anything except Gaussian wavelets, so I had to determine the Gaussian wavelet that fitted the width

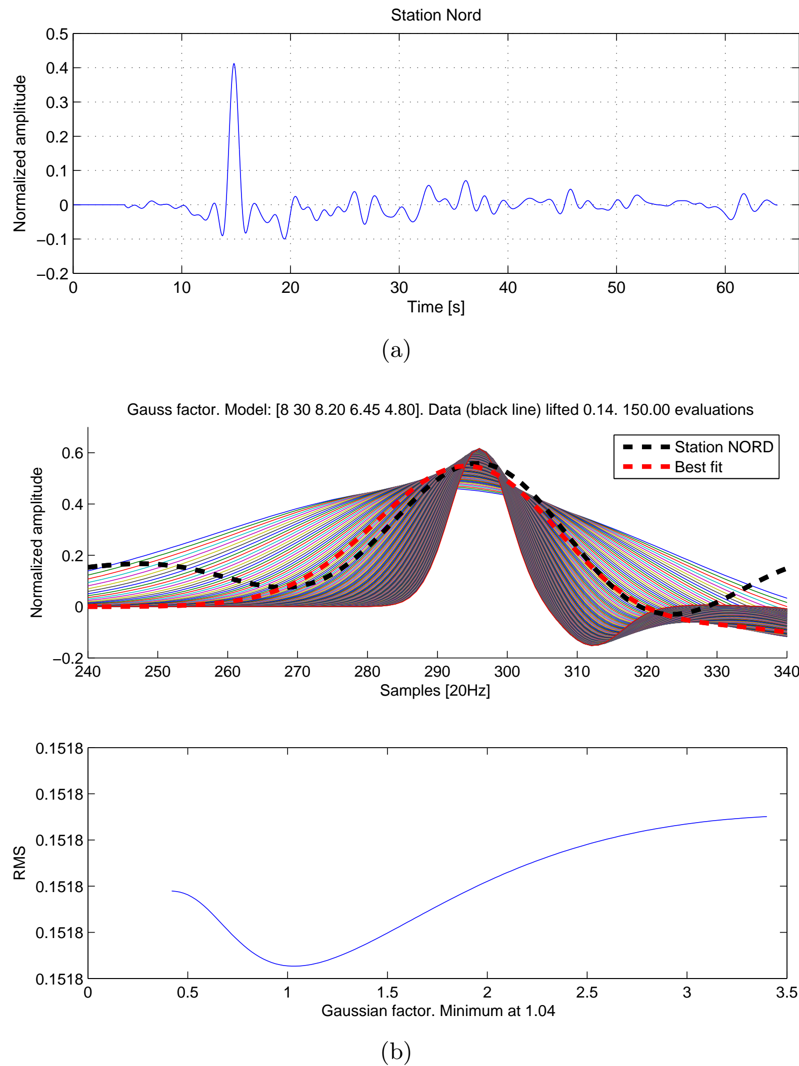


Figure 3.3: *a) The receiver functions from station Nord. b) 150 models width varying Gaussian factors to determine the value at which the data fit was best.*

of the embedded wavelet best. This was done by altering a Gaussian factor in the forward model. The Gaussian factor determines the width of the Gaussian bell that generates the wavelet convolved on the data and hereby also the width of the initial P-arrival. I did this by searching for the Gaussian factor that would make the observed and synthetic data fit best together.

The synthetic data was based on the best model Trine Dahl-Jensen found in her analysis (Dahl-Jensen, 2008). The Gaussian factor was then variated (a in equation 2.5). This was done for station Nord see figure 3.3(b). Afterwards, the Gaussian factor was used in the algorithms when generating synthetic data.

Chapter 4

Inverse problems

One of the main objectives of this thesis is to examine how different optimization schemes apply to the receiver function modeling problem. The problem at hand is what normally is known as an inverse problem, where data are observed and the problem is how to find the model parameters to recreate the data synthetically.

Working with data from the real world can be a daunting task, since the signal can be contaminated with all sorts of noise and instrumental influences. Synthetic data generated from a known model has the advantage that the programmer always is aware that a global minimum exists. This is important for validating algorithms but seldom represents the real world. When working with data from the real world one will rarely be in a position where the mere existence of a global minimum is defined. Often several islands in the model space will contain models that fit data equally well. Using a priori information about the system can help to reduce the amount of possible models by bounding the model within physical reasonable limits.

Generating synthetic data is the opposite process of doing inverse modeling and is called a forward problem. In a forward problem the model and model parameters are known and one wishes to generate data based on the model. Programming a new forward model is beyond the scope of this thesis and fortunately I was kindly provided with a MATLAB based forward model from Bo Holm Jacobsen, (Jacobsen, 2008).

4.1 Inverse problem theory

The majority of physics courses at the university teach how to solve forward problems. An example of a forward problem would be to calculate the electromagnetic field in a given distance from an inhomogeneous conducting rod. If every impurity was mapped and the current was known everywhere inside the rod the electromagnetic field could be calculated uniquely for every point outside the rod. A related inverse problem to this problem would be to determine the structure of the rod by measuring the electromagnetic field outside the rod. Doing this would be a harder problem than the forward problem as

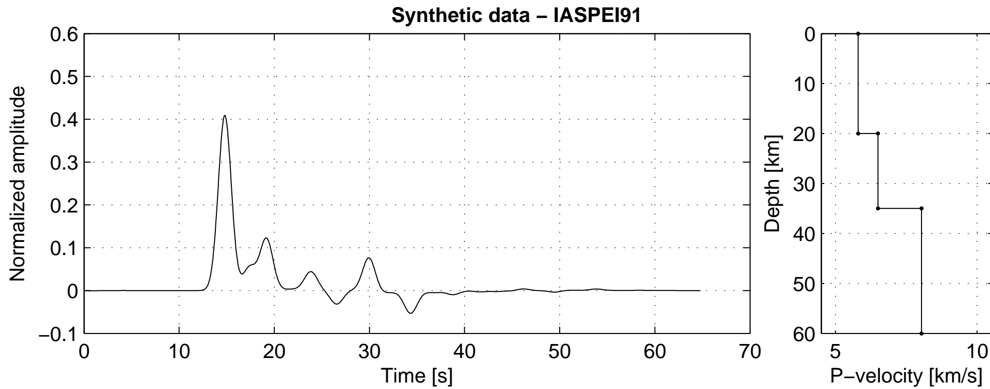


Figure 4.1: *A synthetically generated receiver function based on the reference earth model IASPEI91, (Kennett and Engdahl, 1991). Left: The generated receiver function. Right: The model is defined by two boundaries at a depth of 20 km and 35 km and three P-wave velocities in each of the layers.*

one could imagine several different rod structures that could produce the same electromagnetic field. This is also known as non-uniqueness.

4.1.1 The forward problem

The forward model I will be using in this thesis is provided by Bo Holm Jacobsen (Jacobsen, 2008) in the form of MATLAB code. The code is based on the propagator matrix method described in (Kennett, 1983) and (Shearer, 1999). The propagator matrix is a method to calculate synthetic seismograms from a plane wave propagating through a horizontally stratified underground of homogeneous layers. The implementation of Bo Holm Jacobsen includes all higher order multiples of reflected and refracted waves. The code generates two synthetic seismograms in the vertical and radial direction and by deconvolution of the vertical from the radial component the radial receiver function is constructed.

The input to the code was model parameters given as a vector with the depth to the layers in km as the first N_{layer} vector elements and $N_{layer}+1$ corresponding P-wave velocities in km/s. The forward model then returns the corresponding receiver function.

As an example the code was provided with a model consisting of 2 layers and 3 velocities based on the IASPEI91 model, see (Kennett and Engdahl, 1991). The layer interfaces are located at 20 and 35 km. The P-velocity between the surface and the first layer was 5.8 km/s, between the first and second layer the speed was 6.5 km/s and after the second layer the speed was 8.04 km/s. The resulting receiver function and the model are shown figure 4.1.

The time consumption of the forward algorithm depending on the number of model parameters was found to be

$$t(n) = 0.0032 * n + 0.11$$

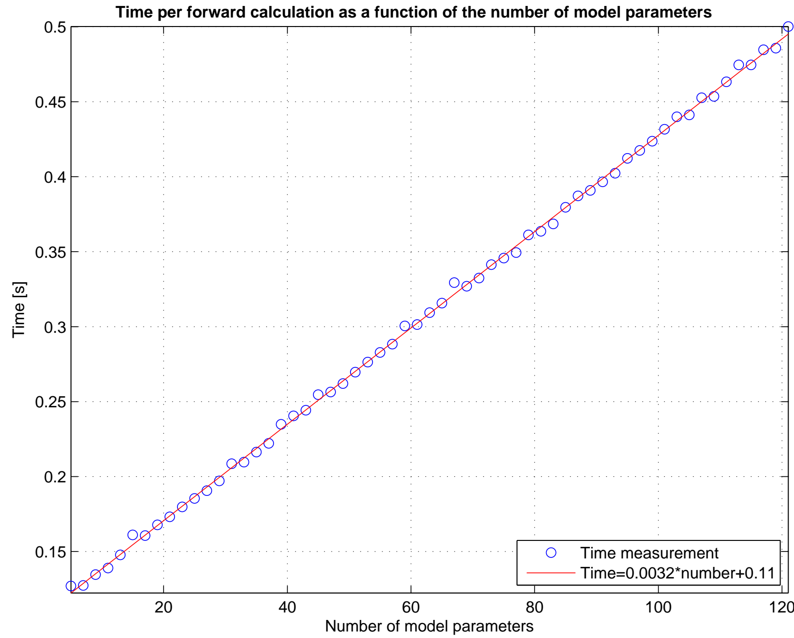


Figure 4.2: *The time consumption of the forward model to generate a receiver function as a function of the number of model parameters.*

where $t(n)$ is the time consumption of the forward model to generate one receiver function and n is the number of model parameters. The norm of the residual was 0.0203 and $t(n)$ is the average of 10 calculations. The relationship can be seen in figure 4.2. The most important aspect of this relationship is the linearity since the actual values depend on the speed of the computer used[1].

Assumptions

Only providing the code with depth and P-wave velocity demands that certain assumptions are made about the V_p/V_s relationship, density etc. I assume that earth behaves as a Poisson solid such that the S-velocities can be related to the P-velocities as

$$V_s = \frac{V_p}{\sqrt{3}}$$

see (Shearer, 1999, p. 21). Furthermore, the density is assumed to be related to the P-velocities by the following empiric relation called the Birch Law:

$$\rho \, [\frac{\mathrm{kg}}{\mathrm{m}^3}] = 0.32 \, [\frac{\mathrm{kg} \cdot \mathrm{s}}{\mathrm{m}^3 \cdot \mathrm{km}}] \cdot V_p \, [\frac{\mathrm{km}}{\mathrm{s}}] + 770 \, [\frac{\mathrm{kg}}{\mathrm{m}^3}];$$

see (Birch, 1964, p. 4380). Finally the results are filtered through a Gaussian filter as defined in section 2.3.2 where the width of the filter can be controlled.

These assumption could be changed when calling the forward code which would make it possible to perturb more variables than the depth and P-velocity

[1]The numbers were obtained on a computer with a Intel Core 2 Duo CPU P8600@2.40GHz.

at the cost of a more complex model space. In my work I have decided only to perturb the depth and velocity parameters.

4.1.2 The inverse problem

With a theoretical forward model one can easily calculate how a receiver function would appear under given circumstances. The problem at hand is however not to find the receiver function but rather to determine the earth velocity structure that generate the receiver function.

Consider a system which is fully described by N_{dim} parameters $\mathbf{m}$, called a model, belonging to a N_{dim}-dimensional space called the model space $\mathcal{M}$. The model space contains all possible parameter sets and each point in this space is a valid description of the system. Given a model and a physical law g, we can predict a set of data values $\mathbf{d}_{cal}$. $g : \mathcal{M} \rightarrow \mathcal{D}$ is the mathematical description of the physical law mapping from the model space $\mathcal{M}$ into the data space $\mathcal{D}$. All possible realizations of data belong to $\mathcal{D}$. The vector $\mathbf{d}$ is a data set with N_d elements and belonging to the N_d a dimensional data space $\mathcal{D}$. In this terminology the forward problem can be written as

$$\mathbf{d}_{cal} = g(\mathbf{m}) \tag{4.1}$$

The inverse problem is to find a model, $\mathbf{m}$ that corresponds to some observed data $\mathbf{d}_{obs}$ and in theory this can be written as:

$$\mathbf{m} = g^{-1}(\mathbf{d}_{obs}). \tag{4.2}$$

However, for most inverse problems it is not possible to create g^{-1}. In general, inversion of receiver functions is a highly non-linear problem[2], see (Ammon et al., 1990), and g^{-1} cannot be constructed. To find the solutions to the problem one has to approach the problem in another way.

4.1.3 Probabilistic formulation

In a probabilistic formulation of a inverse problem a priori information about the model can be described by a probability density, $\rho(\mathbf{m})$. A priori information in this sense is any information on a model given before the modeling begins. As an example of a priori knowledge I use that fact that P-velocities above 7 km/s at depths below 20km are unrealistic, see table 6.1, and prohibit my algorithm from entering this domain of the model space. The a posteriori probability density $\sigma(\mathbf{m})$ over the model space can be seen as the complete solution to the inverse problem as it describes all the information about the problem at hand. $\sigma(\mathbf{m})$ can be very complex and contain many local minima which makes it hard to determine. In figure 4.3 a surface proportional to the a posteriori probability density was created in the case of a five-dimensional model space, where three of the parameters were fixed and two parameters were varied. The pathologies well known in non-linear problems such as valleys, many separate

[2] The non-linearity of the problem has however been challenged in a recent article, (Jacobsen and Svenningsen, 2008), see discussion in section 8.4.

peaks, and troughs are seen. The a priori probability density can be related to the a posteriori density through a Likelihood function, $L(\mathbf{m})$, which evaluates the fit between the data and the model. This connection is usually (Mosegaard and Tarantola, 1995, p. 3) written as

$$\sigma(\mathbf{m}) = k\rho(\mathbf{m})L(\mathbf{m}) \tag{4.3}$$

where k is a normalization constant. In this context the a priori distribution can be seen as a weight given to a model depending on how the model agrees with the a priori information. In the case of the receiver functions in this thesis there is no close-form expression to calculate $\sigma(\mathbf{m})$ and the option is to draw samples from $\sigma(\mathbf{m})$.

The likelihood can have many forms depending on the problem but in my case I have used the following definition, see (Mosegaard and Tarantola, 1995, p. 6)

$$L(\mathbf{m}) = k_l \exp\left(-\frac{S(\mathbf{m})}{s^2}\right) \tag{4.4}$$

k_l is a constant and s^2 is the total noise variance, which in my case is the same for all of the N data values of the data vector, $\mathbf{d}$. The misfit function, $S(\mathbf{m})$, is given as a sum of the element-wise squared difference between the data calculated from the forward model $d_{cal} = g(\mathbf{m})$, and the observed data d_{obs}:

$$S(\mathbf{m}) = \sum_{i=1}^{N} \left(d^i_{cal}(\mathbf{m}) - d^i_{obs}\right)^2 \tag{4.5}$$

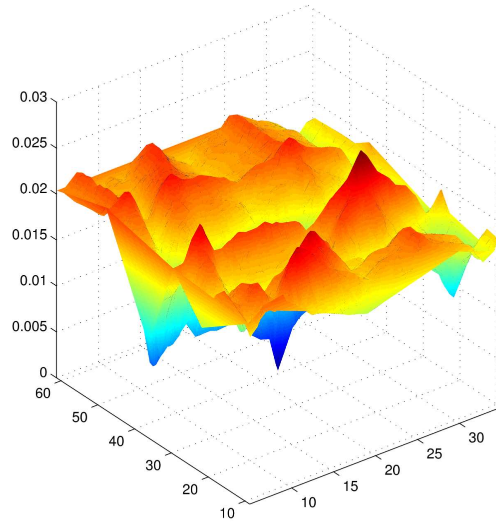
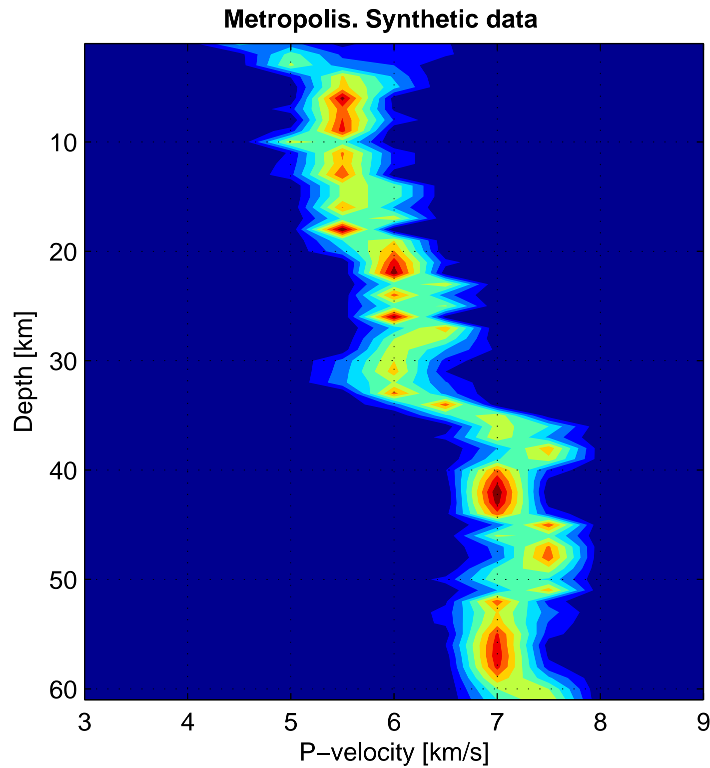


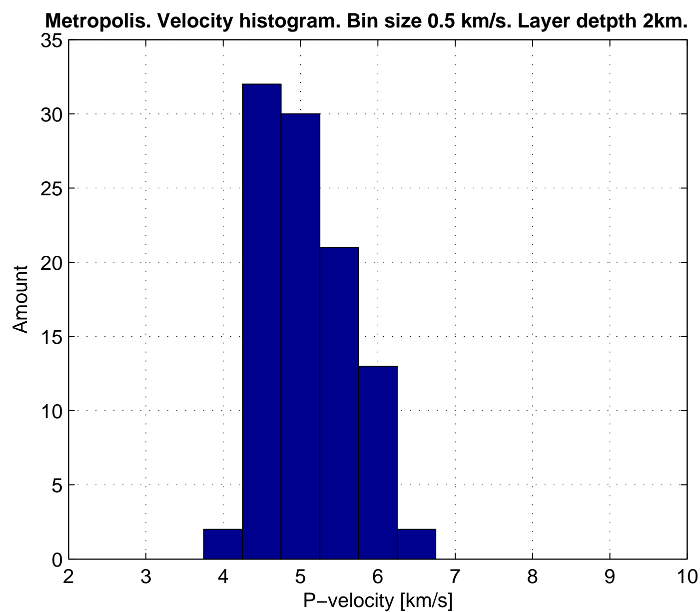
Figure 4.3: *This plot was created by a Uniform sampling with 5000 samples varying two model parameters in five-dimensional model space and comparing the resulting receiver function to the synthetic data. The plot is further described in section 6.2. The surface of this plot shows the misfit value between the synthetic data set and data generated by the algorithm. This surface is proportional to the a posteriori probability density.*

In the probabilistic description the concept of selecting the single best sample from the pool of created samples is meaningless. By drawing independent samples from the a posteriori probability density, it is possible to analyze the samples to find the variance, mean or standard deviation by creating histograms. For each of the model parameters a histogram of the values of model parameters found in the samples is generated. This is also called the marginal probability density and is illustrated in 4.4.

The choice of model parameters in $\mathbf{m}$ for a given problem is very important since the result of the inversion is dependent on the parameterization, see (Menke, 1984, p. 143). In a recent article, (Jacobsen and Svenningsen, 2008), this issue was addressed for receiver functions and is discussed in section 8.4.



(a)



(b)

Figure 4.4: *a) This figure shows the marginal probability velocity distribution for the velocity structure of the synthetic IASPEI91 data set. For each kilometer in depth a histogram velocity value of the samples drawn from* σ *was created. The histograms are then stacked on top of each other and the contour map was created. The red areas illustrate the peak of the histogram. b) The histogram for the layer with a depth of 2 km.*

Chapter 5

Algorithms

In this chapter I shall introduce the algorithms used in the thesis. I have worked with four different algorithms. A simple and unoptimized Uniform sampling algorithm, a hybrid between the Gauss-Newton algorithm and the steepest descent algorithm called the Levenberg-Marquardt algorithm (Levenberg, 1944, Marquardt, 1963), the classic Metropolis algorithm (Metropolis et al., 1953) and finally a newer algorithm called the Neighborhood search algorithm (Sambridge, 1998). The Uniform search, Neighborhood search and Metropolis algorithms all belong to the Monte Carlo family of algorithms which use random numbers to find solutions. I could have decided to investigate many other optimization algorithms such as simulated annealing or genetic algorithms, see (Press et al., 1992) and (Mosegaard and Sambridge, 2002), but with this selection I think some of the most important characteristics from the different types of algorithms are covered.

In figure 5.1 a number of algorithms have been classified by how well they explore or exploit features of the model space. An exploring algorithm does not utilize the information gathered from previous samples of the model space whereas an exploiting algorithm will use the information drawn from previous samples to create a new sample where it is most likely to find a better sample. As it can be seen the Uniform search is highly exploring but not exploiting in any sense. On the other hand, Levenberg-Marquardt, belonging to the group called "Newton-Raphson and other gradient methods" is highly exploiting but can easily get stuck in local minima. Both the Neighborhood search algorithm and the Metropolis algorithm fits in somewhere in between exploitation and exploration, since they can both be tweaked to perform highly exploring or exploitative depending on the choice of control parameters in the algorithm.

5.1 The Monte Carlo Family

The Monte Carlo methods are a family of algorithms and methods which use random numbers to solve a large variety of optimization problems. A famous example of a Monte Carlo method is to determine the value of π by scattering rice grains uniformly onto a circle inscribed in a square. The proportion of grains inside the circle to the grains inside the square is the same as the proportion of

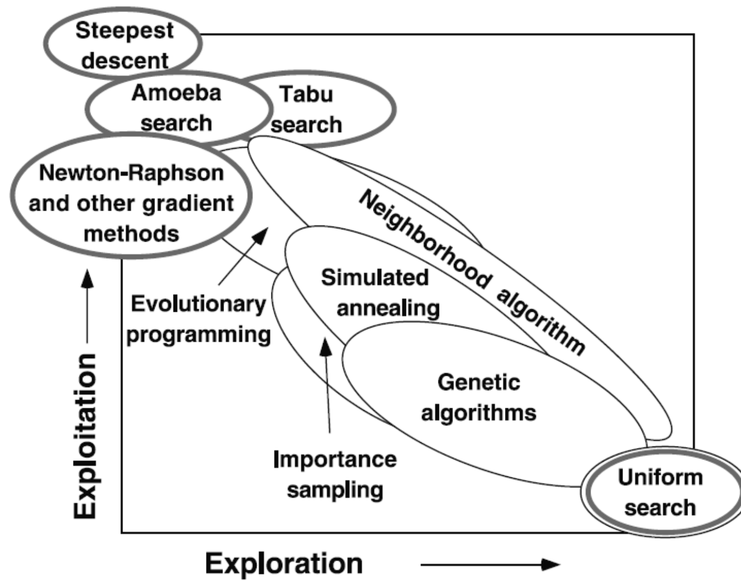


Figure 5.1: *As an example on how to characterize search/optimization algorithms is to describe them either as mainly exploring or exploiting. From (Sambridge and Mosegaard, 2002, p. 3-11).*

areas:

$$\rho = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

By counting the number of grains inside and outside the circle one can determine the value of $\pi/4$ and hence π. This all depends on the randomness of the scattering of the rice grains. If the rice grain scattering is skewed towards hitting the circle, the ratio of grains will no longer be useful in determining π, which implies the need for good random numbers. Another clarifying aspect of this example is that the approximated value of π will come closer to the true value of π as the number of rice grains increases.

The applications of computational Monte Carlo methods started just after the Second World War where Neumann, Ulam and Fermi in 1949 used the method to study nuclear reactions. In 1953 Metropolis published an article (Metropolis et al., 1953) on how to use this method to calculate equations of state on "fast electronic computing machines" – the predecessor to the modern computer. This article was the birth of the what is now known as the Metropolis algorithm and it was among the first algorithms ever to run on a computer. A detailed historical description of development of the Monte Carlo method at Los Alamos National laboratories in the fifties can be found in (Metropolis, 1987).

The example of determining π with rice grains illustrates that the foundation of Monte Carlo method is random numbers. True random numbers based on atmospheric noise or radioactive decay are hard to come by when large amounts are needed. Thus, in the first years of the Monte Carlo methods the generation

of pseudo random numbers was also heavily researched. A pseudo random number sequence is a sequence of numbers that appears to be random but can be calculated by a small set of initial values known as the seed. Today all modern operating systems have some sort of random number generator built in. This is however not a tribute to the Monte Carlo method but instead driven by the need of pseudo random numbers in cryptography. Generally the pseudo random numbers generated are sufficient in most cases, as in mine, but when true randoms are needed there are websites such as `www.random.org` that provide true random numbers based on atmospheric noise.

5.2 Uniform sampling

In search of models that reconstructs the receiver function waveform the simplest method to explore the model space is the Uniform sampling method also known as exhaustive search or grid search, see (Mosegaard and Sambridge, 2002). There are many ways of implementing this algorithm. One of the methods is to design a grid to compartmentalize the parameter space and calculate the misfit in each of the grid points. Depending on the grid spacing this will provide a certain resolution in the parameter space which is controlled by the programmer. This approach can give rise to problems if the grid spacing coincides with a feature in data having a frequency similar to the grid spacing. Finding the best compromise between a fast and coarse grid and a slow but fine grid can be a troublesome problem since the frequency of data features are not known in advance and the algorithm does not exploit local information to converge on the best solutions.

Another approach is to generate a predefined amount of random samples in the parameter space. To achieve a certain mean distance between the samples one can either calculate the mean distance as new samples are generated or just calculate the amount of samples that would generate a certain mean distance in the grid approach.

To generate a fixed number of models, i in a N_{dim} dimensional model space each model, $\mathbf{m}_i$ can be written as:

$$\mathbf{m}_i = \sum_{i=1}^{N} x_i \mathbf{e}_i \tag{5.1}$$

where x_i are the random numbers with appropriate units and e_i are the unit vectors along the i'th dimension of the model space

$$\begin{aligned} \mathbf{e}_1 &= (1, 0, 0, \ldots, 0) \\ \mathbf{e}_2 &= (0, 1, 0, \ldots, 0) \\ &\vdots \end{aligned}$$

Generating the models is a very fast procedure on modern computers but the evaluation of the corresponding misfit is usually much slower. Since the calculation of the misfit is directly proportional to the number of models the calculation time increases linearly with number of models. Considering a coarse grid

of 10 points in each dimension of a N-dimensional model space the number of model evaluations would be 10^N, which makes this algorithm a bad choice for high-dimensional problems.

In the computer industry Moore's law, see (Moore Gordon, 1965), is a widely cited and still useful generalization. It states that the number of transistors that can be placed on a CPU will double every second year and the number of transistors is somewhat proportional to the performance of the CPU. Even though Moore's law explains the advances done in the computer industry which makes calculation that was previously unfeasible simple in a modern light it cannot compensate for the drastic dependence on dimensionality that Uniform search suffers from. The time it takes to create a model in a N_{dim} dimensional space is determined only by the time it takes to create a forward calculation $T_{forward}$.

$$T_{Uniform} = T_{forward}$$

But to sample a N_{dim} dimensional model space with a density high enough to reveal data with a frequency of k peaks in the a posteriori distribution one would need at least

$$T_{Uniform,total} = T_{forward}(2k)^{N_{dim}}$$

samples. My implementation of the algorithm can be found in section B.1.

5.3 The Metropolis algorithm

The Metropolis algorithm is a so-called Markov Chain Monte Carlo method. A Markov chain is a sequence of random models generated by a stochastic process that follows the Markov property. A Markov chain is an example of random walk. The Markov property states that the transition probability from one model $\mathbf{m}_n$ to another $\mathbf{m}_{n+1}$ depends only on the current model $\mathbf{m}_n$, see (Walsh, 2004)

$$P\left(\mathbf{m}_{n+1} = \mathbf{x_{n+1}} | \mathbf{m}_1 = \mathbf{x}_1, \ldots, \mathbf{m}_n = \mathbf{x}_n\right) = P\left(\mathbf{m}_{n+1} = \mathbf{x_{n+1}} | \mathbf{m}_n = \mathbf{x}_n\right) \quad (5.2)$$

where $\mathbf{x}$ is a point in the model space. The memory of the chain is very short since all transitions to all future models are independent of previous models. For a more detailed description of Markov chains see (Walsh, 2004). For a detailed description of Metropolis algorithm see (Mosegaard and Tarantola, 1995) or (Mosegaard and Sambridge, 2002).

The Metropolis algorithm is called an importance sampling algorithm meaning that the Metropolis algorithm will sample a interesting areas of the model space more than the uninteresting areas. Even though no one has coined the term unimportance algorithm it is possible to imagine a method to sample the posteriori probability distribution while not focusing on the interesting areas. A random walker created in the following way does just this.

Imagine a probability density p over the model space and a large number M such that

$$M \geq \max(p(\mathbf{m}))$$

then $p(\mathbf{m})$ could be sampled by accepting a new random model $\mathbf{m}_{n+1}$ succeeding the previous model $\mathbf{m}_n$ with the probability:

$$P_{accept} = \frac{p(\mathbf{m}_{n+1})}{M}$$

The accepted candidates are then samples from p, see (Mosegaard and Sambridge, 2002, p. R34). The problem with this approach is that the acceptance probability will be very small for most points in the model space and the algorithm will move very slow.

Instead of comparing the probability with a large number the Metropolis algorithm compares the probability of a current $\mathbf{m}_{cur}$ model with the probability of the new candidate model $\mathbf{m}_{new}$. The algorithm is best describe by the following steps

1. Select any initial point in the model space, $\mathbf{m}_{cur}$, where the misfit is not zero.

2. Create a new point, $\mathbf{m}_{new}$ in the model space by selecting a random dimension in the model space and adding or subtracting a random value to the existing value. The only requirement is that the probability of jumping from $\mathbf{m}_i$ to $\mathbf{m}_j$ is symmetrical, $P(i \to j) = P(j \to i)$.

3. Calculate the probability of accepting the new model:

$$P_{accept} = \begin{cases} 1 & \text{if } S(\mathbf{m}_{\text{new}}) \leq S(\mathbf{m}_{\text{cur}}) \\ \exp\left(-\frac{\Delta S}{s^2}\right) & \text{if } S(\mathbf{m}_{\text{new}}) > S(\mathbf{m}_{\text{cur}}) \end{cases}$$

This means that when the algorithm suggest a new model, $\mathbf{m}_{cur}$, the model will be accepted immediately if the new model lowers the misfit. If the misfit is not lowered the algorithm makes the transition with a probability of $\exp\left(-\frac{\Delta S}{s^2}\right)$.

When the Markov Chain has been running for a while all traces of the initial model has been lost and the system would have settled down into an area of the model space. The time it takes before the algorithm settles down is called the "Burn-in" period and after this period the algorithm will start to sample the distribution.

In figure 5.2 the Metropolis algorithm has been started at different initial values further and further away from the global minimum in the synthetic data set. The first two parameters of the five parameters in the model, representing the depth parameters, were locked to the same values found in the IASPEI91 model. The remaining three velocity parameters of the synthetic model had the values 5.8, 6.5 and 8.04 km/s respectively. The figure illustrates that the convergence speed towards a stable region in the model space depends on the starting point of the algorithm, $\mathbf{m}_0$ and some element of luck. The program run with values furthest away from the synthetic data seems stuck in a local minimum. If I had continued the simulation the algorithm would eventually jump out of the minimum since the Metropolis algorithm has always a certain probability to jump out of a local or even global minimum even though the probability might be small.

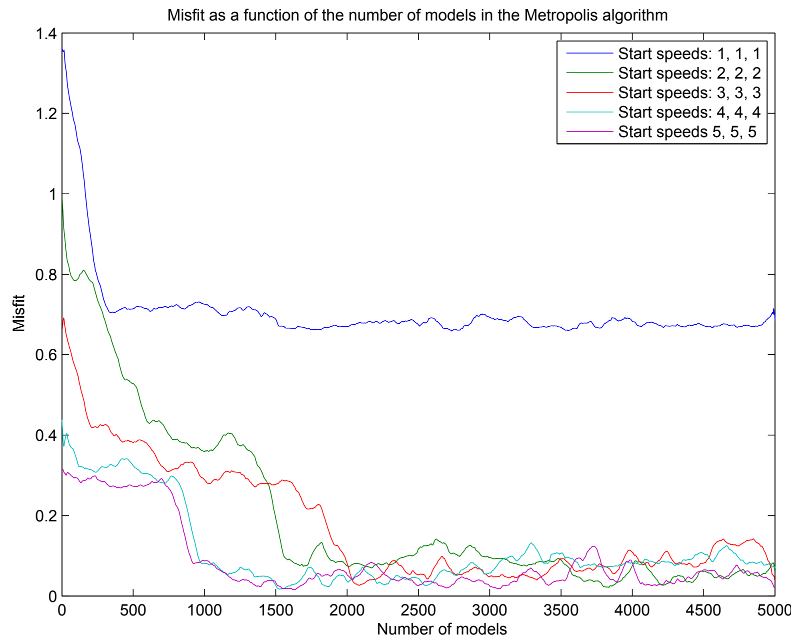


Figure 5.2: *The misfit as a function of the number of steps in the Metropolis algorithm with synthetic data. The blue line appears to be stuck in a local minimum, but in time it would most likely escape. Of the 5 parameters in the model two were constrained and three were free but started with the parameters seen in the graph legend. The synthetic data had the values: 5.8 6.5 8.04.*

One of the problem with Metropolis algorithm is that the efficiency of the algorithm depends on the size of the steps done by the random walker. There is no universal method of determining the correct step size (Mosegaard and Sambridge, 2002, p. R38) and usually the step sizes has to be calibrated such that the algorithm accepts a reasonable amount of transitions. The percentage of accepted step in all the possible steps is called the acceptance rate and should usually be between 30-50%, see (Tarantola, 2005, p. 53). If the accept rate is much lower, the algorithm will spend to much time evaluating models that will not be accepted. A low accept rate is often the case if the step size is too big. In the case that the accept rate is much larger than 50% the algorithm probably has too small a step size and the algorithm will walk too slowly through the model space. A acceptance rate of 30-50% should produce the balance between too small and too large step sizes.

5.4 The Neighborhood search algorithm

In 1998 Malcolm Sambridge proposed a new algorithm to do optimization in inversion problems, see (Sambridge, 1998, 1999). The method was named the Neighborhood search algorithm. The algorithm is a member of the Monte Carlo family of algorithms and in short the idea is to make an initial coarse sampling of the models space, select the most promising regions, and sample

these regions more thoroughly in a manner such that the most heavily sampled areas correspond to the areas where the models fits data best. The basic idea of the algorithm is summarized by Sambridge

> *The philosophy behind the algorithm is that the misfit of each of the previous models is representative of the region of space in its neighborhood.* (Sambridge, 1999, p. 484)

In the case that a given model really represents its neighborhood the algorithm will work fine but if the model is unrepresentative of its neighborhood the algorithm will get into trouble. This issue is discussed further section in 8.3. One of the advantages of the algorithm is that it only relies on two control parameters, namely the number of new samples per iteration N_s and the number of promising models to select for further inspection, N_r. By selecting a N_s/N_r ratio close to 1 the algorithm will be exploring the model space and if the N_s/N_r ratio is closer to 0 the algorithm will behave more exploiting, see (Sambridge, 1999, p. 12). An exploiting algorithm will utilize local information to select better models in the vicinity. An exploring algorithm on the other hand seeks to explore the whole space without much regard for locale areas of interest.

5.4.1 Voronoi cells

A key concept in the Neighborhood search algorithm is the Voronoi cell. A Voronoi cell is defined as the region about a point p in space where all the points in the region are closer to p than to any other point in the space.

Given a discrete set of points P the Voronoi cell belonging to a point $p \in P$ can be formulated more formally as

$$Vor(\mathbf{p}) = \{\mathbf{x} \in \mathcal{M} | dist(\mathbf{p}, \mathbf{x}) \leq dist(\mathbf{q}, \mathbf{x}),\ \forall \mathbf{q} \in P, \mathbf{q} \neq \mathbf{p}\}\,.$$

In figure 5.3a a two-dimensional space with 10 points is shown where the thin lines separate the Voronoi cells of the points.

5.4.2 The behavior of the Neighborhood search algorithm

The behavior of the Neighborhood search algorithm is very easy to illustrate and in figure 5.3 the different stages are shown. At first the algorithm scatters N_s=10 random points in the model space and the Voronoi cells are drawn, see figure 5.3a. After 10 iterations of generating 10 new points inside the most promising regions the Voronoi cell begin to concentrate on the regions with the best models, see 5.3b. Finally after 1000 models have been generated the density of models generated resembles the original distribution, seen in figure 5.3d where the black areas are he most interesting. This behavior can be formulated in five steps

1. Generate N_s initial random models in the model space.
2. Calculate the misfit for each of the models in the model space and find the N_r models with the lowest misfit.

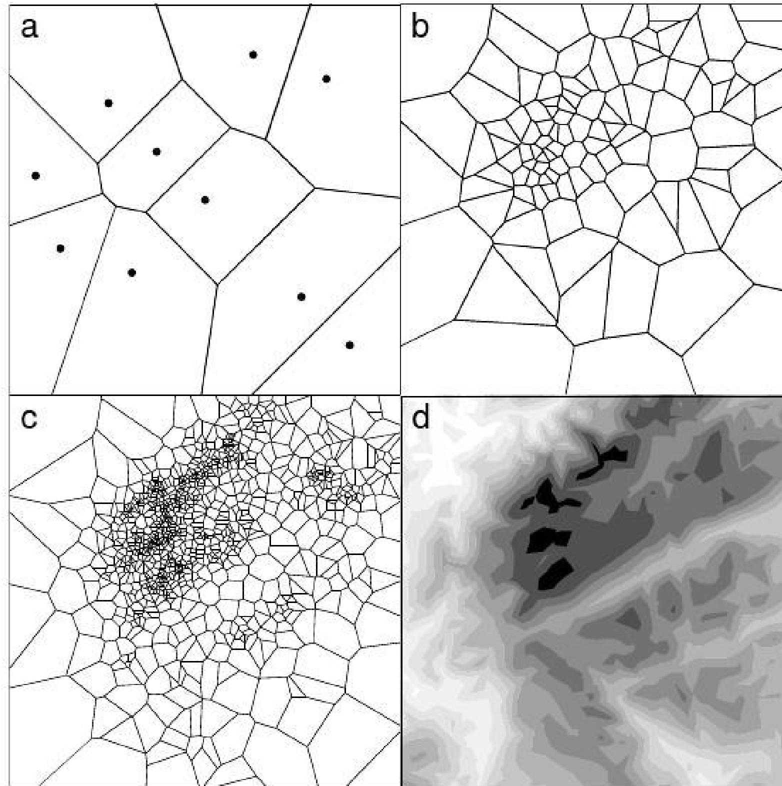


Figure 5.3: *The Neighborhood search algorithm samples the areas of interest more heavily than other areas as the number of iterations increase. a) 10 points. b) 100 points, c) 1000 points. d) Contour of test function. The black areas represent minima. From (Sambridge, 1999).*

3. Determine the boundaries of the Voronoi cell belonging to each of the N_r models.

4. Generate N_s/N_r new random models inside each of the N_r Voronoi cells.

5. Repeat from step 2 until the desired amount of models have been generated.

5.4.3 Exploring the Voronoi cell

The only technical obstacle in implementing the Neighborhood search algorithm is generating new models inside of a Voronoi cell. To do this the boundaries of the cells needs to be known in order to be sure that new models are within the cell. There are many approaches to this problem and working on my thesis I experimented with several approaches. Common for them all is that the new models generated inside a Voronoi cell are constructed using the Gibbs sampler. In the Gibbs sampler each iteration in a N_{dim} dimensional space consists of N_{dim} substeps. Starting inside the Voronoi cell of the point $\mathbf{m}_0$ each of the substeps will make a random change to one of the N_{dim} model parameters in $\mathbf{m}_0$. After all

the substeps have been completed the algorithm will have generated a new point independent of the previous point. The only limitation is that a new random parameter value must be within boundaries of the Voronoi cell to which $\mathbf{m}_0$ belongs.

5.4.4 Sampling the cell using Qhull

One of the three great virtues of a programmer beside impatience and hubris is laziness, (Wall et al., 2000). This should be understood as laziness in regard to redoing work already done once. Major scientific programming environments as MATLAB, Mathematica and GNU octave all use the open source program called Qhull, see (Barber et al., 1996), to compute the Voronoi cells of given number of points in space.

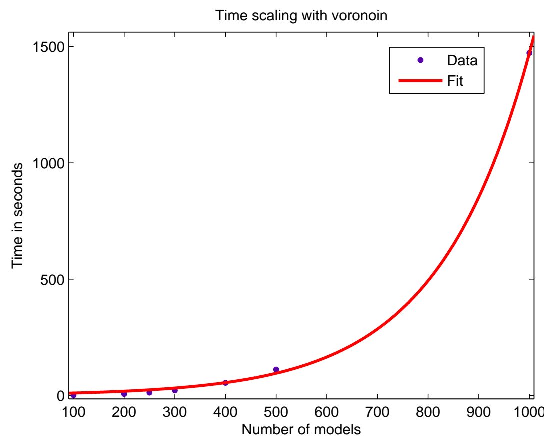


Figure 5.4: *Scaling of the* Voronoin *implementation of Qhull in MATLAB. Data is fitted to exponential function*

Even though implementing Qhull in MATLAB is relatively easy I quickly faced the hubris of the quick solution to my problem. Qhull returns much more information than what is needed at the cost of extra computation time. To do the sampling I only needed information at the boundaries of the cell in one dimension at a time. Qhull calculates the full convex hulls for all cells in every dimension. This is not a problem as long as the number of models is reasonably small but the computation time scales exponentially and hence it is useless for larger models. In figure 5.4 the computation time is compared to the number of models in the space. The computation follows a exponential growth described by:

$$\text{Computation time} = 6.2 \exp^{\frac{\#\text{models}}{181}}$$

It should also be considered that every time a new point is added to the space the Voronoi cell surrounding the new point will change and will need to be recalculated. Furthermore, not all this information is needed as the algorithm only needs to know the boundaries of the Voronoi cell of a point in question to generate new models inside it.

5.4.5 Sampling the cell using discretized axis

The simplest method to find the boundaries of the Voronoi cell was proposed by Sambridge in (Sambridge, 1999, p. 485). The idea is to discretize the model

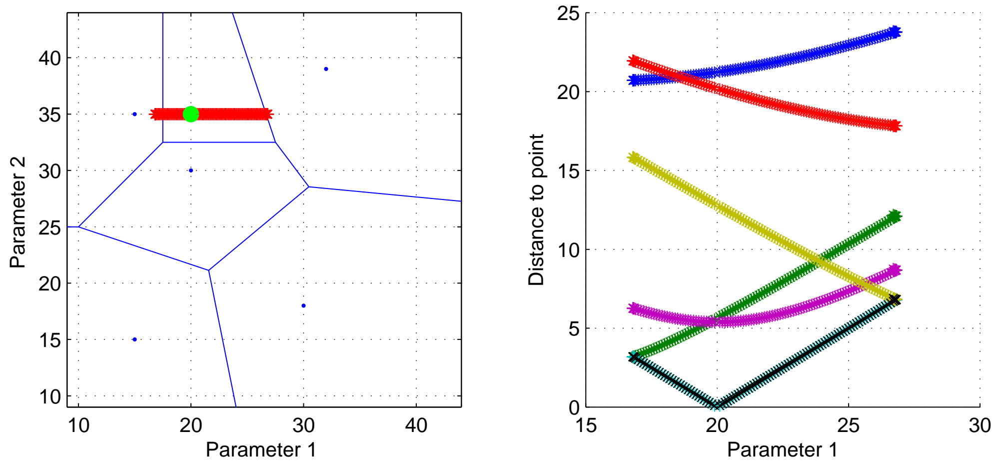


Figure 5.5: *On the left side the Voronoi diagram of a 2D space is shown. The Green spot indicates the cell which is being sampled. The red dots on left shows how the algorithm walks to locate the boundaries of the cell. On the right side the distances to the other models are plotted as a function of the moving point. As an example the black line to the right illustrates the distance to the point in middle of the Voronoi cell from the each of the points on on red line in the left figure.*

space axis into n_a points and determine the distance to all the samples for each of the n_a points. The intersection between two Voronoi cells would then be given by the point in the grid where a neighboring point belonged to another Voronoi Cell.

I have implemented a slightly altered version of this method to ensure that the boundaries would be better resolved. Instead of using a fixed amount of points n_a, I made large steps, s_{orig}, starting in the center Voronoi cell belonging to the model $\mathbf{m}_k$ along a given axis. For each step I calculated the distance to all promising models in space and once the algorithm steps outside the Voronoi cell belonging to $\mathbf{m}_k$ the distance to $\mathbf{m}_k$ would no longer be the smallest. My implementation of the algorithm then takes a step backwards with the original step size s_{orig} to make sure we are still inside the Voronoi cell of $\mathbf{m}_k$. The algorithm then divides the step size by 2 and steps forward again until the algorithm determines that it had stepped outside the Voronoi cell of $\mathbf{m}_k$. The algorithm will then repeat to diminish the step size until the step size gets smaller than a limit given by the programmer. This approach ensures that the boundary can be found with any resolution wanted. In figure 5.5 this approach is illustrated and it can be seen how the distance to the other points increases or decreases as the algorithm walks along an axis. My implementation can be found in section B.3.

The problem with this simple approach is that the number of times it will evaluate the distance to other models per cells, increases with the number of dimensions and the number of models. In the case described in (Sambridge, 1999, p. 485) with a fixed amount evaluation points n_a along each axis and n_p

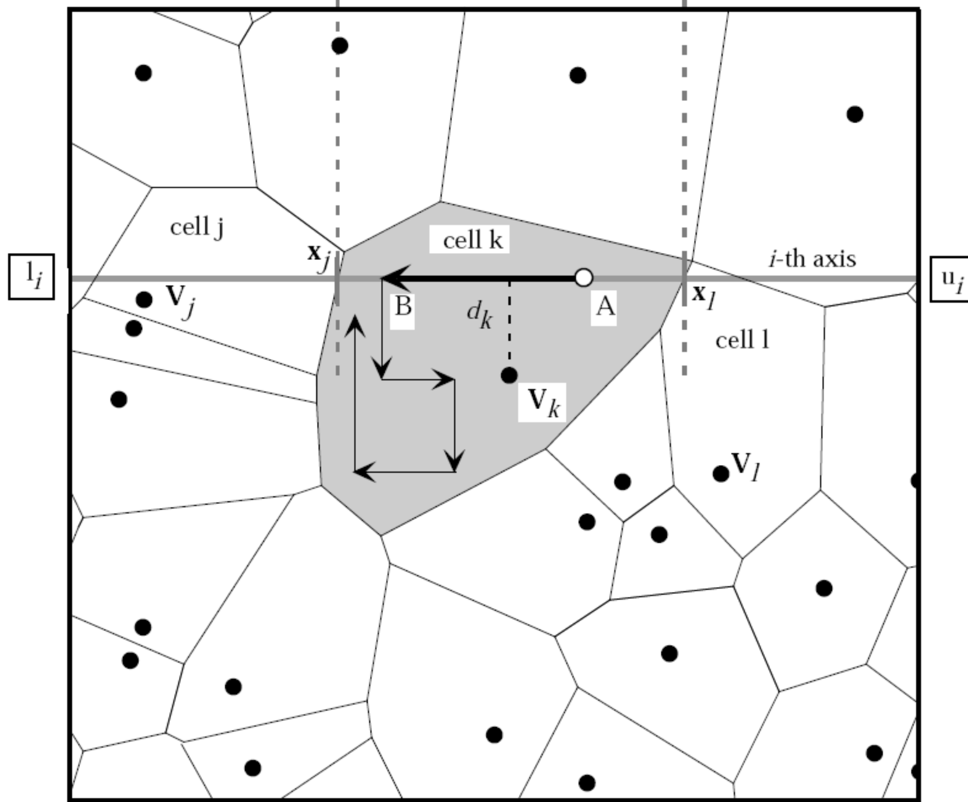


Figure 5.6: *2D illustration of the "refined" approach described in (Sambridge, 1999). Notice that Sambridge uses $\mathbf{V}_k$ where I use $\mathbf{m}_k$ to indicate that they are models.*

models the time to create each independent sample is

$$T_{simple} = T_{forward} + k n_a n_p N_{dim}^2,$$

where k is a constant, $T_{forward}$ is the time it takes to do a forward model and N_{dim} is the amount of dimension in the model space. Since my approach will increase with n_a by a certain amount I would get a better resolution of the boundary at the cost of a higher n_a and hence more evaluations.

5.4.6 Sampling the cell using exact intersections

In (Sambridge, 1999, p. 485) a more refined approach was introduced. Instead of discretizing the model space this method calculates the exact intersection point between the Voronoi cell and the nearest neighbor along a give dimension.

Imagine a 2D space bounded by a black box as the one seen surrounding the Voronoi cells in figure 5.6 with N_s random models scattered around. Let $\mathbf{m}_k$ define the k'th model and the k'th Voronoi cell, $\mathbf{V}_k$, be the one about $\mathbf{m}_k$. For the sake of the argument say that our algorithm starts in the point A[1]. We

[1]It is also possible and maybe easier to start the algorithm in $\mathbf{m}_k$ since this is the one point known to be inside Voronoi cell. But using figure 5.6 I will use A in the description.

wish to determine the intersection point between the k'th and j'th cells and this point shall be called $\mathbf{x}_j$. Since $\mathbf{x}_j$ is define as the point to which the distance from both $\mathbf{m}_j$ and $\mathbf{m}_k$ equal we have:

$$\|\mathbf{m}_k - \mathbf{x}_j\| = \|\mathbf{m}_j - \mathbf{x}_j\|$$

In the i'th dimension of the model space this can be written as

$$d_k^2 + (m_{k,i} - x_{j,i}) = d_j^2 + (m_{j,i} - x_{j,i})$$

where d_k^2 and d_j^2 are the perpendicular distances from $\mathbf{m}_k$ and $\mathbf{m}_j$ to a line going through $\mathbf{m}_k$ in dimension i. This expression can be solved to find the exact intersection point $x_{i,j}$

$$x_{j,i} = \frac{1}{2}\left[m_{k,i} + m_{j,i} + \frac{d_k^2 - d_j^2}{m_{k,i} - m_{j,i}}\right] \tag{5.3}$$

This calculation has to be done for all models in the space and the two closest points on each side of $\mathbf{m}_k$ defining the boundaries of the Voronoi cell. An important part of equation 5.3 is the perpendicular distance to a line in N_{dim} dimensions. The distance from a line to a point in N_{dim} is not hard to calculate but depending on the formulation of the problem the equation can be more less complicated. In the major part of my thesis I used a geometrical approach to calculate the distance from a line to a point in N_{dim} and even though the result is valid the solution is long and complicated, see section A.3. It was only in the last days of my thesis I realized a much simpler approach using a parametrized version of the line, see section A.4. My implementation of the algorithm can be found in section B.4. Even though this method is superior to the method described in section 5.4.5 I ended up not using this method since I experienced numerical problems with my implementation.

Since this approach unlike the method with the discretized axis does not need to calculate the distance two times for each point of the grid this approach is a factor $n_a N_{dim}$ faster. The time to create each independent sample is then

$$T_{exact} = T_{forward} + k_2 n_p N_{dim},$$

where n_p is the amount of models in the space at the time (Sambridge, 1999).

5.4.7 Changing the misfit function

In (Sambridge, 2001, p. 393) two alternative misfit functions is described[2]. By modifying the misfit function the programmer can alter the landscape the algorithm sees. This can serve to make the algorithm center on areas that would not usually be sampled.

One method would be to flatten the misfit function. If the misfit in a point is below a threshold S_t the misfit value will altered to the threshold value. This can formally be written as

[2]The misfit function I am using is defined in equation 4.5.

$$
\begin{array}{lcll}
S_{flat}(\mathbf{m}) & = & S(\mathbf{m}) & S(\mathbf{m}) > S_t \\
S_{flat}(\mathbf{m}) & = & S_t & S(\mathbf{m}) \leq S_t
\end{array}
$$

This definition makes the algorithm “see” good areas as flat areas and will optimize the algorithm to explore the areas uniformly. This misfit function could help the algorithm to behave by exploring on a local level.

Another alternative misfit function is to reverse the misfit function. Sometimes the most interesting feature of an region in the model space is the boundary that defines the region in space. By creating a misfit function that returns a very low value near the edge of an interesting area the algorithm is forced to sample the edge. This can formally be written as

$$
\begin{array}{lcll}
S_{reversed}(\mathbf{m}) & = & S(\mathbf{m}) & S(\mathbf{m}) > S_t \\
S_{reversed}(\mathbf{m}) & = & S_t - S(\mathbf{m}) & S(\mathbf{m}) \leq S_t
\end{array}
$$

In (Sambridge, 2001) it was concluded by numerical test that deploying these alternative misfit functions depended highly on the problem and the dimensions of the problem. I my thesis I have not deployed any of these alternatives.

5.5 The Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm, (Levenberg, 1944, Marquardt, 1963), is a very popular algorithm in small and medium-sized optimization problems and has proved itself to work very well in real world applications. The algorithm combines the best of the Gauss-Newton algorithm and the gradient descent algorithm. When the algorithm is far from a minimum it performs as the fast-moving gradient descent method and when it is closer to a possible solution it gradually turns to a more optimized quadratic form. The method is pseudo second order since it estimates the second order derivatives only using first order derivatives and functional evaluations. This estimation causes problems if used on nonlinear problems but the Levenberg-Marquardt algorithm evaluates the use of this estimation in each iteration and turns to the gradient descent algorithm if the use of the estimation is not valid. Furthermore, since the calculation of second order derivatives is normally computational expensive the approximated second order approach makes the algorithm very fast. The following deviations are based on (Madsen et al., 2004).

The purpose of the algorithm is to find the minimum of the sum of squares function S between observed data $\mathbf{d}_{obs}$ and data generated by the forward model $g(\mathbf{m}) = \mathbf{d}_{cal}$.

$$
S(\mathbf{d_{obs}}, \mathbf{m}) = \sum_{i=0}^{N_{data}} \left(d^i_{obs} - g^i(\mathbf{m})\right)^2 \tag{5.4}
$$

The algorithm starts at a point in the model space $\mathbf{m}_0$ and finds the next model by taking a step from $\mathbf{m}_0$ to $\mathbf{m}_1 = \mathbf{m}_0 + \mathbf{s}_0$. In iteration the algorithm gets closer to the minimum of S.

5.5.1 Finding the minimum of S

The minimum of S is found by creating a sequence of steps s_n converging at the minimum. The key of the algorithm is to find the correct step size and direction such that the algorithm will take optimal steps in the most beneficial direction. The minimum of S is found at the point where the gradient of S with respect to $\mathbf{s}$ is zero.

A linear approximation to $g(\mathbf{m}+\mathbf{s})$ is given by

$$g(\mathbf{m}+\mathbf{s}) \approx g(\mathbf{m}) + \mathbf{J}\mathbf{s}$$

and inspecting the conditions when the gradient of S is zero leads to

$$\begin{aligned} 0 &= \nabla S(\mathbf{d}_{obs}, \mathbf{m}+\mathbf{s}) \\ &\approx \mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m}) + \mathbf{J}\mathbf{s}\right) \\ &= \mathbf{J}^T\mathbf{J}\mathbf{s} + \mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m})\right) \end{aligned} \tag{5.5}$$

where $\mathbf{J}$ is the Jacobian of g. $\mathbf{J}^T\mathbf{J}$ is also know as the as the approximated Hessian $\hat{\mathbf{H}}$ and the approximated Hessian include only first order derivatives which makes $\hat{\mathbf{H}}$ easy to compute. The full Hessian of S can be written as

$$(\mathbf{H}(\mathbf{m}))_{ij} = \frac{\partial^2 S}{\partial m^j \partial m^k}(\mathbf{m}) = \sum_{i=1}^{N}\left(\frac{\partial g^i}{\partial m^j}(\mathbf{m})\frac{\partial g^i}{\partial m^k}(\mathbf{m}) + g^i(\mathbf{m})\frac{\partial^2 g^i}{\partial m^j \partial m^k}(\mathbf{m})\right).$$

but as mentioned earlier computing the second derivative term $g^i(\mathbf{m})\frac{\partial^2 g^i}{\partial m^j \partial m^k}(\mathbf{m})$ is computational expensive. By discarding the second order derivative terms and focusing on the first terms in this equation the approximate Hessian is thus

$$\hat{\mathbf{H}} = \mathbf{J}^T\mathbf{J} \tag{5.6}$$

Remembering that the goal was to find the optimal step s in equation 5.5 yields

$$\begin{aligned} 0 &= \mathbf{J}^T\mathbf{J}\mathbf{s} + \mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m})\right) \qquad \Rightarrow \\ \mathbf{s} &= -(\hat{\mathbf{H}})^{-1}\mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m})\right) \end{aligned}$$

Writing this in a iterative form highlights how the step from $\mathbf{m}_n$ to $\mathbf{m}_{n+1}$ is performed

$$\begin{aligned} \mathbf{m}_{n+1} &= \mathbf{m}_n + \mathbf{s}_n \\ &= \mathbf{m}_n - \hat{\mathbf{H}}^{-1}\mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m_n})\right). \end{aligned} \tag{5.7}$$

This is the quadratic approach also called the Gauss-Newtons method.

In the next section I will refer to the steepest decent method which is much simpler than the quadratic approach. The steepest decent method is based on the fact that a function will decrease the fastest in the direction of the negative gradient of the function. More formally the algorithm in our context updates as

$$\mathbf{m}_{n+1} = \mathbf{m}_n - \lambda\mathbf{J}^T\left(\mathbf{d}_{obs} - g(\mathbf{m_n})\right.$$

5.5.2 Levenberg's contribution

The improvement made by Levenberg (Levenberg, 1944) was to change $\hat{\mathbf{H}}^{-1}$ to $(\hat{\mathbf{H}} + \lambda\mathbf{I})^{-1}$ such that a new model could be written as

$$\mathbf{m}_{n+1} = \mathbf{m}_n - (\hat{\mathbf{H}}^{-1} + \lambda\mathbf{I})^{-1}\mathbf{J}^T(\mathbf{d}_{obs} - g(\mathbf{m_n})) \tag{5.8}$$

where $\mathbf{I}$ is the identity matrix and λ is a mixing factor. By controlling the mixing factor the algorithm can be tuned from behaving as a quadratic approach

$$\mathbf{m}_{n+1} = \mathbf{m}_n - (\hat{\mathbf{H}}^{-1})^{-1}\mathbf{J}^T(\mathbf{d}_{obs} - g(\mathbf{m_n})) \qquad \lambda << 1$$

to behaving as a steepest decent method

$$\mathbf{m}_{n+1} = \mathbf{m}_n - \frac{1}{\lambda}\mathbf{J}^T(\mathbf{d}_{obs} - g(\mathbf{m_n})) \qquad \lambda >> 1$$

In this way the algorithm can change the mixing factor to vary between the less complicated steepest decent method in flat regions and turn to the more advanced quadratic approximation near a minimum.

The tricky part of the algorithm is to determine the scaling factor λ at a given time. This is usually done by the following method

1. Make an iteration as described in equation 5.8.
2. Evaluate the value of S for the new model $\mathbf{m}_{n+1}$.
   (a) If S has increased discard the model $\mathbf{m}_{n+1}$ and increase λ by a factor 10. Return to (1).
   (b) If S has decreased save the model $\mathbf{m}_{n+1}$ and decrease λ by a factor 10. Return to (1).

Described in another way: if the algorithm is not getting closer to a minimum it should increase λ such that the algorithm behaves more as the steepest decent method. If the new model is getting better the chances are that the algorithm is getting closer to a minimum and λ should decrease such that the algorithm behaves more like the quadratic approach.

5.5.3 Marquardt's contribution

Marquardt (Marquardt, 1963) realized that with high values of λ the algorithm is performing as the steepest decent method and do not benefit from the information in the approximated Hessian. His contribution was that the approximated second derivatives could still be useful. By replacing $\mathbf{I}$ with the diagonal of the Hessian $\hat{\mathbf{H}}$

$$\mathbf{m}_{n+1} = \mathbf{m}_n - (\hat{\mathbf{H}} + \lambda \mathrm{diag}\hat{\mathbf{H}})^{-1}\mathbf{J}^T(\mathbf{d}_{obs} - g(\mathbf{m_n}))$$

In this way the algorithm can use the gradient information in the approximated Hessian even when it is behaving as the steepest decent method and take larger steps in the directions where the gradient is smaller.

Even though the algorithm saves a lot of time by not calculating the full Hessian the algorithm has the drawback that matrix inversion is needed which is a slow process when the number of dimensions grow. The maximal number of dimensions used in this thesis is $60+61=121$ which doesn't cause problems with the Levenberg-Marquardt method but for larger problems with thousands of dimensions this would be more troublesome.

The algorithm is one of the most common algorithms in optimization and curve-fitting problems and I found that reimplementing this algorithm would not be fruitful considering the three other algorithms also needed. I have used the implementation built-in to MATLAB and my implementation can be found in section B.5.

Chapter 6

Algorithm and plotting considerations

In this chapter I will describe how the algorithms were initialized and some considerations regarding the plotting of the data.

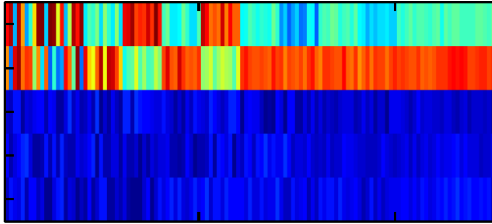


Figure 6.1: *The stacked plot of models illustrates the behavior of the algorithm. Here the Neighborhood search algorithm is shown and it is clear how the algorithm jumps between different promising areas until the algorithm settles on one area. The 2 top rows are depth parameters and the 3 bottom rows are velocity parameters. Each column represents a model and the models are generated such that new models are added to the right. The colors show the parameter value, red is high and blue is low.*

6.1 Plotting the results

For all the algorithms except Levenberg-Marquardt I have plotted the models as the algorithm produces them in a horizontally stacked plot. The color represents the value and each column is a model, see examples in figure 6.1 and 6.2. Depending on the layer boundaries being fixed or not the depth parameters are also shown. This plotting method resembles the "movie making method" described in (Mosegaard and Tarantola, 1995, p. 12). It makes it easier to show large numbers of models and identify common features within them. Another benefit from this kind of plot is that the inner workings of the algorithm are exposed. As an example it is easy to see how the Neighborhood algorithm alternates between different promising areas of the model space in figure 6.1.

For a given number of layers, N_{layer} there will be $2N_{layer} + 1$ parameters. The first N_{layer} parameters define the depth in km and following $N_{layer} + 1$ define the P-velocity km/s. The reason for the $N_{layer} + 1$ velocity parameters and not N_{layer} is that the one more velocity parameter is needed to define the velocity from the surface to the first interface.

In the Metropolis algorithm I have locked the depth parameters in the models and consequently the depth parameter will not be shown in the stacked plot.

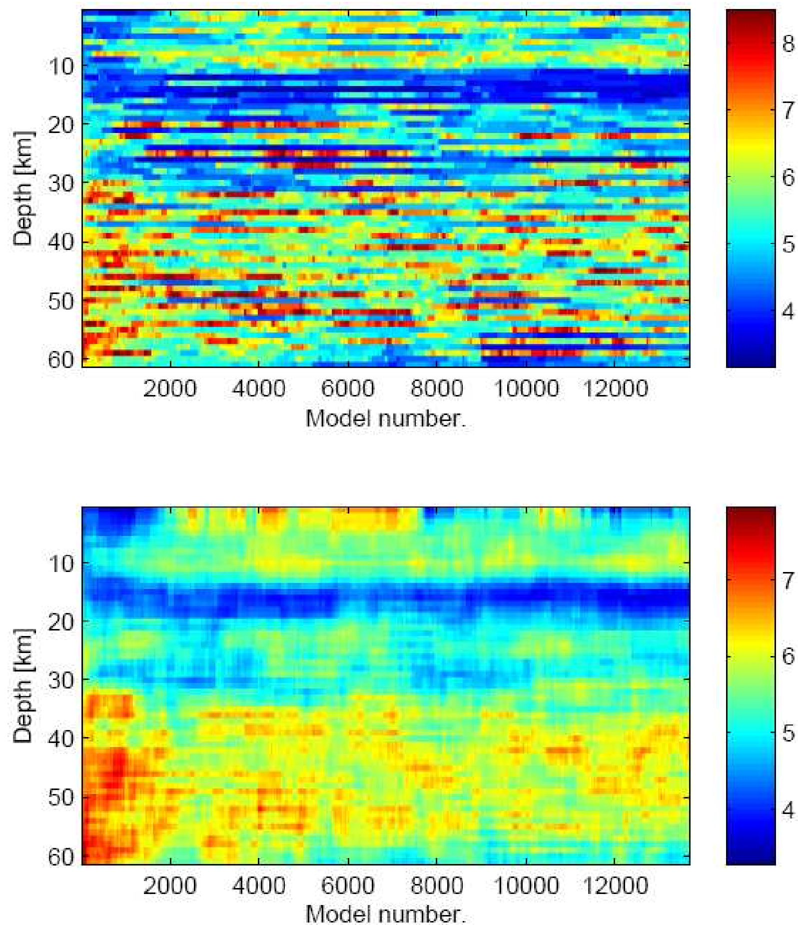
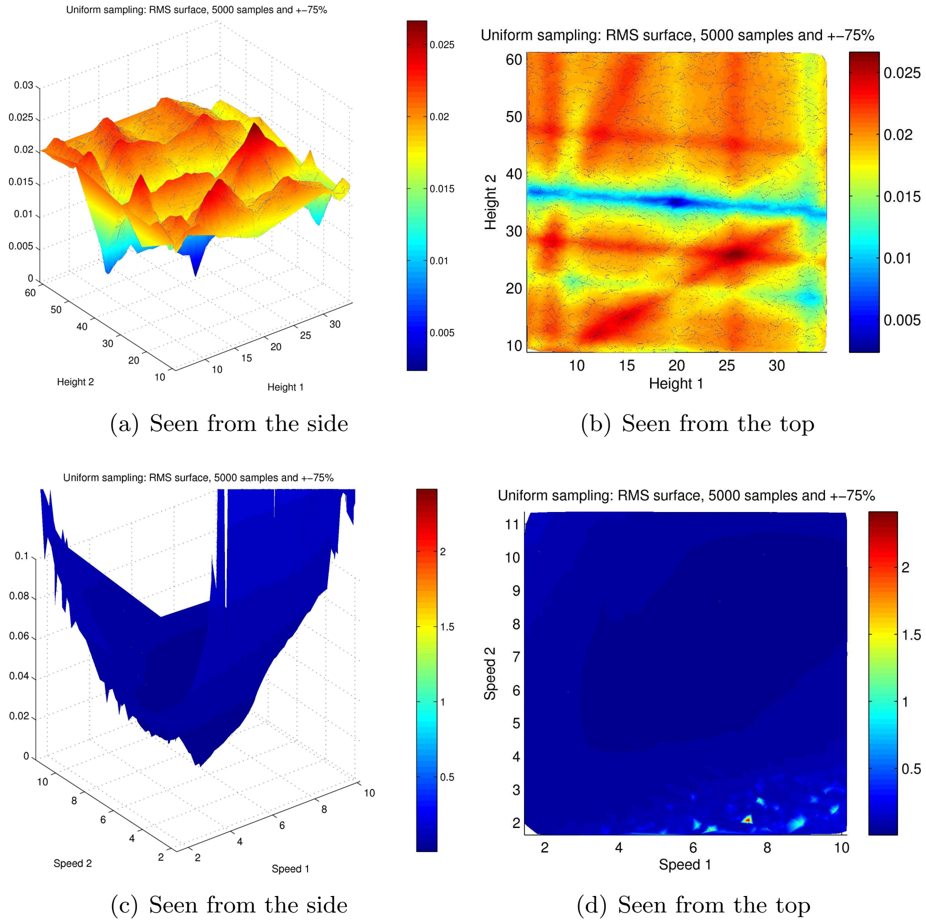


Figure 6.2: *In the top figure the raw output from the Metropolis algorithm is shown. In the bottom figure the result has been smoothed by a running mean of 5 km. In this plot the layer boundaries are fixed 1 km apart and not shown. The rows show the 61 velocity parameters.*

The stacked plot for the Metropolis algorithm shows the P-velocity as a function of depth. For all the other algorithms the depth parameter is not locked and I will plot all the parameters in the stacked plot.

In the case of the Metropolis algorithm I will be using 60 fixed layer boundaries spaced 1 km apart and 61 velocity parameters. With such a large number of layers the resulting velocity structure can be hard to interpret since the velocity of layers can vary strongly between adjacent layers. Furthermore the algorithm seems to prefer layers that alternates much in P-velocity from one layer to another. To create more physically realistic models I have chosen to filter the velocities with running average of the 5 layers. In figure 6.2 the smoothed and non-smoothed velocity structures are shown. The overall features in the velocity structure are much clearer in the smoothed plot. The marginal probability plots used with the Metropolis algorithm are created from the smoothed velocities. This has the effect that a sharp transition will become a more smooth



(a) Seen from the side (b) Seen from the top

(c) Seen from the side (d) Seen from the top

Figure 6.3: *5000 random samples ±75% of the synthetic parameters created using the Uniform search. While two parameters were varied the other three parameters were locked. The surface shows the root mean square of the distance to the real synthetic data. a,b) The two depth (named height in this plot) parameters varied. c,d) Two velocity parameters varied.*

transition and this should be kept in mind when interpreting the results.

6.2 An insight into the complexity of the problem

To illustrate the complexity of the misfit surface the algorithms are navigating on I created some simple examples seen in figure 6.3. I created 5000 random models in a five dimensional model space and varied two model parameters at a time and bounded them to be within ±75% of the IASPEI91 model. The other three parameters remained constant during the sampling. For each of the models I calculated the misfit between the IASPEI91 model and the generated models. The misfit values form the surface of the 2D planes shown in figure 6.3.

In figure 6.3(a) and 6.3(b) the two depth parameters have been perturbed. The surface is intimidating from a optimization view since it is full of local

minima and valleys which could potentially trap the algorithms.

In figure 6.3(c) and 6.3(d) two of the three velocity parameters have been perturbed. The surface appears more simple with a single minimum. Notice that in the bottom of figure 6.3(d) there is some isolated peaks indicating that the model is experiencing numerical difficulties when the value of one the velocity parameters is close to 2 km/s.

6.3 General description of the algorithms used on data

Not surprisingly the algorithms performed quite differently both in computing time and in ability to fit the waveform. This meant that each algorithm had to be fed with different starting conditions.

6.3.1 Uniform search

The Uniform search is a exclusively exploring algorithm and as described earlier the number of samples needed to resolve the structures with a certain frequency is highly depended on the dimensions of the model space. For a model with two layer boundaries and three velocity parameters the model space have five dimensions and to ensure a meager 10 models per dimension, the algorithm would need to generate 10^5 models. To provide the same model density in 61 dimensions is a gargantuan task. On this background I have decided only to use a 5-dimensional space with the Uniform search algorithm. Thus, I have created 100,000 models and selected 25 with the lowest misfit to look for a general pattern. I experimented with selecting a higher number of models but the models were so different that it was hard to interpret the results. It should also be noticed that the stacked plot for the Uniform search only shows the first 1000 models. This was done since stacking 100000 random models next to each other on a A4 paper would appear as a grey mass. By only showing 1000 models the random nature of the algorithm is visible.

The boundaries of the model space in which the algorithm could generate models was defined as between 1 and 60 km in the depth parameters and between 3 and 9 km/s in the velocity parameters. The depth parameters were generated randomly and afterwards sorted such that the highest values of depth always was found in the bottom of the depth vector. This was done to ensure that the algorithm would not feed the forward algorithm, a model where e.g. the first layer boundary was deeper than the second one.

6.3.2 Metropolis

The Metropolis algorithm has almost no computational overhead beside solving the forward problem for each generated model. This feature makes the algorithm easy to use in a high dimensional space since the computation time of one forward calculation is linear in the number of model parameters, see figure 4.2.

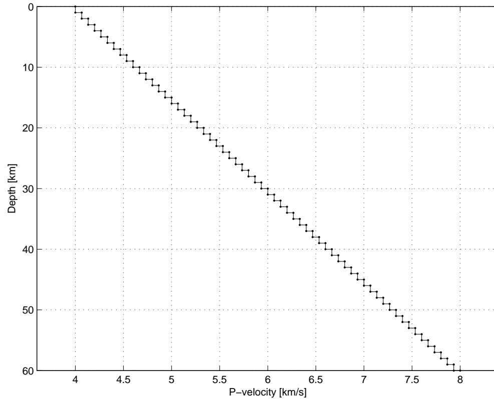


Figure 6.4: *The starting model used in the Metropolis algorithm with 60 boundaries.*

The starting model for the Metropolis algorithm is an almost continuous model with velocities starting at 4 km/s climbing slowly to 8 km/s, see figure 6.4. To achieve an acceptance rate of 30-60 % I perturbed the step sizes in both velocity and depth until the suitable size was found. I ended up allowing the Metropolis algorithm to take steps up to 1 km in the depth parameter and 1 km/s in the velocity parameter. The burn-in period varied between data sets from around 2000 models in synthetic case to 5000-7000 models for the real data set. I generated approximately 13,000 models to be sure that the algorithm was not beyond the burn-in period as well as to have enough samples after the burn-in period to draw statically independent samples.

Since the dimensions of the model space is not very important to the Metropolis algorithm I have decided to use a model with 60 layer boundaries with fixed depths which gives the Metropolis 61 velocity parameters to perturb. The choice of 60 layers of 1 km each was based on the previous studies that indicated the depth to the Moho at station Nord, would be in a depth of 30-40 km (Dahl-Jensen et al., 2003, Dahl-Jensen, 2008). In the Metropolis algorithm the acceptance probability was given as $\exp^{-\Delta S/s^2}$ where s^2 is the variance of the noise. I have used a value of $s = 0.1$ on all data sets as an estimate of the noise variance since it returned good results.

When I initially started to work with the Metropolis algorithm I had no upper or lower bounds on values for the model parameters to take. It turned out that the algorithm had no problem finding models that would fit the waveform but the velocities in a single layer could be as high as 30 km/s, which is not physical realizable. To circumvent this problem I bounded the velocities such that they were inside a physically realistic range provided by Trine Dahl-Jensen, (Dahl-Jensen, 2008)

Depth interval	Maximal P-velocity
0 - 20 km	7.0 km/s
20 - 60 km	8.5 km/s

Table 6.1: *Realistic range of P-velocities.*

Bounding the velocity values is an example of using a priori knowledge of the system to constrain the algorithm.

6.3.3 Neighborhood search

In my implementation I have used the method described in section 5.4.5. The Neighborhood search algorithm also encountered problems with a large number of model parameters. This is in part due to the implementation but mostly due to the problem of finding the boundaries of the Voronoi cells which increased rapidly with the amount of dimensions as well as the number of models. Preferably I would have used the same model with 60 layer boundaries, as for the Metropolis algorithm, but since the computation time was very high[1] I decided to use a smaller model. By inspecting the result from using the Metropolis algorithm with 60 layer boundaries I found that the overall structure for station Nord could be approximated with a model with three layer boundaries equal to a 7 dimensional model space.

The most important algorithm parameters in the Neighborhood algorithm is the number of models generated at each iteration N_s and N_r describing the number of Voronoi cells to focus on. In (Sambridge, 1999) the author applied the algorithm on a receiver function problem with, $N_s = 20$ and $N_r = 2$, generating 10,000 models in total in a 24-dimensional space. I experimented with the same values of N_s and N_r on the synthetic data set and found that the algorithm seemed to get stuck in the wrong minimum. This is discussed further in section 8.3. For this reason I selected $N_s = 250$ and $N_r = 25$ with 40 iterations. I chose 40 iterations by trial and error to find the number of iterations where the Voronoi cells had become so small that the misfit was not changing considerably.

The space in which the models could be generated was bounded in the depth parameters between 10 and 40 km and in the velocity parameters between 3 and 9 km/s.

6.3.4 Levenberg-Marquardt

The Levenberg-Marquardt algorithm was started with the same start model as for the Metropolis algorithm, see figure 6.4. The depth parameters were also locked 1 km apart. The algorithm was allowed to evaluate the forward problem 20,000 times and since the algorithm had found the gradients in 61 dimensions which amounts to around 320 iterations. I stopped the algorithm at 20,000 models after ensuring that the misfit seemed to have stabilized. However, the algorithm would also stop if the misfit could not be lowered further.

[1]The Neighborhood algorithm took 13 hours to generate 10,000 models in 7 dimensional space where the time taken to solve the forward problem 10,000 times was 45 minutes.

Chapter 7

Results

In this chapter I shall present the results from four different algorithms on both synthetic and real data.

7.1 Description of the plots

The plots are very heavy in information but I have aimed to create the figures with a single design for all the four algorithms to the extent it was possible. In a figure such as figure 7.1(a) the top figure shows the waveform of the receiver function and the selected models are shown. In the middle to the left side the generated models are shown stacked horizontally together. In the bottom to the left side the misfit is shown as a function of the number of generated models. Finally on the bottom right side the selected generated models are depicted with velocity as a function of depth.

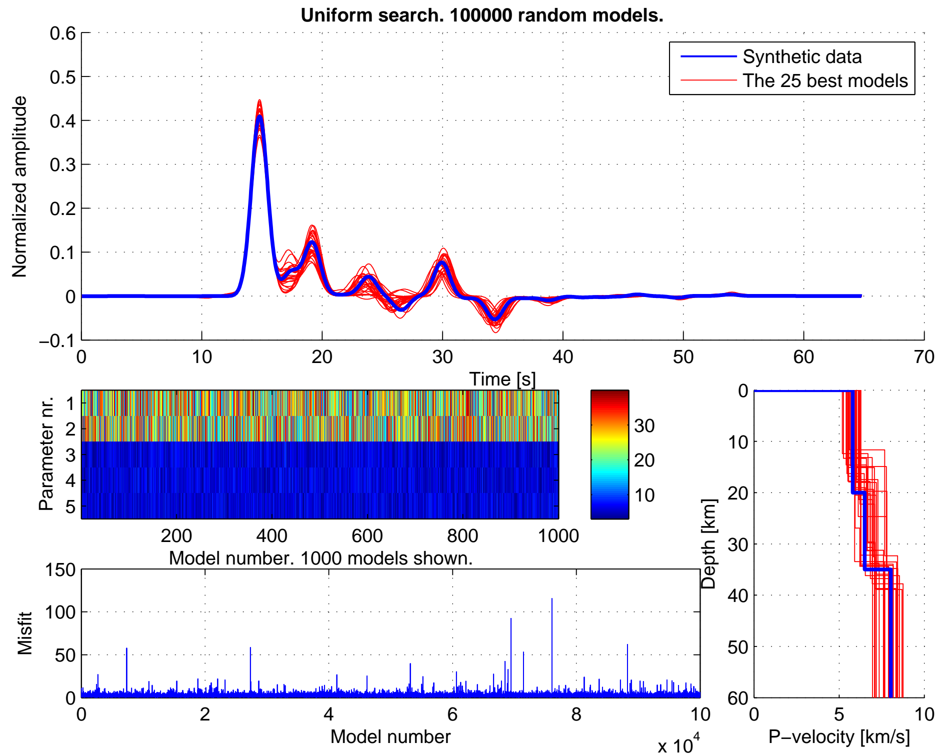
For the Metropolis algorithm the marginal probability density is shown instead of the stair case velocity-depth plot. For the Levenberg-Marquardt algorithm both a smoothed and non-smoothed version of the generated models are shown. The stacked plot is not present for the Levenberg-Marquardt algorithm since I could not extract the individual models as they were generated from the built-in MATLAB function.

7.2 Synthetic data

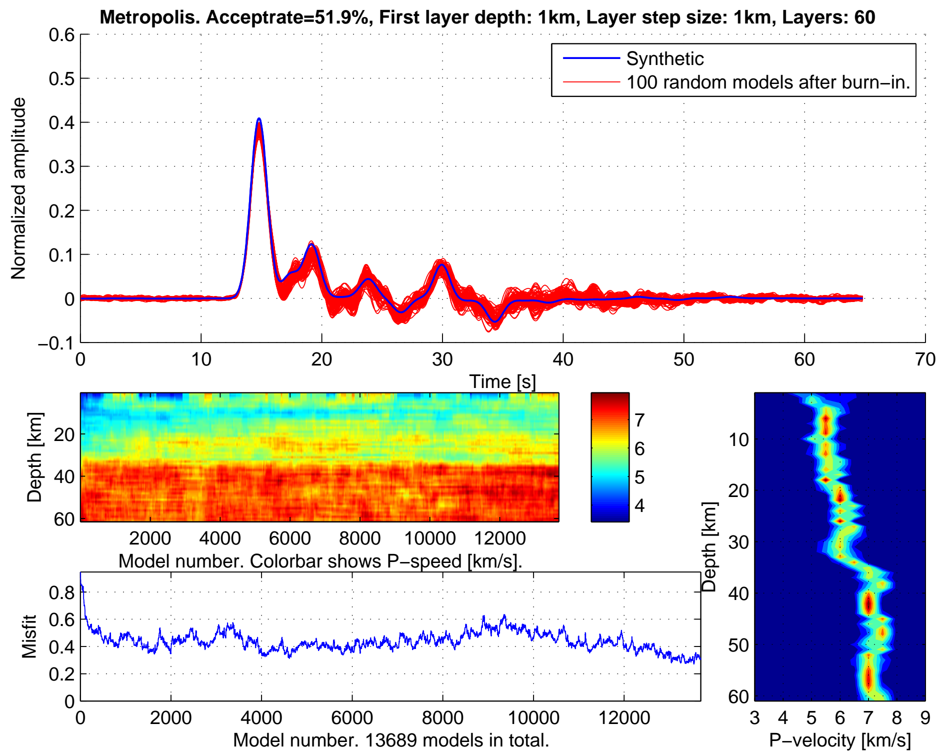
As described earlier the synthetic data created with the forward model are based on the IASPEI91 earth model shown in figure 4.1. This implies that the wavelet used to create the synthetic data set based on IASPEI91, is the same as the wavelets used in the algorithm when creating new models and thus it is reasonable to expect good fit.

7.2.1 Uniform search

From the 100,000 models generated the 25 models with the lowest misfit were selected and is seen in figure 7.1(a). These 25 models seem to have fit the waveform nicely. But a common feature seems to be that the depth to the first

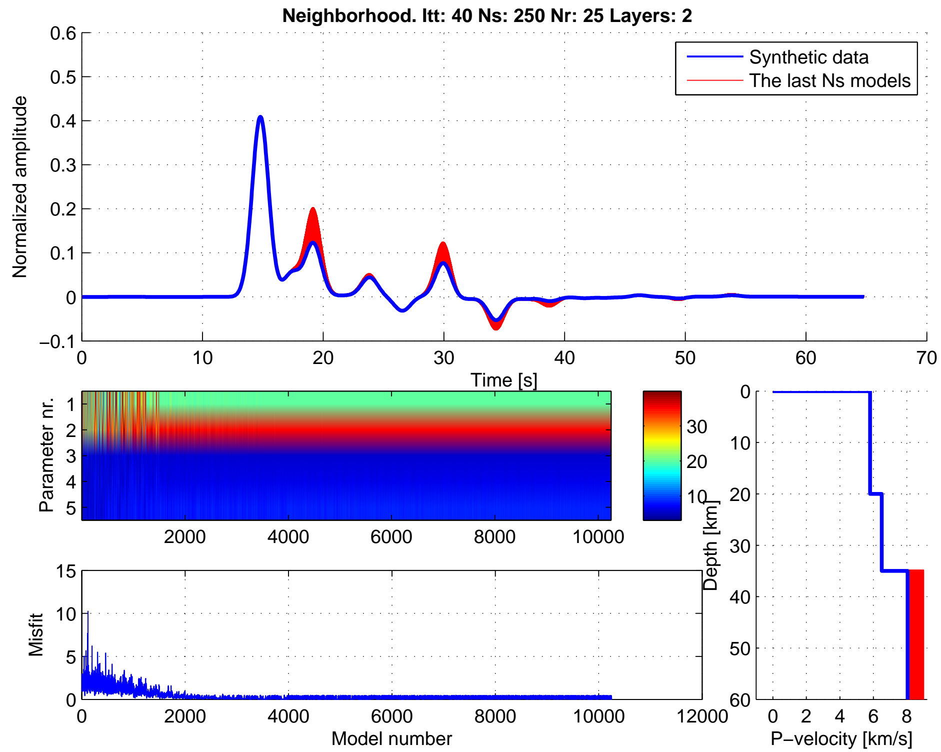


(a) Uniform search

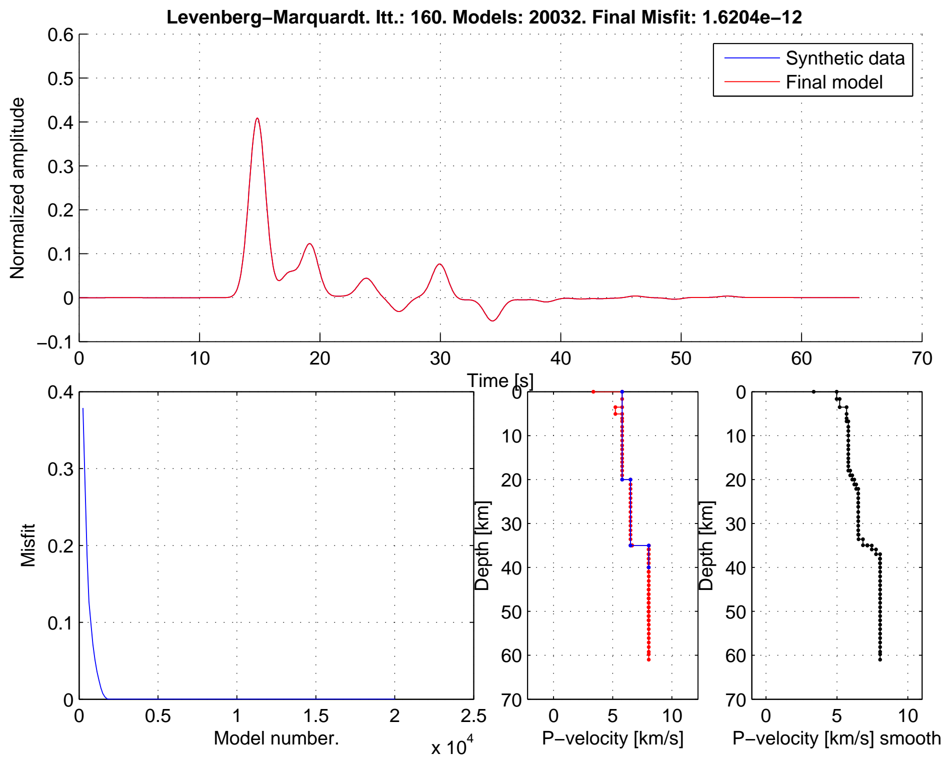


(b) Metropolis

Figure 7.1: *The result obtained through the Uniform search and the Metropolis algorithm on the synthetic data set. See description in section 7.1.*



(a) Neighborhood search



(b) Levenberg-Marquardt

Figure 7.2: *The result obtained through the Neighborhood search and Levenberg-Marquardt algorithms on the synthetic data set.*

boundary is a bit lower than the true model. As expected the misfit and the stacked plot varies wildly.

7.2.2 Metropolis

In figure 7.1(b) the Metropolis has generated approximately 13,000 models with a burn-in period of around 2000 models. The depth to the first of two layers seems to have been well resolved while the transition to the second layer is more uncertain. The deepest boundary at 35 km is the boundary with the highest velocity change and especially this boundary is quite distinct. The two layers are clearly visible when inspecting the marginal probability plot, located in the lower right corner. The center of the marginal probability plot also follows three straight lines as the ones found in the IASPEI91 model, figure 4.1.

7.2.3 Neighborhood search

The Neighborhood algorithm was successful in determining the depths to the two layers and the first two velocity parameters in figure 7.2(a). The last velocity parameter varied a bit but this might be due to a numerical problem near the border of the model space. Overall the Neighborhood algorithm did a convincing job in reconstructing the waveform and the model parameters.

7.2.4 Levenberg-Marquardt

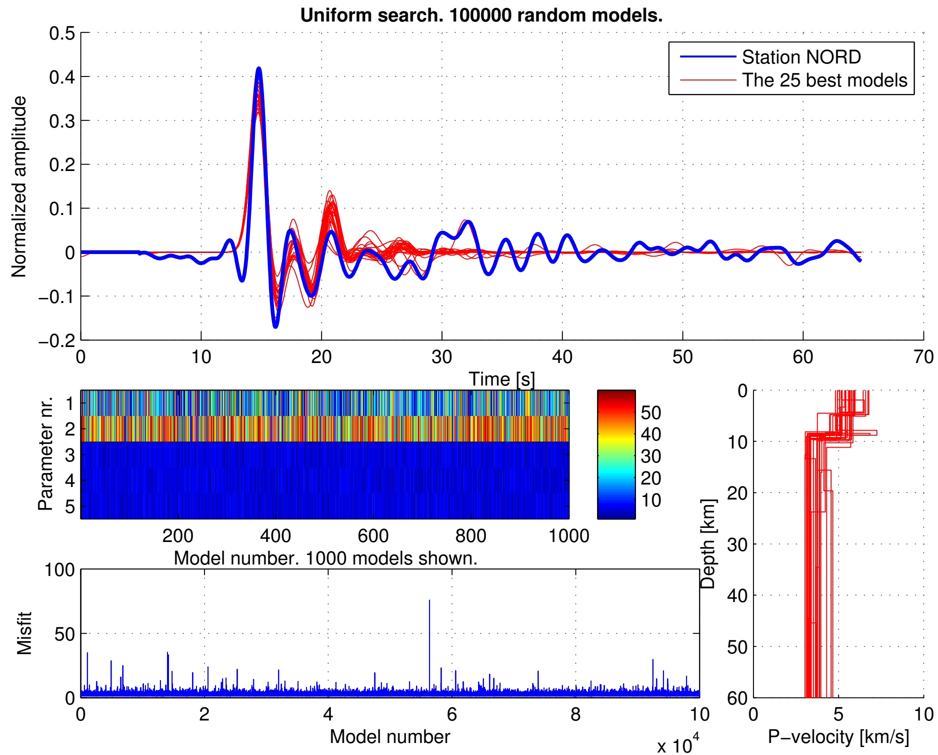
In figure 7.2(b) the Levenberg-Marquardt algorithm did a surprisingly good job in reconstructing the synthetic data considering the apparent complicated surface of the misfit function, see figure 6.3(a). The algorithm came very close to the true velocity structure in only 20 iterations. Inspecting the model and the underlying data, it is clear that beside a single layer being slightly to low at a depth of 4 km the fit is perfect. The effect of smoothing the data is very clear in this figure as well. The transitions between layers are done gradually in the smoothed model.

7.3 Data from station Nord

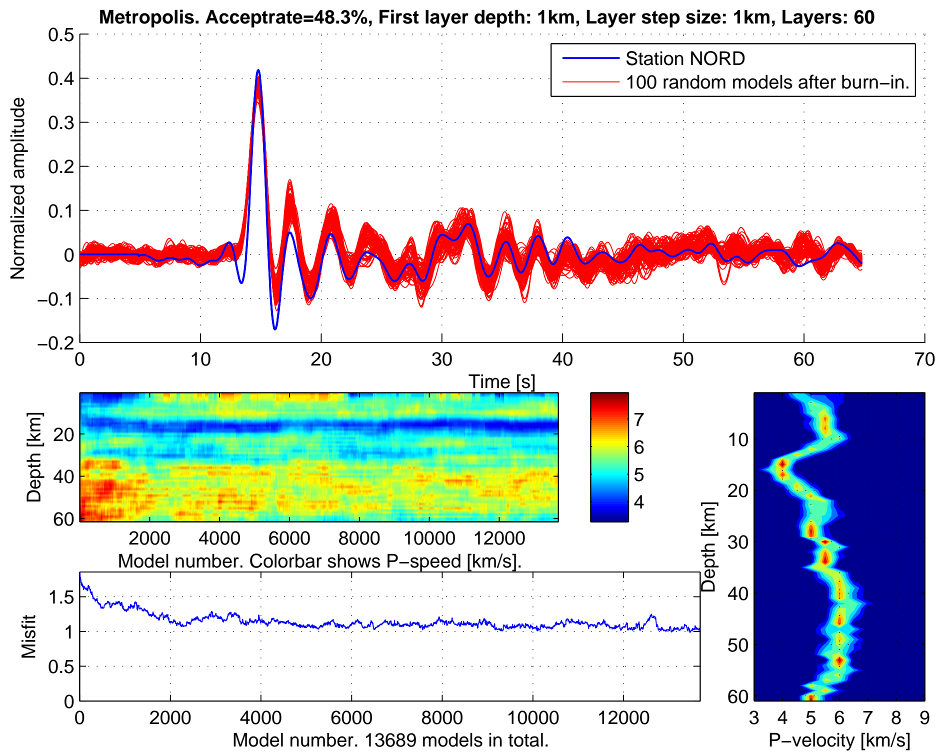
A common feature for all the results obtained for station Nord is a low-velocity layer in a depth of 10 and 20 km. This is not a feature found in other studies (Dahl-Jensen, 2008) and is most likely an artifact of the difference between the wavelet in the data and the wavelet in the forward model. This is discussed further in chapter 8.

7.3.1 Uniform search

The 25 best models in figure 7.3(a) seems to agree on a boundary in the depth of approximately 10 km corresponding to the lower velocity layer mentioned earlier. This apparent low velocity layer is very clear in the Metropolis algorithm where it is the most dominating feature. Since the Uniform search only has two depth parameters the models with the lowest misfit are the models that fit the

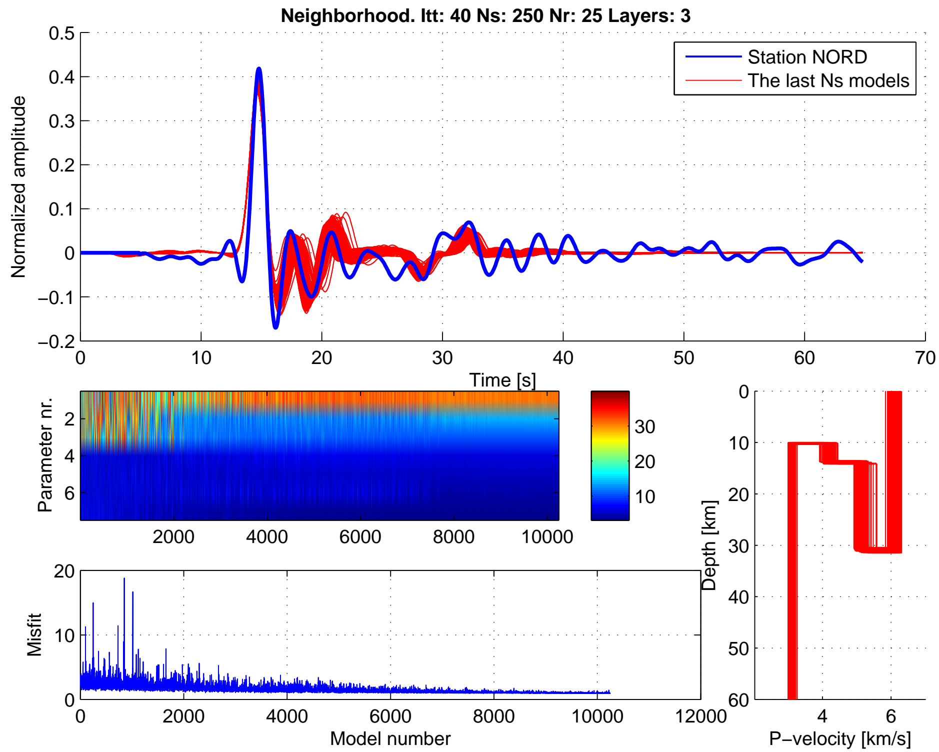


(a) Uniform search

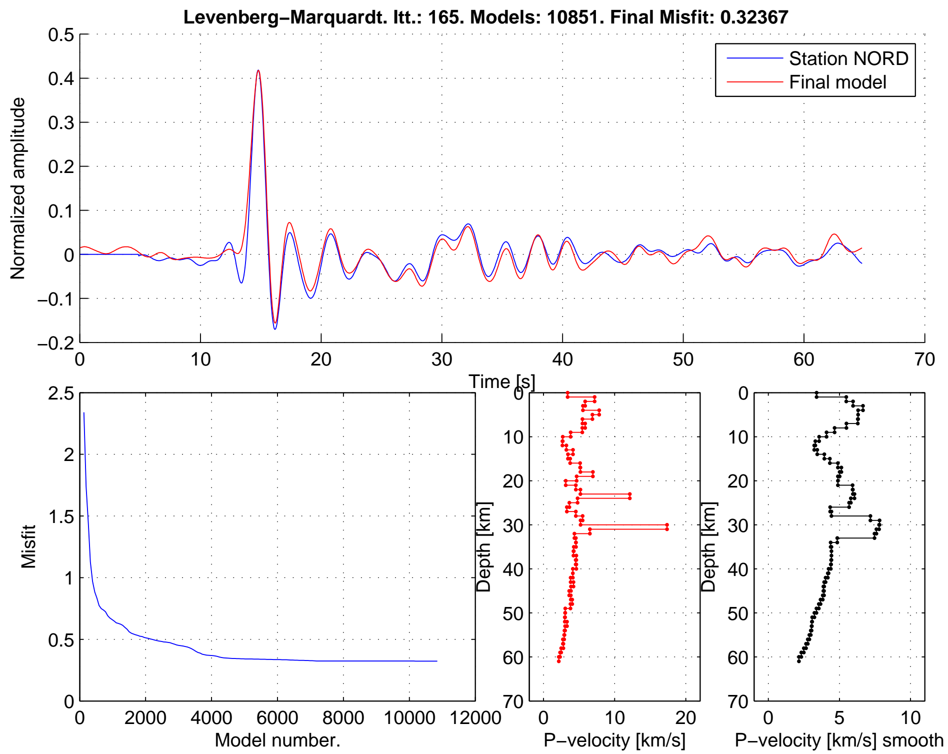


(b) Metropolis

Figure 7.3: *The result obtained through the Uniform search and the Metropolis algorithm on data collected at Station Nord.*



(a) Neighborhood search



(b) Levenberg-Marquardt

Figure 7.4: *The result obtained through the Neighborhood search and Levenberg-Marquardt algorithms on data from station Nord.*

low velocity layer. The wave does not reconstruct the multiples of the waves seen around 30 second and the models estimate an unrealistic low velocity for everything beneath 10 km. All in all the algorithm did not do a convincing job.

7.3.2 Metropolis

As in the synthetic data section the samples generated by the Metropolis algorithm in figure 7.3(b) fits the waveform convincingly. However as mentioned in the beginning of this section the apparent low velocity layer between 10 and 20 km is most likely not existing in the real world. This illustrates that even though it is possible to fit the waveform using a Gaussian wavelet the resulting models show features arising from this problem and not from the underground structure it self.

After the first 20 layers the velocity is almost constantly 5 km/s down to around 29 km where the velocity increases to 5.5 km/s and remains at this level until 35-37 km where the velocity changes to gradually to 6 km/s. The velocity remains at 6 km/s until the approximately 55 km where it gradually drops to 5 km/s. This last drop is strange and most likely also an artifact of the wavelet problem or noise in the data. The velocities are lower than what would be expected which could also be related to the wavelet problem.

7.3.3 Neighborhood search

I tuned the parameters of the model with a N_s/N_r ratio of 10 with $N_s = 250$ and $N_r = 50$. I have experimented other combinations of N_s and N_r including $N_s = 20$ and $N_r = 2$ (Sambridge, 2001) but without satisfying results.

The result of the algorithm is shown in figure 7.4(a). The most important and disturbing element is the model on the right side. It is clear that the algorithm has chosen a model that is not realistic in any sense. The first layer boundary depth is at 31 km, the second at 14 km, the third at 10 km. Even though the forward model is able to calculate the corresponding receiver function the result is not realistic. This is further discussed in chapter 8.

7.3.4 Levenberg-Marquardt

In figure 7.4(b) the Levenberg-Marquardt algorithm has managed to fit the waveform surprisingly well. Unfortunately this does not mean that the model that fit the data is very realistic. At around 10 km the velocity is very low corresponding to the low velocity layer described earlier. The most distinct features are the very fast layers at 25 and 30 km. The velocity in these layers are well outside the range described in table 6.1. In the MATLAB implementation of the algorithm it is not possible to impose bounds on the values, so this behavior could not have been avoided. Even the resulting model is not realistic, I do believe that the result is an insight into the structure. I believe that the boundary around 25 km is the lower boundary of the artificial low velocity layer and it could be speculated that the boundary around 30 km could represent the Moho.

Chapter 8

Discussion

In this chapter I will discuss the results obtained and the possible pitfalls and problems that should be considered in relation to both the workings of the algorithm and the data.

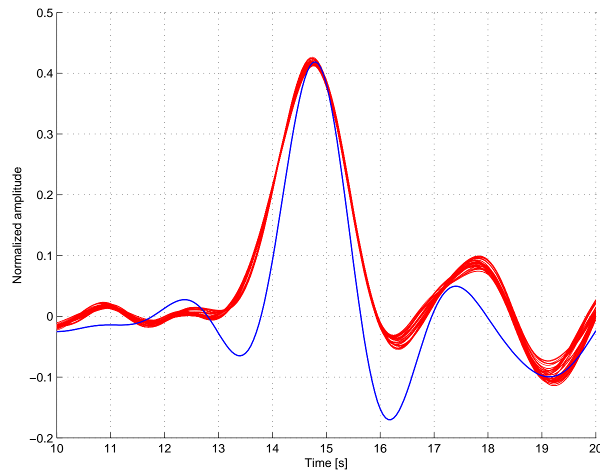
8.1 Comparison of algorithm performance on synthetic data

All the algorithms were successful in fitting synthetic data. Not surprisingly the Uniform search did not fit the data perfectly but came close to the overall result. Using the Metropolis algorithm three distinct layers were also visible both in the marginal probability plot and the stacked plot. The Neighborhood search was much more successful than Uniform search and succeeded pinpointing four of the five model parameters. I was surprised to see how well the Leveberg-Marquardt algorithm did since I would not initially have guessed that a gradient-based method would do so well in such a complex environment, see figure 6.3.

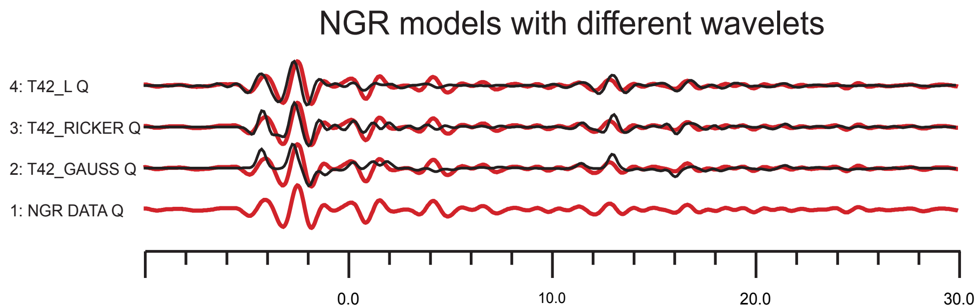
Based on these results it is clear that the Uniform search is not really a viable algorithm even for synthetic data. Considering that the algorithm did not find better models among 100,000 models, underlines the need for optimization. Using the Metropolis algorithm to sample the a posterior probability distribution provides a very interesting marginal probability plot. This plot gives an estimate of how well the different structures are resolved. The Neighborhood algorithm could be seen as an attractive algorithm when expecting the results but the computational overhead makes this algorithm the slowest of them all. The Levenberg-Marquardt algorithm was both accurate and fast since the computation of first order derivatives is simple and the computational time was almost the same as the time to solve the forward problem.

8.2 The wavelet problem

Applying the algorithm on the real world data set was not surprisingly less successful than for the synthetic data. As I have already mentioned this was most likely due to the problem with the wavelets.



(a) Gaussian wavelet and station NORD data.



(b) Effect of different wavelets

Figure 8.1: *a) The black line is the data from Station NORD and the red lines are 50 models generated using the Metropolis algorithm. The red synthetically generated models uses a Gaussian wavelet. b) 1: Shows the data recorded in Greenland, station NGR. 2: Data fitted with a Gaussian wavelet. 3: Data fitted with a Ricker Wavelet. 4: The wavelet contained in the data. For 2-4 the same depth-model was used and only the wavelet was changed. b) From (Dahl-Jensen, 2008).*

In figure 8.1(b) data from station Nord GRIP in Greenland is fitted using the same model with varying wavelets (Dahl-Jensen, 2008). As it can be seen the data fit depends heavily on the used wavelet. The Gaussian, Ricker and autocorrelated wavelets can be seen in their raw form in figure 2.8.

The forward model used in this thesis applied a Gaussian wavelet and even though it would have been interesting to see the effect of using another wavelet it was beyond the scope of this thesis to change the wavelet in the forward code.

In figure 8.1(a) data from station Nord and models generated synthetically using the Metropolis algorithm are shown together. Inspecting the fit at 13.5 and 16 seconds it is clear that a Gaussian wavelet have problems fitting the data since the original wavelet does not have these dips before the major peak. The

effect of this problem to the data fits are unknown but since the low velocity layer found at station Nord is not found in other surveys (Dahl-Jensen, 2008) it is reasonable to expect that the low velocity layer is an effect of the wavelet problem.

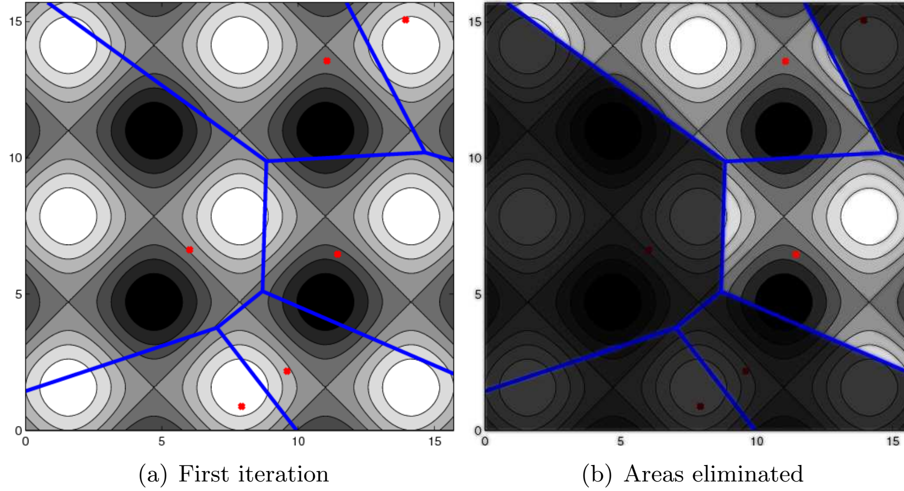
Bearing this problem in mind, it is understandable that all the algorithms had problems fitting the waveform and hence generate reasonable models. The Uniform search failed in creating any meaningful results since it used the depth parameters to fit the low velocity layer. Overall the Metropolis algorithm predicted very small velocities and found the low velocity layer as well. However it was possible to find a increase in velocity around 30 km and again at 35. This could be interpreted as the Moho as it is expected to be found in this depth and is defined as abrupt increase in seismic velocity. The Levenberg-Marquardt algorithm displayed results similar to the Metropolis algorithm and found the low velocity layer as well as two very fast layers. The layer at 25 km could be described as the lower boundary of the low velocity layer and the other fast layer at 30 km could be the Moho discontinuity.

Ultimately the wavelet problem is influencing all the algorithms and all the generated models and it is hard to tell whether or not the depth of the layer boundaries and the corresponding velocity changes is comparable to other studies.

The Neighborhood algorithm did not create useful results with the data set from Greenland. Unfortunately I discovered this problem very late due to an erroneous plotting script. The problem was that the forward algorithm could easily accept models where the layer boundaries were not sorted such that the deepest layer could be the first in the model vector. The receiver function calculated from such a defect model could appear reasonable but only when inspecting the model parameters themselves it was clear that these were not acceptable models. Even though I am glad to have found this error I did so in the last days before my deadline and managed to solve the problem for all algorithms but the Neighborhood algorithm. The fix would not take more than a few hours but since I needed to generate (13 hours) and interpret the results I could not manage to do so within the time limit. If I had more time the solution would have been to lock the depth parameters with a reasonable spacing such as 10 or 5 km apart and only let the Neighborhood algorithm perturb the velocity parameters.

8.3 The importance of N_s in the Neighborhood search algorithm

In figure 6.3 cross-sections of the model space reveals a highly irregular landscape. When starting the Neighborhood algorithm N_s random models are placed in the model space and the misfit is evaluated for each of the models. In the first iteration the algorithm samples the Voronoi cells belonging to the N_r models with the lowest misfit. This implies that the Neighborhood algorithm eliminates $N_s - N_r$ cells already in the first iteration. This behavior makes the algorithm concentrate on the regions in the model space it believes is the most



(a) First iteration (b) Areas eliminated

Figure 8.2: *a) A contour plot of the function* $Z = \sin(X) + \sin(Y)$ *and* $N_s = 6$ *randomly generated points. The white areas are peaks and black areas are holes. b)* $N_r = 2$ *and 4 of the regions will be discarded.*

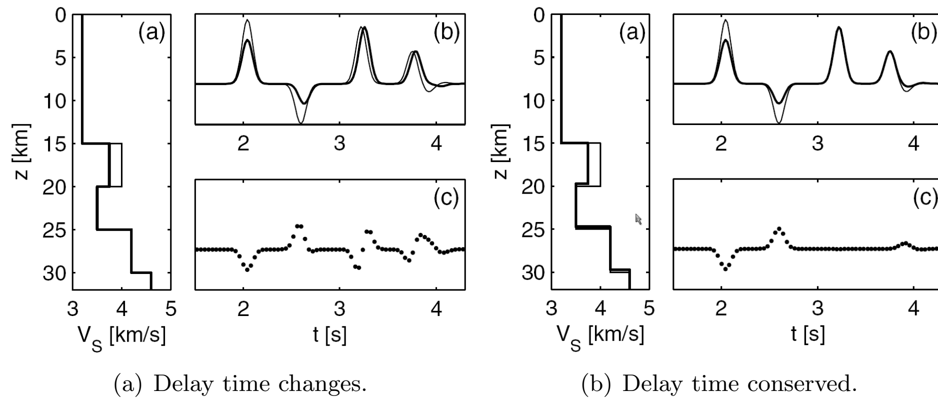
interesting at a very early stage where the Voronoi cells are still very large, see figure 5.3. It is quite possible that the algorithm will miss interesting regions if N_s is too low. As an example consider a 2D space with four holes we would like to map, see figure 8.2. If $N_s = 6$ and $N_r = 2$ the algorithm would divide the space into 6 Voronoi cells, figure 8.2(a), and generate $N_s/N_r = 3$ models inside the two best Voronoi cells. The four other regions will be discarded.

As quoted in chapter 5 the philosophy behind the algorithm is that the misfit of each of the previous models is representative of the region of space in its neighborhood. One can never be sure that a neighborhood really represents every model inside. If the misfit varies highly as seen in figure 6.3 it would be necessary with a higher value of N_s to be sure that the N_s Voronoi cells would represent their neighborhood.

The Nyquist rate is the minimum sampling rate needed to ensure that a signal with frequency of f_{peaks} is not aliased

$$f_{nyquist} = 2f_{peaks},$$

see (Lay and Wallace, 1995). In figure 6.3a at least three peaks per dimension can be seen and as such the minimum amount of samples per dimension must be $2 \cdot 3$. In five dimensions this is $(2 \cdot 3)^5 = 7776$ samples. So to ensure that the Neighborhood algorithm does not eliminate an interesting area N_s should be at least 7776. Since I have chosen $N_s = 250$ it is clear that the there will a high risk of that the Voronoi cells will not represent the regions they define. The estimate of three peaks were based on existing survey of the misfit surface for the synthetic data. For real data such a survey does not exist and either a survey should be done before the Neighborhood algorithm was applied to find the correct value of N_s or N_s should be as high as if the initial sampling was a Uniform sampling. In either case this renders the algorithm a doubtful choice.



(a) Delay time changes.

(b) Delay time conserved.

Figure 8.3: *Left: a) Two almost identical velocity models with one having a slightly lower velocity in one of the layers. b) The two corresponding receiver functions. c) The difference between the two receiver functions. Right: The same figures as to the left but here the layer depths have changed as wells as the velocity such that the delay time is conserved. It is seen that the change in velocity to the left in one layer shifts all the peaks of all other layers. From (Jacobsen and Svenningsen, 2008, p. 1759).*

The inventor of the algorithm Malcolm Sambridge is well aware of the problem and in an article from 2001, (Sambridge, 2001, p. 392) he presents an empirical formula for the number of samples, s, needed to fill up a space as function of the number of dimension N_{dim}

$$s(N_{dim}) \approx N_{dim}^{3.5}$$

This underlines the problems that the algorithm will experience in high-dimensional spaces.

8.4 A non-linear problem

The inverse problem of finding the models corresponding to a given waveform is usually regarded as highly non-linear problem. This is mostly based on an article from 1990 (Ammon et al., 1990) which addresses this problem. But very recently in 2008 (Jacobsen and Svenningsen, 2008) the non-linearity of inversion has been challenged. In the article the non-linearity of the problem is seen as a product of a unfavorable model parameterization. In the 2008 article the results from (Ammon et al., 1990) are compared with results using a new parameterization. Instead of parameterizing the problem separately in velocity and depth as in (Ammon et al., 1990) Jacobsen and Svenningsen generated a quasi-continuous model parameterized in delay time. This is done by changing the depth layer (thickness) together with the layer velocity such that the delay time of the other peaks are conserved. In the article the two parametrizations were compared, see figure 8.3, using two almost identical velocity models with one of the models having a slightly lower velocity in one of the layers. As seen

in the figure a small change in velocity in one layer affects not only the first peak but shifts all later peaks which makes the problem highly non-linear.

For all the algorithms changing the depth and velocity parameters independently the problem would appear as a highly non-linear problem. This is the case for all my algorithms beside the Levenberg-Marquardt method. In the Levenberg-Marquardt method the algorithm performs steps in the optimal directions in the multidimensional space. Knowing the gradients in all directions it is reasonable to believe that the algorithm can see, that a step changing both depth and velocity together minimizes the problem better than only changing one type of parameter per iteration.

8.5 Removing the P-pulse from the receiver function

Inspecting the receiver function it is obvious that the direct P-wave is the dominant spike in the data. And it has been speculated that removing the pulse would make the problem easier to solve. In an article from 2003 (Reading et al., 2003) this idea of removing the direct P-pulse from the receiver function was proposed. But later (Svenningsen and Jacobsen, 2004) it was shown that the difference between the usual LQT rotation, see section 2.1.1, which retains the P-pulse, and the newly proposed rotation without the P-pulse was very small in theory and it was argued that the effect would be absent in practice.

Chapter 9

Conclusions

Given the nature of synthetic data it is easy to validate whether or not the algorithms did recover model results in the synthetic data. The case was that all the algorithms did a fairly good job in recovering the waveform and the underlying model.

Validating the results from the real world data is more problematic. Fortunately, Greenland was investigated in 2003 using receiver function analysis (Dahl-Jensen et al., 2003, p. 390). In the survey an approximated depth to the Moho underneath a series of seismic stations was provided. The depth to the Moho at station Nord was approximated to 30±2 km. Later (Dahl-Jensen, 2008) the depth to the Moho for the same station was found to be 33 km. Neither the Uniform nor Neighborhood search algorithms resulted in reasonable models but both the Metropolis and Levenberg-Marquardt algorithms showed a velocity increase around 30 km. Due to the wavelet problem this result does not necessarily mean that the algorithms found the Moho. But it does give some confidence that the algorithm could perform quite well if the wavelet embedded in the data had been used.

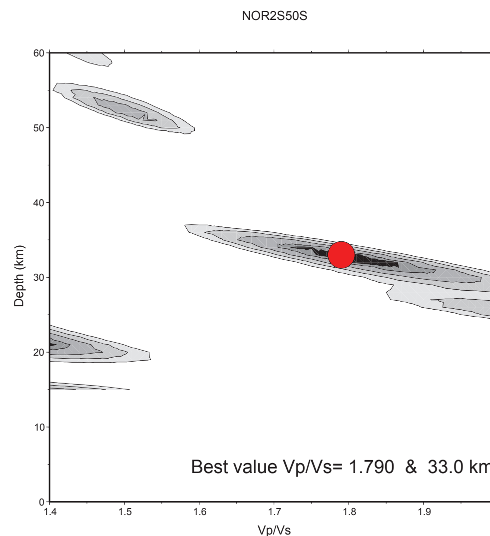


Figure 9.1: *The result of grid search in a two-dimensional model space following the method (Zhu and Kanamori, 2000). The parameters are the Poisson ratio V_p/V_s and the depth to the Moho. The data were from station Nord and the forward model used the autocorrelated wavelet. From (Dahl-Jensen, 2008).*

The Levenberg-Marquardt method fitted the receiver function surprisingly well. Actually the algorithm fitted the data too well by overfitting the data. The data contains noise and possible 3D effects structures in the underground not accounted for in the forward model. The algorithm has created non-existing

velocity changes to fit the waveform perfectly. This should be considered when interpreting the results from the Levenberg-Marquardt algorithm. In my opinion the best method to apply on real data is the Metropolis algorithm. Not only is it fast but it also provides the much needed uncertainties on the results. From the marginal probability plot it is easy to extract the mean and standard deviations for each layer.

It was expected that the Uniform search algorithm would perform badly in high dimensional problems and it can only be recommended for very small problems. On the other hand I had expected much from the Neighborhood algorithm. It turned out that this algorithm also faces great problems with high dimensional model spaces and I would not consider using this algorithm in other problems.

In light of the recent speculation in oil exploitation in Greenland and the increased interest in the arctic area receiver function analysis is a highly relevant method for extracting geological information from terrain otherwise difficult to survey by conventional means.

9.1 Further work

If time was not an issue the most interesting thing to investigate in this thesis would be to change the forward model so I could use any given wavelet. If the forward model was using the embedded wavelets I would expect a much better data fit and removal of the low-velocity layer.

Another subject for further work would be to implement the parametrization described in (Jacobsen and Svenningsen, 2008) and evaluate the results from the algorithms.

While investigating the Levenberg-Marquardt algorithm it became clear that other more similar but more advanced methods such as the trust region method (Celis et al., 1984) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (Broyden, 1970, Fletcher, 1970, Goldfarb, 1970, Shanno, 1970) existed. Considering the surprisingly good results from the Levenberg-Marquardt method I would be interested in implementing these alternatives.

Bibliography

Ammon, C. J. (1991). The isolation of receiver effects from teleseismic P waveforms. *Bulletin of the seismological Society of America*, 81(6):2504–2510.

Ammon, C. J. (1997). An overview of receiver-function analysis. Webpage. Department of Geosciences Pennsylvania State University. Available from: http://eqseis.geosc.psu.edu/~cammon/HTML/RftnDocs/rftn01.html.

Ammon, C. J., Randall, G. E., and Zandt, G. (1990). On the nonuniqueness of receiver function inversions. *Journal of Geophysical Research*, 95(B10):15303–15318.

Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22(4):469–483. Available from: http://www.qhull.org.

Birch, F. (1964). Density and Composition of Mantle and Core. *Journal of Geophysical Research*, 69.

Broyden, C. (1970). The convergence of a class of double-rank minimization algorithms. II: The new. *J. Inst. Math. Appl.*, 6:222–231.

Celis, M., Dennis, J., and Tapia, R. (1984). A trust region strategy for nonlinear equality constrained optimization. *Numerical Optimization*, pages 71–82.

Clayton, R. and Wiggins, R. (1976). Source shape estimation and deconvolution of teleseismic body waves. *Geophys. JR Astron. Soc*, 47:151–177.

Dahl-Jensen, T. (2008). Personal communication. GEUS – De Nationale Geologiske Undersøgelser for Danmark og Grønland.

Dahl-Jensen, T., Larsen, T., Woelbern, I., Bach, T., Hanka, W., Kind, R., Gregersen, S., Mosegaard, K., Voss, P., and Gudmundsson, O. (2003). Depth to Moho in Greenland: receiver-function analysis suggests two Proterozoic blocks in Greenland. *Earth and Planetary Science Letters*, 205(3-4):379–393.

Ekrem, Z. (2002). *The shear wave velocity structure of the eastern Marmara region by using receiver function analysis*. PhD thesis, Department of Geophysics, Istanbul Technical University.

Fletcher, R. (1970). A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322.

Goldfarb, D. (1970). A family of variable metric updates derived by variational means. *Mathematics of Computing*, 24(109):23–26.

Jacobsen, B. H. (2008). Personal Communications. Department of Earth Sciences, University of Aarhus.

Jacobsen, B. H. and Svenningsen, L. (2008). Enhanced Uniqueness and Linearity of Receiver Function Inversion. *Bulletin of the Seismological Society of America*, 98(4):1756.

Kennett, B. L. N. (1983). *Seismic Wave Propagation in Stratified Media*. Cambridge University Press.

Kennett, B. L. N. and Engdahl, E. R. (1991). Traveltimes for global earthquake location and phase identification. *Geophysical Journal International*, 105(2):429–465.

Kreysig, E. (1999). *Advanced Engineering Mathematics*. John Wiley & Sons, Inc, 8 edition.

Langston, C. A. (1979). Structure under mount Rainier, Washington, inferred from teleseismic body waves. *J. Geophys. Res.*, 84(B9):4749–4762.

Lay, T. and Wallace, T. C. (1995). *Modern Global Seismology*, volume 58 of *International geophysics series*. Academic Press.

Levenberg, K. (1944). A method for the solution of certain nonlinear problems in least squares. *Q. Appl. Math*, 2(2):164–168.

Madsen, K., Nielsen, H. B., and Tingleff, O. (2004). Methods for non-linear least squares problems. Lecture notes, Institute of Informatics and Mathematical Modeling, Technical University of Denmark. Available from: www2.imm.dtu.dk/pubdb/views/edoc_download.php/3215/pdf/imm3215.pdf.

Marquardt, D. (1963). An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math*, 11(2):431–441.

Menke, W. (1984). *Geophysical Data Analysis: Discrete Inverse Theory*. Academic Press.

Metropolis, N. (1987). The beginning of the Monte Carlo method. *Los Alamos Science*, 15(Special Issue):125–130.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087.

Mohorovičić, A. (1909). Das beben vom 8. *Jahrbruch met. Obs. Zagreb*, 9:1–63.

Moore Gordon, E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

Mosegaard, K. and Sambridge, M. (2002). Monte Carlo analysis of inverse problems. *Inverse Problems*, 18:R29–R54.

Mosegaard, K. and Tarantola, A. (1995). Monte carlo sampling of solutions to inverse problems. *Journal of Geophysical Research*, 100(B7):12431–12447.

Phinney, R. A. (1964). Structure of the earth's crust from spectral behavior of long-period body waves. *Journal of Geophysical Research*, 69:2997.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in Fortran 77*, volume 1 of *Fortran Numerical Recipes*. Cambridge University Press, 2 edition.

Reading, A., Kennett, B., and Sambridge, M. (2003). Improved inversion for seismic structure using transformed, S-wavevector receiver functions: Removing the effect of the free surface. *Geophys. Res. Lett*, 30(19):1981.

Sambridge, M. (1998). Exploring multidimensional landscapes without a map. *Inverse Problems*, 14:427–440.

Sambridge, M. (1999). Geophysical inversion with a neighborhood algorithm – I. search a parameter space. *Geophysical Journal International*, 138:479–494.

Sambridge, M. (2001). Finding acceptable models in nonlinear inverse problems using neighborhood algorithm. *Inverse Problems*, 17:387–403.

Sambridge, M. and Mosegaard, K. (2002). Monte Carlo methods in geophysical inverse problems. *Rev. Geophys*, 40(3):1009.

Shanno, D. (1970). Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation*, 24(111):647–656.

Shearer, P. M. (1999). *Introduction to Seismology*. Cambridge University Press.

Svenningsen, L. and Jacobsen, B. H. (2004). Comment on "Improved inversion for seismic structure using transformed, S-wavevector receiver functions: Removing the effect of the free surface" by Anya Reading, Brian Kennett and Malcolm Sambridge. *Geophys. Res. Lett*, 31:L24609.

Tarantola, A. (2005). *Inverse problem theory and methods for model parameter estimation*. SIAM.

Upadhyay, S. K. (2004). *Seismic reflection processing: With special reference to Anisotropy*. Springer.

Wall, L., Christiansen, T., and Orwant, J. (2000). *Programming Perl*. O'Reilly.

Walsh, B. (2004). Markov chain Monte Carlo and Gibbs sampling. Lecture notes. Available from: http://www.stat.columbia.edu/~liam/teaching/neurostat-spr07/papers/mcmc/mcmc-gibbs-intro.pdf.

Weisstein, E. W. (2008). Point-line distance – 3-dimensional. MathWorld – A Wolfram Web Resource. Available from: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html.

Yilmaz, Ö. and Doherty, S. (1987). *Seismic data processing*. Society of Exploration Geophysicists.

Zhu, L. and Kanamori, H. (2000). Moho depth variation in southern California from teleseismic receiver functions. *Journal of Geophysical Research*, 105(B2).

Appendix A

Mathematics

A.1 Convolution theorem

Theorem 1 (Convolution) *Let $f(t)$ and $g(t)$ satisfy the hypothesis of the existence theorem. The product of their transform $F(s) = \mathscr{L}(f)$ and $G(s) = \mathscr{L}(g)$ is the transform $H(s) =$ of the* ***convolution*** *$h(t)$ of $f(t)$ and $g(t)$, which is denoted by $(f * g)(t)$ and defined by*

$$h(t) = (f * g)(t) = \int_0^t f(\tau)g(t-\tau)\mathrm{d}\tau \tag{A.1}$$

From (Kreysig, 1999, p. 279).

A.2 Approximate Hessian

$$\frac{\partial F}{\partial p_j}(\mathbf{p}) = \sum_{i=1}^{m} f_i(\mathbf{p})\frac{\partial f_i}{\partial p_j}(\mathbf{p})$$

It can be seen that the gradient of F is

$$F'(\mathbf{p}) = \mathbf{J}(\mathbf{p})^T f(\mathbf{p})$$

We will also need the second derivatives of F, so we calculate the Hessian.

$$\frac{\partial^2 F}{\partial p_j \partial p_k}(\mathbf{p}) = \sum_{i=1}^{m} \left(\frac{\partial f_i}{\partial x_j}(p)\frac{\partial f_i}{\partial x_k}(p) + f_i(p)\frac{\partial^2 f_i}{\partial x_j \partial p_k}(p) \right)$$

such that

$$F''(p) = \mathbf{J}(p)^T\mathbf{J}(p) + \sum_{i=1}^{m} f_i(x)f_i''(x). \tag{A.2}$$

A approximation to the Hessian can be made by only considering the first term, $\mathbf{J}^T\mathbf{J}$, in this equation, which is computational much easier than calculating the second derivatives.

By looking at the derivative of F

$$x_{i+1} = x_i - \left(\mathbf{J^T J} + \lambda \mathrm{diag}\left[\mathbf{H}\right]\right)^{-1} d \tag{A.3}$$

A.3 The distance from a line to a point in 5 dimensions

This following is based distance in 3D given by (Weisstein, 2008) but expanded to 5D.

Consider two points $\mathbf{a} = (a_1, a_2, a_3, a_4, a_5)$ and $\mathbf{b} = (b_1, b_2, b_3, b_4, b_5)$ such that the vector along the line would be given by

$$\mathbf{v} = \begin{pmatrix} a_1 & (b_1 - a_1) \cdot t \\ a_2 & (b_2 - a_2) \cdot t \\ a_3 & (b_3 - a_3) \cdot t \\ a_4 & (b_4 - a_4) \cdot t \\ a_5 & (b_5 - a_5) \cdot t \end{pmatrix}$$

The squared distance between a point on the line with parameter t and point in space $\mathbf{c} = (c_1, c_2, c_3, c_4, c_5)$ is hence

$$d^2 = \begin{matrix} (a_1 - c_1) + (b_1 - a_1) \cdot t + \\ (a_2 - c_2) + (b_2 - a_2) \cdot t + \\ (a_3 - c_3) + (b_3 - a_3) \cdot t + \\ (a_4 - c_4) + (b_4 - a_4) \cdot t + \\ (a_5 - c_5) + (b_5 - a_5) \cdot t + \\ (a_1 - c_1) + (b_1 - a_1) \cdot t \end{matrix} \tag{A.4}$$

To find the minimum distance I differentiate d^2 with regard to t and find the t value where $\frac{\mathrm{d}d^2}{\mathrm{d}t} = 0$. This t_{best} is put back into equation A.4 which gives a expression for the minimum distance.

The following MATLAB code solves the problem of finding the distance from a line to a point in 5D.

```
syms t
syms a1 a2 a3 a4 a5;
syms b1 b2 b3 b4 b5;
syms c1 c2 c3 c4 c5;

dsq = ...
    ((a1-c1)+(b1-a1)*t)^2 + ...
    ((a2-c2)+(b2-a2)*t)^2 + ...
    ((a3-c3)+(b3-a3)*t)^2 + ...
    ((a4-c4)+(b4-a4)*t)^2 + ...
    ((a5-c5)+(b5-a5)*t)^2;

dsqdiff  = diff(dsq,t) - 0;
tbest    = solve(dsqdiff,t);
solution = subs(dsq, t, tbest)
```

The solution is quite simple for 2D or 3D but in 5D is rather unpleasant.

```
>> solution =
(a1-c1+(b1-a1)*(-a1*b1+a1^2+c1*b1-c1*a1-a2*b2+a2^2+c2*b2-c2*
a2-a3*b3+a3^2+c3*b3-c3*a3-a4*b4+a4^2+c4*b4-c4*a4-a5*b5+a5^2+
c5*b5-c5*a5)/(b1^2-2*a1*b1+a1^2+b2^2-2*a2*b2+a2^2+b3^2-2*a3*
b3+a3^2+b4^2-2*a4*b4+a4^2+b5^2-2*a5*b5+a5^2))^2+(a2-c2+(b2-
```

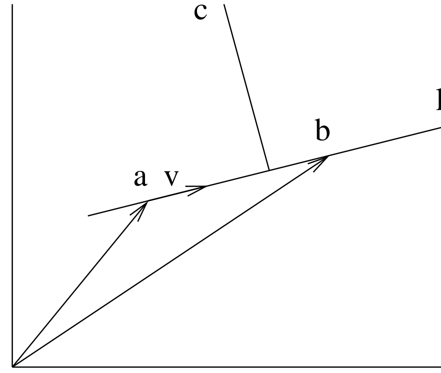


Figure A.1: *The distance from a line to a point in N_{dim} dimensions from parametrized view point*

```
a2)*(-a1*b1+a1^2+c1*b1-c1*a1-a2*b2+a2^2+c2*b2-c2*a2-a3*b3+
a3^2+c3*b3-c3*a3-a4*b4+a4^2+c4*b4-c4*a4-a5*b5+a5^2+c5*b5-c5*
a5)/(b1^2-2*a1*b1+a1^2+b2^2-2*a2*b2+a2^2+b3^2-2*a3*b3+a3^2+
b4^2-2*a4*b4+a4^2+b5^2-2*a5*b5+a5^2))^2+(a3-c3+(b3-a3)*(-a1*
b1+a1^2+c1*b1-c1*a1-a2*b2+a2^2+c2*b2-c2*a2-a3*b3+a3^2+c3*b3-
c3*a3-a4*b4+a4^2+c4*b4-c4*a4-a5*b5+a5^2+c5*b5-c5*a5)/(b1^2-
2*a1*b1+a1^2+b2^2-2*a2*b2+a2^2+b3^2-2*a3*b3+a3^2+b4^2-2*a4*
b4+a4^2+b5^2-2*a5*b5+a5^2))^2+(a4-c4+(b4-a4)*(-a1*b1+a1^2+
c1*b1-c1*a1-a2*b2+a2^2+c2*b2-c2*a2-a3*b3+a3^2+c3*b3-c3*a3-
a4*b4+a4^2+c4*b4-c4*a4-a5*b5+a5^2+c5*b5-c5*a5)/(b1^2-2*a1*
b1+a1^2+b2^2-2*a2*b2+a2^2+b3^2-2*a3*b3+a3^2+b4^2-2*a4*b4+
a4^2+b5^2-2*a5*b5+a5^2))^2+(a5-c5+(b5-a5)*(-a1*b1+a1^2+c1*
b1-c1*a1-a2*b2+a2^2+c2*b2-c2*a2-a3*b3+a3^2+c3*b3-c3*a3-a4*
b4+a4^2+c4*b4-c4*a4-a5*b5+a5^2+c5*b5-c5*a5)/(b1^2-2*a1*b1+
a1^2+b2^2-2*a2*b2+a2^2+b3^2-2*a3*b3+a3^2+b4^2-2*a4*b4+a4^2+
b5^2-2*a5*b5+a5^2))^2
```

A.4 The distance from a line to a point in N_{dim} dimensions

Given two points $\mathbf{a}$ and $\mathbf{b}$ constituting a line l in a N_{dim} dimensional space. The unit vector along l is given by

$$\mathbf{v} = \frac{\mathbf{b} - \mathbf{a}}{|\mathbf{b} - \mathbf{a}|}$$

and the points on l can be written as

$$\mathbf{p}(t) = \mathbf{a} + \mathbf{v}t \qquad t \in \mathcal{R}.$$

The goal is to find the perpendicular distance to l from a point $\mathbf{c}$. The distance to a point $\mathbf{p}(t)$ on l can be written as

$$\begin{aligned} d(\mathbf{c},\mathbf{p}(t)) &= \left(\sum_{i=1}^{N}(c_i - p_i(t))^2\right)^{1/2} \\ &= \left(\sum_{i=1}^{N}(c_i - (a_i + v_i t))^2\right)^{1/2}. \end{aligned} \tag{A.5}$$

Defining the squared distance as

$$\begin{aligned} d^2 &= (d(\mathbf{c},\mathbf{p}(t)))^2 \\ &= \sum_{i=1}^{N}(c_i - (a_i + v_i t))^2 \end{aligned}$$

the minimum distance to can be found by differentiating d^2 with respect to t and setting it equal to zero

$$\begin{aligned} \frac{\mathrm{d}d^2}{\mathrm{d}t} &= -2\sum_{i=1}^{N}(c_i - (a_i + v_i t_0))\, v_i \\ &= -2\sum_{i=1}^{N}(c_i - a_i)\, v_i - 2t_0\sum_{i=1}^{N} v_i^2 \\ &= -2\sum_{i=1}^{N}(c_i - a_i)\, v_i - 2t_0 \\ &= 0 \end{aligned}$$

Solving this equation yields the t that minimizes the distance

$$t_0 = -\sum_{i=1}^{N}(c_i - a_i)\, v_i \tag{A.6}$$

Inserting A.6 into A.5 reveals the minimum distance from $\mathbf{c}$ to the line l as

$$d_{min} = \left(\sum_{i=1}^{N}\left(c_i - \left(a_i + v_i\sum_{i=1}^{N}(c_i - a_i)\, v_i\right)\right)^2\right)^{1/2}.$$

Appendix B

Code

B.1 Uniform Search

```
% $Id: UniformSearch.m 146 2008-12-12 16:20:31Z tjansson $
clc; clear all; clc;
tegn          = 1; % Save the figures to harddisk
setuppath     % Redefines the path include the lib folder
outputpath    = '/home/gfy-1/tjansson/'; % Folder create images are ↩
    saved to.
time          = datestr(now, 'yyyy-mmdd-HHMM');


%% Algorithm parameters
m_amount          = 100000; % The amount of samples
m_timepersample   = 0.127; % in seconds
m_sorted_amount   = min(m_amount*0.1,25);
onlyvelocity      = 0;


%% Load the data

% Loads the data
data              = load('R_NOR_THOMAS_CLEAN_1296');
data_name         = 'Station NORD';
data_name_file    = 'NORD';
data_length       = length(data);

% Define the data sets that you are trying to fit the function to.
%data_para        = [20, 35, 5.80, 6.5, 8.04];
%data             = rf_forward(data_para(1:2), data_para(3:5));
%data_length      = length(data);
%data_name        = 'Synthetic data';
%data_name_file   = 'synthetic';



%% Setup the parameter space

% The upper and lower bounds of the parameter space
% The starting model
zmin              = 1;
zmax              = 60;
layers_stepsize   = 30;
layers            = length(zmin:layers_stepsize:zmax);
modelyaxis        = 'Depth [km]';
m_zs              = zmin:layers_stepsize:zmax;
```

```
m_VP              = 4:(8-4)/layers:8;
m_start           = [m_zs m_VP];
startmodel        = rf_forward(m_zs, m_VP);

ZSbound_lo = 01; % Minimum depth
ZSbound_up = 60; % Maximum depth
VPbound_lo = 03; % Minimum velocity
VPbound_up = 09; % Maximum velocity

% Generate the models array - to hold model parameters and misfit
% (depth1, depth2, ..., vel1, vel2, ..., misfit)
models                        = zeros(m_amount,length(m_start)+1);
models(:,length(m_start)+1)   = 10; % The initial misfit

for i=1:m_amount
    models(i,1:length(m_zs)) = sort((1 + (ZSbound_up-1)*rand(1,length(↩
        m_zs))));
end

for i=1:m_amount
    models(i,length(m_zs)+1:length(m_start)) = VPbound_lo+(VPbound_up-↩
        VPbound_lo)*rand(1,length(m_VP));
end

%% Calculating
tic
for i=1:m_amount
    models(i,length(m_start)+1)= ...
            rms_data_many(...
            models(i,1:length(m_zs)),...
            models(i,length(m_zs)+1:length(m_start)), ...
            data);

    if mod(i,25) == 0
        fprintf('Iterations left: %4.0f', m_amount-i )
        fprintf(' Time left: %4.2f min.\n', ...
            ((m_amount-i)*m_timepersample)/60 )
    end
end
timespend = toc;
fprintf('Total time spend: %5.0f seconds\n', timespend )
fprintf('Time per mode    : %5.3f seconds\n', timespend/m_amount )



%% Processing

%Find the best positions
[value,position] = sort(models(:,length(m_start)+1));
m_sorted          = models(position(1:m_sorted_amount),:);
m_sorted_length   = size(m_sorted,1);

% Calculate the receiver function for last amount of samples
rf_functions = zeros(m_sorted_length,length(data));
for i=1:m_sorted_length;
    rf_functions(i,:) = rf_forward(...
        m_sorted(i,1:length(m_zs)), ...
        m_sorted(i,length(m_zs)+1:length(m_start)));
end
```

B.2 Metropolis

```
% $Id: Metropolis.m 148 2008-12-14 00:48:38Z tjansson $
clc; clf; clear all;
tegn=1; % Save the figures to harddisk
anim=0; % Create animated GIF's?
wavelet=0; % Draws wavelet;
setuppath % Redefines the path include the lib folder
time = datestr(now, 'yyyy-mmdd-HHMM');

%% Initialize observede data and start model
% Loads the data
data              = load('R_NOR_THOMAS_CLEAN_1296');
data_name         = 'Station NORD';
data_name_file    = 'NORD';
data_length       = length(data);

%data_zs           = [20,35]; %IASPEI layers in crust
%data_VPs          = [5.8,6.5,8.04]; %IASPEI velocities
%data              = rf_forward(data_zs, data_VPs);
%data_name         = ['Synthetic'];
%data_name_file    = ['synthetic'];
%data_length       = length(data);

% The starting model
zmin              = 1;
zmax              = 60;
layers_stepsize = 1;
layers            = length(zmin:layers_stepsize:zmax);
modelyaxis        = 'Depth [km]';
m_zs              = zmin:layers_stepsize:zmax;
m_VP              = 4:(8-4)/layers:8;
m_start           = [m_zs m_VP ];
startmodel        = rf_forward(m_zs, m_VP);

%% Initialize the Metropolis algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

runtime   = 180.0; % in minuttes
tsample   = 0.417; % time per sample in seconds 0.101 with 2 layers and ↩
    0.306 with 4
nsamples = ceil((runtime * 60) / tsample);
samples   = zeros(length(m_start)+1, nsamples);
accept    = zeros(nsamples, 1);

% This generates the random number, both positive and negative, to be ↩
    added
% at each itteration. The random number is ie. between 0 and maxZ for ↩
    the
% heights and between 0 and maxVP for the heights
pct          = 0.05;
%maxZ          = 40*pct;
%maxVP         = 10*pct;
maxZ         = 1.0;
maxVP        = 1.0;
randZstep    = (0+(maxZ-0))*rand(nsamples,1).*sign(rand(nsamples,1)-0.5);
randVPstep   = (0+(maxVP-0))*rand(nsamples,1).*sign(rand(nsamples,1)↩
    -0.5);
randdir      = fix((length(m_start)-1)*rand(nsamples,1)+1);
randdirVP    = fix((length(m_zs)+1)+(length(m_start)+1-(length(m_zs)+1))*↩
    rand(nsamples,1));

%% The itteration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic

m_cur         = m_start; % The first model is the startmodel
m_cur_misfit = 10000; % The first models misfit is bad
```

```
misfit        = zeros(nsamples,1); % Array to store models

for i=1:nsamples;

    % The new model is based on the old model
    m_new = m_cur;

    % Shows the process
    if mod(i,25)== 2
        done=(i/nsamples)*100; % Shows the progress in procent
        timeleft_min=((nsamples-i)*tsample);
        clf; %If this is not done this image will use 3GB of memory in ↩
            10000 models.
        subplot(2,1,1)
        imagesc(samples(length(m_zs)+1:length(m_start),:)); colorbar; %↩
             show only speed
        xlabel(modelxaxis);
        ylabel(modelyaxis);
        hold on
        subplot(2,1,2)
        plot(samples(length(m_start)+1,:))
        grid
        xlabel(modelxaxis);
        ylabel('RMS');
        axis([0 length(samples(length(m_start)+1,:)) 0 max(samples(↩
            length(m_start)+1,:))])
        drawnow
        fprintf('\n %3.2f seconds left, ', timeleft_min)
        fprintf('%3.2f%% done, ', done)
        fprintf('rms = %3.8f%.', m_cur_misfit)
    end

    % If this loop is included both depth and velociy are changed.
    %if randdir(i) <= length(m_zs)
    %    %Varies the depth
    %    m_new(randdir(i)) = m_new(randdir(i)) - randZstep(i);
    %else

        %Varies the velocity
        % Pick out a random element of m_new and some randomness
        % But only in speeds the depths do not change
        if randdirVP(i)-length(m_zs)+1 < 20 % The depth is less than ↩
            20;
            %If the speed gets larger than 8.5km/s subtract instead of ↩
                add
            % Trine would like to have 7 but I am affraid it will ↩
                interfer.
            if m_new(randdirVP(i)) + randVPstep(i) > 7
                m_new(randdirVP(i)) = m_new(randdirVP(i)) - randVPstep↩
                    (i);
            else
                m_new(randdirVP(i)) = m_new(randdirVP(i)) + randVPstep↩
                    (i);
            end
        else

            % If the speed is larger than 8.5km/s subtract instead of ↩
                add
            if m_new(randdirVP(i)) + randVPstep(i) > 8.5
                m_new(randdirVP(i)) = m_new(randdirVP(i)) - randVPstep(↩
                    i);
            else
                m_new(randdirVP(i)) = m_new(randdirVP(i)) + randVPstep(↩
                    i);
            end
        end
```

```
    %end

    % Make sure that the models are never zero or negative
    m_new = abs(m_new);
    for j=1:length(m_new);
        if (m_new(j) == 0)
            m_new(j) = 0.1;
        end
    end

    % Calculate the acceptance of the model
    % Mosegaard & Tarantola, 1995,
    % Journal of Geophysical Research, side 6
    m_new_misfit = misfit_data_many(m_new(1:length(m_zs)), ...
        m_new(length(m_zs)+1:length(m_start)), ...
        data);

    if m_new_misfit <= m_cur_misfit
        P_accept = 1;
    else
        deltaS = m_new_misfit - m_cur_misfit;
        P_accept = exp(-(deltaS)/(0.1)^2);
    end

    if (rand < P_accept)
        m_cur = m_new;
        m_cur_misfit = m_new_misfit;
        accept(i) = 1;
    end

    % Save all the models and the misfit for later use.
    samples(1:length(m_start),i) = m_cur;
    samples(length(m_start)+1,i) = m_cur_misfit;
end

% Running average on the data.
% This is a hack since filter would run the average from 1 in the ↩
    begning
% and not the value it should, so I copy the first row in 5 times and
% remove them again after the filtering.
tmp1 = samples(length(m_zs)+1,:);
tmp2 = samples(length(m_zs)+1:length(m_start),:);
tmp3 = [tmp1; tmp1; tmp1; tmp1; tmp1; tmp1; tmp1; tmp1; tmp1; tmp1;tmp2↩
    ];
windowSize = 5;
samples_smooth = filter(ones(1,windowSize)./windowSize, 1, tmp3);
samples_smooth = samples_smooth(11:end,:);

% Calculate the acceptrate
naccept = 0;
for i=1:nsamples
    if accept(i) == 1
        naccept=naccept+1;
    end
end
acceptrate = (naccept/nsamples)*100;

% Show some statistics
fprintf('\n\n Acceptrate: %3.2f', acceptrate)
fprintf('\n Number of samples: %3.0f', nsamples)
totaltime = toc;
timepersample = totaltime/nsamples;
fprintf('\n Time per itteration: %3.3f', timepersample)
fprintf('\n Totaltime:  %3.3f \n\n', totaltime)
```

B.3 Neighborhood search – Discretizing the axis

```
% $Id: NeighborhoodSmallSteps.m 149 2008-12-14 12:06:08Z tjansson $
clear all; clc;
tegn          = 1; % Save the figures to harddisk
draw_voronoi = 0; % Draw the Voronoi cells?
setuppath         % Redefines the path include the lib folder
time          = datestr(now, 'yyyy-mmdd-HHMM');
testing       = 0;
if testing
    profile on
end

%% Algorithm parameters
Itt = 40;  % The number of itterations this process should continue.
N_s = 250;  % Amount of models to create at begining and each level.
N_r = 25;  % The amount of models with the best misfit.

% This ensures that no bad N_s N_r combos are chosen
if mod(N_s,N_r) ~=0
    fprintf(1, 'ERROR - mod(N_s,N_r) is not 0\nexiting\n')
    break
end

% Defines the stepsize in size the cells
stepsize_orig      = 1.0000; % First stepsize
stepsize_min       = 0.0010; % Stepsize should get small than this
border_resolution = 0.0010; % The maximum distance to the border

%% Load the data
data              = load('R_NOR_THOMAS_CLEAN_1296');
data_name         = 'Station NORD';
data_name_file   = 'NORD';
data_length       = length(data);

% Define the data sets that you are trying to fit the function to.
%data_para         = [20, 35, 5.80, 6.5, 8.04];
%data              = rf_forward(data_para(1:2), data_para(3:5));
%data_length       = length(data);
%data_name         = 'Synthetic data';
%data_name_file   = 'synthetic';

%% Setup the parameter space
% The upper and lower bounds of the parameter space
% The starting model

zmin              = 1;
zmax              = 60;
layers_stepsize = 25;
layers            = length(zmin:layers_stepsize:zmax);
modelyaxis        = 'Depth [km]';
m_zs              = zmin:layers_stepsize:zmax;
m_VP              = 4:(8-4)/layers:8;
m_start           = [m_zs m_VP];
startmodel        = rf_forward(m_zs, m_VP);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% VERY IMPORTANT
% When the following loop as well as the special version of dim loop ↩
    below
% is active only the velocity parameter will change.
onlyvelocity = 0;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Set the boundary of the space
bounds(1:length(m_zs),1) = 10; % Depth lower
bounds(1:length(m_zs),2) = 40; % Depth upper
bounds(length(m_zs)+1:length(m_start),1) = 03; % Speed lower
bounds(length(m_zs)+1:length(m_start),2) = 09; % Speed upper

% Generate the models array - to hold model parameters and misfit
% (depth1, depth2, ..., vel1, vel2, ..., misfit)
models                          = zeros(Itt*N_s,length(m_start)+1);
models(:,length(m_start)+1)     = 10; % The initial misfit
for i=1:length(m_start)
    models(1:N_s,i) = bounds(i,1)+(bounds(i,2)-bounds(i,1))*rand(N_s,1)↩
        ;
end
tmpmodel                        =  zeros(1,length(m_start));

if onlyvelocity
    for i=1:((Itt+1)*N_s)
        models(i,1:length(m_zs)) = m_zs;
    end
end

%% Testing arrays and parameters
counterlooptoc     = zeros(1, Itt);
lowupranddim_num   = length(m_start)*(N_s/N_r)*N_r*Itt;
lowupranddim_tmp   = 1;
lowupranddim       = zeros(lowupranddim_num,6);
testdim            = 1;

%% Calculating Cell

% This function calculates the euclidean function
distance = @(a,b) sqrt(sum((a(1:length(m_start))-b(1:length(m_start)))↩
    .^2));

% Loop through the amount of itterations speciefied
for counter=1:Itt

    clf
    subplot(2,1,1)
    imagesc(models(:,1:length(m_start))');
    colorbar
    subplot(2,1,2)
    plot(models(:,length(m_start)+1))
    xlabel('Model number')
    ylabel('Misfit')
    grid on
    drawnow

    tic
    clear distplot distplot2
    itmp = 0; %Used for saving the new points to models

    %Calculate the misfit of the new points
    for countermisfit=(1+(counter-1)*N_s):(counter*N_s)
        models(countermisfit,length(m_start)+1) = ...
            rms_data_many(...
            models(countermisfit,1:length(m_zs)),...
            models(countermisfit,length(m_zs)+1:length(m_start)), ...
            data);
    end
    lastmisfit=models(countermisfit,length(m_start)+1);

    %The models is sorted acording to misfit value
    [value,position] = sort(models(:,length(m_start)+1));
```

```
    % Just for the plotting part
    if counter == 1
        firstbestmodels = models(position(1:N_r),:);
    end

    % For each of the promising models
    for index=1:N_r

        % Define the promising models, corresponds to point A in the ↩
            article
        promising = models(position(index),1:length(m_start));

        %The gibs sampler is started at the center of promising point.
        tmpmodel_orig = promising;

        % Generate N_s/N_r model inside each promising model
        for ins=1:N_s/N_r

            % Loops through the dimensions
            if onlyvelocity % Varies only velocity
                dimtmp = length(m_zs)+1:length(m_start);
            else % Varies both depth and velocity
                dimtmp = 1:length(m_start);
            end

            for dim=dimtmp;

                y=1;

                %%% Upper boundary

                if onlyvelocity
                    tmpmodel_orig(1,1:length(m_zs)) = m_zs;
                end

                uppertmp    = 100;
                movingpoint = tmpmodel_orig;
                stepsize    = stepsize_orig;
                dist        = zeros(counter*N_s,1);

                % Stop at upper bound
                while movingpoint(dim) < bounds(dim,2)
                    % Find the distance to the owner of the Voronoi ↩
                        cell.
                    distpromis = norm(movingpoint-promising);

                    % Find the distance to all other models
                    for j=1:(counter*N_s)
                        dist(j,1)   = norm(movingpoint-models(j,1:length↩
                            (m_start)));
                    end
                    sortdist = sort(dist);

                    % Bound is met if another model is closer
                    if sortdist(1) < distpromis
                       % Stop the search if the distance to the border ↩
                           is
                       % less than the border resolution or the ↩
                           stepsize
                       % can not be reduced further.
                       if abs(sortdist(1)-distpromis) < ↩
                           border_resolution || stepsize < stepsize_min
                           uppertmp = movingpoint(dim);
                           break
                       else
                           % The border resolution is not met or the
```

```
                % stepsize can be reduced further.
                movingpoint(dim) = movingpoint(dim) - ↩
                    stepsize;
                stepsize = stepsize/2;
            end
        end

        % Take a step
        movingpoint(dim) = movingpoint(dim) + stepsize;

        %TESTING - the distances to movingpoint
        if testing
            if dim==testdim && ins==1
                distplot(:,y)=dist(:,1);
                distplot2(:,y)=movingpoint(dim);
                y=y+1;
            end
        end
    end

    %%% Lower boundary

    lowertmp    = -100;
    movingpoint = tmpmodel_orig;
    stepsize    = stepsize_orig;
    dist        = zeros(counter*N_s,1);

    while movingpoint(dim) > bounds(dim,1) % Stop at lower ↩
        bound
        % Find the distance to the owner of the Voronoi ↩
            cell.
        distpromis = norm(movingpoint-promising);

        % Find the distance to all other models
        for j=1:(counter*N_s)
            dist(j,1)   = norm(movingpoint-models(j,1:length↩
                (m_start)));
        end
        sortdist = sort(dist);

        % Bound is met if another model is closer
        if sortdist(1) < distpromis
           % Stop the search if the distance to the border ↩
                is
           % less than the border resolution or the ↩
                stepsize
           % can not be reduced further.
           if abs(sortdist(1)-distpromis) < ↩
                border_resolution || stepsize < stepsize_min
                lowertmp = movingpoint(dim);
                break
           else
                % The border resolution is not met or the
                % stepsize can be reduced further.
                movingpoint(dim) = movingpoint(dim) + ↩
                    stepsize;
                stepsize = stepsize/2;
            end
        end

        % Take a step
        movingpoint(dim) = movingpoint(dim) - stepsize;

        %TESTING - the distances to movingpoint
        if testing
            if dim==testdim && ins==1
```

```
                            distplot(:,end+1)=dist(:,1);
                            distplot2(:,end+1)=movingpoint(dim);
                            y=y+1;
                        end
                    end
                end

                % Find the boundaries.
                upper = max(min(uppertmp,bounds(dim,2)),bounds(dim,1));
                lower = min(max(lowertmp,bounds(dim,1)),bounds(dim,2));
                if lower > upper
                    error('The lower boundary is larger than the upper'↩
                        )
                end

                % Find a new random point between the boundaries
                randstep        = lower+(upper-lower)*rand;
                tmpmodel(1,dim) = randstep;

                % Testing
                lowupranddim(lowupranddim_tmp,[1 2 3 4 5 6]) = ...
                [lowertmp,lower,uppertmp,upper,randstep,dim];
                lowupranddim_tmp = lowupranddim_tmp+1;
            end %ends dim loop

            % Save the new model in models.
            itmp = itmp+1;

            if onlyvelocity
                models(counter*N_s+itmp,length(m_zs)+1:length(m_start))↩
                     = ...
                tmpmodel(length(m_zs)+1:length(m_start));
            else
                models(counter*N_s+itmp,1:length(m_start)) = tmpmodel;
            end

        end %ends ins loop
    end %ends index loop

    counterlooptoc(counter) = toc;
    % Printing some progress
    fprintf('%5.0f iterations of', counter*N_s)
    fprintf('%5.0f done. ', Itt*N_s+N_s)
    fprintf('Time used: %5.2f seconds. ', counterlooptoc(counter))

    time_used    = sum(counterlooptoc);
    time_first   = counterlooptoc(1);
    time_est     = 0;
    for i=2:Itt
        time_est = time_est +time_first*2;
    end
    time_est = time_est*2;
    fprintf('Estimated time left: %5.2f seconds. ', time_est - ↩
        time_used)
    fprintf('Misfit: %5.2f \n', lastmisfit)

end %ends counter loop

time_finished = sum(counterlooptoc);
fprintf('Time used: %5.2f seconds \n', time_finished)

% Calculate the misfit of the last N_s points
for countermisfit=(counter*N_s+1):(counter*N_s+N_s)
    models(countermisfit,length(m_start)+1) = ...
        rms_data_many(...
        models(countermisfit,1:length(m_zs)),...
```

```
        models(countermisfit,length(m_zs)+1:length(m_start)), ...
        data);
end

% Sort the models a final time
[final_value,final_position] = sort(models(:,length(m_start)+1));

if testing
    profile viewer
end
```

B.4 Neighborhood search – Exact intersection

```
% $Id: NeighborhoodRefined.m 137 2008-12-10 16:39:15Z tjansson $
clear all; clc;
setuppath    % Redefines the path include the lib folder
outputpath   = '/home/tjansson/'; % Folder create images are saved to.
time         = datestr(now, 'yyyy-mmdd-HHMM');
tegn         = 1; % Save the figures to harddisk

%% Algorithm parameters

Itt = 10;  % The number of itterations this process should continue.
N_s = 20;  % Amount of models to create at begining and each level.
N_r = 2;  % The amount of models with the best misfit.

%% Load the data
%data              = load('R_NOR_THOMAS_CLEAN_1296');
%data_name         = ['Station NORD'];
%data_name_file    = ['NORD'];
%data_length       = length(data);

% Define the data sets that you are trying to fit the function to.
data_para         = [20, 35, 5.80, 6.5, 8.04];
data              = rf_forward(data_para(1:2), data_para(3:5));
data_length       = length(data);
data_name         = ('Synthetic data');
data_name_file    = ('synthetic');

%% Setup the parameter space
% The upper and lower bounds of the parameter space
% The starting model
zmin              = 1;
zmax              = 60;
layers_stepsize = 1;
layers            = length(zmin:layers_stepsize:zmax);
modelyaxis        = 'Depth [km]';
m_zs              = zmin:layers_stepsize:zmax;
m_VP              = 4:(8-4)/layers:8;
m_start           = [m_zs m_VP];
startmodel        = rf_forward(m_zs, m_VP);

% Set the boundary of the space
bounds(1:length(m_zs),1) = 01; % Depth lower
bounds(1:length(m_zs),2) = 60; % Depth upper
bounds(length(m_zs)+1:length(m_start),1) = 03; % Speed lower
bounds(length(m_zs)+1:length(m_start),2) = 09; % Speed upper

%% Calculating Cell
% Generate the models array - to hold model parameters and misfit
% (depth1, depth2, ..., vel1, vel2, ..., misfit)
models                        = zeros(Itt*N_s,length(m_start)+1);
```

```
models(:,length(m_start)+1)   = 10; % The initial misfit

% Populate the first N_s models with random models
for i=1:length(m_start)
    models(1:N_s,i) = bounds(i,1)+(bounds(i,2)-bounds(i,1))*rand(N_s,1)↩
        ;
end

%% Calculating Cell
tic
for counter=1:Itt

    %The misfit of the new points are calcualted
    for countermisfit=(1+(counter-1)*N_s):(counter*N_s)
            models(countermisfit,length(m_start)+1) = ...
        rms_data_many(...
        models(countermisfit,1:length(m_zs)),...
        models(countermisfit,length(m_zs)+1:length(m_start)), ...
        data);

        if mod(countermisfit,50) == 0
            fprintf('%5.0f misfit of', countermisfit)
            fprintf('%5.0f\n', counter*N_s)
        end
    end

    %The models is sorted acording to misfit value
    [value,position] = sort(models(:,length(m_start)+1));
    itmp = 0;

    % Just for the plotting part
    if counter == 1
        firstbestmodels = models(position(1:N_r),:);
    end

    % Printing some progress
    fprintf('%5.0f iterations of', counter*N_s)
    fprintf('%5.0f done. ', Itt*N_s+N_s)
    fprintf('Timeleft around %3.0f seconds. \n', 0.15*(Itt*N_s+N_s - ↩
        counter*N_s))

    for index=1:N_r % For each of the promising models

        % Printing some progress
        fprintf('Promising model number %5.0f of', index)
        fprintf('%5.0f\n',N_r)

        % promising corresponds to point A in the article
        promising = models(position(index),1:length(m_start));

        for ins=1:N_s/N_r % Generate N_s/N_r model inside each ↩
            promising model

            %The gibs sampler is started at the center of promising ↩
                point.
            tmpmodel = promising;

            for dim=1:length(m_start)% We wish to take a step in each ↩
                dimension

                % Create the points the define the i'th axis.
                steadyL       = tmpmodel;
                steadyU       = tmpmodel;
                steadyL(dim) = bounds(dim,1);
                steadyU(dim) = bounds(dim,2);

```

```
for i=1:(counter*N_s)
    % For each of the other models we calculate the ↩
        distance to
    % every other model to find the boundary of the ↩
        Voronoi cell

    if i==1 % This array should be cleared everytime i↩
        =1
        boundary = zeros(counter*N_s,1);
    end

    % This is current model from the whole set of ↩
        models
    cur = models(i,1:length(m_start));

    %distk is the perpendicular distance from sample k ↩
        from
    %the current axis. See Sambridge 1999, page 485 ↩
        (19).
    %the current axis is defined as the one going ↩
        through
    %the promising and the steadypoint.

    % a and b defines the line. v is in the direction ↩
        of
    % the line
    a     = steadyL;
    b     = steadyU;
    v     = (b-a)/norm(b-a);
    % Distance from promising to line
    c     = promising;
    distk = sqrt(sum((c-(a+v*((c-a)*v'))).^2));
    % Distance from current point to line
    c     = cur;
    distj = sqrt(sum((c-(a+v*((c-a)*v'))).^2));

    if not(cur(dim) == promising(dim))
        %Calculating the boundary points, Sam. 1999, p↩
            .485 (19)
        boundary(i,1) = 0.5*(  promising(dim) + cur(dim↩
            ) +...
          ((distk^2 - distj^2)/(promising(dim) - cur(↩
              dim))));
    end

end %ends i loop

%This removes the zero element of boundary
boundary = boundary(find(boundary));

% With the newly calculated distances one can the find ↩
    the
% upper and lower boundaries of the Voronoi cell.
% See Sambridge 1999, page 485 (20, 21)

% First I divde the boundaries into the ones larger and
% smaller than the promissing point.
boundary_higher = bounds(dim,2)*ones(length(boundary)↩
    ,1);
boundary_lower  = bounds(dim,1)*ones(length(boundary)↩
    ,1);
for j=1:length(boundary)
    if     boundary(j) >= promising(dim)
               boundary_higher(j) = boundary(j);
    elseif boundary(j) <= promising(dim)
               boundary_lower(j)  = boundary(j);
```

```
                end
            end

            %lower = max(max(boundary_lower,bounds(dim,1)));
            %upper = min(min(boundary_higher,bounds(dim,2)));

            lower = max(boundary_lower);
            upper = min(boundary_higher);

            if lower > upper
                error('Lower limit is larger than upper limit!')
            end

            if lower < bounds(dim,1)
                lower = bounds(dim,1);
            end
            if upper > bounds(dim,2)
                upper = bounds(dim,2);
            end

            % Select a random number between upper and lower
            randstep = lower+(upper-lower)*rand;
            % and saves this into a temporary holder until this has↩
                 been
            % done for every dimension.
            tmpmodel(1,dim) = randstep;

            % This distance should always be zero
            %sqrt(distk^2+(promising(dim)-tmpmodel(dim))^2) - sqrt(↩
                distj^2+(cur(dim)-tmpmodel(dim))^2)

        end %ends dim loop

        % Save the new model in models.
        itmp = itmp+1;
        models(counter*N_s+itmp,1:length(m_start)) = tmpmodel(1,1:↩
            length(m_start));
    end %ends ins loop
  end %ends index loop
end %ends counter loop

%The misfit of the new points are calcualted
for countermisfit=(counter*N_s+1):(counter*N_s+N_s)
                models(countermisfit,length(m_start)+1) = ...
        rms_data_many(...
        models(countermisfit,1:length(m_zs)),...
        models(countermisfit,length(m_zs)+1:length(m_start)), ...
        data);
end
toc
```

B.5 Levenberg-Marquardt

```
% $Id: LevenbergMarquardt.m 137 2008-12-10 16:39:15Z tjansson $
clf; clear all; clc;
tegn=1; % Save the figures to harddisk
setuppath % Redefines the path include the lib folder
time = datestr(now, 'yyyy-mmdd-HHMM');
testing = 0;

%% Load the data
data           = load('R_NOR_THOMAS_CLEAN_1296');
```

```
data_name        = 'Station NORD';
data_name_file   = 'NORD';
data_length      = length(data);

% Define the data sets that you are trying to fit the function to.
%data_zs             = [20, 35];
%data_VP             = [5.8, 6.5, 8.04];
%data                = rf_forward(data_zs, data_VP);
%data_name           = 'Synthetic data';
%data_name_file      = 'synthetic';

%% Setup the parameter space
% The upper and lower bounds of the parameter space
% The starting model
zmin                = 1;
zmax                = 60;
layers_stepsize     = 1;
layers              = length(zmin:layers_stepsize:zmax);
modelyaxis          = 'Depth [km]';
m_zs                = zmin:layers_stepsize:zmax;
m_VP                = 4:(8-4)/layers:8;
m_start             = [m_zs m_VP];
startmodel          = m_VP;

%% Algorithm parameters
% Set an options file for LSQNONLIN to use the medium-scale algorithm
options = optimset( 'LevenbergMarquardt','on', ...
    'LargeScale', 'off', ...
    'Diagnostics', 'on', ...
    'MaxFunEvals', 20000, ...
    'MaxIter',1000, ...
    'TolX',1e-6, ...   %Termination tolerance on x. default: 1e-6
    'TolFun',1e-6, ... % Termination tolerance on the function value. ↩
        default: 1e-6
    'Display','iter');

%% Calculating cell
diary([outputpath, 'Levenberg-Marquardt-Vel-',time,'-',data_name_file,'↩
    -diary.txt'])

tic
% Calculate the new coefficients using LSQNONLIN.
[x,resnorm,residual,exitflag,output]=lsqnonlin( ...
    @misfit_data_many_levenberg_vel,...
    startmodel,...
    options,...
    data);
calctime = toc;

fprintf('\n\n Number of itt.: %3.0f, ', output.iterations)
fprintf('func. evals : %4.0f. ', output.funcCount )
fprintf('Time: %3.2f seconds\n', calctime )
final_misfit = rms_data(x,data);

diary off
```