An Algorithm for Full Waveform Inversion of Vector Acoustic Data

by

©Seyed Mostafa Akrami

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of science (Geophysics).

Department of Earth Sciences
Memorial University of Newfoundland

Memorial University of Newfoundland

May 2017

ST. JOHN'S NEWFOUNDLAND

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am truly grateful to my supervisor **Dr. Alison Malcolm** for my Masters program in Geophysics and my thesis. Her scientific vision, kindness and brilliance have been exemplary model for my life. As a student with condensed matter Physics and Engineering background, studying Geophysics was a journey for me and during this journey, the greatest help was from Alison. Alison taught me an interdisciplinary area of science which is a combination of Mathematics, Physics and Computer sciences and I really enjoyed this field. I wholeheartedly thank her for her help and guidance during my study and endeavour.

I also deeply appreciate **Dr. Charles Hurich** for being in my committee and his help and teaching some Seismology courses which opened my eyes to Earth sciences.

I wish to express my gratitude to **Dr. Colin Farquharson** for his help and inversion course during my study.

The other scientists who helped me and inspired me were **Dr. Laurent Demanet** (**MIT**), **Dr. Russell Hewett** (**Total S. A.**), **Dr. Felix Herrmann** (**SLIM group, UBC**). The especial thanks go to Laurent and Russell for **PySIT** package.

During my study, I met a lot of people and I found good friends. I wish to thank all of them for scientific and non-scientific stuffs, especially some of our group members: **Dr. Polina Zheglova** (**MUN**) and **Bram Willemsen** (**MIT**) and the other friends at **UBC**: **Felix Oghenekohwo**, **Ragiv Kumar** and **Ben Bougher**.

Last but not by any means least, I humbly thank my immediate family especially my parents for their support and love. Although they are physically thousands of miles away from me, but they are deeply in my heart. I am also truly blessed to have my beloved **Ghazal** in my life and words cannot describe how much I love, admire and thank her.

Abstract

In exploration seismology, constructing an accurate velocity model is imperative. One of the algorithms which can lead to an accurate velocity model is Full Waveform Inversion (FWI). FWI takes advantage of full wave information that is, direct, reflection and refraction waveforms and tries to construct the model parameters that best fit the data and obtain the best-fit images of the Earth's subsurface. Depending on the environment, these parameters could be compressional or shear wave velocities, density, Lame parameters, etc. Acoustic FWI uses only scalar data such as pressure to construct a velocity model and does not provide any directivity information about the wavefields. Mimicking the recent experiments in seismic acquisition, which allow for recording different types of data (scalar and vector data) in terms of FWI scheme is crucial for complex imaging problem. This is because, extending FWI to vector data allows us to use both pressure and velocity components at the same time, giving directivity information about the wavefields. By extending FWI to vector data and thus improving the input data to FWI, we obtain both improved resolution and directivity information. This can be done by employing monopole as well as dipole sources and regularized joint objective functions. I demonstrate my algorithm with four models.

Chapter 1

Introduction

1.1 Motivation and Challenges

In Full Waveform Inversion (FWI) the goal is to reconstruct the unknown model parameters, namely properties of the subsurface of the Earth, from the waveforms recorded at the surface of the Earth. In the simplest formulation, it is assumed that the wave propagation inside the Earth is governed by the acoustic constant density wave equation, and the recorded data are pressure waves. Recently, in marine acquisition, there has been interest in recording different types of data including velocity and acceleration. In Vector acoustic Full Waveform Inversion (VFWI) we record the multi-component pressure and velocity data using usual and point-force sources. Since point-force source components (e. g. velocity components) are proportional to the spatial gradient of pressure, wavefields in the VFWI scheme can take advantage of directivity information contained in point-force source components. In addition, the contribution of pressure and velocity components (vector data) rather than only pressure data (scalar data) can improve the images' resolution.

1.2 Literature Review

Generally in inverse problems, we consider the process of obtaining a model of some sort, for example a geophysical image of the earth, a medical image of the body, etc [2]. The process starts with some measuring device that measures the data. Depending on the measuring device and the underlying physics, the data can have many different forms. For example it can be projections of an object, electromagnetic or seismic waves that went through the object that are recorded at some point, and more. Usually, such data are noisy because any measurement process introduces some random or systematic noise. In the next step a code that implements some algorithm takes the data, processes it and transforms it into an image that represents the object under investigation. This algorithm can be composed of multiple parts, for example, it may include preprocessing of the data. After the initial image is obtained we may want to extract certain features from the image or perform comparison of images. Segmentation, interpolation and registration are often used to achieve these goals.

My specific inverse problem in this thesis is inversion of seismic waves. In seismic inversion several types of algorithms have been used, all of which require an estimate of the background wave velocity. The book by Yilmaz and Doherty [1], gives an overview of standard seismic data processing.

Wave velocity can be estimated using Normal Moveout (NMO) analysis as well as iterative prestack migration velocity analysis. Both of these methods suffer from some disadvantages. Normal moveout analysis may not be suitable for complicated media, particularly, when we are faced with strong lateral variations in the velocity [1].

A model of wave velocity can be estimated using Migration Velocity Analysis

(MVA) [47] and Wave Equation based Migration Velocity Analysis (WEMVA) [48] which extract additional information from the reflections by extending the migrated image or angle domains. Although the evolution of these reflection-based methods resulted in velocity models with increased resolution, they use only a subset of the recorded seismic data. Moreover, these methods may not be able to very accurately propagate waves in the presence of laterally heterogeneous media, so they may not be the best methods for complex areas. Biondi et al. [51] proposed a method which combines the MVA and FWI algorithms to overcome the mentioned challenges associated with WEMVA as well as the shortcoming of FWI in estimating low velocity changes. They have also integrated FWI and WEMVA into Tomographic Full Waveform Inversion (TFWI) workflow to gain a robust convergence to high-resolution models. Also, by extending the velocity model along the time-lag axis in TFWI, they achieved strong convergence properties when both reflected and refracted waves are present. In addition to that, they reduced the FWI sensitivity to the starting model [51].

Recently, there have been some advances in marine data acquisition. Multicomponent seismic data has become more interesting due to introduction of dual sensors and new marine seismic acquisition techniques [40–42]. Instead of recording only conventional seismic data (scalar data), one can record both scalar and vector data (pressure and particle velocity components) at the same time [40]. Several authors have proposed new advances in data-domain processing of vector data for the purposes of noise attenuation improvement [43], signal reconstruction [44], 3D deghosting and multiple attenuation [45, 46] and wavefield separation and ghost removal [38].

To better exploit these data, [31] propose a method for multicomponent Reverse-Time Migration (RTM) which is based on an adjoint-state formulation using the vector acoustic wave equations for pressure and the corresponding displacement field. This approach is more computationally expensive than other migration methods since there are different source types, but gives additional information about directionality allowing for directionally targeted imaging. Also, in the case of 4-component data only on the receiver side, the increase in cost comes in the form of additional memory requirements and additional terms in the imaging condition, which are negligible increases compared to the cost of extrapolation, which remains unchanged. In this thesis, we extend this algorithm to Full-Waveform Inversion (FWI).

The main difference between the Vector-Acoustic (VA) and acoustic data (we will formulate acoustic FWI in the second chapter) is that VA data contains the pressure and displacement or particle velocity at the same time whereas for acoustic data we record only pressure. In addition, by employing a dipole point-force source and dual receivers, which are necessary to generate and record VA data, we can suppress imaging artefacts arising from ambiguities in the direction of wavefield propagation [31].

Fleury and Vasconcelos [31] presented an adjoint state VA method in which multi-component seismic source and receiver data are used in a finite-frequency formulation of reverse time migration. However in our case, we present a VFWI algorithm in which 4-component seismic source and receiver data are used to obtain velocity models.

In FWI, the optimization process requires the minimization of the difference between modeled synthetic waveforms (u) and the observed data (d). Where

$$u = \mathcal{F}m,$$

and $\mathcal{F}$ is a forward modelling operator. u, m and d are in general continuous functions of space (in case of m) and space-time (in case of u, d), but they are discretized in numerical implementations of FWI. The recorded data $d \in \mathbb{R}^n$ is always finite dimensional and we assume that it belongs to a vector space $\mathcal{D}$. The model, m is some attribute of the object we want to image. Generally it is assumed to be a function on $\mathbb{R}^d$, $d = 1, ..., 4$ (space and possibly time), and also it belongs to a functional space $\mathfrak{M}$. Forward operator $\mathcal{F}$ can be linear or nonlinear function that maps elements in $\mathfrak{M}$ into $\mathcal{D}$. From these descriptions we can say that there is no unique model for given the data. The reason is that we try to recover a function from a discrete set of data, i. e. the data, d, is finite dimensional while the model is infinite dimensional. Thus unless more information is given, the problem does not have a unique solution. Moreover, even if we restrict m, the problem of recovering m from d is ill-posed, i.e., a very small perturbation in d can result in a very large perturbation in m. Nevertheless, we could compute a solution to the problem given some assumptions on the space $\mathfrak{M}$. If our assumptions are correct we may obtain an approximation to the solution.

Generally, the squared difference between u and d is called the misfit function $\mathcal{J}$ and depends on the model parameter (m),

$$\mathcal{J}(m) = 1/2\|u - d\|^2.$$

By this definition the misfit function is expressed in l^2 norm. However it can be calculated in l^1 or l^2 norm depending on the specific problem. For example, in compressive sensing problems [7], it is usually computed in l^1 norm whereas in FWI problems it is calculated in l^2 norm.

One of the pioneers in formulating a full wave inversion method was Tarantola [35], who realized that the model could be improved iteratively by back-propagating the data residuals and correlating the result with forward-propagated wave in the time domain. This is called adjoint-state method [20].

Full waveform inversion also can be done in the frequency domain by using an implicit frequency domain numerical forward modeling algorithm [10, 12, 19]. The whole workflow of FWI consists of four main steps [35]; (i) Obtaining the modeled data by solving the wave equations for all the sources, (ii) Calculation of misfit function, (iii) calculation of the gradient by back propagating the data residuals and cross-correlating the back-propagated wavefields with the modelled wavefields (adjoint state method) to obtain the model update. (iv) Regularization/conditioning. Figures 1.1 and 1.2 illustrate the inversion workflow and FWI process. Figure 1.1 shows the general inversion process in which on the left hand side, we have an unknown medium that we measure data from and on the right hand side, we have modeled data computed from the guessed model, that is the forward problem, and we use computed data and misfit from the real measured data on the left to update the guessed model to obtain a final model which when residuals are small, will resemble the true model.

Although FWI is a promising method to obtain subsurface parameters, it is also accompanied by challenges. The first challenge is to the computational cost of the algorithm. Computational operations in the frequency domain cost less than in the time domain as one can perform computational operations for a few frequencies and the convolution operation is replaced by multiplication [10, 11]. This is generally only true in 2D, since solving 3D wave-equations in the frequency-domain, for highly heterogeneous media, can require very sophisticated approaches to pre-conditioning very large, sparse linear systems that ultimately can compete in cost to 3D computations in the time-domain. When enough memory is available, high-efficiency computational

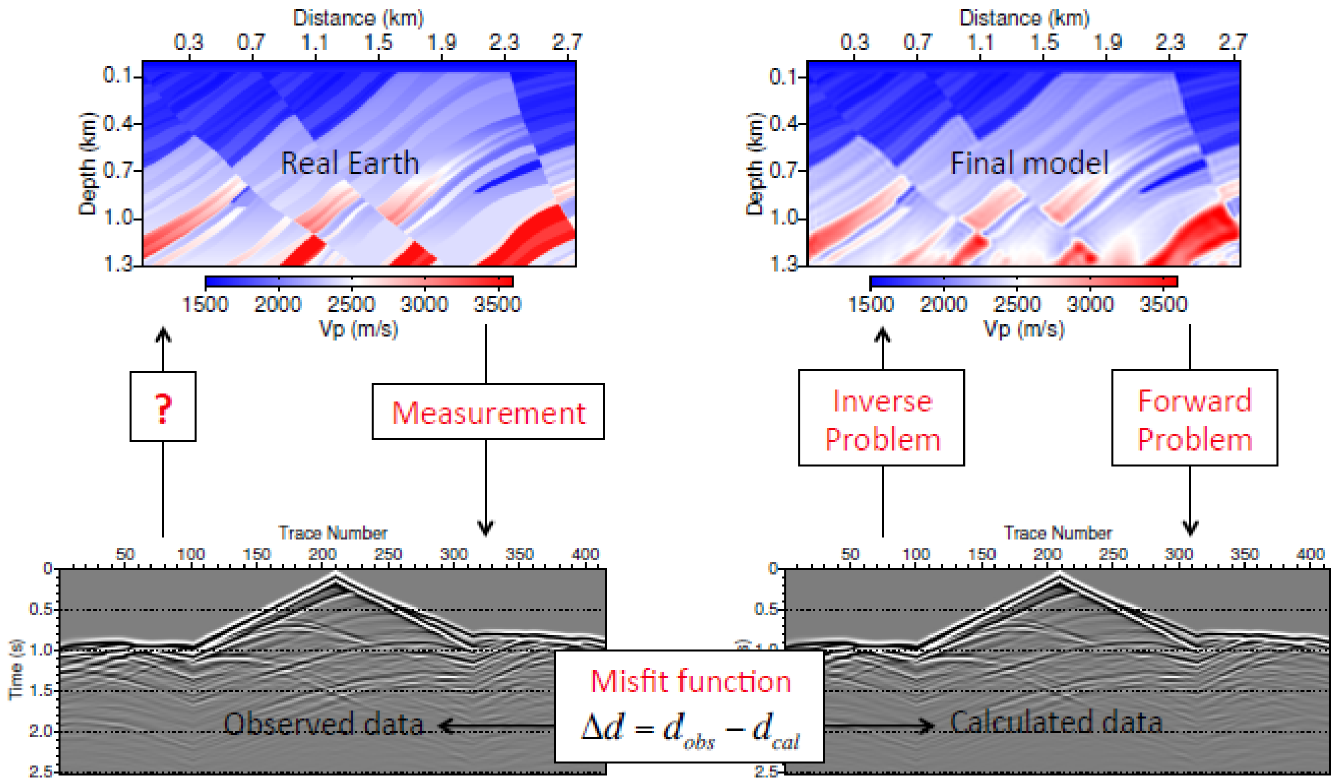


Figure 1.1: Inversion workflow: The whole diagram shows an inversion problem workflow. In this process one tries to minimize the misfit ($\|d_{obs} - d_{cal}\|^2$). The left hand side of diagram shows the measurement of the observed data whereas the right hand side explains obtaining synthetic data by applying forward modeling to the velocity model. The velocity model estimation is achieved through an inversion process. The question mark on the left hand side represents the inverse problem we want to solve.

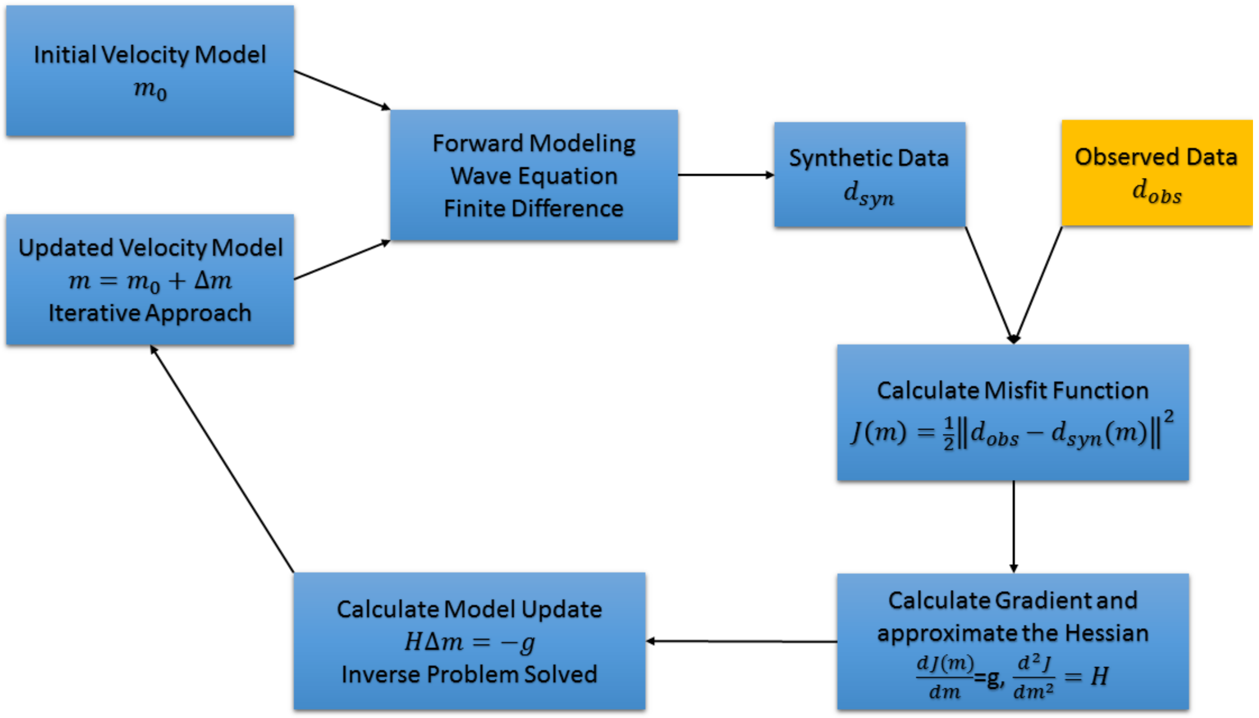

Initial Velocity Model m_0
Forward Modeling Wave Equation Finite Difference
Synthetic Data d_{syn}
Observed Data d_{obs}
Updated Velocity Model $m = m_0 + \Delta m$ Iterative Approach
Calculate Misfit Function $J(m) = \frac{1}{2}\|d_{obs} - d_{syn}(m)\|^2$
Calculate Model Update $H\Delta m = -g$ Inverse Problem Solved
Calculate Gradient and approximate the Hessian $\frac{dJ(m)}{dm}$=g, $\frac{d^2J}{dm^2} = H$

Figure 1.2: Standard FWI algorithm process: The whole diagram explains the iterative process of standard FWI algorithm. We start with initial model (m_0) and by using forward modeling operator solve the wave equation and generate synthetic data. The next step is computing the difference between observed and synthetic data, calculate the gradient and estimate the Hessian and finally obtaining the updated velocity. By repeating this process several times one can achieve the estimated updated model.

methods such as direct solvers [24, 26] or iterative solvers [25] can be implemented for this problem. However, when the problem size becomes too large such as in elastic FWI [21] and 3D FWI [22], these methods fail.

Another issue associated with FWI comes from the non-linearity of the inverse problem. The existence of local minima is simply a result of the physics of the problem combined with the choice of the objective function. Cycle-skipping is only a problem because we use gradient-based, iterative methods and if we were able to use global, statistical sampling inverse methods (which we cannot afford), local minima and cycle-skipping would not be a problem. Cycle-skipping is illustrated in Figure1.3. The solid black line is a monochromatic seismogram of period T plotted as a function of time which represents the recorded data and the upper dashed line represents the modeled monochromatic seismograms with a time delay greater than $T/2$. When the time delay between the two waves is more that $T/2$, FWI tries to update the model in such a way that the $n+1st$ cycle of the modeled seismogram matches the nth cycle of the observed seismogram, which leads to an erroneous model update. This cycle skipping problem is mitigated by following a multiscale approach. Bunks et al. [18] suggested successive inversion of data sets of increasing frequency content in the time domain, since low frequencies are less sensitive to cycle-skipping. The long wavelength parts of the data are fit first, which gives a starting model for higher frequency data that is closer than $T/2$ to the true model, allowing for successive improvements to the velocity model [13].

There are several approaches toward FWI which combine it with migration velocity analysis and they can circumvent conventional cycle-skipping by providing a better initial model for FWI [15]. An appropriate initial model provides matching of

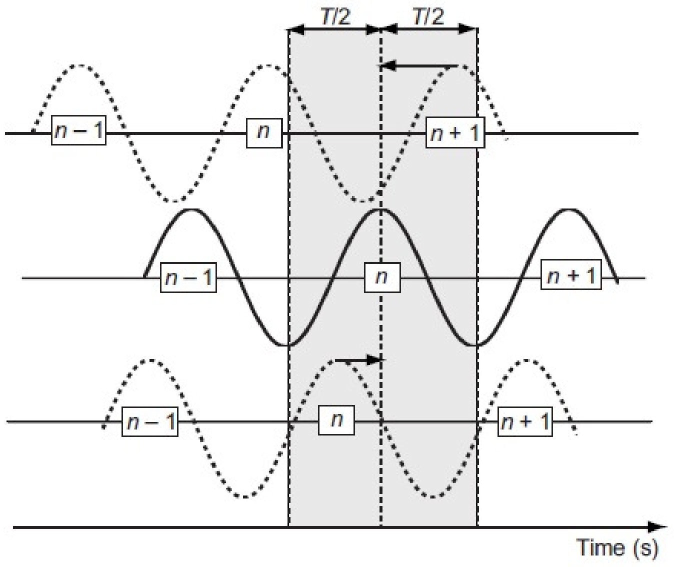


Figure 1.3: This diagram illustrates cycle-skipping phenomenon in FWI. The solid black line is a recorded seismogram trace of period T as a function of time. The upper dashed line indicates the modeled seismogram trace which has a time delay larger than $T/2$. The FWI algorithm tries to update the model in such a way that the $n + 1st$ cycle of the modeled seismogram trace matches the nth cycle of the observed data. In the bottom there is another model in which the modeled and recorded nth cycle have time delay less than $T/2$. In this case, FWI is able to correctly update the model. (This figure is taken from [16])

the observed seismogram with an error less than half of the period. Otherwise, cycle skipping issue will lead to convergence toward a local minimum as discussed above.

In spite of the fact that the standard FWI algorithm takes advantage of the large amount of scalar data contained in the seismic traces, it fails to extract explicit directivity information from the wavefields. In this study, we present an extension of the algorithm proposed by [31] to FWI of vector acoustic data. Thus we are able to gain more complete information from the wavefields, namely, directionality and therefore, better lateral resolution in estimating velocities. In this approach, we use dipole sources as well as monopole sources in different orientations and our wave solver for a complete acoustic wave equation to generate vector acoustic data. We then derive and test an FWI algorithm with synthetic data.

It is worth mentioning that due to free surface in the FWI model the gradient of the misfit function is strongly affected by ghost arrivals in the data [38]. To mitigate this, in this study the model is padded with a PML on all sides, eliminating free surface multiples in the true and the synthetic data so that ghost contamination phenomena are removed.

1.3 RTM versus FWI

In this section we briefly review the basic ingredients of RTM and FWI and then we compare them to each other.

Reverse-time migration is established as a famous technique in seismic imaging due to its capability to tackle large dips of reflectors and strong velocity contrasts [8]. In time domain, the image is formed by cross-correlating the source and receiver wavefields, at zero time shift and summing over all time steps and shots. Unlike FWI, RTM does not do inversion for the full model m. However, it separates the scales as

$$m \approx m_0 + \delta m.$$

where m_0 is the smooth background velocity model that is assumed known and kinematically correct, and delta m is the model perturbation, containing reflectors. Thus, in RTM the forward modelling operator is linearized as we now describe. Starting from forward modelling operator ($\mathcal{F}$), we can use a Taylor expansion to expand the operator with respect to the background model m_0

$$\mathcal{F}(m) = \mathcal{F}(m_0) + \frac{\partial \mathcal{F}(m_0)}{\partial m}\delta m + \mathcal{O}(\delta m^2).$$

The left hand side of this equation is the observed field data and the first term on the right hand side represents modelled data in the background model. We can rewrite this equation as

$$\mathcal{F}(m) - \mathcal{F}(m_0) \approx \frac{\partial \mathcal{F}(m_0)}{\partial m}\delta m.$$

The left hand side is now the data residual (δd) and $\frac{\partial \mathcal{F}(m_0)}{\partial m}$ is called the Jacobian ($\mathbf{J}$). So it leads to

$$\delta d \approx \mathbf{J}\delta m. \tag{1.1}$$

RTM finds image by applying the adjoint Jacobian operator on date residual, i. e.

$$\delta m \approx \mathbf{J}^{\top}\delta d. \tag{1.2}$$

Least-Squares Reverse-Time Migration (LSRTM) is another migration algorithm in which instead of using the adjoint of the Jacobian, equation (1.1) is solved in the least squares sense.

$$\underset{\delta m}{\text{minimize}} \quad \|\hat{\mathbf{J}}(m_0)\,\delta m - \delta d\|^2, \tag{1.3}$$

where m_0 is fixed and does not change during the iteration, delta m is optimized in this problem. $\hat{\mathbf{J}}$ is linearized modelling operator.

In FWI for a single source q, time-domain inversion using the adjoint-state method (that we will describe in more detail in the next chapter) [20] solves the following PDE-constrained optimization problem

$$\begin{aligned} &\underset{m,u}{\text{minimize}}\ \frac{1}{2}\|u - d\|^2 \\ &\text{subject to } \mathbf{A}(m)u = q. \end{aligned} \tag{1.4}$$

Here $\mathbf{A}$ denotes the Helmholtz operator. The difference between $\mathbf{A}$ (defined here) and the forward modelling operator $\mathcal{F}$ (defined earlier) is that the Helmholtz operator ($\mathbf{A}$) takes the field u and maps it into the source q, whereas the forward modelling operator ($\mathcal{F}$) takes the model m and maps it into data. The adjoint-sate method solves the above problem by eliminating the PDE constraint, so we can rewrite the FWI least squares objective function $\mathcal{J}(m)$ as

$$\underset{m}{\text{minimize}}\, \mathcal{J}(m) = \frac{1}{2}\|\mathbf{A}^{-1}(m)q - d\|^2. \tag{1.5}$$

The gradient of the above objective function is given by the action of the adjoint of the Jacobian on the data residual $\delta d = (u - d)$,

$$\nabla \mathcal{J}(m) = -\sum_{t=1}^{n_t} \left\{ (\mathbf{D}u)^\top \operatorname{diag}(v) \right\} = \mathbf{J}^\top \delta d, \tag{1.6}$$

where D is a second time derivative operator and *diag* is a diagonal operator [39]. u is the forward wavefield computed forward in time via

$$\mathbf{A}(m)u = q, \tag{1.7}$$

and v is the adjoint wavefield computed backwards in time via [20]

$$\mathbf{A}^*(m)v = \delta d. \tag{1.8}$$

By comparing RTM and LSRTM with FWI we can conclude that RTM is a linear inverse problem, it gets a structural image and deals with the high frequency contribution to the model (δm). On the contrary, FWI is a non-linear problem, it solves for the full model and tries to update both low and high frequencies in the model m. For an extensive overview of of FWI, one can refer to [3].

1.4 Thesis Outline

Here we briefly describe the structure of the thesis.

In **Chapter 1**, I describe the problem which I want to solve (i. e . extending FWI to vector data) by introducing vector-acoustic data, the method (VFWI) and the limitations and challenges toward estimating velocity. In **Chapter 2**, I reformulate FWI and then explain the methodology of VFWI. In **Chapter 3**, I present results from four numerical examples of VFWI to demonstrate our algorithm. A discussion and conclusions are included in the last chapter (**Chapter 4**).

Chapter 2

Methodology

In this chapter we formulate the full waveform inversion problem using vector acoustic data and monopole pressure and dipole point force sources. In our method we depart from the conventional formulation of FWI in two ways:

- The data to be used in the inversion method consists of 3 or 4 components in 2D and 3D respectively, namely, acoustic pressure and particle velocity, in contrast to the conventional FWI, in which only pressure data is used.
- We use two types of sources: monopole pressure source and dipole point force sources.

For simplicity, we develop our method in two spatial dimensions, but generalization to the three-dimensional case is straightforward. Also, to simplify the methodology and implementation, we consider the constant density acoustic case. In this thesis we use similar notations as Fleury and Vasconcelos [31] in terms of writing the resulting equations, however they differ in that we do not apply the receiver weighting in our implementation. Also, we go far beyond their derivations to derive a full FWI algorithm whereas they went only so far as RTM. Aside from being different scheme, our VFWI algorithm differs from Vector Acoustic Reverse Time Migration (VARTM) proposed by Fleury and Vasconcelos [31] in both methodology and results as well as recovered model parameter. To this end, we start with our problem description and forward model.

2.1 Problem description and forward model

Our goal is to reconstruct the model parameter m:

$$m = \rho\kappa = \frac{1}{c^2},$$

where ρ is density, κ is compressibility and c is the pressure wave velocity of the rocks.

Generally, acoustic wave propagation is governed by the following system of linear differential equations [33]

$$\begin{aligned} &p_{q,\mathbf{f}}(t,z,x) + \frac{1}{\kappa(z,x)}\nabla\cdot\mathbf{v}_{q,\mathbf{f}}(t,z,x) = q(t,z,x), \\ &\rho(z,x)\frac{\partial^2}{\partial t^2}\mathbf{v}_{q,\mathbf{f}}(t,z,x) + \nabla p_{q,\mathbf{f}}(t,z,x) = \mathbf{f}(z,x,t), \end{aligned} \tag{2.1}$$

subject to the initial conditions:

$$p_{q,\mathbf{f}}(t,z,x) = 0, \quad \mathbf{v}_{q,\mathbf{f}}(t,z,x) = 0 \quad \text{for} \quad t < 0. \tag{2.2}$$

Here

- $p_{q,\mathbf{f}}(t,z,x)$ is pressure;
- $\mathbf{v}_{q,\mathbf{f}}(t,z,x)$ is particle displacement, consisting of two components: $\mathbf{v}^z_{q,\mathbf{f}}(t,z,x)$ and $\mathbf{v}^x_{q,\mathbf{f}}(t,z,x)$;

- q and $\mathbf{f}$ are respectively monopole pressure and dipole point-force sources, where $\mathbf{f}$ has two components;
- subscripts q and $\mathbf{f}$ for the fields denote that the fields are generated by pressure and point-force sources respectively, i.e. p_q is the pressure field generated by the pressure source, $p_{\mathbf{f}}$ is the pressure field generated by the point-force source, and analogously for $\mathbf{v}_q$ and $\mathbf{v}_f$.
- (z, x) denote spatial position and t denotes time.

Assuming that density is constant, we can eliminate it from equation (2.3) in the following way. First, we rewrite (2.3) as:

$$\begin{aligned} &\frac{1}{c^2(z,x)} p_{q,\mathbf{f}}(t,z,x) + \nabla \cdot (\rho\, \mathbf{v}_{q,\mathbf{f}}(t,z,x)) = \frac{1}{c^2(z,x)} q(t,z,x), \\ &\frac{\partial^2}{\partial t^2}(\rho\, \mathbf{v}_{q,\mathbf{f}}(t,z,x)) + \nabla p_{q,\mathbf{f}}(t,z,x) = \mathbf{f}(z,x,t), \end{aligned} \tag{2.3}$$

and then replace in the above equation:

$$\begin{aligned} &\rho\, \mathbf{v}_{q,\mathbf{f}}(t,z,x) \longmapsto \mathbf{v}_{q,\mathbf{f}}(t,z,x), \\ &\frac{1}{c^2(z,x)} q(t,z,x) \longmapsto q(t,z,x), \end{aligned} \tag{2.4}$$

so that $\mathbf{v}_{q,\mathbf{f}}(t,z,x)$ and $q(t,z,x)$ are now scaled displacement and scaled monopole source. For brevity we will in what follows drop the word "scaled" and call these quantities simply "displacement" and "monopole source". Then we obtain the following system in the new variables:

$$\begin{aligned} &m(z,x)\, p_{q,\mathbf{f}}(t,z,x) + \nabla \cdot \mathbf{v}_{q,\mathbf{f}}(t,z,x) = q(t,z,x), \\ &\frac{\partial^2}{\partial t^2} \mathbf{v}_{q,f}(t,z,x) + \nabla p_{q,f}(t,z,x) = \mathbf{f}(z,x,t). \end{aligned} \tag{2.5}$$

We use equations (2.5) as our vector-acoustic forward problem. These equations have

Perfectly Matched Layer (PML) absorbing boundary conditions [6] on all sides of the computational domain to mimic an infinite medium. In matrix form, the set of equations (2.5) becomes

$$\mathbf{L}^{VA}(m)\mathbf{u}_{q,\mathbf{f}} = \mathbf{s}, \tag{2.6}$$

where

$$\mathbf{L}^{VA}(m) = \begin{pmatrix} m & \nabla\cdot \\ \nabla & \frac{\partial^2}{\partial t^2}\mathbf{I} \end{pmatrix}, \quad \mathbf{u}_{q,f} = \begin{pmatrix} p_{q,\mathbf{f}} \\ \mathbf{v}_{q,\mathbf{f}} \end{pmatrix}, \quad \mathbf{s} = \begin{pmatrix} q \\ \mathbf{f} \end{pmatrix}. \tag{2.7}$$

Following Eq. 2.7 we can define adjoint source as

$$\mathbf{s}^{\dagger} = \begin{pmatrix} q^{\dagger} \\ \mathbf{f}^{\dagger} \end{pmatrix}. \tag{2.8}$$

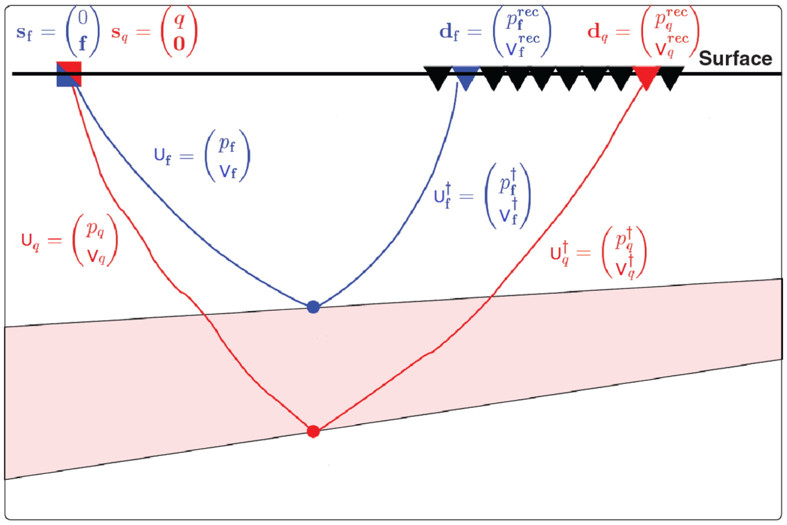
Figure 2.1 shows a configuration of source and receiver positions. It represents the wave paths for pressure sources in red and for point-force sources in blue. Also, the generated wavefields, their adjoints and recorded data by receivers are shown in this figure.

2.2 Objective function

Following Fleury and Vasconcelos, the general form of the objective function for our problem is as follows:

$$\begin{aligned} \mathcal{J}(m) = &\frac{1}{2}w_s^q \sum_{s,r} \int_0^T \|\mathbf{W}_r[\mathbf{u}_q(\mathbf{x_s},\mathbf{x_r},t) - \mathbf{d}_q(\mathbf{x_s},\mathbf{x_r},t)]\|_2^2\, dt + \\ &+ \frac{1}{2}w_s^f \sum_{s,r} \int_0^T \|\mathbf{W}_r[\mathbf{u}_f(\mathbf{x_s},\mathbf{x_r},t) - \mathbf{d}_f(\mathbf{x_s},\mathbf{x_r},t)]\|_2^2\, dt, \end{aligned} \tag{2.9}$$

where



$\mathbf{s}_f = \begin{pmatrix} 0 \\ \mathbf{f} \end{pmatrix}$
$\mathbf{s}_q = \begin{pmatrix} q \\ \mathbf{0} \end{pmatrix}$
$\mathbf{d}_f = \begin{pmatrix} p_f^{rec} \\ \mathsf{v}_f^{rec} \end{pmatrix}$
$\mathbf{d}_q = \begin{pmatrix} p_q^{rec} \\ \mathsf{v}_q^{rec} \end{pmatrix}$
Surface
$\mathsf{u}_f = \begin{pmatrix} p_f \\ \mathsf{v}_f \end{pmatrix}$
$\mathsf{u}_f^{\dagger} = \begin{pmatrix} p_f^{\dagger} \\ \mathsf{v}_f^{\dagger} \end{pmatrix}$
$\mathsf{u}_q = \begin{pmatrix} p_q \\ \mathsf{v}_q \end{pmatrix}$
$\mathsf{u}_q^{\dagger} = \begin{pmatrix} p_q^{\dagger} \\ \mathsf{v}_q^{\dagger} \end{pmatrix}$

Figure 2.1: This diagram represent sources and receivers' positions. Symbols and ray paths in red indicate fields associated to physical pressure sources s_q, whereas blue symbols and lines are associated with the point-force sources s_f . u_q and u_f are the wavefields generated by point and dipole sources respectively. Triangles represent receivers at the surface which record the observed data from either pressure d_q or point-force-dipole sources d_f. Adjoint wavefields $u_q^{\dagger}$ and $u_f^{\dagger}$ are shown by the right side red and blue paths, respectively. Having the source wavefields and the receiver wavefields from either source type which are composed of scalar pressure and vector displacement data in the subsurface we can construct the image.(Modified after [31]).

- $\mathbf{d}_{q,\mathbf{f}} = [p_{q,\mathbf{f}}^{meas} \quad \mathbf{v}_{q,\mathbf{f}}^{meas}]^T$ are measured data;
- $\mathbf{x_s} = (z_s, x_s)$ and $\mathbf{x_r} = (z_r, x_r)$ are source and receiver coordinates;
- w_s^q and w_s^f are source weights necessary to balance the contributions of the different source types in the objective;
- $\mathbf{W}_r$ is a 3×3 receiver weight matrix that weights the contributions of different data components in the objective.

In general, the source weights are determined in such a way that different sources produce waves that carry comparable energy. The receiver weighting matrix is determined in such a way that the contributions from different data components are comparable and have the same physical dimensions. However, in order to simplify our objective function and the subsequent derivation of the adjoint state gradient, we introduce the following modifications. In what follows we set the $w_s^q = w_s^f = 1$ and weigh the sources directly in the forward modelling equations. In the numerical examples we use only one source type per experiment, so that source weighting becomes less important. Also, we set the matrix $\mathbf{W}_r$ to identity. Therefore, the objective functions used in this thesis is as follows:

$$\mathcal{J}(m) = \frac{1}{2}\sum_{s,r}\int_0^T \left[\|\mathbf{u}_q(\mathbf{x_s},\mathbf{x_r},t) - \mathbf{d}_q(\mathbf{x_s},\mathbf{x_r},t)\|_2^2 + \|\mathbf{u}_f(\mathbf{x_s},\mathbf{x_r},t) - \mathbf{d}_f(\mathbf{x_s},\mathbf{x_r},t)\|_2^2\right] dt. \tag{2.10}$$

This function is to be minimized over the model space.

2.3 Adjoint problem and gradient: theoretical framework

Following Fichtner's notations [52], in order to derive the gradient of the objective function $\mathcal{J}(m)$ in (2.10), we rewrite it as follows:

$$\begin{aligned}\mathcal{J}(m) = \frac{1}{2}\sum_{s}\int_0^T \int_G &\left[\|\mathbf{u}_q(\mathbf{x_s},\mathbf{x},t) - \mathbf{d}_q(\mathbf{x_s},\mathbf{x},t)\|_2^2 + \right.\\ &\left. + \|\mathbf{u}_f(\mathbf{x_s},\mathbf{x},t) - \mathbf{d}_f(\mathbf{x_s},\mathbf{x},t)\|_2^2\right] \delta(\mathbf{x}-\mathbf{x}_r) d\mathbf{x}\, dt.\end{aligned} \tag{2.11}$$

Then, it can be written as

$$\mathcal{J}(m) = \int_0^T \int_G J_1(m)\, d\mathbf{x}\, dt = \langle J_1(m)\rangle\,. \tag{2.12}$$

where

$$\mathcal{J}_1(m) = \frac{1}{2} \sum_s \left[\|\mathbf{u}_q(\mathbf{x_s}, \mathbf{x}, t) - \mathbf{d}_q(\mathbf{x_s}, \mathbf{x}, t)\|_2^2 + \|\mathbf{u}_f(\mathbf{x_s}, \mathbf{x}, t) - \mathbf{d}_f(\mathbf{x_s}, \mathbf{x}, t)\|_2^2 \right] \delta(\mathbf{x} - \mathbf{x}_r), \tag{2.13}$$

and

$$\langle\, f(x,t),\ g(x,t)\, \rangle = \int_0^T \int_G f(x,t)\ g(x,t)\ d\mathbf{x}\ dt. \tag{2.14}$$

Then we can show that

$$\nabla_m \mathcal{J}(m) \delta m = \langle \nabla_{\mathbf{u_{q,f}}} \mathcal{J}_1(m), \delta \mathbf{u_{q,f}} \rangle. \tag{2.15}$$

We prove equation (2.15) in appendix A.1.

We then differentiate equation (2.6) with respect to m, using the chain rule and keeping in mind that the source $\mathbf{s}$ does not depend on m:

$$\nabla_m \mathbf{L}^{VA}\ \delta m + \nabla_{\mathbf{u}_{q,\mathbf{f}}} \mathbf{L}^{VA}\ \nabla_m \mathbf{u}_{q,\mathbf{f}}\ \delta m = 0.$$

Using relationship (A.1) we obtain:

$$\nabla_m \mathbf{L}^{VA}\ \delta m + \nabla_{\mathbf{u}_{q,\mathbf{f}}} \mathbf{L}^{VA}\ \delta \mathbf{u}_{q,\mathbf{f}} = 0. \tag{2.16}$$

Now we introduce the adjoint fields

$$\mathbf{u}_{q,\mathbf{f}}^{\dagger} = \begin{pmatrix} p_{q,\mathbf{f}}^{\dagger} \\ \mathbf{v}_{q,\mathbf{f}}^{\dagger} \end{pmatrix}. \tag{2.17}$$

By taking the dot product of the adjoint field $\mathbf{u}_{q,\mathbf{f}}^{\dagger}$ with equation (2.16) and integrating over space and time we obtain:

$$\left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \nabla_m \mathbf{L}^{VA}\ \delta m \right\rangle + \left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \nabla_{\mathbf{u}_{q,\mathbf{f}}} \mathbf{L}^{VA}\ \delta \mathbf{u}_{q,\mathbf{f}} \right\rangle = 0. \tag{2.18}$$

Now we add together equations (2.15) and (2.18), we obtain:

$$\nabla_m \mathcal{J}(m)\delta m = \left\langle \nabla_{\mathbf{u_{q,f}}} \mathcal{J}_1(m), \delta \mathbf{u_{q,f}} \right\rangle + \left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \nabla_m \mathbf{L}^{VA}\ \delta m \right\rangle + \left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \mathbf{L}^{VA}\ \delta \mathbf{u}_{q,\mathbf{f}} \right\rangle. \tag{2.19}$$

The goal of the adjoint state method is to eliminate $\delta \mathbf{u}_{q,\mathbf{f}}$ from equation (2.19) in order to avoid the calculation of Jacobian $\nabla_m \mathbf{u}_{q,\mathbf{f}}$. It means that by using adjoint state method we find the gradient of our objective function only by taking adjoints of state variables which results in less computational cost [20]. To this end, we first note that using the definition of adjoint the third term on the right hand side of (2.19) can be rewritten as:

$$\left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \mathbf{L}^{VA}\ \delta \mathbf{u}_{q,\mathbf{f}} \right\rangle = \left\langle \mathbf{L}^{VA\dagger}\ \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \delta \mathbf{u}_{q,\mathbf{f}} \right\rangle, \tag{2.20}$$

where $\mathbf{L}^{VA\dagger}$ is the adjoint of $\mathbf{L}^{VA}$. Equation (2.19) now takes the form:

$$\nabla_m \mathcal{J}(m)\delta m = \left\langle \nabla_{\mathbf{u_{q,f}}} \mathcal{J}_1(m), \delta \mathbf{u_{q,f}} \right\rangle + \left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \nabla_m \mathbf{L}^{VA}\ \delta m \right\rangle + \left\langle \mathbf{L}^{VA\dagger}\ \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \delta \mathbf{u}_{q,\mathbf{f}} \right\rangle. \tag{2.21}$$

The first and the third terms on the right hand side of (2.21) add up to zero if we can now find the adjoint field $\mathbf{u}^{\dagger}_{q,\mathbf{f}}$ that satisfies the following adjoint equation

$$\mathbf{L}^{VA\dagger}\ \mathbf{u}^{\dagger}_{q,\mathbf{f}} = -\nabla_{\mathbf{u_{q,f}}} \mathcal{J}_1(m). \tag{2.22}$$

The right hand side of equation (2.22) is called adjoint sources. We then can simplify it using equation (2.13) gives

$$\mathbf{s}^{\dagger}_{q,\mathbf{f}}(\mathbf{x}, \mathbf{x_s}, t) = \sum_r [\mathbf{u}_{q,\mathbf{f}}(\mathbf{x}_r, \mathbf{x}_s, T-t; m) - \mathbf{d}_{q,\mathbf{f}}(\mathbf{x}_r, \mathbf{x}_s, T-t)]\delta(\mathbf{x} - \mathbf{x}_r), \tag{2.23}$$

we then can rewrite equation (2.22) as

$$\mathbf{L}^{VA\dagger}\ \mathbf{u}^{\dagger}_{q,\mathbf{f}} = -\mathbf{s}^{\dagger}_{q,\mathbf{f}}(\mathbf{x}, \mathbf{x_s}, t). \tag{2.24}$$

Therefore equation (2.21) can be written as:

$$\nabla_m \mathcal{J}(m)\delta m = \left\langle \mathbf{u}^{\dagger}_{q,\mathbf{f}}, \nabla_m \mathbf{L}^{VA}\ \delta m \right\rangle = \int_0^T \int_G \mathbf{u}^{\dagger}_{q,\mathbf{f}} \cdot \nabla_m \mathbf{L}^{VA}\ \delta m\ d\mathbf{x}\ dt, \tag{2.25}$$

so that the gradient can be computed as follows:

$$\nabla_m \mathcal{J}(m) = \int_0^T \mathbf{u}^{\dagger}_{q,\mathbf{f}} \cdot \nabla_m \mathbf{L}^{VA}\ dt. \tag{2.26}$$

In order to compute the gradient equation (2.26) we need to differentiate the $\mathbf{L}^{VA}(\mathbf{u}_{q,\mathbf{f}})$ from equation (2.7) with respect to m, which gives

$$\nabla_m \mathbf{L}^{VA}(\mathbf{u}_{q,\mathbf{f}}) = (p_{q,\mathbf{f}}, 0)^T,$$

then equation (2.26) becomes

$$\nabla_m J(m) = \sum_s \int_T p^{\dagger}_{q,\mathbf{f}}(\mathbf{x}, \mathbf{x}_s, t) p_{q,\mathbf{f}}(\mathbf{x}, \mathbf{x}_s, t)\ dt. \tag{2.27}$$

Where in equation (2.27) $\mathbf{u}^{\dagger}_{q,\mathbf{f}}$ is $p^{\dagger}_{q,\mathbf{f}}$ since $\mathbf{v}^{\dagger}_{q,\mathbf{f}}$ vanishes as a result of inner product operation between the integrand in equation (2.26).

In order to complete derivations for our problem, we need to derive the adjoint operator $\mathbf{L}^{VA\dagger}$ based on our forward operator $\mathbf{L}^{VA}$ and our objective function. We do this in the following section.

2.3.1 Adjoint problem

In this section we derive the adjoint operator $\mathbf{L}^{VA\dagger}$.

For simplicity we drop the VA superscript from operator $\mathbf{L}$ i.e.,

$$\mathbf{L}^{VA} = \mathbf{L}.$$

By invoking the definition of an adjoint in the form of an inner product [4] we have

$$\langle \mathbf{L}\ \delta \mathbf{u}_{q,\mathbf{f}}, \mathbf{u}^{\dagger}_{q,\mathbf{f}} \rangle = \langle \delta \mathbf{u}_{q,\mathbf{f}}, \mathbf{L}^{\dagger}\ \mathbf{u}^{\dagger}_{q,\mathbf{f}} \rangle, \tag{2.28}$$

expanding the inner product of left hand side of equation (2.28) we get

$$\begin{aligned} \langle \mathbf{L}\ \delta \mathbf{u}_{q,\mathbf{f}}, \mathbf{u}^{\dagger}_{q,\mathbf{f}} \rangle = & \underbrace{\int_0^T \int_G (m\, \delta p_{q,\mathbf{f}} + \nabla \cdot \delta \mathbf{v}_{q,\mathbf{f}}) p^{\dagger}_{q,\mathbf{f}}\ dxdt}_{\text{A}} + \\ & + \underbrace{\int_0^T \int_G (\nabla \delta p_{q,\mathbf{f}} + \frac{\partial^2}{\partial t^2} \delta \mathbf{v}_{q,\mathbf{f}}) \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}}\ dxdt}_{\text{B}}. \end{aligned} \tag{2.29}$$

The second term of A is

$$A_1 = \int_0^T \int_G \nabla \cdot \delta \mathbf{v}_{q,\mathbf{f}}\, p^{\dagger}_{q,\mathbf{f}}\ dxdt = \int_0^T \int_G (\nabla \cdot \delta \mathbf{v}_{q,\mathbf{f}})\ (p^{\dagger}_{q,\mathbf{f}})\ dxdt, \tag{2.30}$$

If we integrate the equation (2.30) using Green's identities [5], after simplification it gives

$$A_1 = -\int_0^T \int_G \delta \mathbf{v}_{q,\mathbf{f}} \cdot \nabla (p^{\dagger}_{q,\mathbf{f}})\, dxdt\ +\ \oint_{\partial t} \partial_n (\delta \mathbf{v}_{q,\mathbf{f}}) \cdot p^{\dagger}_{q,\mathbf{f}}\, ds. \tag{2.31}$$

Since we use PML absorbing boundary conditions, that implies $p^{\dagger}_{q,\mathbf{f}}$ in the second integral of equation (2.31) vanishes as pressure components and their adjoints are zero at the boundary. Finally we get

$$A = \int_0^T \int_G (m\,\delta p_{q,\mathbf{f}}\; p^{\dagger}_{q,\mathbf{f}} - \delta\mathbf{v}_{q,\mathbf{f}} \cdot \nabla(p^{\dagger}_{q,\mathbf{f}}))\;\; dxdt. \tag{2.32}$$

As for B by having these relations between $\nabla\cdot$ and ∇, namely, that the adjoint of divergence is negative gradient [34], we obtain

$$\begin{cases} \nabla\delta p_{q,\mathbf{f}} \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}} \;\longmapsto -\delta p_{q,\mathbf{f}}\;(\nabla \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}}) \\ \frac{\partial^2}{\partial t^2}\delta\mathbf{v}_{q,\mathbf{f}} \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}} \;\longmapsto \delta\mathbf{v}_{q,\mathbf{f}} \cdot \frac{\partial^2}{\partial t^2}\mathbf{v}^{\dagger}_{q,\mathbf{f}} \end{cases}, \tag{2.33}$$

and again boundary conditions yield

$$\begin{cases} \mathbf{v}^{\dagger}_{q,\mathbf{f}}(T) = 0, \\ \delta\mathbf{v}_{q,\mathbf{f}}(0) = 0. \end{cases} \tag{2.34}$$

The second equation in the set (2.33) can be obtained by taking integral by parts as follows:

$$\begin{aligned} \int_0^T \frac{\partial^2}{\partial t^2}\,\delta\mathbf{v}_{q,\mathbf{f}} \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}}\,dt &= \frac{\partial}{\partial t}\,\delta\mathbf{v}_{q,\mathbf{f}} \cdot \mathbf{v}^{\dagger}_{q,\mathbf{f}}\Big|_0^T - \int_0^T \frac{\partial}{\partial t}\,\delta\mathbf{v}_{q,\mathbf{f}}\,\frac{\partial}{\partial t}\,\mathbf{v}^{\dagger}_{q,\mathbf{f}} = \\ &= \underbrace{\frac{\partial}{\partial t}\,\delta\mathbf{v}_{q,\mathbf{f}}\,\mathbf{v}^{\dagger}_{q,\mathbf{f}}\Big|_0^T}_{\alpha} - \underbrace{\delta\mathbf{v}_{q,\mathbf{f}}\,\frac{\partial}{\partial t}\,\mathbf{v}^{\dagger}_{q,\mathbf{f}}\Big|_0^T}_{\beta} + \int_0^T \delta\mathbf{v}_{q,\mathbf{f}}\,\frac{\partial^2}{\partial t^2}\,\mathbf{v}^{\dagger}_{q,\mathbf{f}}\,dt. \end{aligned} \tag{2.35}$$

if we use an initial and boundary conditions for forward and adjoint problems as follows

$$\begin{cases} \delta\mathbf{u}_{q,\mathbf{f}}(0) = \frac{\partial}{\partial t}\,\delta\mathbf{u}_{q,\mathbf{f}}(0) = 0, \\ \mathbf{u}^{\dagger}_{q,\mathbf{f}}(T) = \frac{\partial}{\partial t}\,\mathbf{u}^{\dagger}_{q,\mathbf{f}}(T), \end{cases} \tag{2.36}$$

then the terms α and β in equation (2.35) vanish. Therefore we get

$$\int_0^T \int_G \frac{\partial^2}{\partial t^2}\, \delta \mathbf{v}_{q,\mathbf{f}} \cdot \mathbf{v}^\dagger_{q,\mathbf{f}}\, dxdt = \int_0^T \int_G \delta \mathbf{v}_{q,\mathbf{f}}\, \frac{\partial^2}{\partial t^2}\, \mathbf{v}^\dagger_{q,\mathbf{f}}\, dxdt. \tag{2.37}$$

Which denotes the second equation in the set (2.33). Therefore they give

$$B = \int_0^T \int_G (-\delta p_{q,\mathbf{f}}\ (\nabla \cdot \mathbf{v}^\dagger_{q,\mathbf{f}}) + \delta \mathbf{v}_{q,\mathbf{f}} \cdot\ \frac{\partial^2}{\partial t^2} \mathbf{v}^\dagger_{q,\mathbf{f}})\ dxdt. \tag{2.38}$$

Finally for the left hand side of equation (2.28) we get

$$\begin{aligned} \langle \mathbf{L}\ \delta \mathbf{u}_{q,\mathbf{f}}, \mathbf{u}^\dagger_{q,\mathbf{f}} \rangle &= \int_{\mathbb{R}} \int_{\mathbb{T}} [\delta p_{q,\mathbf{f}}\ (m\, p^\dagger_{q,\mathbf{f}} - \nabla \mathbf{v}^\dagger_{q,\mathbf{f}}) + \delta \mathbf{v}_{q,\mathbf{f}} \cdot (-\nabla (p^\dagger_{q,\mathbf{f}}) + \frac{\partial^2}{\partial t^2} \mathbf{v}^\dagger_{q,\mathbf{f}})]\ drdt = \\ &= \langle \delta \mathbf{u}_{q,\mathbf{f}}, \mathbf{L}^\dagger\ \mathbf{u}^\dagger_{q,\mathbf{f}} \rangle, \end{aligned} \tag{2.39}$$

which equals to the right hand side of equation (2.28).

Finally we can find $\mathbf{L}^\dagger$

$$\mathbf{L}^\dagger \mathbf{u}_{q,\mathbf{f}} = \begin{pmatrix} m\, p^\dagger_{q,\mathbf{f}} - \nabla \cdot \mathbf{v}^\dagger_{q,\mathbf{f}} \\ -\nabla (p^\dagger_{q,\mathbf{f}}) + \frac{\partial^2}{\partial t^2} \mathbf{v}^\dagger_{q,\mathbf{f}} \end{pmatrix} = \begin{pmatrix} m & -\nabla \cdot \\ -\nabla & \frac{\partial^2}{\partial t^2} \mathbb{I} \end{pmatrix} \begin{pmatrix} p^\dagger_{q,\mathbf{f}} \\ \mathbf{v}^\dagger_{q,\mathbf{f}} \end{pmatrix}, \tag{2.40}$$

which gives that

$$\mathbf{L}^\dagger = \begin{pmatrix} m & -\nabla \cdot \\ -\nabla & \frac{\partial^2}{\partial t^2} \mathbb{I} \end{pmatrix}. \tag{2.41}$$

Now we review the source signatures and corresponding modeled data in the forward modeling scheme.

2.4 Sources and Data

In this section we explain the sources and then by means of forward modeling we generate pressure and velocity data. It is worth mentioning that these sources have potential applications in real life, however the development of this technology (designing these sources) is intrinsically connected to the design of acquisition geometries, i.e., where to best place a distribution of a limited number of sources and receivers within certain practical constraints. For example, one of the real life applications of these sources is in marine seismic acquisition in which designing such dipole point force sources helps to overcome some challenges including noise, illumination, bandwidth and signal aliasing on the source-side. Such sources also contribute to wavefield separation and ghost removal. Meier et al. (2015) discuss the design of a marine dipole source and explain how its technical realization could help to overcome the ghost issues at low frequencies [32]. Also in seismic imaging by employing such sources we can image down-going waves as well as up-going waves. Another application of these sources can be in medical imaging where the dipole sources can generate wavefields in specific directions. In this thesis we employ these sources to generate vector acoustic data for the purpose of VFWI. To this end we begin with explaining the different type of sources.

We consider two main types of sources: *Monopole Pressure* (Figure 2.3) and *Dipole Point Force Sources*. We utilize three types of dipole sources, namely: *Vertical Dipole Point Force Source* (Figure 2.4), *Horizontal Dipole Point Force Source* (Figure 2.5) and *Angle Dipole Point Force Source* (Figure 2.6).

Having these sources ensures that we have a fully vector data set, on both the source and receiver sides.

2.4.1 Monopole Pressure and Dipole Point Force Sources

The conventional seismic source is defined by the multiplication of the Ricker wavelet $w(t)$ in time and Dirac delta function in space:

$$s(t, z, x) = w(t)\ \delta(\mathbf{x} - \mathbf{x}_s). \tag{2.42}$$

In the conventional second order acoustic wave equation, which is used to solve the forward modeling problem in our method, we therefore need to input $\kappa\ \frac{\partial^2}{\partial t^2} q$ for the monopole pressure source and $\rho^{-1}\ \nabla \cdot \mathbf{f}$, for the dipole point force source. Thus our sources have the following signatures:

$$\beta\ \frac{\partial^2 w(t)}{\partial t^2}\ \delta(\mathbf{x} - \mathbf{x}_s), \tag{2.43}$$

for the monopole pressure source, and

$$n_z\ \gamma\ w(t) \frac{\partial \delta(z - z_s, x - x_s)}{\partial z} + n_x\ \gamma\ w(t) \frac{\partial \delta(z - z_s, x - x_s)}{\partial x}, \tag{2.44}$$

for the dipole point force source, where $\mathbf{x} = (z, x)$, $\mathbf{x}_s = (z_s, x_s)$, β and γ are source weights and $(n_z, n_x)^T$ is a unit vector. By varying this unit vector we can arbitrarily set the orientation of the dipole point force source, e.g. $(n_z, n_x)^T = (0, 1)^T$ for the horizontal dipole source and $(1, 0)^T$ for the vertical dipole source. Introducing β and γ ensures that different sources have correct total output energy:

$$\begin{cases} \beta = 2.0\kappa \frac{\Delta t}{\Delta x^2}, \\ \gamma = 2.0\rho^{-1} \frac{\Delta t}{\Delta x^2}. \end{cases} \tag{2.45}$$

We then define the vertical dipole source by

$$s_{vertical} = \gamma \begin{pmatrix} w(t)\frac{\partial \delta(z-z', x-x')}{\partial z}\hat{k} \\ 0 \end{pmatrix}.$$

Our horizontal dipole source similarly is

$$s_{horizontal} = \gamma \begin{pmatrix} 0 \\ w(t)\frac{\partial \delta(z-z', x-x')}{\partial x}\hat{i} \end{pmatrix}.$$

Finally our angle dipole source is

$$s_{angle} = w(t) \begin{pmatrix} \frac{\partial \delta(z-z', x-x')}{\partial z}\hat{k} \\ \frac{\partial \delta(z-z', x-x')}{\partial x}\hat{i} \end{pmatrix}.$$

Compared to standard data, the monopole pressure source generates the same data with the same polarity as usual seismic source, up to a scalar multiplier except that standard data only record the pressure and not the velocity data.

In order to generate vector data and solve the wave equation of our system in the time domain, we use a two-dimensional acoustic solver from PySIT [56]. Figure 2.2 shows a graphical representation of the staggered-grid implementation for computing the velocity. V_x and V_z represent the particle displacement, and P represents the acoustic pressure. The grids of the V_x and V_z wavefields are positioned in between the P grids [27].

Generated scalar and vector data by using different sources are shown in Figures 2.3, 2.4, 2.5 and 2.6. Figure 2.3 shows a graphical representation of monopole source (Eq. 2.43) orientation and its associated wavefield. Also data generated by this source and a standard seismic source (equation 2.42). The data are similar as expected. Figure 2.4 represents a vertical dipole source orientation and its associated wavefield. In addition, it shows the generated data by this source which depicts the polarity in vertical direction. The two last figures (2.5 and 2.6) also give the same information as figure 2.4, however they clearly differ in source orientations, their associated wavefields and generated data polarities. As can be seen from Figure 2.4, the wavefields are in vertical direction and data polarity indicates the vertical direction. Also Figure 2.5 shows that its associated wavefields are in the horizontal direction and its data polarity is asymmetric respect to vertical axis with plus/minus polarity. Figure 2.6 indicates more or less the same data polarity as Figure 2.5, however the direction of wavefields clearly makes an angle with respect to the horizontal axis.

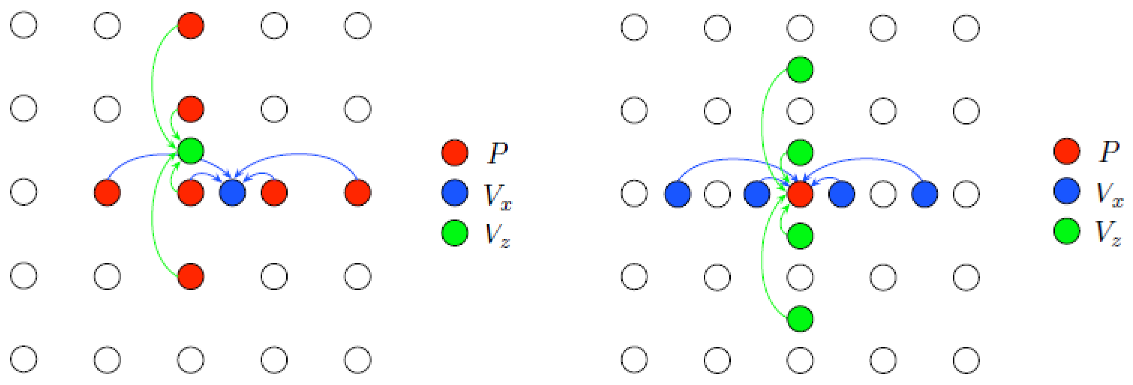


Figure 2.2: Acoustic staggered calculation grid for a fourth-order scheme in space. The grid points needed to update the V_x and V_z (left) and P (right) wavefields. The wavefields all have a unique grid position. This means that the grids of the V_x and V_z wavefields are positioned in between the P grid. (The figure is taken from [27].)

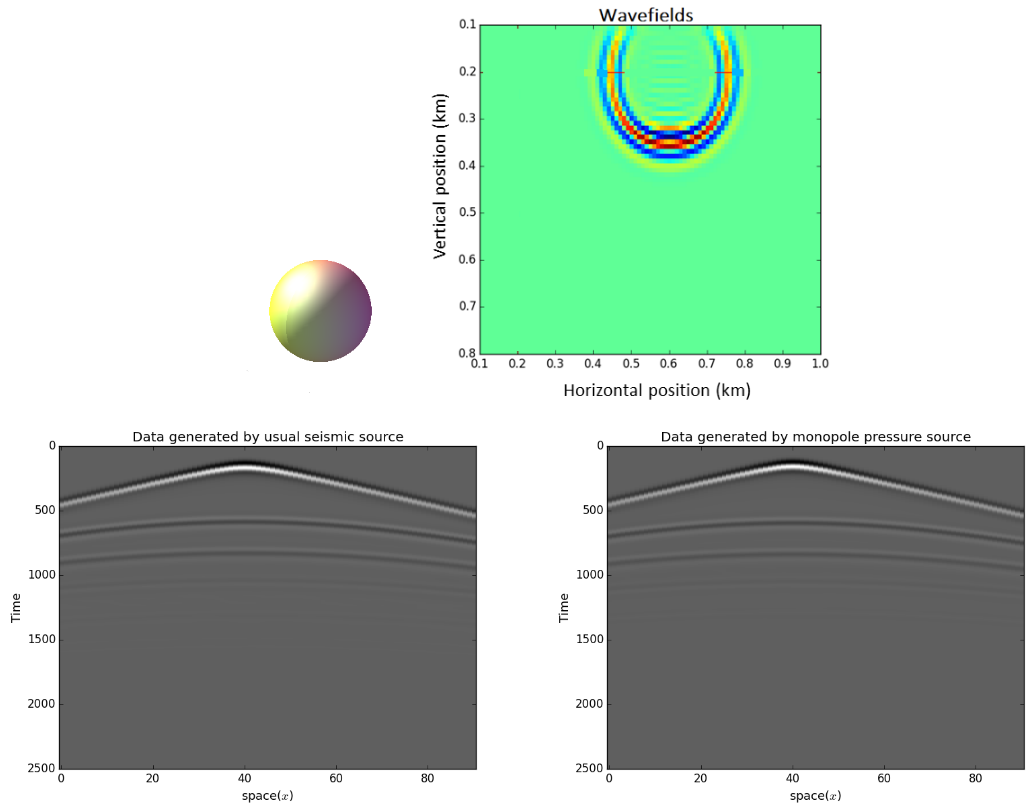


Figure 2.3: Monopole source orientation and a snapshot of its wavefields. Also scalar data generated by usual seismic (bottom left) and monopole pressure (bottom right) sources are shown (generated data should be similar). The data is generated for a single layered model where sources and receivers are equally spaced and spread over the entire top surface of computational domain. The pressure component of data is shown in the bottom left and right. The polarities of generated data by the usual seismic and monopole pressure sources are roughly the identical. The polarities are denoted clearly by the direct wave (first arrival events) and the other events are reflections.

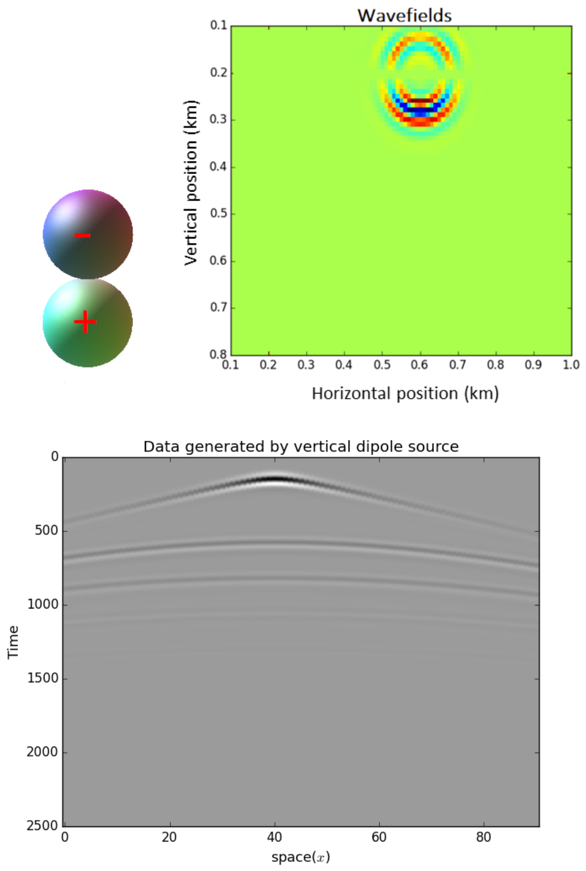


Figure 2.4: Vertical Dipole source orientation and a snapshot of its wavefields. Vector data generated by vertical dipole source is plotted. The data is generated for a single layered model where sources and receivers are equally spaced and spread over the entire top surface of computational domain. The pressure component of data is shown in the bottom. Direction of generated wavefields indicates downward force direction along vertical axis.

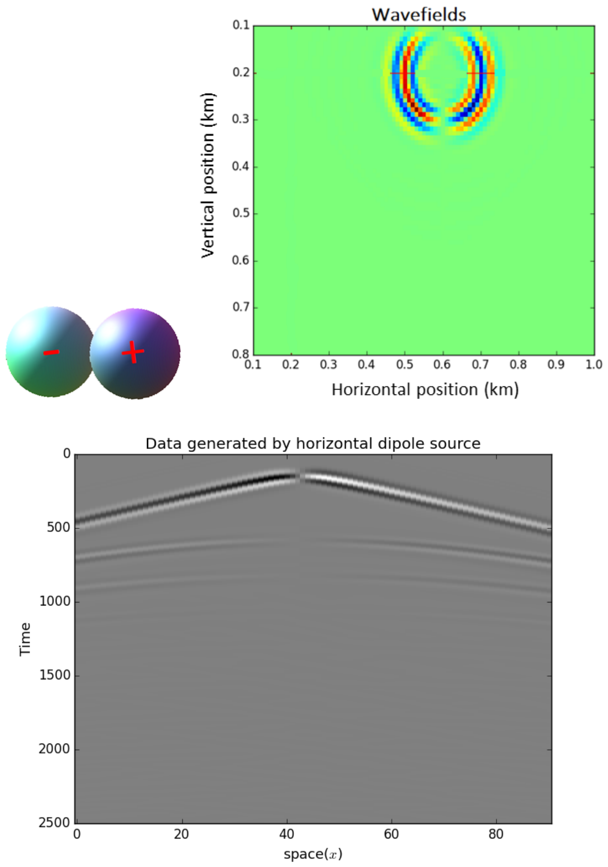
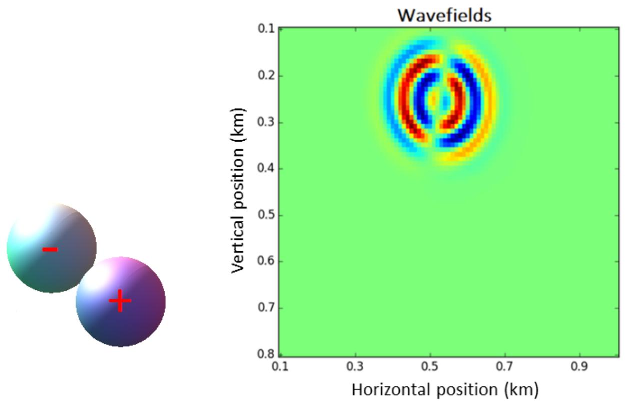


Figure 2.5: Horizontal Dipole source orientation, a snapshot of its wavefields and associated vector data generated are represented. Again the data is generated for a single layered model where sources and receivers are equally spaced and spread over the entire top surface of computational domain. The pressure component of data is shown in the bottom. In this case, the polarity of data is similar to the angle dipole source (figure 2.6).



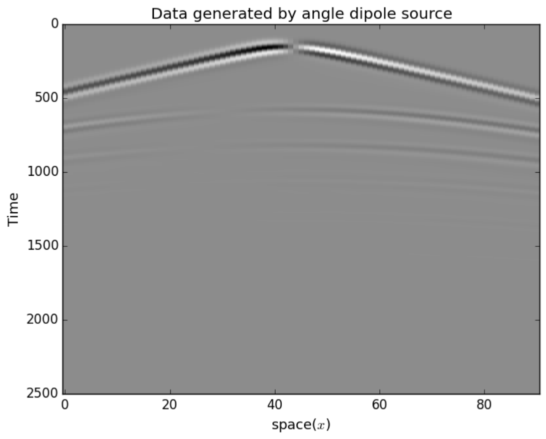


Figure 2.6: Angle Dipole source orientation and a snapshot of its wavefields. Also The pressure component of data generated by this source is shown which clearly represent the vector data recorded by the receivers. The data is generated for a single layered model where sources and receivers are equally spaced and spread over the entire top surface of computational domain.

2.5 Optimization and Algorithm

In this section, we briefly review the optimization method which we used.

We rewrite the Taylor expansion of objective function $\mathcal{J}$ with respect to model parameter m as

$$\mathcal{J}(m+\delta m) = \mathcal{J}(m) + \nabla\mathcal{J}(m)\delta m + \frac{1}{2}\delta m\mathcal{H}(m)\delta m + \mathcal{O}(\delta m)^3, \tag{2.46}$$

where $\mathcal{H}(m) = \nabla^2\mathcal{J}(m)$. If we assume that Hessian information is unavailable, as is usually the case, then we can approximate $\mathcal{H}^{-1}$ directly:

$$\nabla_m\mathcal{J}(m+\delta m) \approx \nabla_m\mathcal{J}(m) + \mathcal{H}\delta m, \tag{2.47}$$

which leads to

$$\mathcal{H}^{-1}\{\nabla_m\mathcal{J}(m+\delta m) - \nabla_m\mathcal{J}(m)\} = \mathcal{H}^{-1}\delta\nabla_m\mathcal{J}(m) \approx \delta m, \tag{2.48}$$

where

$$\delta\nabla_m\mathcal{J}(m) = \nabla_m\mathcal{J}(m+\delta m) - \nabla_m\mathcal{J}(m). \tag{2.49}$$

However, the calculation of a very dense matrix $\mathcal{H}$ is unfeasible. This issue becomes very important especially in the case of FWI which is a large scale problem, and the storage of approximated $\mathcal{H}$ and its inverse is very expensive. This is also an important issue for VFWI algorithm.

One of the methods which can be used is l-BFGS (Low-memory BFGS) which is called after Broyden, Fletcher, Goldfarb and Shanno. The BFGS method is based on finding the minimum Frobenius norm correction to the Hessian [37]. In order to overcome the aforementioned challenge for our algorithm, we need to use l-BFGS which never stores $\mathcal{H}^{-1}$. The salient point of using this method is that one does not require to compute $\mathcal{H}$ in any way and only a few gradients of non-linear iterations need to be stored. l-BFGS provides an appropriate scaling of the computed gradients for VFWI which is computationally efficient compared to the other optimization methods.

Algorithm 1 summarizes the sequence of steps in our time domain VFWI method, where we note that steps in the inner loop are performed in parallel. We modified PySIT implementation of the l-BFGS method to our interest.

Algorithm 1 Time domain VFWI algorithm

```
Input: Measured vector acoustic data $\mathbf{d}_{q,\mathbf{f}}$
Output: $\arg\min_m \; J(m)$
Starting model $\longleftarrow m_0$
For k=1:$N_{iter}$
    For s=1:$N_{src}$
        Compute forward wavefields $\mathbf{u}_{q,\mathbf{f}}$ via Equation 2.5;
        Compute data residuals and the objective function;
        Compute back-propagated residual wavefields;
        Compute the gradient using Equation 2.27;
        Add to the summation over all sources;
    End
    Calculate the model update using l-BFGS and update the model
End
```

We clearly see that our VFWI algorithm is a new method to take advantage of vector acoustic data and recover the model. Our algorithm differs from VARTM in four ways:

- Scheme, i.e. FWI vs RTM;
- Methodology;
- Model parameter;
- Results.

We implemented a forward modeling as well as inversion codes of our algorithm which are compatible with PySIT. We briefly explain the discretization of our system and one of our codes in the appendices of this thesis. Also we explain our code implementations by presenting a list of our sub-routines in the following section.

2.6 Implementations

In this section we briefly explain our code implementation by showing our code's sub-routines.

In algorithm 2 we summarize the sub-routines that we coded in order to run our VFWI algorithm. We divide them into two main categories: Forward Modeling and Inversion. We implemented all of these codes in Python language in order to be compatible with PySIT. Most of our codes were not available in PySIT so that we had to write from scratch. For some of them e.g. LBFGSMODIF, we were able to create a modified version of the corresponding PySIT routine (e.g. PySIT's l-BFGS algorithm).
For forward modelling, we essentially designed our sources including monopole pressure and dipole point force sources. We also implemented two classes for vector data shots and generating vector data as well as pulse functions. Also for our large scale models e.g. BP (chapter 3), we used some parallelization techniques including vector data shot level parallelism. In order to plot specifically some of data we did not follow the conventional plotting tools in Python instead, we wrote some codes e.g. Convenient-plot-functions and Ximage in order to visualize our data properly. Also, we created a sub-routine (vis-plot) for making movie of our generated and save wavefields. The other sub-routines for example normalized amplitude is implemented to scale the amplitudes. We explain the implementation of our smoothing operator in

Algorithm 2 List of code sub-routines

1. **Forward Modeling**
    - SOURCES: design monopole pressure source and dipole point-force sources;
    - VECTOR-DATA-SHOT-CLASS: create a class to handle vector data shots;
    - PARALLELISATION: parallel vector data shots by using MPI4PY;
    - PULSE-FUNCTIONS: design pulse functions to handle different source signatures;
    - GENERATING-VECTOR-DATA-CLASS: create a class for generating vector data;
    - CONVENIENT-PLOT-FUNCTIONS: create some functions to plot data corresponding to some specific formats;
    - XIMAGE: create some functions to plot data corresponding to some specific formats;
    - UTIL: including all the tools e.g. derivative operators;
    - VIS-PLOT: to generate and save wavefields movie;
    - SMOOTHING-OPERATOR: create an operator to obtain suitable initial models;
    - NORMALIZED-AMLITUDE: create a class to normalize wavefields amplitudes and obtain average energy ratio;
2. **Inversion Modeling**
    - JOINT-OBJECTIVE-FUNCTION: to handle our joint objective function corresponding to vector data misfit function;
    - TEMPORALMODELLINGVDPOINTFORCESOURCE: create a temporal vector data modelling to handle dipole point force source;
    - TEMPORALMODELLINGVDMONOPOLEPRESSURESOURCE: create a temporal vector data modelling to handle monopole pressure source;
    - ADJOINTS: to handle our adjoint parameters;
    - LBFGSMODIF: create a modified version of the corresponding PySIT routine (e.g. PySIT's l-BFGS algorithm);
    - REGULARIZATION: design a quadratic regularization term for joint objective function;

the next section.

The second category of our coding refers to inversion modeling where we wrote some sub-routines to deal with our algorithm. We first implemented our joint objective function equation (2.10) and then two temporal modeling routines as we wrote our codes in time-domain. We also implemented adjoints, gradients and regularization. We explain our penalty term (regularization) in the following sections and its discretization in appendix B. For optimization, since PySIT's l-BFGS algorithm only handles standard shots and objective functions, we needed to modify some of its routines in order to implement vector data shots and the joint objective function.

2.7 Smoothing Operator

Even direct solutions to linear inverse problems often require smoothing and the ill-posedness of an inverse problem increases the need for smoothing. Besides, we always are not able to recover the earth's model using FWI algorithm unless we use suitable starting model for our algorithm. VFWI algorithm is an ill-posed problem also strongly affected by the chosen initial model. So, in order to overcome the ill-posedness issue and start with a reasonable initial model, we implemented a smoothing operator as a tool to start with a good model. Our smoothing operator is based on convolving two matrices, so that it takes length and number of times to convolve. Then we use Kronecker tensor product to obtain our smoothed matrix. The result of applying smoothed matrix for the BP model is presented in chapter 3.

2.8 Regularization

Our inverse problem is highly ill-conditioned. Rather than obtaining a solution for our inverse problem we acknowledge the fact that there are infinitely many acceptable solutions. So the strategy is to use optimization to find a suitable answer for our problem. The solution we want to recover minimizes a functional $\mathcal{R}(\mathbf{m}-\mathbf{m_{ref}})$, where $\mathcal{R}(\cdot)$ is a function from $\mathbb{R}^2 \to \mathbb{R}$ which is called the regularizer. Usually the best choice for $\mathcal{R}$ is a convex function as we do optimization. The reason is that a convex function does not have multiple local minima so the iterations in algorithms do not stuck in local minima. The choice of regularizer is significant since different choices lead to very different solutions. Obviously, for a meaningful solution of the problem we need to have $\mathcal{R}(\mathbf{m}-\mathbf{m_{ref}})$ small for the true solution.

Regularization operators which have been successfully used for many problems include the gradient and the Laplacian and variations and combination of thereof. These operators imply that the solution is expected to be smooth, with no discontinuities. We used a standard quadratic regularization technique for our problem. Setting

$$\mathcal{R}(m) = \|Lm\|^2,$$

where L is a gradient operator. Therefore, our joint objective function becomes

$$\begin{aligned}\mathcal{J}(m) = \frac{1}{2}\sum_{s,r}\int_0^T \left[\|\mathbf{u}_q(\mathbf{x_s},\mathbf{x_r},t)-\mathbf{d}_q(\mathbf{x_s},\mathbf{x_r},t)\|_2^2 + \|\mathbf{u}_f(\mathbf{x_s},\mathbf{x_r},t)-\mathbf{d}_f(\mathbf{x_s},\mathbf{x_r},t)\|_2^2\right] dt + \\ + \frac{\mu}{2}\|\nabla m\|^2,\end{aligned} \tag{2.50}$$

where μ is a regularization weight. The result is a well-behaved objective function. To illustrate the benefit of this function we apply it to the BP velocity model in the next chapter. Also we explain the discretization of our regularization in appendix B.5.

Chapter 3

Results and discussions

In this chapter we demonstrate our algorithm by giving four examples with different models: two isolated perturbations, horizontal reflector, Marmousi [49] and BP [50] velocity models.

In all of the examples sources and receivers are equally spaced and spread over the entire top surface of our computational domain and below the PML. Depending on the case, each receiver records velocity or/and pressure. The peak frequency associated with the source signature is 10 Hz and our solver has a spatial accuracy order of 4. In all of the examples the generated data are without noise. There are a lot of metrics by which one can estimate the error in the recovered model. We use *RMS* velocity errors as one of the simplest metrics to do that. We show the *RMS* velocity error for the Marmousi and the BP model in Tables 3.1 and 3.2. *RMS* velocity errors can be expressed by

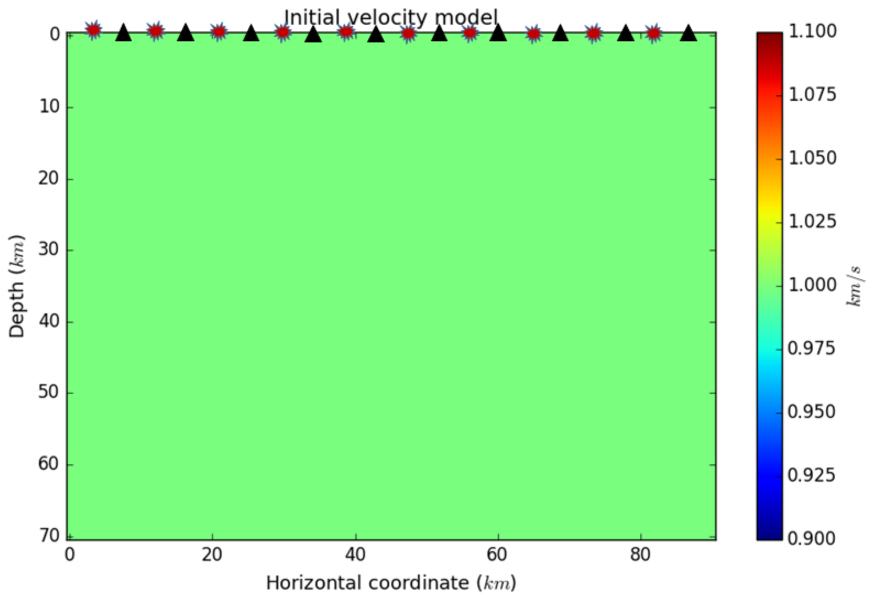
$$\text{RMS velocity error} = \frac{||\text{True velocity} - \text{Estimated velocity}||}{||\text{True velocity}||}.$$

3.1 Two Isolated Perturbations

For the first example we tested our algorithm on a synthetic model shown in Figure 3.1(bottom).

As can be seen from this plot there are two isolated perturbations which violate the homogeneous model. This velocity model is discretized using 91 nodes in the z direction and 71 nodes in the x direction. We start with the uniform model as an initial model to recover the ultimate model using reflected waves. For this purpose, we use 10 equally-spaced sources and receivers and 30 iterations of the l-BFGS scheme for each source type as it is enough to get to the minimum of the objective functions.

Figures 3.2 and 3.3 show the results. For all of the source types we obtain a good reconstruction, with the fewest number of artifacts in the case of vertical and angle sources. Because of the radiation pattern of the horizontal sources, we have less energy interacting with the perturbations so we obtain a poorer reconstruction. For the monopole source, which uniformly radiates energy, we see more artifacts; these are caused by edge effects at the corners of the computational domain that are shown by black arrows in all the figures. Also orange arrows in vertical sources case (Figure 3.3) indicate the areas which have been best illuminated by the radiation of the sources. We also plotted the velocity slices for all the sources at the same plot shows the difference between true and estimated velocities (figure 3.4).



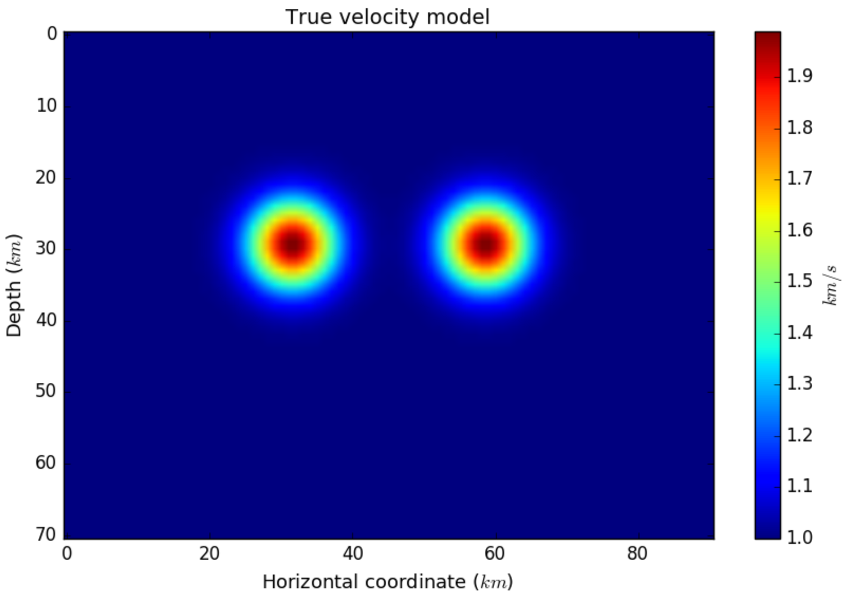


Figure 3.1: Initial (top) and true velocity models (bottom) of two isolated perturbations model. 10 equally-spaced sources and receivers are placed at the top surface of the computational domain. They are indicated by red explosion signs (sources) and black triangles (receivers). As can be seen from true model, there are two isolated perturbations like islands which violate the uniform background model.

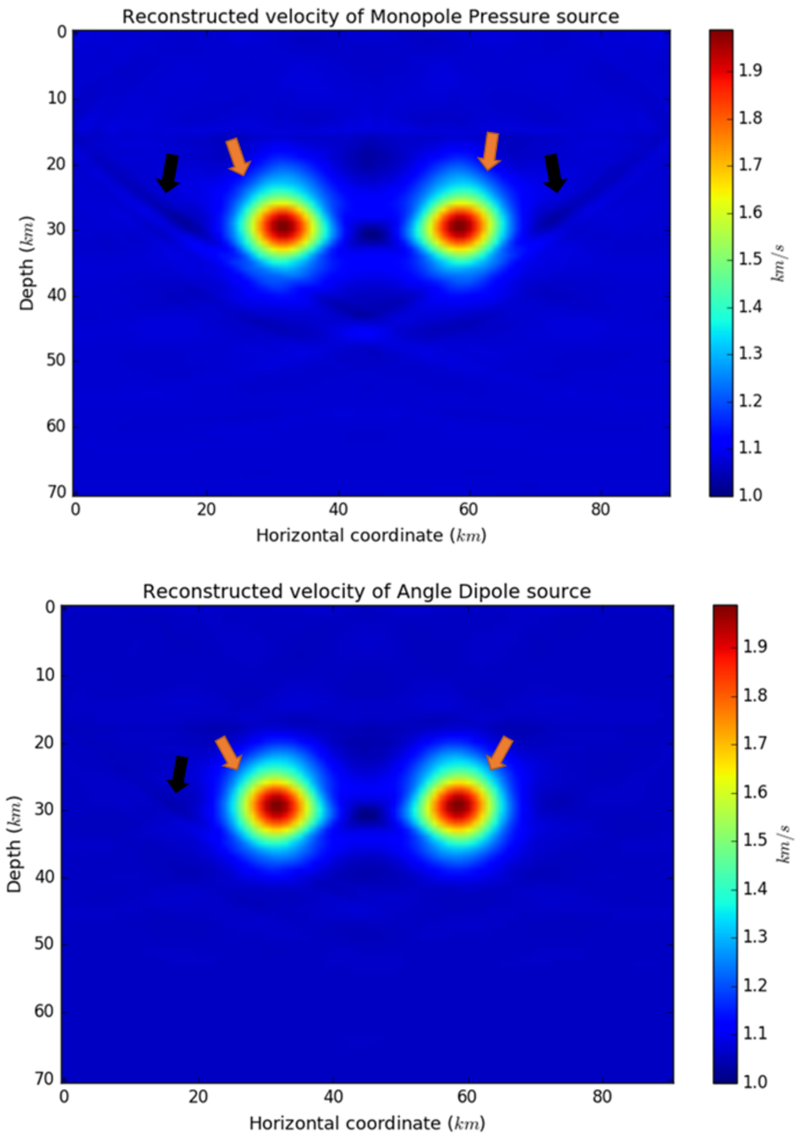
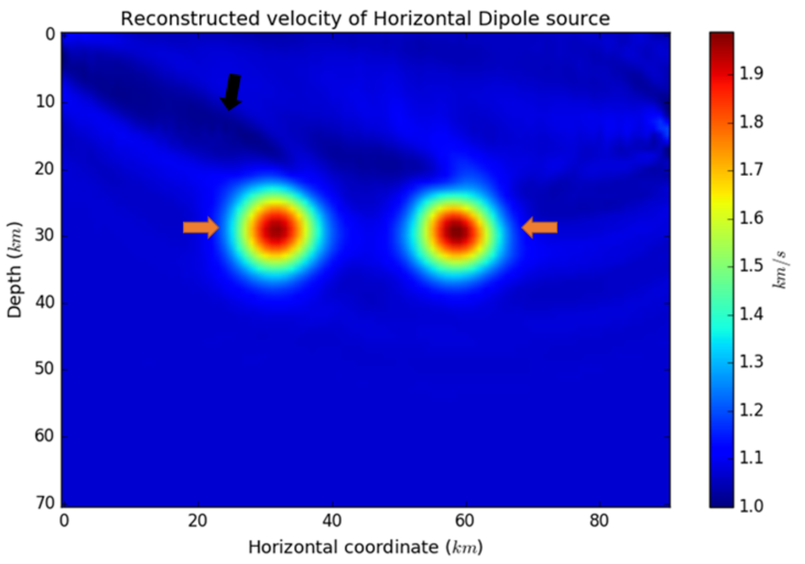


Figure 3.2: Estimated velocities for the two isolated perturbations model by using monopole pressure (top) and angle dipole sources (bottom). Black arrows show the artifacts caused by the edge effect. In the case of monopole pressure source, the artifacts are more clear at both sides of the perturbations, whereas for angle source we only have artifacts at the left side of the perturbation. In both cases (monopole pressure and angle dipole sources), the orange arrows indicate the areas which have been best illuminated by the radiation patters of the sources.



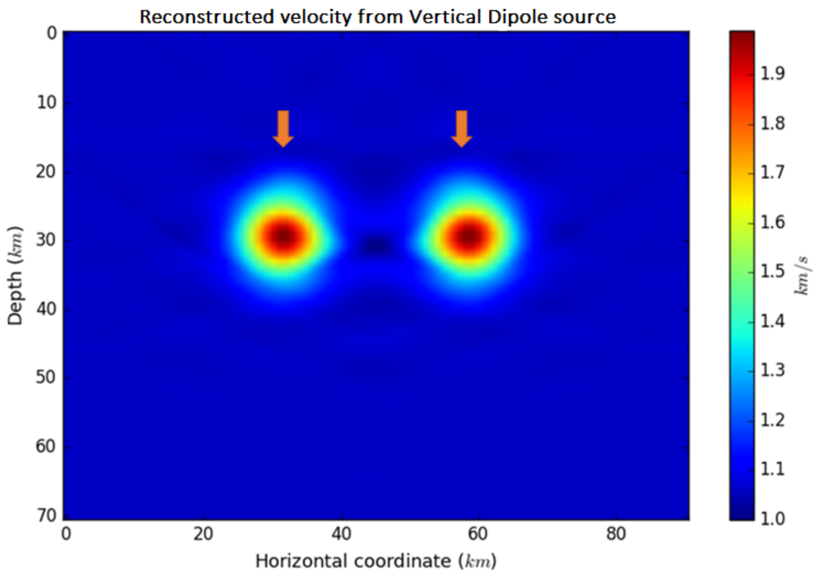


Figure 3.3: Estimated velocities for the two isolated perturbations model by using horizontal (top) and vertical dipole sources (bottom). In the case of horizontal source, the artifacts appear a bit higher compared to the other cases, which is shown by the black arrow. In both cases (horizontal and vertical dipole sources), the orange arrows indicate the areas which have been best illuminated by the radiation patters of the sources.

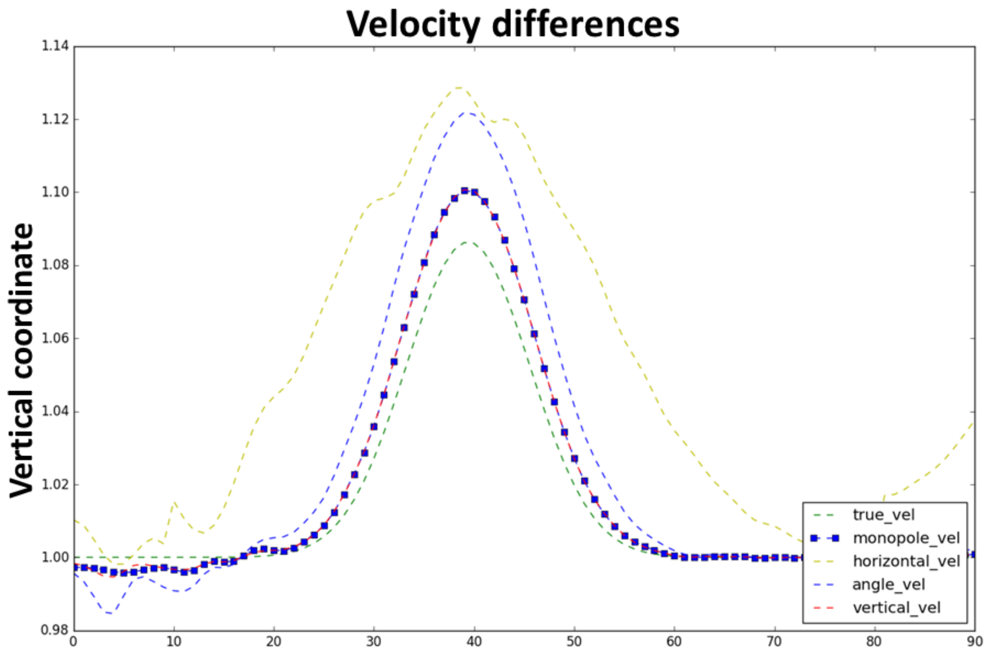


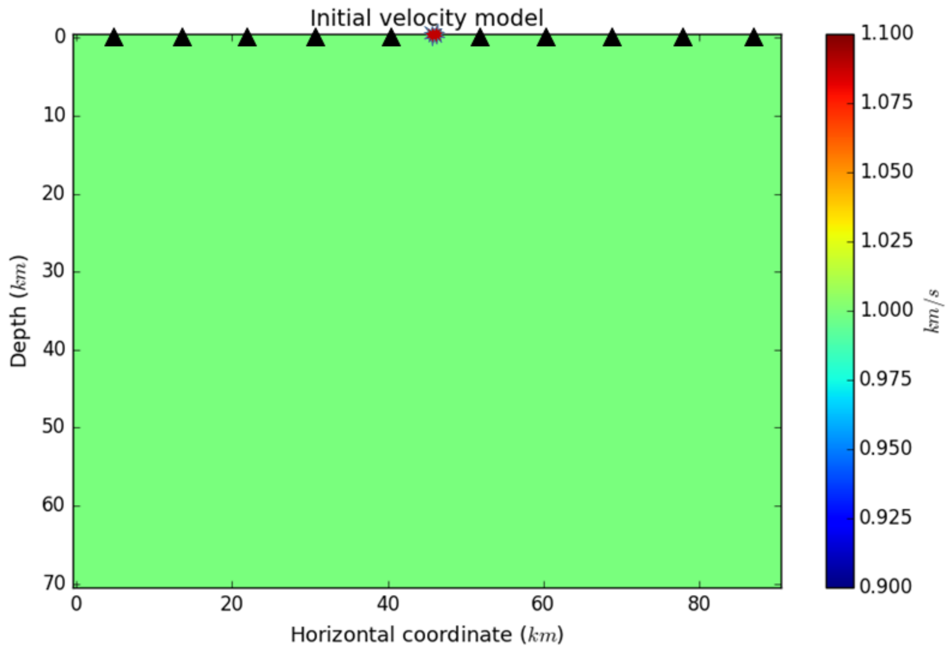
Figure 3.4: Velocity slices for all the sources at the same plot shows the difference between true and estimated velocities.

3.2 Horizontal Reflector

Another synthetic model that we used is horizontal reflector. This model consists of two identical parallel reflection layers. The initial and true velocity models are shown in Figure 3.5. It is worth mentioning that this is not a realistic earth model, but it gives a nice illustration of the effects of the radiation patters of the source. Like the previous example, this model is discretized using 91 nodes in the z direction and 71 nodes in the x direction.

In this case we use 1 source and 10 receivers and 10 iterations to construct the different images. Using 10 iterations in this case gives us desirable convergence, i.e. the objective function values reach a plateau. The results of VFWI corresponding to different sources are shown in Figures 3.6 and 3.7.

The Expected directivity information is very clear in this case. For instance the model estimated using a horizontal dipole source clearly shows the radiation pattern of this source in the recovered image. Also, it can be seen from Figure 3.6 that the reconstructed model by using angle dipole source is lopsided which comes from the angle orientation. The final estimated model using vertical dipole sources also shows strong radiation in vertical direction (Figure 3.7). Like previous example, we plotted the velocity slices for all the sources at the same plot shows the difference between true and estimated velocities (figure 3.8).



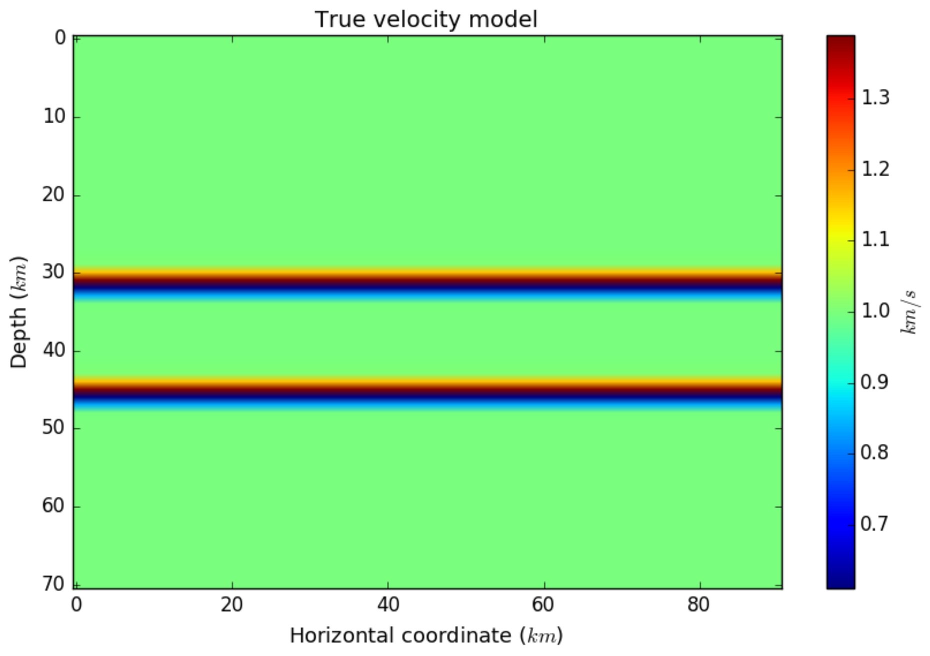
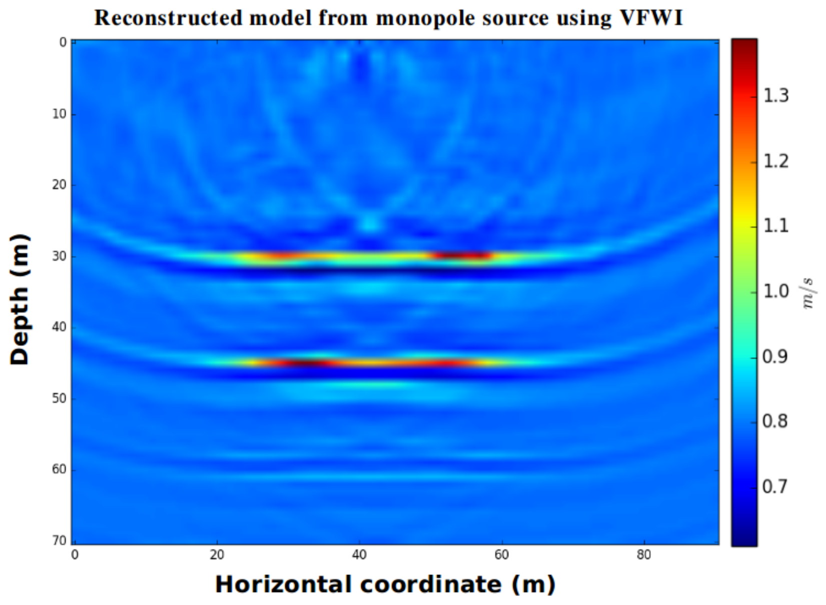


Figure 3.5: Initial (top) and true velocity (bottom) of the horizontal reflector model. 1 source and 10 receivers are placed at the top surface of the computational domain. They are indicated by red explosion sign (source) and black triangles (receivers). As can be seen from the bottom figure, there are two layers located at 30 and 45 kilometer depth in the true model.



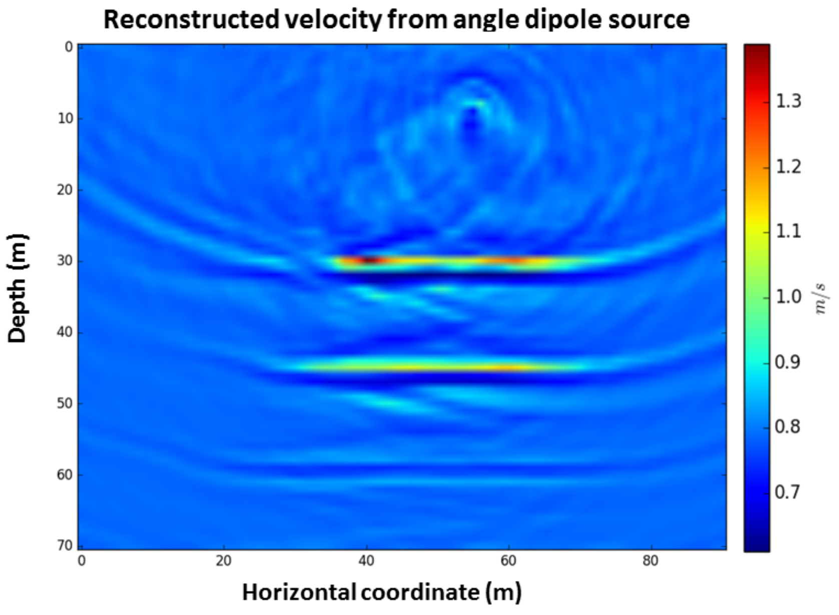
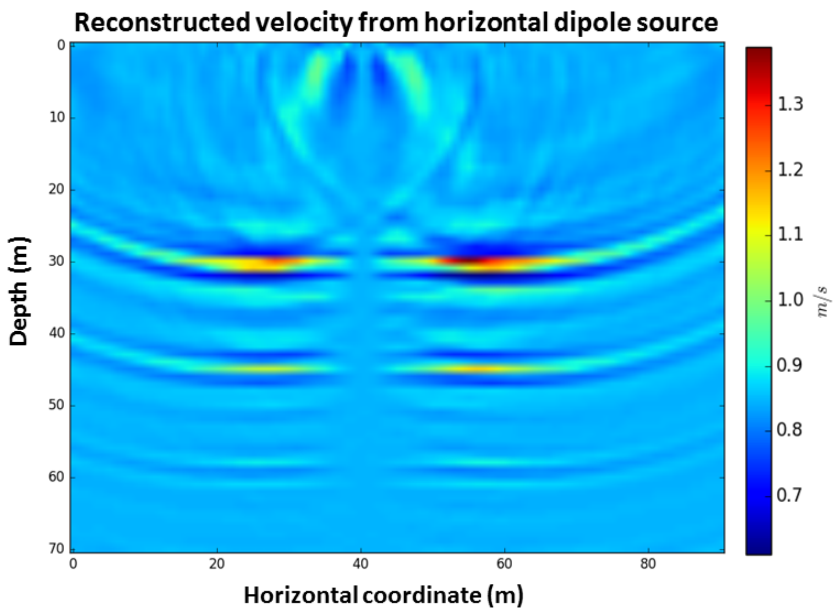


Figure 3.6: Estimated velocities for the horizontal reflector model by using monopole pressure (top) and angle dipole sources (bottom). In the bottom figure the reconstructed model by using angle dipole source is lopsided which comes from the angle orientation.



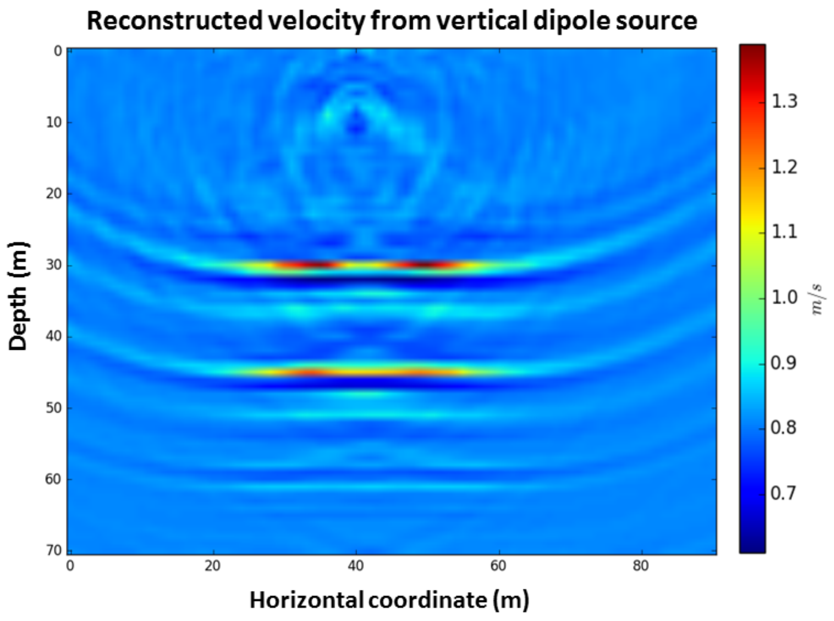


Figure 3.7: Estimated velocities for the horizontal reflector model using horizontal (top) and vertical dipole sources (bottom). The top figure clearly shows the radiation pattern of horizontal source in the recovered image.

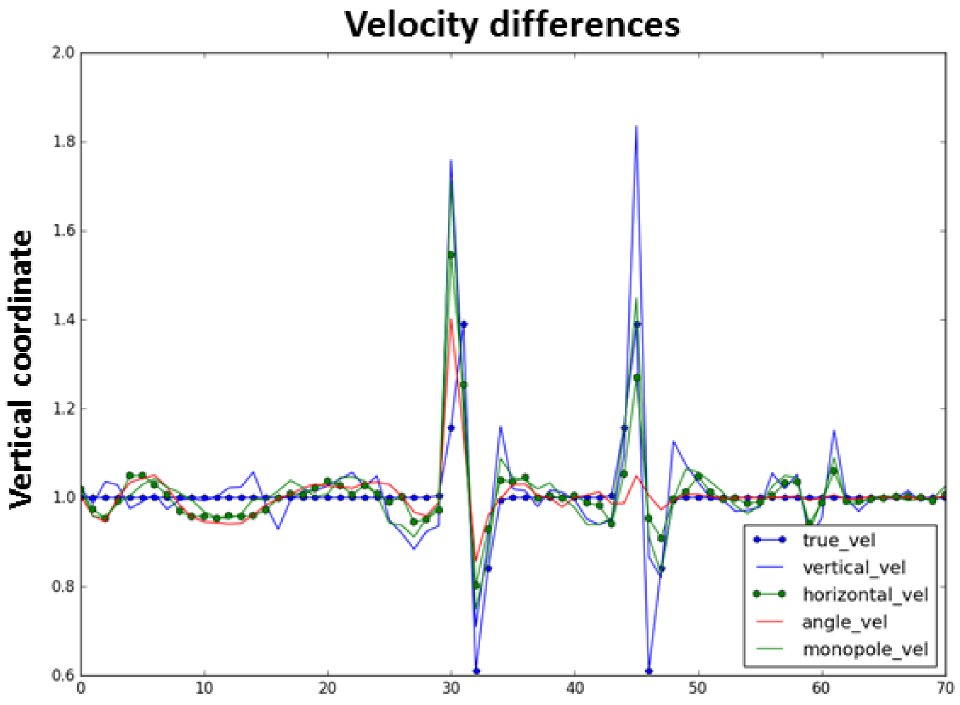


Figure 3.8: Velocity slices for all the sources at the same plot shows the difference between true and estimated velocities.

3.3 Marmousi

In this example we use the VFWI algorithm to reconstruct the Marmousi velocity model [49]. The Marmousi velocity model is discretized using 151 nodes in the z direction and 461 nodes in the x direction. Node spacing is 20m. The inverse crime is committed by using the same solver for generating the 'true' data and the 'synthetic' data. This may make our results appear better than they would be for a real data set.

As in the previous examples, we estimate velocity models using four different sources: monopole pressure, vertical dipole, horizontal dipole and angle dipole sources.

Initial and true velocities are plotted in Figure 3.9. For this example we used 10 sources and the receivers are placed all the way across the top of the computational domain which means in a fixed spread acquisition. We use 30 iterations in this case.

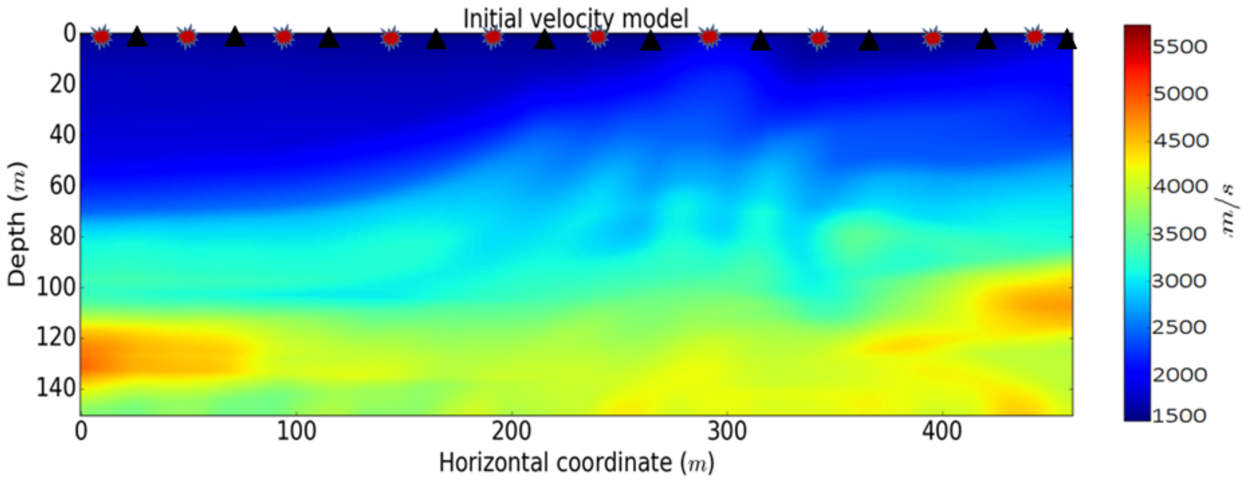
As can be seen from Figure 3.12, the model generated by the monopole pressure sources has better resolution compared to the other cases. The horizontal dipole sources generate poor recovery (Figure 3.12) but show the directivity of wavefields as gives us only the sides velocity recovery. In this case, since recorded VA data contains more horizontal components of velocity, therefore both sides of the model are recovered better than the other areas. Also as for the case of angle dipole source (Figure 3.13), we can see some artefacts in the direction of source orientation. Generating VA data by using these point-force sources and finally recording such data enables us to have clear directivity information about the wavefields and its impact on final estimated model for the Marmousi model. It also has some disadvantages in this case as it cannot provide good resolution compared to the usual monopole seismic source.

Estimated Velocity Error for Marmousi Velocity Model		
Source type	**FWI**	**VFWI**
Monopole Pressure	0.004	——-
Monopole Pressure	——-	0.008
Vertical Dipole	——-	0.012
Horizontal Dipole	——-	0.222
Angle Dipole	——-	0.013

Table 3.1: *RMS* velocities of different source types using FWI and VFWI algorithms for Marmousi model.

Misfit values (in figures 3.10, 3.11, 3.12 and 3.13) corresponding to monopole pressure, vertical, horizontal and angle dipole sources do not converge to zero, which point out a crucial point about our objective functions. Practically, we can not reach zero for misfit values as there are three different issues regarding objective functions; Local minima, saddle point and ill-conditioning issues [54]. Here our objective functions are highly ill-conditioned. I should mention that the misfit values corresponding to the monopole and dipole sources for Marmousi model, are levelled off which can be seen in figures 3.10, 3.11, 3.12 and 3.13.

The resulting *RMS* velocity errors between each recovered model and the true model are shown in Table 3.1. It proves that in this case, the final reconstructed velocity model associated with the monopole pressure source using FWI algorithm is better than in other cases. In other words, although using VFWI for the Marmousi model can provide us with the directivity of wavefields, it has some shortcomings in proving good resolution in the final images.



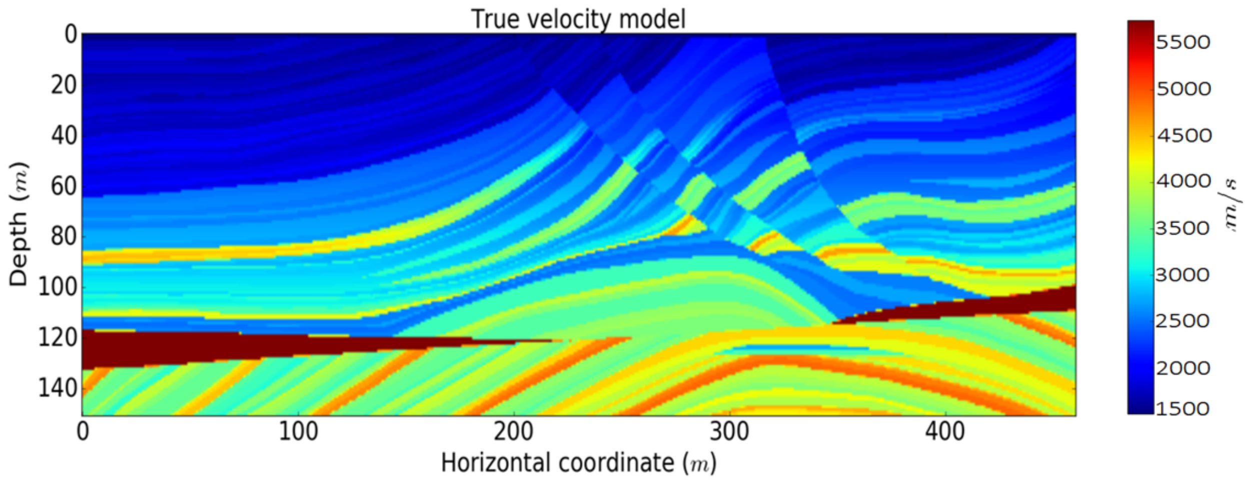
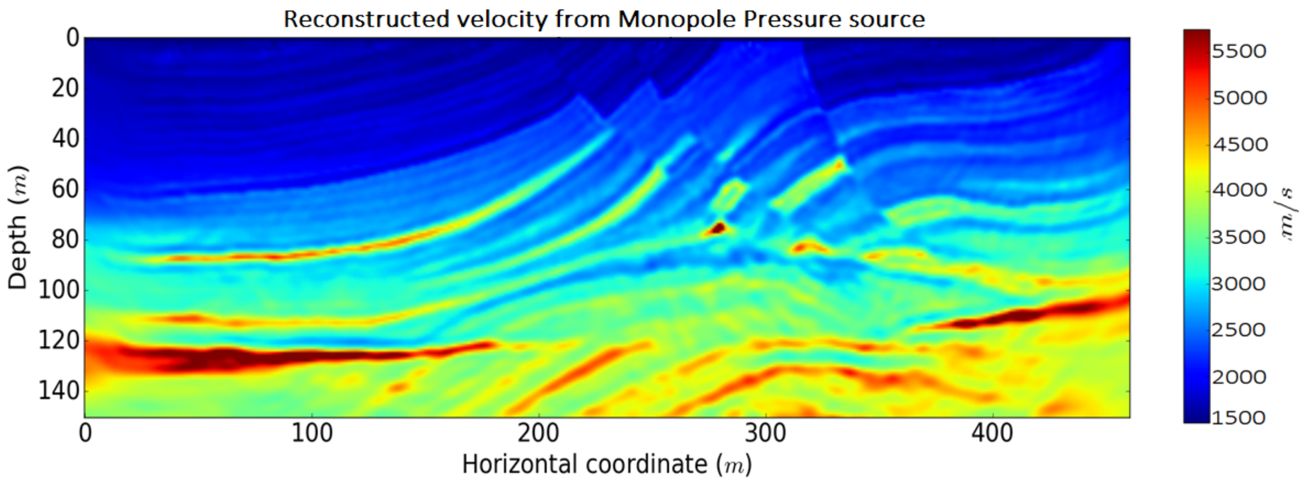


Figure 3.9: Initial and true Marmousi velocity model. 10 equally-spaced sources and receivers are placed at the top surface of the computational domain. They are indicated by red explosion signs (sources) and black triangles (receivers).



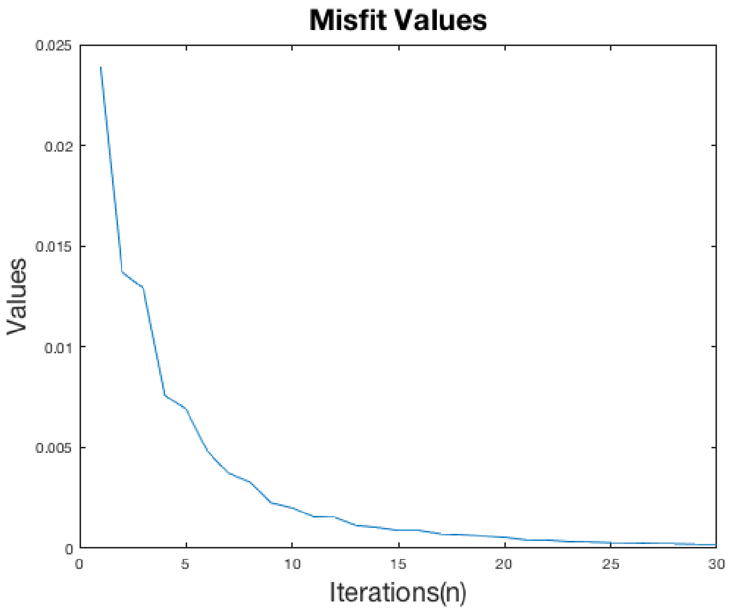
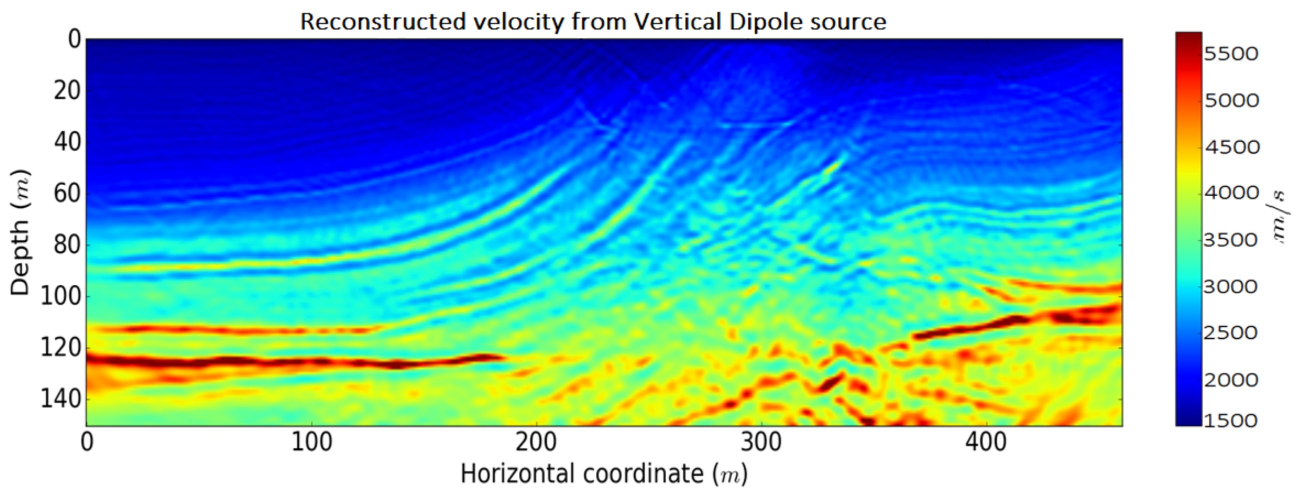


Figure 3.10: Reconstruction of velocity and corresponding misfit values by using monopole pressure source (The result is similar to ordinary seismic source, i.e. monopole pressure source using FWI algorithm).



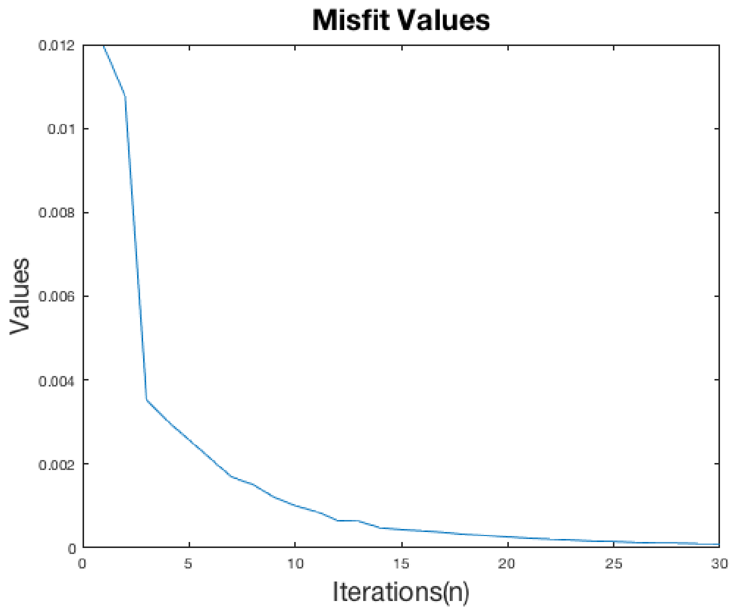
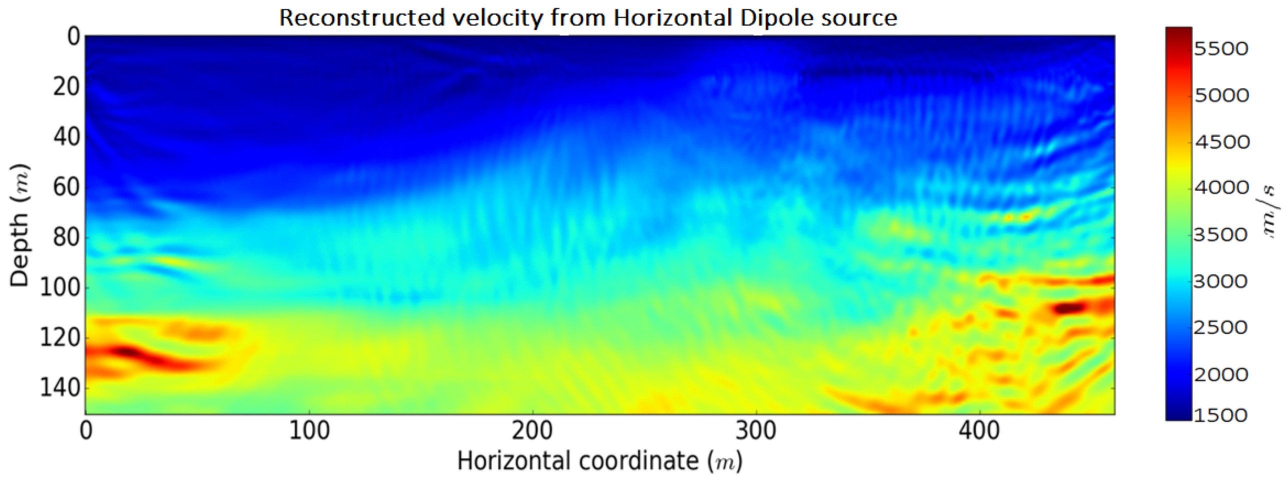


Figure 3.11: Reconstruction of velocity and corresponding misfit values by using vertical dipole source (directionality information).



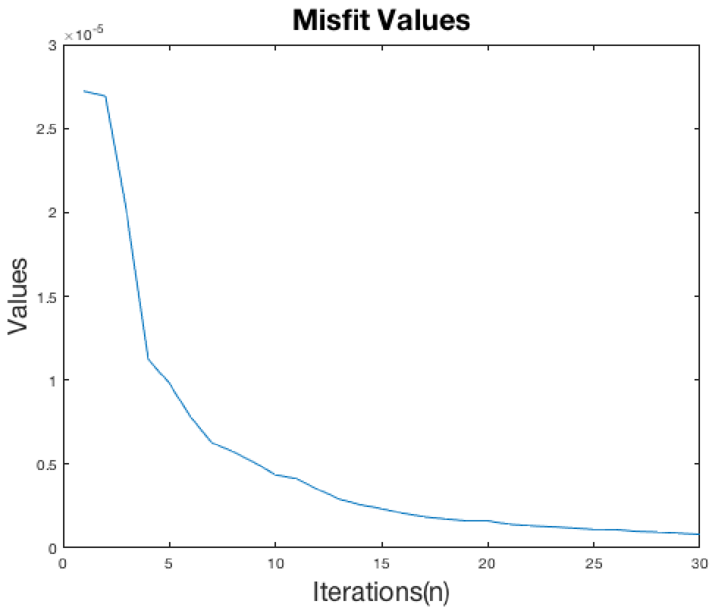
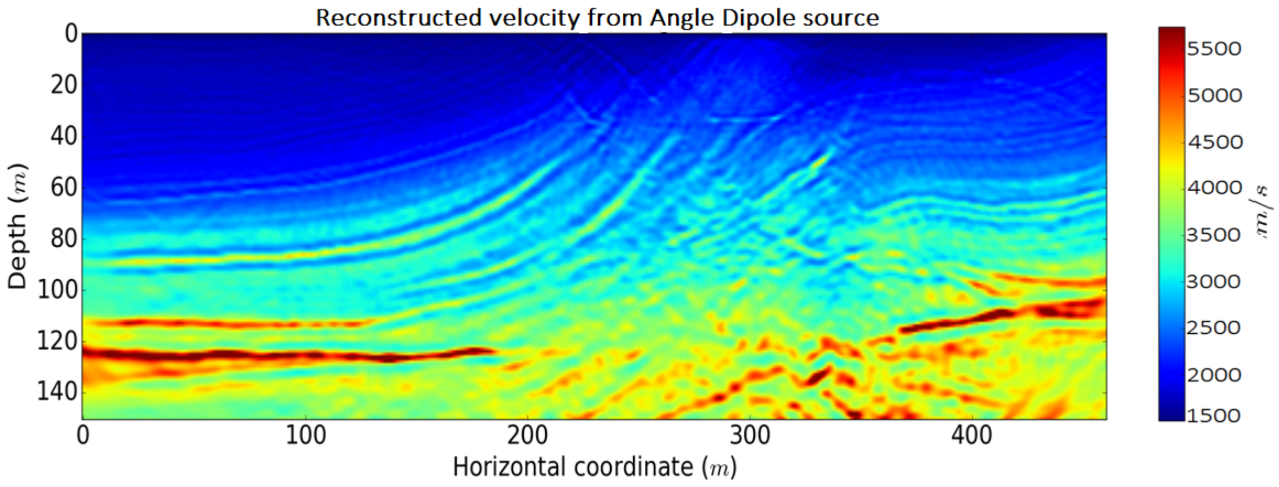


Figure 3.12: Reconstruction of velocity and corresponding misfit values by using horizontal dipole source (directionality information).



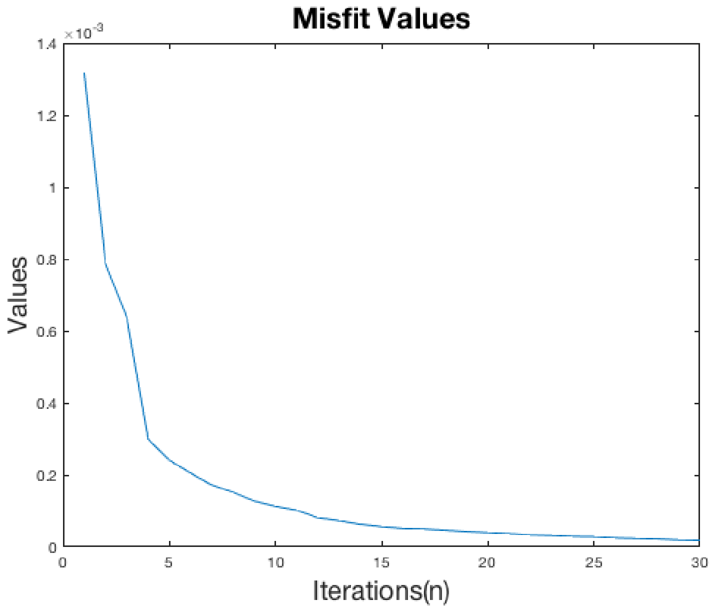


Figure 3.13: Reconstruction of velocity and corresponding misfit values by using angle dipole source (directionality information).

3.4 BP

The last example is the BP velocity model which contains salt bodies with steep flanks and irregular shapes [50]. The BP velocity model is discretized using 115 nodes in the z direction and 205 nodes in the x direction. Like in the previous example, the inverse crime is committed by using the same solver for generating the 'true' data and the 'synthetic' data.

As in the previous examples, we recover velocity models using four different sources: monopole pressure, vertical dipole, horizontal dipole and angle dipole sources. Initial and true velocity models are plotted in Figures 3.14, respectively. Since the BP model is large, we subsampled it so that only 12% of the samples remained, to reduce the computational cost.

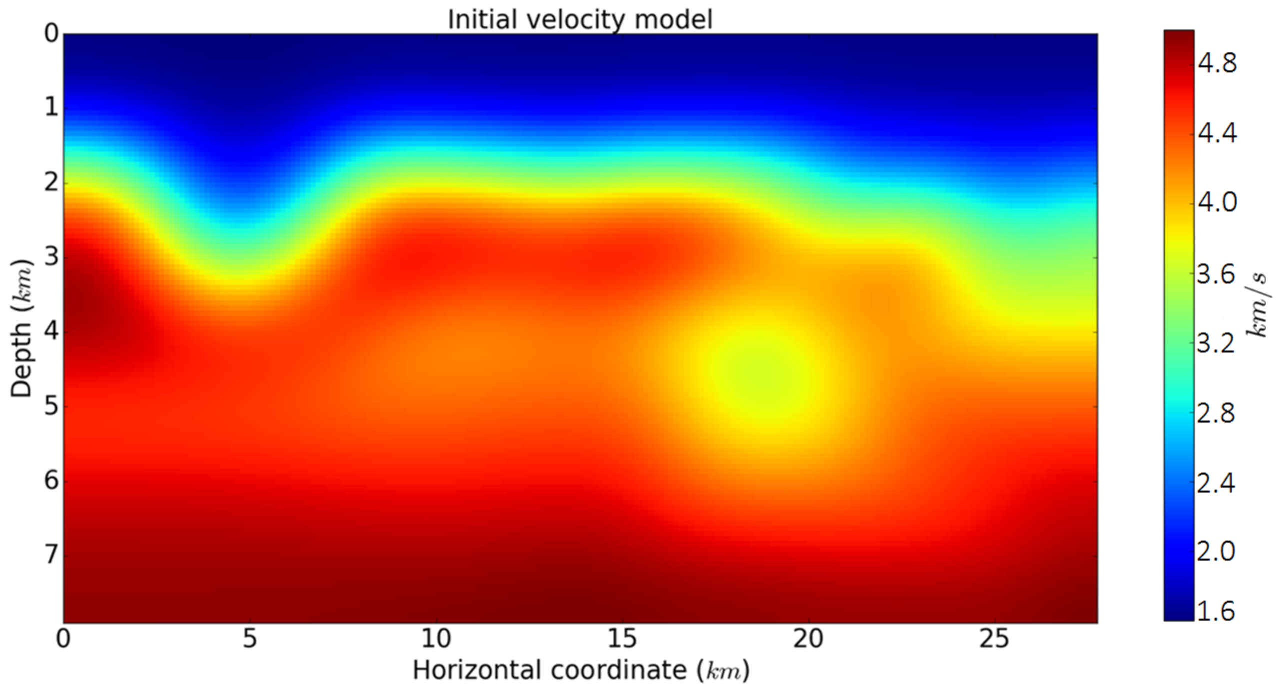
Reconstruction of velocity in the case of BP model is very hard as the model is so complicated. We used our smoothing operator (explained in Chapter 2) to smooth the true model and and thereby obtain an appropriate starting model initial model (Figure 3.14). This is the kind of starting model one would expect to obtain by other processing techniques prior to FWI.

For this example we used 50 sources and 50 receivers placed all the way across the top of the computational domain for monopole pressure, vertical dipole and angle dipole sources. We used 100 sources and 100 receivers for the case of horizontal dipole source. We used 30 l-BFGS iterations to invert this model.

The recoveries from the monopole pressure and horizontal dipole sources, Figure 3.15, are quite good and look similar. As can be seen from Figures 3.16 and 3.18, the recovered velocity models associated with vertical and angle dipole sources have oscillations across the models, especially in the case of vertical sources, where we have a poorer overall reconstruction. However, we also see that we are getting sharper edges on some of the smaller features especially at the top of the salt body and near the edges of the model. In order to remove the artefacts, we add a regularization term in our objective functional (Eq. 2.50) as explained in Chapter 2. After applying regularization we obtain more reliable results shown in Figures 3.16 (bottom), 3.17, 3.18 (bottom) and 3.19. We performed regularization process for two different weights: $\mu = 6$ and $\mu = 10$ for both angle and vertical dipole sources. The corresponding plots demonstrate how increasing μ results in better artefact removal.

As can be seen from Figure 3.15, the model generated by horizontal dipole sources has better lateral resolution compared to the model generated by the vertical dipole sources, Figure 3.17. Similar to the Marmousi model, recorded VA data are more sensitive to the scatterers at the sides of the computational domain. However, the difference between our recovery of the Marmousi and BP models is that in the BP case we used more horizontal dipole sources to obtain a better recovery of other areas of the model as well, whereas in the Marmousi case we did not increase the number of the horizontal dipole sources compared to other source types. The resulting *RMS* velocity errors between each recovered model and the true model are shown in Table 3.2, which demonstrates the best velocity recovery corresponding to the horizontal dipole source. The second best recovery refers to the monopole pressure source moreover both monopole pressure and horizontal dipole sources do not seem to require regularization. It should be noted that the resulting *RMS* velocity errors in the case of vertical and angle dipole are calculated for regularized BP model.

We also plotted the velocity slices for each case show the difference between true and estimated velocities (Figures 3.20, 3.21, 3.22 and 3.23). The last plot (Figure 3.23) denotes a reliable velocity reconstruction corresponding to the horizontal dipole source.



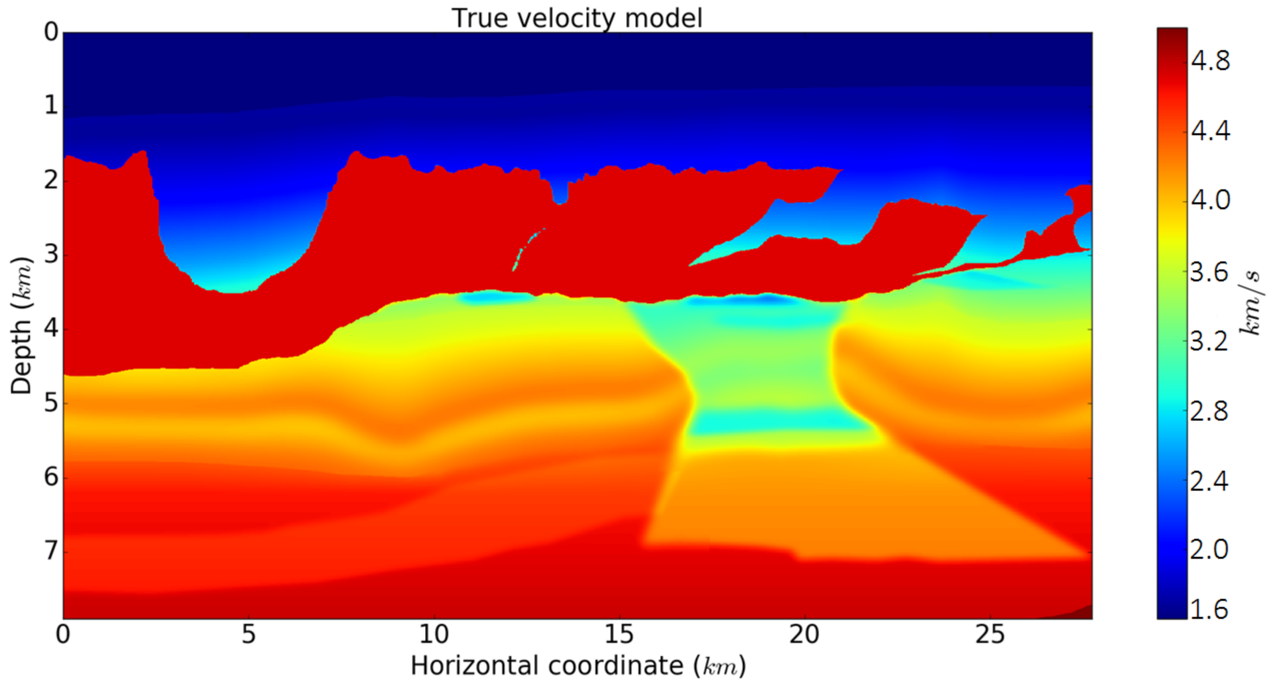
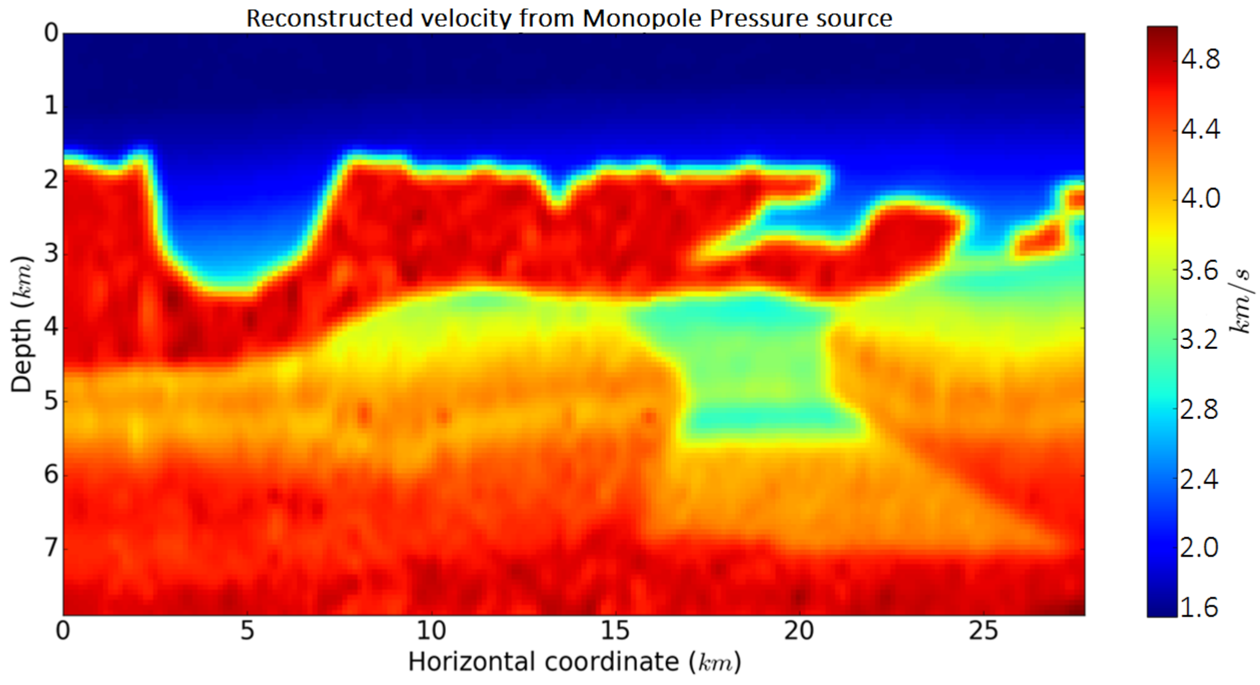


Figure 3.14: A sub-sampled initial and true BP velocity model (with 12% of samples remaining).



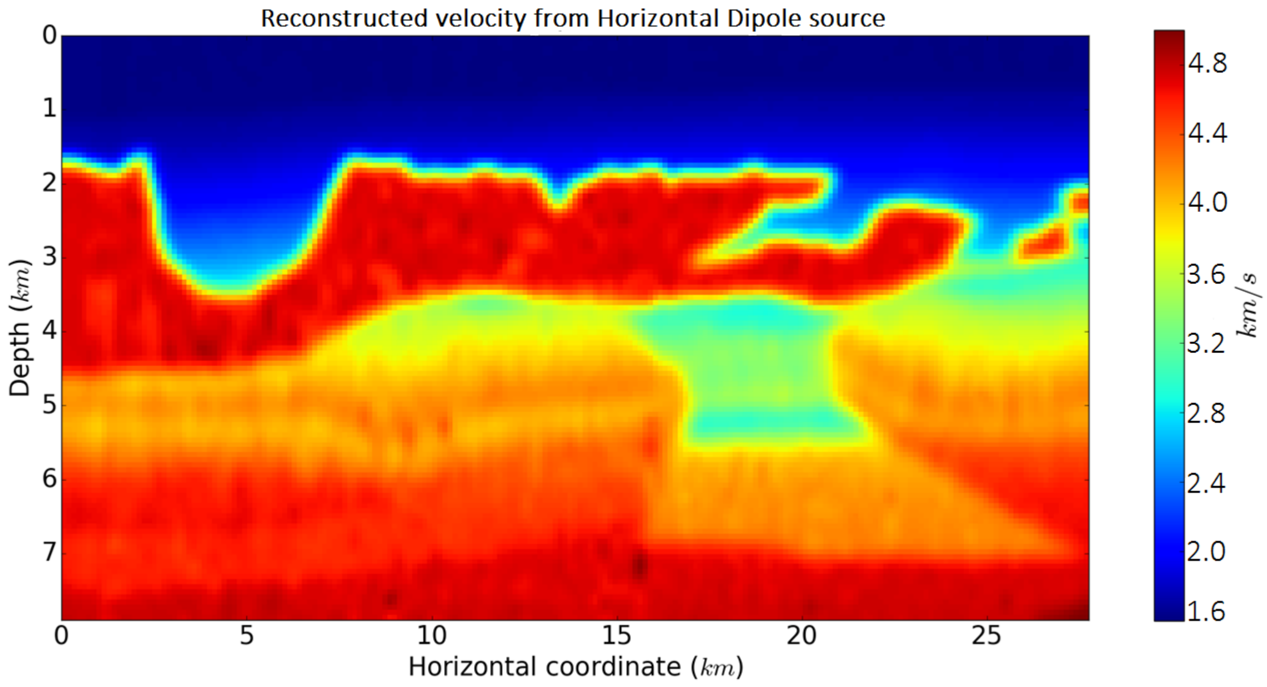
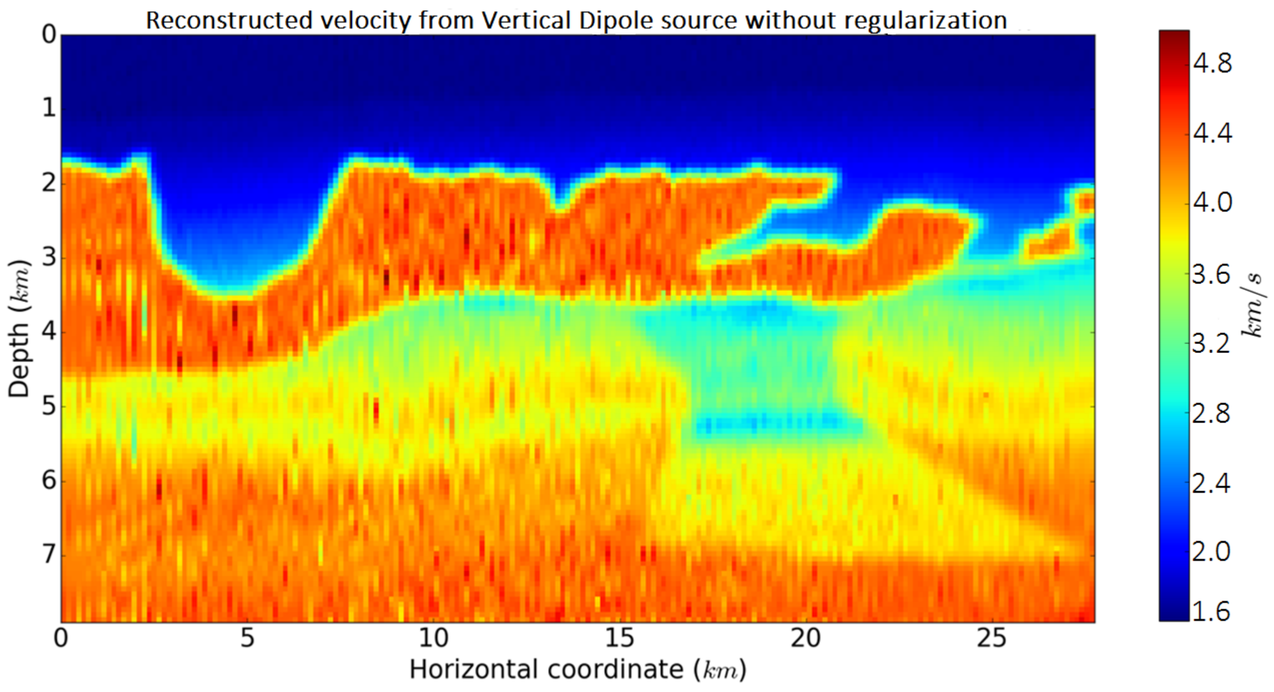


Figure 3.15: Reconstruction of a sub-sampled BP velocity model by using monopole pressure and horizontal dipole sources.



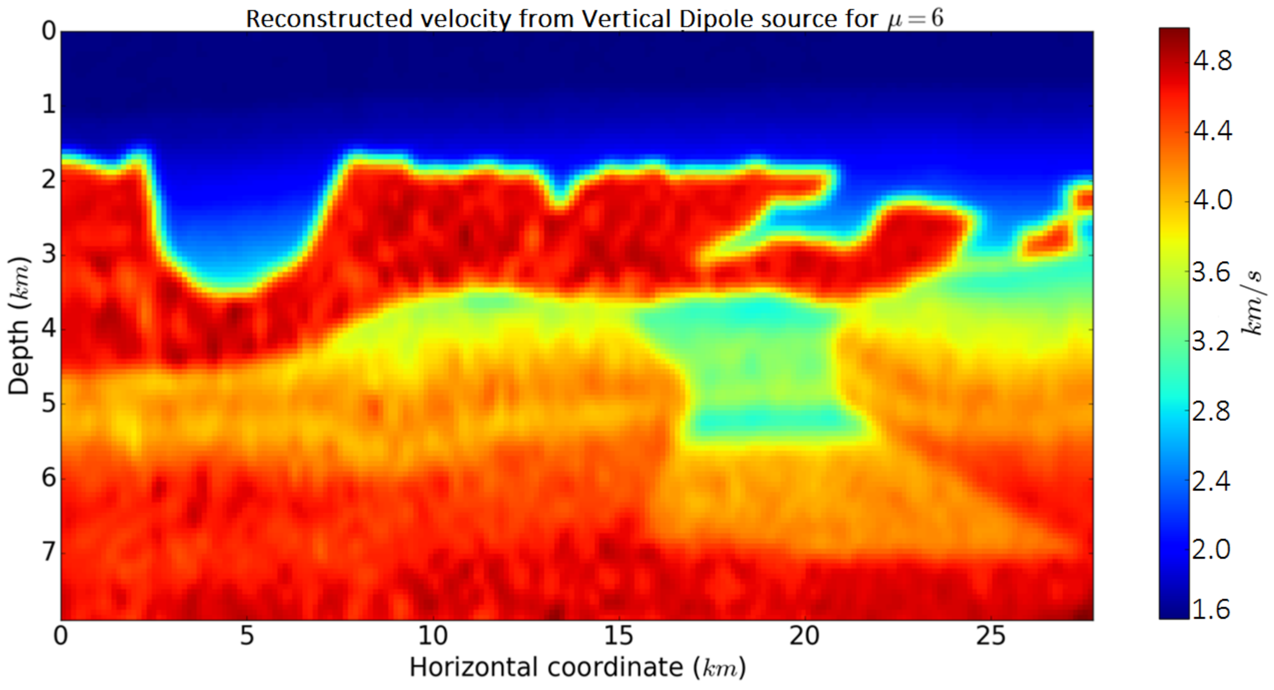


Figure 3.16: Reconstruction of a sub-sampled BP velocity model by using vertical dipole source without regularization (top) and with regularization for $\mu = 6$ (bottom).

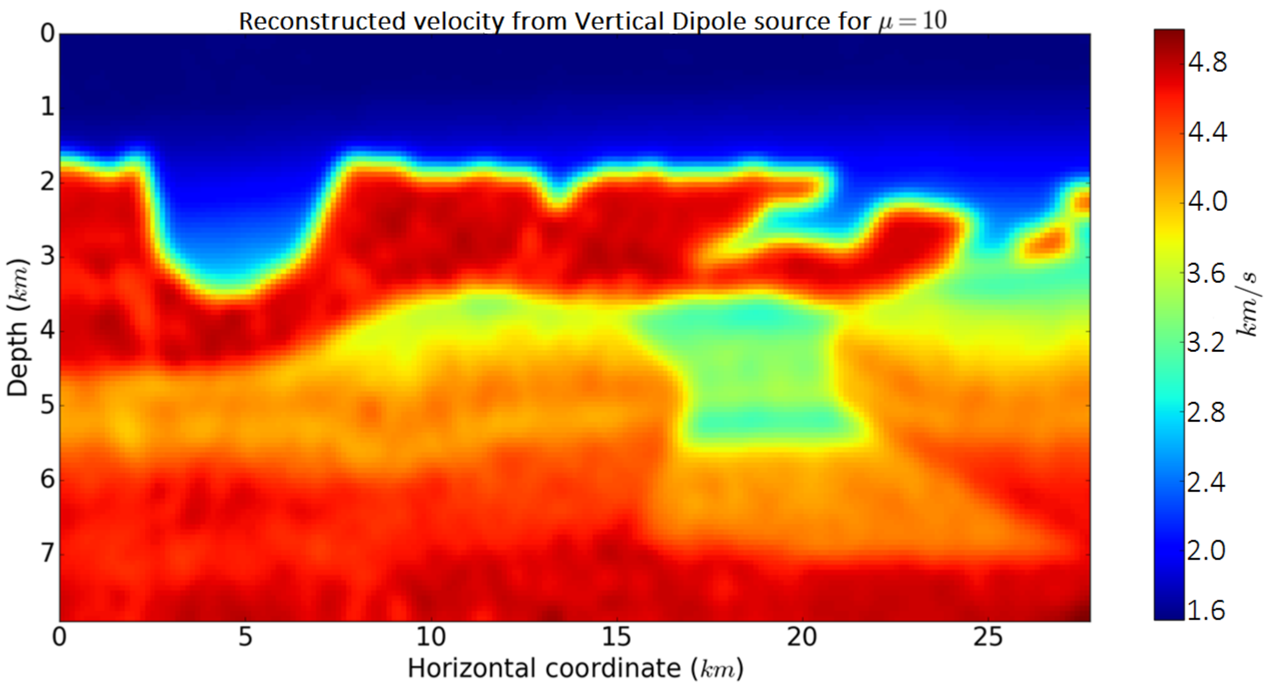
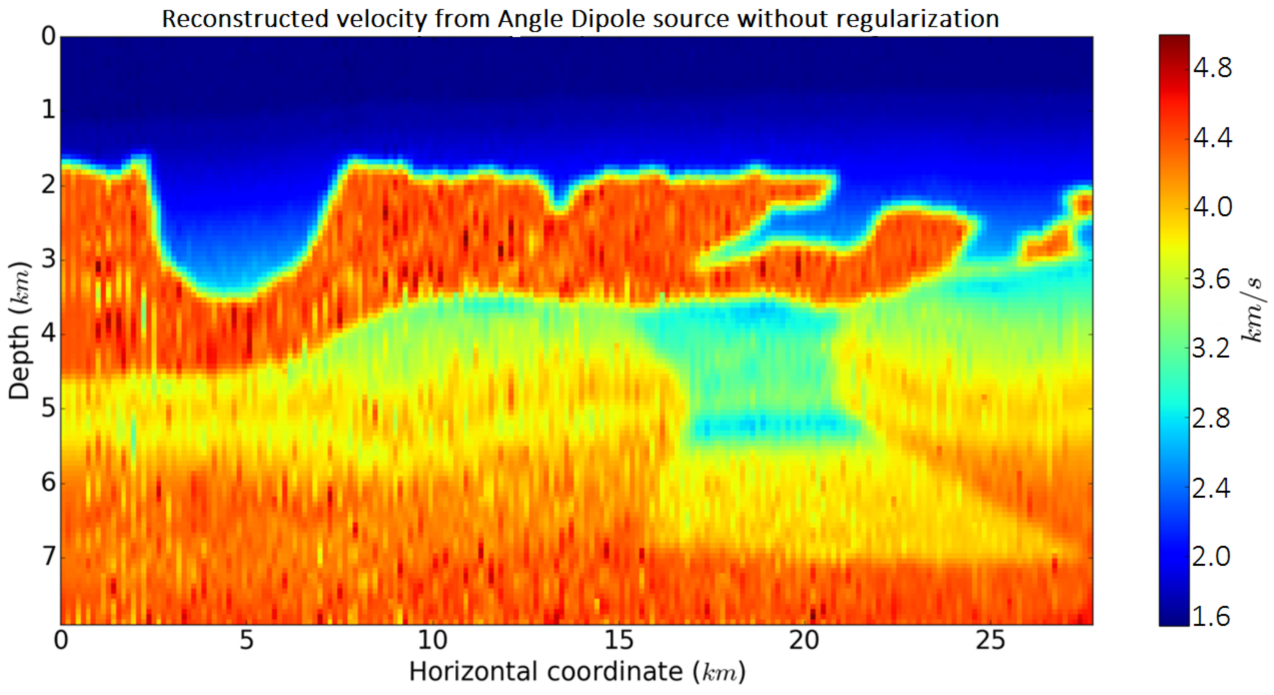


Figure 3.17: Reconstruction of a sub-sampled BP velocity model by using vertical dipole source with regularization ($\mu = 10$).



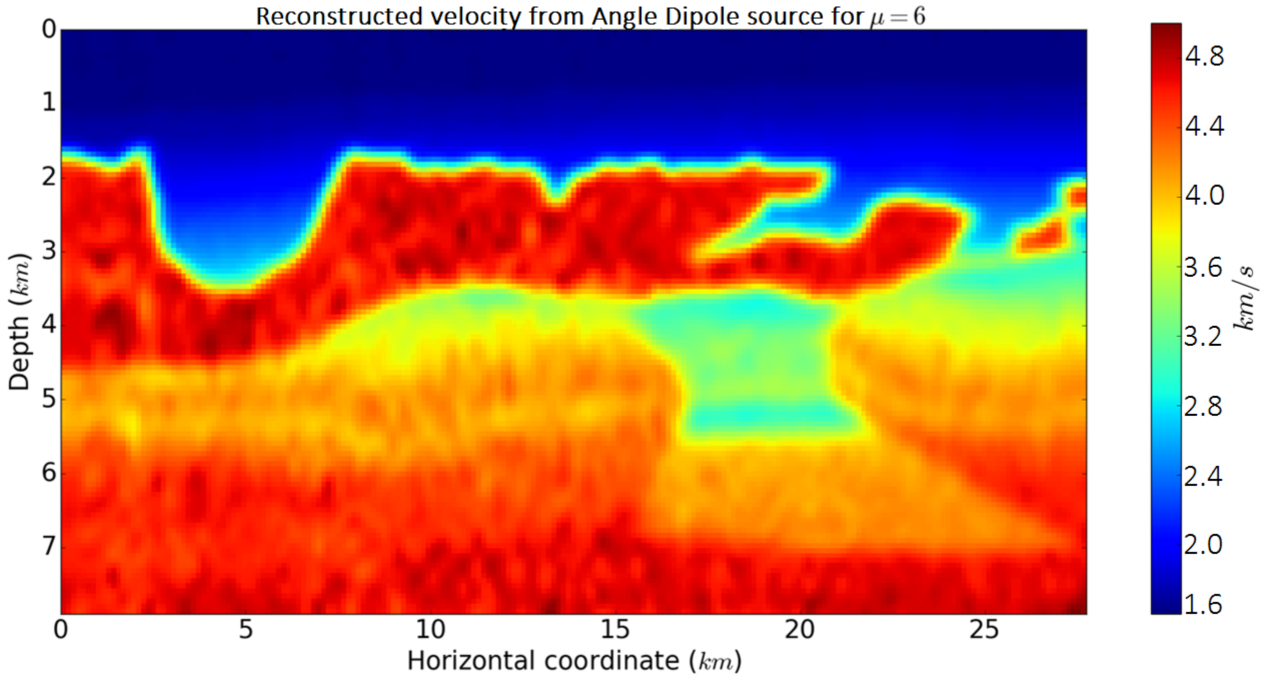


Figure 3.18: Reconstruction of a sub-sampled BP velocity model by using angle dipole source without regularization (top) and with regularization for $\mu = 6$ (bottom).

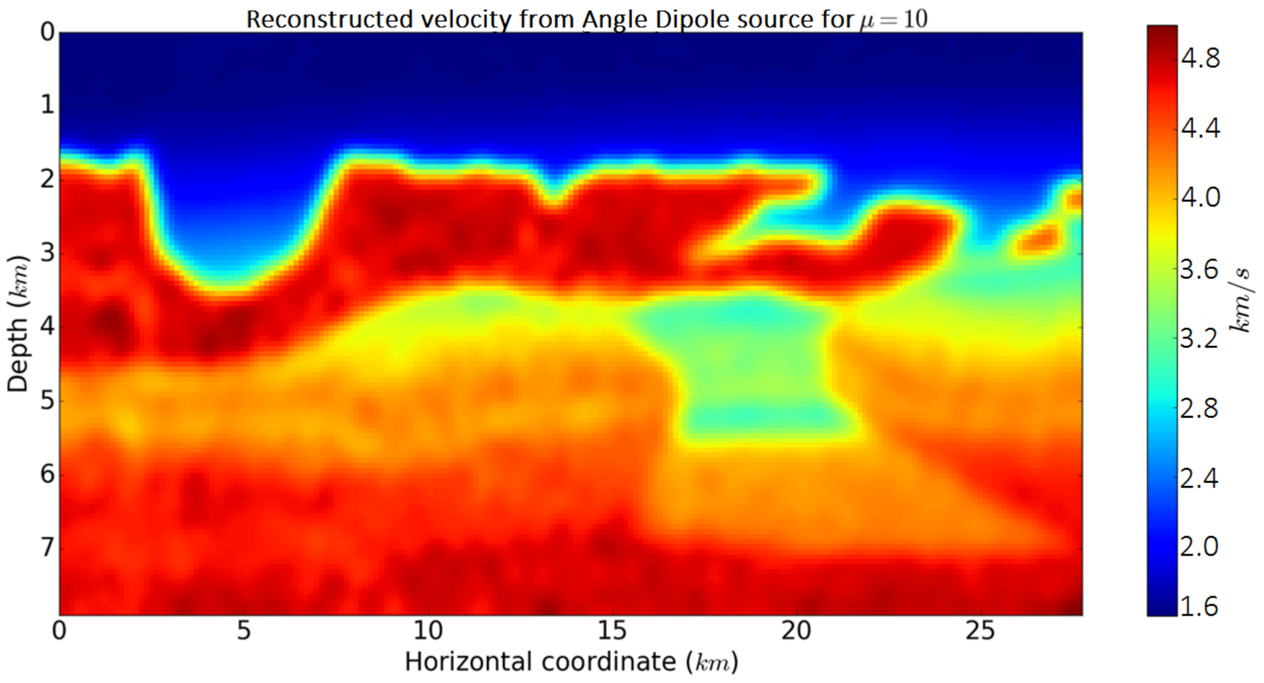


Figure 3.19: Reconstruction of a sub-sampled BP velocity model by using one angle dipole source with regularization ($\mu = 10$).

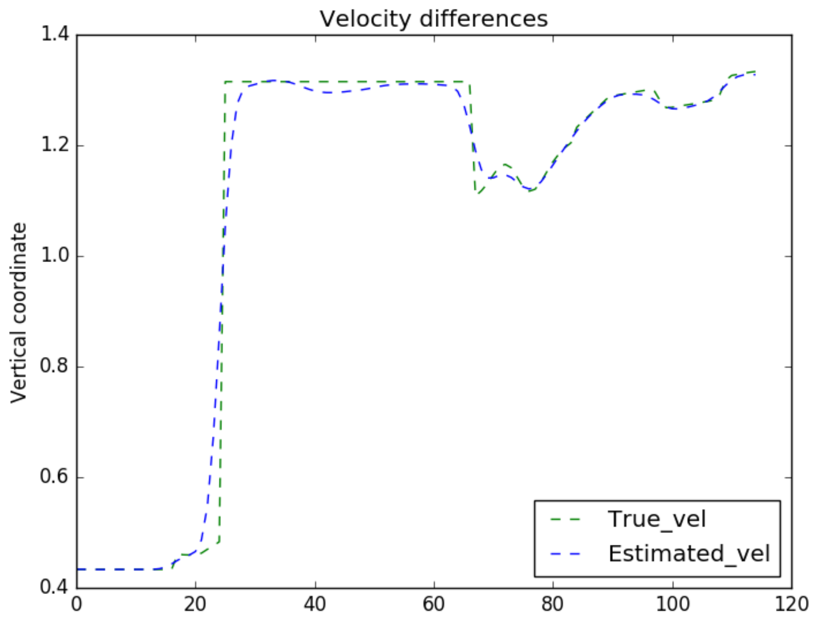


Figure 3.20: The velocity difference between true and estimated velocities of monopole pressure source.

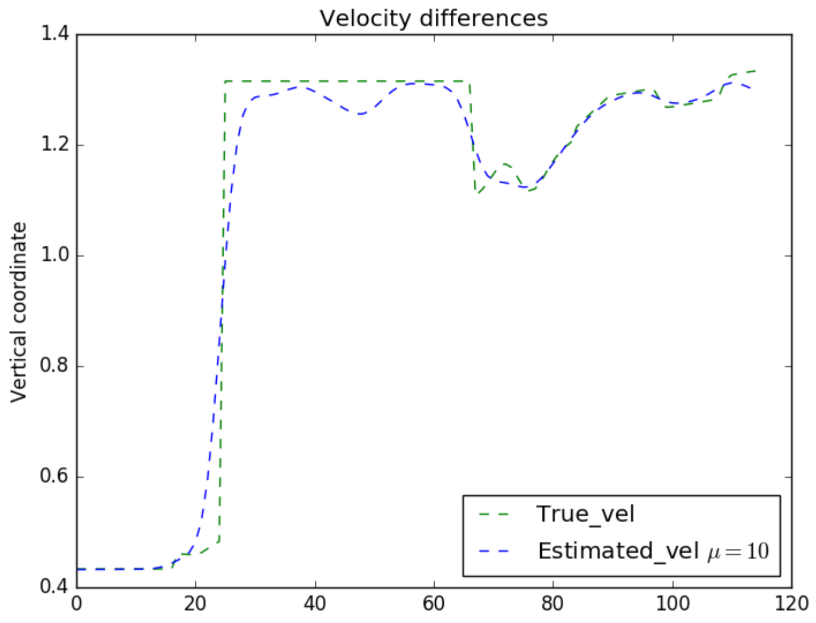


Figure 3.21: The velocity difference between true and estimated velocities of angle dipole source.

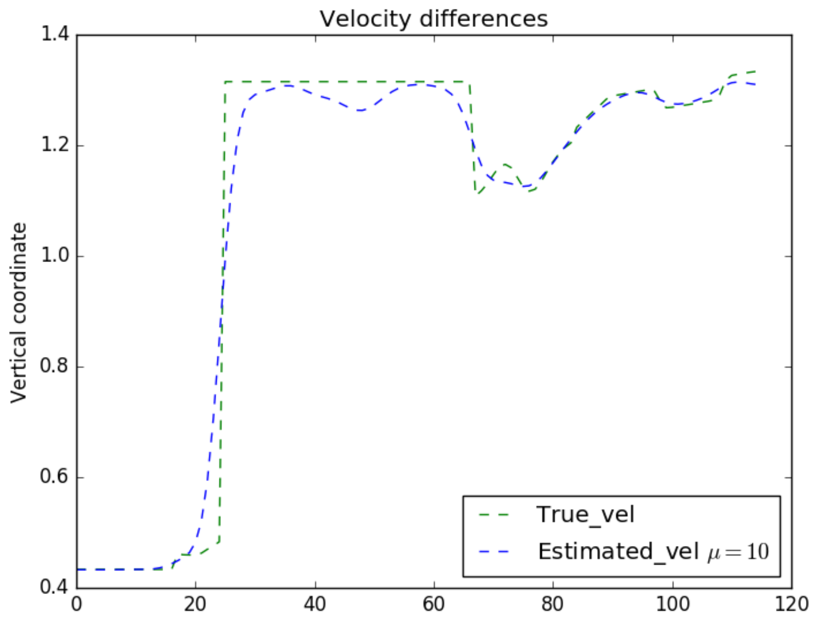


Figure 3.22: The velocity difference between true and estimated velocities of vertical dipole source.

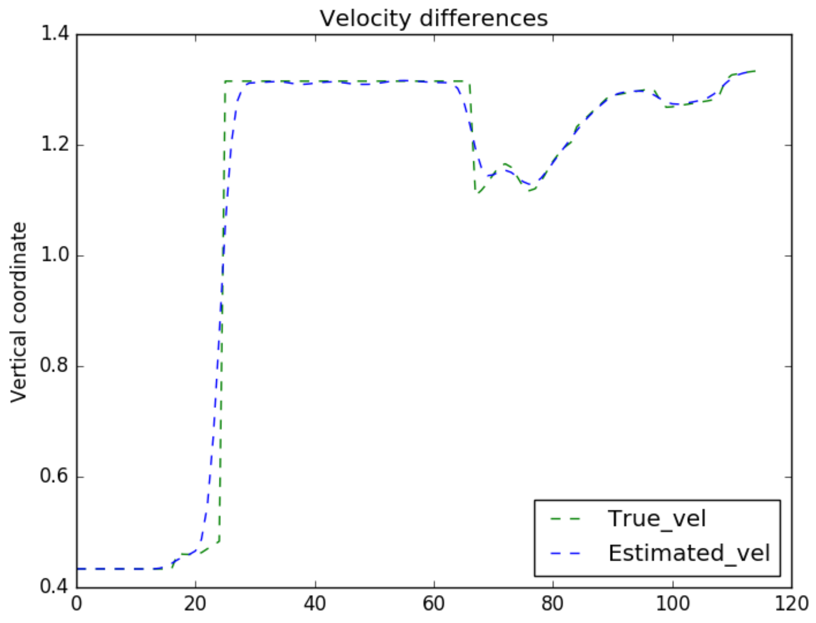


Figure 3.23: The velocity difference between true and estimated velocities of horizontal dipole source.

Estimated Velocity Error for the BP Velocity Model	
Source type	**VFWI**
Monopole Pressure	0.052
Vertical Dipole	0.055
Horizontal Dipole	0.050
Angle Dipole	0.055

Table 3.2: *RMS* velocities of different source types using VFWI algorithm for the BP model.

Chapter 4

Conclusions and future work

In this thesis we have extended full waveform inversion to vector data. In the methodology Chapter we introduced our VFWI algorithm in detail followed by derivation of first order adjoint-state method. In Chapter 3 we demonstrated our algorithm by implementing the codes into PySIT package so as to obtain our results. We have investigated four different velocity models: two isolated perturbations, horizontal reflector, Marmousi and BP.

In the case of horizontal reflector and the Marmousi models horizontal dipole source shows the strong directivity of the wavefields. The directivity of these models can be mitigated by including more sources and receivers in the calculation. We use a single source/receiver pair here to highlight the differences in illumination between the different source types.

As for the image resolution, in the two isolated perturbations case, vertical and angle dipole sources generate the highest resolution image and the true model is reconstructed very well. In the horizontal reflector example, however, as can be seen in Figure 3.6, the most accurate recovered model is that made with the monopole pressure source.

In the third example, our recovered Marmousi model (in the case of using standard FWI) has an *RMS* velocity error that is less than the *RMS* velocity error of our VFWI algorithm. This difference becomes more important when we compare the error associated with usual seismic source with horizontal dipole source as shown in Table 3.1. In addition, the monopole pressure source provides better lateral resolution as shown in Figure 3.10. One reason might be in regular FWI we use the standard Ricker wavelet for the source signature which transmits uniform energy across our computational domain. Whereas in VFWI the source signatures are no longer Ricker wavelet to ensure directionality of wavefields. Therefore, the transmitted energy is not as uniform as FWI case.

The last example was the reconstruction of the BP velocity model. In this example we needed to define a smoothing operator in order to deal with the true model. Since the BP model is so large and computationally expensive, first we had to subsample the true model in such a way that only 12 percent of sample remaining. The BP model recovery is almost impossible unless we start with an appropriate initial model. Therefore by using our smoothing operator we could obtain a suitable starting model. Recording VA data in the case of using horizontal dipole sources provided better sides and edges recovery. In the BP case, we used double the number of sources and receivers all across the surface to compensate the poor recovery in the middle areas of our model.

Another issue which we resolved was the presence of artefacts across our recovered model in the case of using angle and vertical dipole sources. In order to overcome this issue, we used a regularization term with the definition of the gradient of our model and a variable to control over its weight. The regularization process successfully mitigates this problem.

VFWI is a novel imaging technique that allows one to obtain directivity information from the wave fields about the subsurface scatterers. Moreover, by introducing dipole sources, one can create source radiation patters that better illuminate specific parts of the model that might be of interest to the researcher. The effect of the source radiation patters on the model recovery was demonstrated in the synthetic examples in Chapter 3. We discovered that some source types work better for some models, and not so well for other models, for example, the horizontal dipole sources in the case of Marmousi and BP models. In some cases, especially for the BP model, we obtained a lot of oscillatory artefacts with the vertical and angle dipole sources that we were able to mitigate using regularization. More experiments need to be performed in order to determine the most appropriate source type for a particular model.

Other possible future research directions include:

- applying different kinds of regularization, for example TV that might allow us to preserve sharper boundaries of the homogeneous salt bodies in the regularized recovery of models such as BP with vertical and angle dipole sources;
- Investigating how combining different source types in one experiment could lead to higher resolution in the recovered models and better cancellation of artifacts;
- Extending the method to the variable density acoustic case with appropriate receiver weighting.

Bibliography

[1] *Ö*zdogan Yilmaz and Stephen M. Doherty, Seismic data analysis: Prcessing, inversion and interpretation of seismic data, V. 1, Society of Exploration Geophysicists, (2001).

[2] F. Jones and D. Oldenberg, Inversion theory, Course notes, (2007).

[3] J. Virieux and S. Operto, An overview of full waveform inversion in exploration geophysics, Geophysics, 74(6), WCC1-WCC26, (2009).

[4] Reed, M. and Simon, B., Functional Analysis, Elsevier, (2003).

[5] Strauss, W. A., Partial Differential Equations: An Introduction, Wiley, (2007).

[6] Collino, F. and Tsogka. C, Application of the perfectly matched absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media, Geophysics, 66(1), 294-307, (2001).

[7] Yonina C. Eldar and Gitta Kutyniok, Compressive Sensing: Theory and Applications, Cambridge University Press, (2012).

[8] Baysal, E., D. D. Kosloff, and J. W. C. Sherwood, Reverse time migration: Geophysics, 48, 1514-1524, (1983).

[9] Hyunggu Jun, Youngseo Kim, Jungkyun Shin, Changsoo Shin and Dong-Joo Min, Laplace-Fourier-domain elastic full-waveform inversion using time-domain modeling, GEOPHYSICS, VOL. 79, NO. 5, (2014).

[10] Pratt, R. G., Seismic waveform inversion the frequency domain, Part 1: Theory and verification in a physical scale model. Geophysics 64, 888-910, (1999).

[11] Pratt, R., Shin, C., and Hicks, G., Gauss-Newton and full newton methods in frequency-space seis- mic waveform inversion: Geophysical Journal International, 133, no. 2, 341–362, (1998).

[12] Sirgue, L., Pratt, R. G., Efficient waveform inversion and imaging : A strategy for selecting temporal frequencies, (2004).

[13] Tristan Van Leeuwen and Felix J. Herrmann, Mitigating local minima in full-waveform inversion by expanding the search space, GEOPHYSICS, J. Int, (2013).

[14] Tarantola, A. and Valette, A., Generalized nonlinear inverse problems solved using the least squares criterion, Reviews of Geophysics and Space Physics, (1982).

[15] Symes, W. W., Migration velocity analysis and waveform inversion, Geophysical Prospecting, 56(6), 765–790. (2008).

[16] Virieux, J., and S. Operto, An overview of full-waveform inversion in exploration geophysics: Geophysics, 74, no. 6, WCC1–WCC26, (2009).

[17] Mora, P. R., Nonlinear two-dimensional elastic inversion of multioffset seismic data: Geophysics, 52, 1211–1228 (1987).

[18] Bunks, C., F. M. Saleck, S. Zaleski, and G. Chavent, Multiscale seismic waveform inversion: Geophysics, 60, 1457–1473,(1995).

[19] Pratt, R. G., C. Shin, and G. J. Hicks, Gauss-Newton and full newton methods in frequency-space seismic waveform inversion: Geophysical Journal International, 133, 341–362, (1998).

[20] Plessix, R., A review of the adjoint-state method for computing the gradient of a functional with geophysical applications: Geophysical Journal International, 167, 495–503, (2006).

[21] Operto, S., Y. Gholami, R. Brossier, L. Metivier, V. Prieux, A. Ribodetti, and J. Virieux, A guided tour of multiparameter full-waveform inversion with multicomponent data: From theory to practice: The Leading Edge, 32, 1040–1054, (2013).

[22] Operto, S., J. Virieux, P. Amestoy, J. L'Excellent, L. Giraud, and H. Ben-Hadj-Ali,3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study: Geophysics, 72, no. 5, SM195–SM211, (2007).

[23] Kim, Y., C. Shin, H. Calandra, and Min D-J, An algorithm for 3D acoustic time-Laplace-Fourier-domain hybrid full-waveform inversion: Geophysics, 78, no. 4, R151–R166, (2013).

[24] Davis, T., UMFPACK version 5: University of Florida. Gardner, G. H. F., L.W. Gardner, and A. R. Gregory, 1974, Formation velocity and density— The diagnostic basics for stratigraphic traps: Geophysics, 39, 770–780, (2006).

[25] Trefethen, L. L., and D. Bau, Numerical linear algebra: SIAM. Virieux, J., 1986, P-sv wave propagation in heterogeneous media: Velocitystress finite-difference method: Geophysics, 51, 889–901, (1997).

[26] Kim, J. H., and S. J. Kim, A multifrontal solver combined with graph partitioner: The American Institute of Aeronautics and Astronautics (AIAA) Journal, 37, 964–970, (1999).

[27] Jan Thorbecke, 2D Finite-Difference Wavefield Modelling, (2015).

[28] Guitton, A., andW.W. Symes, Robust inversion of seismic data using the Huber norm: Geophysics, 68, 1310–1319, (2003).

[29] Huber, P. J., Robust regression: Asymptotics, conjectures, and Monte Carlo: Annals of Statistics, 1, 799–821, (1973).

[30] Shin, C., and D. Min, Waveform inversion using a logarithmic wavefield: Geophysics, 71, no. 3, R31–R42, (2006).

[31] Clement Fleury and Ivan Vasconcelos, Adjoint-state reverse time migration of 4C data:Finite-frequency map migration for marine seismic image: Geophysics, 78, no. 2, WA159-WA172, (2013).

[32] Meier, M. A., R. E. Duren, K. T. Lewallen, J. Otero, S. Heiney, and T. Murray., A marine dipole source for low frequency seismic acquisition, SEG Technical Program Expanded Abstracts, Society of Exploration Geophysicists, 176–180, (2015).

[33] De Hoop, AT, Handbook of radiation and scattering of waves: Acoustic waves in fluids, elastic waves in solids, electromagnetic waves: with corrections, Former publisher: Academic Press, (2017).

[34] Allaire, Grégoire, Numerical analysis and optimization: an introduction to mathematical modelling and numerical simulation, Oxford University Press, (2007).

[35] Tarantola, A., Inversion of seismic reflection data in the acoustic approximation: Geophysics, 49, 1259–1266, (1984).

[36] Symes,W., The seismic reflection inverse problem: Inverse Problems, 25, 123008, (2009).

[37] Haber, E., Computational Methods in Geophysical Electromagnetics, University of British Columbia, Vancouver, British Columbia, Canada, (2015).

[38] Sun, D. and Jiao, K. and Vigh, D., Compensating for source and receiver ghost effects in full waveform inversion and reverse time migration for marine streamer data, V. 201, 3, Journal of Geophysical Research, 1507-1521, (2015).

[39] Louboutin, Mathias and Herrmann, Felix J, Time compressively sampled full-waveform inversion with stochastic optimization, SEG Technical Program Expanded Abstracts 2015, Society of Exploration Geophysicists, 5153-5157, (2015).

[40] Robertsson, J. O. A. and Moore, I. and Vassallo, M. and Kemal, Van Manen, D.-J. and Ozbek, A., On the use of multicomponent streamer recordings for reconstruction of pressure wavefields in the crossline direction, V. 73, 5, Journal of Geophysical Research, A45–A49, (2008).

[41] Carlson, D. H., A. Long, W. Sollner, H. Tabti, R. Tenghamn, and N. Lunde, Increased resolution and penetration from a towed dual-sensor streamer: First Break, 25, 71–77, (2007).

[42] Tenghamn, R., S. Vaage, and C. Borresen, A dual-sensor towed marine streamer: Its viable implementation and initial results: 77th Annual International Meeting, SEG, Expanded Abstracts, 989–993, (2007).

[43] Cambois, G., D. Carlson, C. Jones, M. Lesnes, W. Sollner, and H. Tabti, Dual-sensor streamer data: Calibration, acquisition QC and attenuation of seismic interferences and other noises: 79th Annual International Meeting, SEG, Expanded Abstracts, 142–146, (2009).

[44] Vassallo, M., A. Ozbek, K. Ozdemir, and K. Eggenberger, Crossline wavefield reconstruction from multicomponent streamer data: Part 1 Multichannel interpolation by matching pursuit (MIMAP) using pressure and its crossline gradient: Geophysics, 75, no. 6, WB53–WB67, (2010).

[45] Ozbek, A., M. Vassallo, D. J. van Manen, and K. Eggenberger, Crossline wavefield reconstruction from multicomponent streamer data: Part 2 Joint interpolation and 3D up/down separation by generalized matching pursuit: Geophysics, 75, no. 6, WB69–WB85, (2010).

[46] Frijlink, M., R. van Borselen, and W. Sollner, The free surface assumption for marine data-driven demultiple methods: Geophysical Prospecting, 59, 269–278, (2011).

[47] Brandsberg-Dahl S., and De Hoop, M. V., Velocity analysis in the common scattering-angle/azimuth domain, 51, Geophysical Prospecting, 295–314, (2003).

[48] Sava, P., and Biondi, B., Wave-equation migration velocity analysis, Geophysical Prospecting, (2004).

[49] Brougois, A and Bourget, Marielle and Lailly, Patriek and Poulet, Michel and Ricarte, Patrice and Versteeg, Roelof, Marmousi, model and data, EAEG Workshop-Practical Aspects of Seismic Data Inversion, 1990.

[50] Billette, FJ and Brandsberg-Dahl, Sverre, The 2004 BP velocity benchmark, 67th EAGE Conference & Exhibition, 2005.

[51] Biondi, B., Almomin, Ali, Tomographic Full Waveform Inversion (TFWI) by Extending the Velocity Model Along the Time-Lag Axis, Society of Exploration Geophysicists, SEG Annual Meeting, 22-27 September, Houston, Texas, (2013).

[52] Fichtner, A., Full Seismic Waveform Modelling and Inversion, Springer, Heidelberg Dordrecht London New York, (2011).

[53] Nocedal, J. and Wright, S. J., Numerical Optimization, Springer, Library of Congress Control Number, (2006).

[54] Boyd, Stephen and Vandenberghe, Lieven, Convex optimization, Cambridge university press, (2004).

[55] Gary Margrave, Matt Yedlin and Kris Innanen, Full waveform inversion and the inverse hessian, 23, CREWES Research Report, (2011).

[56] Russell J. Hewett and Laurent Demanet, an open source toolbox for seismic inversion and seismic imaging, Imaging and Computing Group in the Department of Mathematics at MIT, http://pysit.bitbucket.org/.

Appendix A

Some derivations

In this appendix I derive equation (2.15) that we used in chapter 2 of my thesis.

A.1 Derivation of equation 2.15

In order to prove formula (2.15), we use the following relationship:

$$\delta \mathbf{u}_{q,\mathbf{f}} = \nabla_m \mathbf{u}_{q,\mathbf{f}} \; \delta m, \tag{A.1}$$

which means that $\delta \mathbf{u}_{q,\mathbf{f}}$ is the linear differential of $\mathbf{u}_{q,\mathbf{f}}$ with respect to m and $\nabla_m \mathbf{u}_{q,\mathbf{f}}$ is the Jacobian. Then we can write:

$$\begin{aligned}\nabla_m \mathcal{J}(m)\delta m &= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_0^T \int_G \left[\mathcal{J}_1(m + \epsilon \delta m) - \mathcal{J}_1(m)\right] d\mathbf{x}\; dt = \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \int_0^T \int_G \left[\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m + \epsilon \delta m)) - \mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\right] d\mathbf{x}\; dt\end{aligned} \tag{A.2}$$

Then, expanding $\mathbf{u}_{q,\mathbf{f}}(m + \epsilon \delta m)$ around m, we obtain:

$$\mathbf{u}_{q,\mathbf{f}}(m + \epsilon \delta m) = \mathbf{u}_{q,\mathbf{f}}(m) + \nabla_m \mathbf{u}_{q,\mathbf{f}}(m)\; \epsilon \delta m + O(\epsilon^2) = \mathbf{u}_{q,\mathbf{f}}(m) + \epsilon \delta \mathbf{u}_{q,\mathbf{f}}(m) + O(\epsilon^2), \tag{A.3}$$

where we used (2.15) so that

$$\nabla_m \mathcal{J}(m)\delta m = \lim_{\epsilon\to 0}\frac{1}{\epsilon}\int_0^T\int_G \Big[\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m)+\epsilon\delta\mathbf{u}_{q,\mathbf{f}}(m)+O(\epsilon^2)) - \mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\Big]\, d\mathbf{x}\; dt. \tag{A.4}$$

In this last equation we expand $\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m)+\epsilon\delta\mathbf{u}_{q,\mathbf{f}}(m)+O(\epsilon^2))$ around $\mathbf{u}_{q,\mathbf{f}}(m)$:

$$\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m)+\epsilon\delta\mathbf{u}_{q,\mathbf{f}}(m)+O(\epsilon^2)) = \mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))+\epsilon\nabla_{\mathbf{u}_{q,\mathbf{f}}}\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\cdot\delta\mathbf{u}_{q,\mathbf{f}}(m)+O(\epsilon^2), \tag{A.5}$$

so that

$$\begin{aligned}\nabla_m \mathcal{J}(m)\delta m &= \lim_{\epsilon\to 0}\frac{1}{\epsilon}\int_0^T\int_G \Big[\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m)) + \epsilon\nabla_{\mathbf{u}_{q,\mathbf{f}}}\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\cdot\delta\mathbf{u}_{q,\mathbf{f}}(m)+\\ &+ O(\epsilon^2) - \mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\Big]d\mathbf{x}\; dt =\\ &= \lim_{\epsilon\to 0}\frac{1}{\epsilon}\int_0^T\int_G \Big[\epsilon\nabla_{\mathbf{u}_{q,\mathbf{f}}}\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\cdot\delta\mathbf{u}_{q,\mathbf{f}}(m)+O(\epsilon^2)\Big]\, d\mathbf{x}\; dt =\\ &= \int_0^T\int_G \nabla_{\mathbf{u}_{q,\mathbf{f}}}\mathcal{J}_1(\mathbf{u}_{q,\mathbf{f}}(m))\cdot\delta\mathbf{u}_{q,\mathbf{f}}(m) d\mathbf{x}\; dt = \langle\nabla_{\mathbf{u_{q,f}}}\mathcal{J}_1(m), \delta\mathbf{u_{q,f}}\rangle,\end{aligned} \tag{A.6}$$

which completes the proof.

Appendix B

Coding Description

In this chapter we briefly describe our coding. We used PySIT package [56] and develop it for our own purpose. PySIT is a python package for seismic inversion and imaging especially designed for FWI. Our contribution to PySIT consists of **forward problem and discretization**, **inverse problem and joint objective function**, **regularization** and **optimization**.

B.1 Forward Problem and Discretization

In this section we describe our forward problem and discretization. The forward problem is a parameter identification problem since in our case, it is a Partial Differential Equation (PDE) and the data is the solution of that PDE. As we explained before in Chapter 2, in order to make sure that we have vector data rather than only scalar data in acoustic FWI, we need to have dipole sources to generate vector data and then record them by the receivers. In forward problem first we need to discretize the wave equation. In our vector-acoustic equation (2.5), we assume that density is constant which is a similar assumption in a standard acoustic wave equation:

$$\Big(m(z,x)\,\frac{\partial^2}{\partial t^2} - \nabla^2_{z,x}\Big)\,u_s(t,z,x) = s(t,z,x), \tag{B.1}$$

where $\nabla^2_{z,x}$ is a two dimensional Laplacian operator and $u_s(t,z,x)$ is a scalar wavefield generated by a conventional seismic source $s(t,z,x)$, i.e. equation (2.42). The solver for equation B.1 was already existed in PySIT [56], so we used the same two dimensional constant density solver for our problem. However, vector-acoustic equation is different in the sense that we needed to also compute gradient of pressure and discretize the right-hand-side of equation, i.e. different sources and data (as we explained in Chapter 2). All of them had to be implemented in PySIT so as to have a working forward problem. For example, in order to compute gradient of pressure for different components we implemented a staggered grid or stencil using finite difference method. In order to discretize our forward problem we assume that the domain in $\mathbb{R}^2$, is divided into n^2 voxels of size h. If we consider u as a general wavefield, then the derivative of u in the x direction can be written as

$$\partial^h_x u(x_i + \frac{h}{2}, z_j) \approx \frac{1}{h}(u(x_{i+1}, z_j) - u(x_i, z_j)).$$

We can assume that D is the 1D derivative matrix and U is the 2D wavefield ordered as a matrix

$$D_x \approx \frac{1}{h}\begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}.$$

So, the 1D derivative of U is DUI, where I is identity matrix. Similarly, in vertical direction z we have

$$D_y \approx \frac{1}{h}\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix},$$

which leads to IUD^{T}. Finally by using Kronecker product we can construct gradient operator ∇ as

$$\nabla = \begin{bmatrix} \partial_x \\ \partial_z \end{bmatrix} \simeq \begin{bmatrix} I \otimes D_x \\ D_z \otimes I \end{bmatrix}.$$

It is easy to verify that divergence operator can be written as

$$\nabla\cdot = -\nabla^{\mathsf{T}}.$$

Going to the higher order, a second order finite difference in 1D of the second derivative can be written as

$$\frac{\partial^2}{\partial x^2}u(x_i, z_j) = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h} + \mathcal{O}(h^2).$$

Similarly, second order derivatives are used in the z direction. So, the finite difference matrix for the Laplacian is

$$\nabla_h^2 = D_2 \otimes I_n + I_n \otimes D_2,$$

where, assuming Neumann Boundary conditions,

$$D_2 = \frac{1}{h_2}\begin{bmatrix} -1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & & & & \vdots \\ 0 & 1 & -2 & 1 & 0 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 0 & 1 & -2 & 1 & 0 \\ \vdots & & & & 0 & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1 \end{bmatrix}.$$

Using these operators, we can compute gradient or laplacian of our parameters and wavefields. Setting up the right-hand-side (sources), finally we can solve and visualize the results i. e. generated data. We already showed the generated and recorded data in Chapter 2.

B.2 Forward Modelling Python Code for Horizontal Reflector

Here we show one of our forward modelling codes for horizontal reflector model.

```

from __future__ import absolute_import

import numpy as np
import scipy as sp
from scipy.sparse import spdiags
import scipy.sparse as spsp
import matplotlib.pyplot as cm
import scipy.io
import scipy.sparse as spsp

from my_extensions.my_sources import *
from my_extensions.convenient_plot_functions import *
from my_extensions.ximage import *
from solvers.wavefield_vector import *
from pysit.solvers.constant_density_acoustic.time.scalar.
    constant_density_acoustic_time_scalar_base import *
from pysit.solvers.constant_density_acoustic.time.scalar.
    constant_density_acoustic_time_scalar_2D import
    ConstantDensityAcousticTimeScalar_2D_cpp
from my_util import Bunch
from my_util import PositiveEvenIntegers
from my_util.derivatives import build_derivative_matrix
from my_util.matrix_helpers import build_sigma, make_diag_mtx
from my_util.solvers import inherit_dict


__all__ = ['HorizontalReflectorModel',
  'horizontal_reflector_horizontal_x_vector']

def _gaussian_derivative_pulse(XX, threshold, **kwargs):
    """ Derivative of a Gaussian at a specific sigma """
    T = -100.0*XX*np.exp(-(XX**2)/(1e-4))
    T[np.where(abs(T) < threshold)] = 0
    return T

def _gaussian_pulse(XX, threshold, sigma_in_pizels=1.0, **kwargs):
    """ Gaussian function, in X direction, with sigma specified in terms
         of pizels """
    xdelta = XX[np.where((XX-XX.min) != 0.0)].min() - XX.min()
    sigma = sigma_in_pizels*xdelta
    T = np.exp(-(XX**2) / (2*sigma**2)) / (sigma*np.sqrt(2*np.pi))
    T = T * xdelta
    T[np.where(abs(T) < threshold)] = 0
    return T

_pulse_functions = { 'gaussian_derivative' : _gaussian_derivative_pulse,
                     'gaussian' : _gaussian_pulse
                   }


class HorizontalReflectorModel(GeneratedGalleryModel):

    """ Gallery model for constant background plus simple horizontal
        reflectors. """

    model_name = "Horizontal Reflector"

    valid_dimensions = (1,2,3)

    @property
    def dimension(self):
        return self.domain.dim

    supported_physics = ('acoustic',)

    def __init__(self, mesh,
                         reflector_depth=[0.45, 0.65], # as percentage of
                             domain
                         reflector_scaling=[1.0, 1.0],
                         background_velocity=1.0,
                         drop_threshold=1e-7,
                         pulse_style='gaussian_derivative',
                         pulse_config={},
                         ):
        """ Constructor for a constant background model with horizontal
            reflectors.

        Parameters
        ----------
        mesh : mesh
            Computational mesh on which to construct the model
        reflector_depth : list
            Depths of the reflectors, as a percentage of domain depth
        reflector_scaling : list
            Scale factors for reflectors
        background_velocity : float
        drop_threshold : float
            Cutoff value for evaluation of reflectors
        pulse_style : {'gaussian_derivative', 'gaussian_pulse'}
            Shape of the reflector
        pulse_config : dict
            Configuration of the pulses.



        GeneratedGalleryModel.__init__(self)


        self.reflector_depth = reflector_depth
        self.reflector_scaling = reflector_scaling

        self.background_velocity = background_velocity

        self.drop_threshold = drop_threshold

        self.pulse_style = pulse_style
        self.pulse_config = pulse_config

        self._mesh = mesh
        self._domain = mesh.domain
        # Set _initial_model and _true_model
        self.rebuild_models()

    def rebuild_models(self, reflector_depth=None, reflector_scaling=
        None, background_velocity=None):
        """ Rebuild the true and initial models based on the current
            configuration."""

        if reflector_depth is not None:
            self.reflector_depth = reflector_depth

        if reflector_scaling is not None:
            self.reflector_scaling = reflector_scaling

        if background_velocity is not None:
            self.background_velocity = background_velocity

        C0 = self.background_velocity*np.ones(self._mesh.shape())

        dC = self._build_reflectors()

        self._initial_model = C0
        self._true_model = C0 + dC

    def _build_reflectors(self):

        mesh = self.mesh
        domain = self.domain

        grid = mesh.mesh_coords()
        XX = grid[-1]

        dC = np.zeros(mesh.shape())

        # can set any defaults here
        if self.pulse_style == 'gaussian_derivative':
            pulse_config = {}
        elif self.pulse_style == 'gaussian':
            pulse_config = {}

        # update to any user defined defaults
        pulse_config.update(self.pulse_config)

        for d,s in zip(self.reflector_depth, self.reflector_scaling):

            # depth is a percentage of the length
            depth  = domain.x.lbound + d * domain.x.length

            pulse = _pulse_functions[self.pulse_style](XX-depth, self.
                drop_threshold, **pulse_config)
            dC += s*pulse

        return dC

def horizontal_reflector_horizontal_x_vector( mesh, **kwargs):
    """ Friendly wrapper for instantiating the horizontal reflector
        model. """

    # Setup the defaults
    model_config = dict(reflector_depth=[0.45, 0.65], # as percentage of
         domain
                        reflector_scaling=[1.0, 1.0],
                        background_velocity=1.0,
                        drop_threshold=1e-7,
                        pulse_style='gaussian_derivative',
                        pulse_config={},)

    # Make any changes
    model_config.update(kwargs)

    return HorizontalReflectorModel(mesh, **model_config).get_setup()

class PML(Domain):
   """ Perfectly Matched Layer (PML) domain boundary condition.

   def __init__(self, length, amplitude, ftype='quadratic', boundary='
       dirichlet'):
      # Length is currently in physical units.
      self.length = length

      self.amplitude = amplitude

      # Function is the PML function
```

```
        self.ftype = ftype

        if (ftype == 'b-spline'):

            self.pml_func = self._bspline
        elif (ftype == 'quadratic'):
            self.pml_func = self._quadratic
        else:
            raise NotImplementedError(.format(ftype))

        if boundary in ['neumann', 'dirichlet']:
            self.boundary_type = boundary
        else:
            raise ValueError("'{0}' is not 'neumann' or 'dirichlet'.".
                format(boundary))

    def _bspline(self, x):
        x = np.array(x*1.0)
        if (x.shape == () ):
            x.shape = (1,)

        retvec = np.zeros_like(x)

        loc = np.where(x < 0.5)
        retvec[loc] = 1.5 * (8./6.) * x[loc]**3

        loc = np.where(x >= 0.5)
        retvec[loc] = 1.5 * ((-4.0*x[loc]**3 + 8.0*x[loc]**2 - 4.0*x[loc]
            + 2.0/3.0))

        return retvec

    def _quadratic(self, x):
        return x**2

    def evaluate(self, n, orientation='right'):
        """Evaluates the PML profile function on `n` points over the range
            [0,1].

        val = self.amplitude * self.pml_func(np.linspace(0., 1., n,
            endpoint=False))
        if orientation is 'left':
            val = val[::-1]

        return val

class CartesianMesh(StructuredMesh):

    @property
    def type(self): return 'structured-cartesian'

    def __init__(self, domain, *args):

        StructuredMesh.__init__(self, domain, *args)

        self.parameters = dict()

        for (i,k) in _cart_keys[self.dim]:

            n = int(args[i])
            delta = domain.parameters[i].length / n

            param = Bunch(n=n, delta=delta)

            param.lbc = MeshBC(self, domain.parameters[i].lbc, i, 'left',
                delta)
            param.rbc = MeshBC(self, domain.parameters[i].rbc, i, 'right',
                delta)

            self.parameters[i] = param # d.dim[-1]
            self.parameters[k] = param # d.dim['z']
            self.__setattr__(k, param) # d.z

        self._shapes = dict()
        self._dofs = dict()
        self._slices = dict()

        self._spgrid = None
        self._spgrid_bc = None

    def nodes(self, include_bc=False):

        return np.hstack(self.mesh_coords())

    def mesh_coords(self, sparse=False, include_bc=False):

        sphash = lambda g: reduce(lambda x,y:x+y,map(lambda x: x.hexdigest
            (), map(hashlib.sha1, g)))

        if sparse:
            if (self._spgrid is not None) and (self._spgrid_hash == sphash(
                self._spgrid)):
                return self._spgrid
            if include_bc and (self._spgrid_bc is not None) and (self.
                _spgrid_bc_hash == sphash(self._spgrid_bc)):
                return self._spgrid_bc

        if include_bc:
            assemble_grid_row = lambda dim: np.linspace(self.domain.
                parameters[dim].lbound+self.parameters[dim].lbc.n*self.
                domain.parameters[dim].delta,
                                                                self.domain.
                                                                    parameters[dim].
                                                                    rbound+self.
                                                                    parameters[dim].
                                                                    rbc.n*self.domain
                                                                    .parameters[dim].
                                                                    delta,
                                                                self.parameters[dim
                                                                    ].n+self.
                                                                    parameters[dim].
                                                                    lbc.n+self.
                                                                    parameters[dim].
                                                                    rbc.n,
                                                                endpoint=False)
        else:
            assemble_grid_row = lambda dim: np.linspace(self.domain.
                parameters[dim].lbound, self.domain.parameters[dim].rbound,
                self.parameters[dim].n, endpoint=False)

        if(self.dim == 1):

            tup = tuple([assemble_grid_row('z')])
        elif(self.dim == 2):
            xrow = assemble_grid_row('x')
            zrow = assemble_grid_row('z')
            tup = np.meshgrid(xrow, zrow, sparse=sparse, indexing = 'ij' )
        else:
            xrow = assemble_grid_row('x')
            yrow = assemble_grid_row('y')
            zrow = assemble_grid_row('z')
            tup = np.meshgrid(xrow, yrow, zrow, sparse=sparse, indexing = 'ij' )

        if sparse:
            if not include_bc and self._spgrid is None:
                self._spgrid = tup
                self._spgrid_hash = sphash(tup)
            if include_bc and self._spgrid_bc is None:
                self._spgrid_bc = tup
                self._spgrid_bc_hash = sphash(tup)

        if sparse:
            return tup
        else:
            return tuple([x.reshape(self.shape(include_bc)) for x in tup])

    def get_deltas(self):
        return tuple([self.parameters[i].delta for i in xrange(self.dim)])

    deltas = property(get_deltas, None, None, "Tuple of grid deltas")

    def _compute_shape(self, include_bc):

        sh = []
        for i in xrange(self.dim):
            p = self.parameters[i]

            n = p.n
            if include_bc:
                n += p.lbc.n
                n += p.rbc.n

            sh.append(n)

        # pml, ghost_padding, as_grid
        self._shapes[include_bc, True] = sh
        self._shapes[include_bc, False] = (int(np.prod(np.array(sh))), 1)
        self._dofs[include_bc] = int(np.prod(np.array(sh)))

    def shape(self, include_bc=False, as_grid=False):

        if (include_bc, as_grid) not in self._shapes:
            self._compute_shape(include_bc)
        return self._shapes[(include_bc, as_grid)]

    def dof(self, include_bc=False):
        if include_bc not in self._dofs:
            self._compute_shape(include_bc)
        return self._dofs[include_bc]

    def unpad_array(self, in_array, copy=False):

        sh_unpadded_grid = self.shape(include_bc=False, as_grid=True)
        sh_unpadded_dof = self.shape(include_bc=False, as_grid=False)

        if in_array.shape == sh_unpadded_grid or in_array.shape == sh_unpadded_dof:
            out_array = in_array
        else:
            sh_grid = self.shape(include_bc=True, as_grid=True)

            sl = list()
            for i in xrange(self.dim):
                p = self.parameters[i]

                nleft  = p.lbc.n
                nright = p.rbc.n

                sl.append(slice(nleft, sh_grid[i]-nright))

            out_array = in_array.reshape(sh_grid)[sl]

        if in_array.shape[1] == 1:
            out = out_array.reshape(-1,1)
        else:
            out = out_array

        return out.copy() if copy else out

    def pad_array(self, in_array, out_array=None, padding_mode=None):

        sh_dof  = self.shape(True, False)
        sh_grid = self.shape(True, True)
        sh_in_grid = self.shape(False,True)
        sl = list()
        for i in xrange(self.dim):
            p = self.parameters[i]

            nleft  = p.lbc.n
            nright = p.rbc.n

            sl.append(slice(nleft, sh_grid[i]-nright))

        if out_array is not None:
            out_array.shape = sh_grid
        else:
            out_array = np.zeros(sh_grid, dtype=in_array.dtype)
        out_array[sl] = in_array.reshape(sh_in_grid)

        if padding_mode is not None:
            _pad_tuple = tuple([(self.parameters[i].lbc.n, self.parameters[i].rbc.n) for i in xrange(self.dim)])
            out_array = np.pad(in_array.reshape(sh_in_grid), _pad_tuple, mode=padding_mode).copy()


        if in_array.shape[1] == 1: # Does not guarantee dof shaped, but suggests it.
            out_array.shape = sh_dof
        else:
            out_array.shape = sh_grid
        return out_array

    def inner_product(self, arg1, arg2):

        return np.dot(arg1.T, arg2).squeeze() * np.prod(self.deltas)

```

```

_sqrt2 = math.sqrt(2.0)


def _arrayify(arg):
    if not np.iterable(arg):
        return True, np.array([arg])
    else:
        return False, np.asarray(arg)


class SourceWaveletBase(object):

    __call__(self, t=None, nu=None, **kwargs)



    @property
    def time_source(self):
        """bool, Indicates if wavelet is defined in time domain."""
        return False

    @property
    def frequency_source(self):
        """bool, Indicates if wavelet is defined in frequency domain."""
        return False

    def __init__(self, *args, **kwargs):
        raise NotImplementedError('')

    def __call__(self, t=None, nu=None, **kwargs):

        if t is not None:
            if self.time_source:
                return self._evaluate_time(t)
            else:
                raise TypeError('Sources of type {0} are not time-domain
                    sources.'.format(self.__class__.__name__))
        elif nu is not None:
            if self.frequency_source:
                return self._evaluate_frequency(nu)
            else:
                raise TypeError('Sources of type {0} are not time-domain
                    sources.'.format(self.__class__.__name__))
        else:
            raise ValueError('Either a time or frequency must be provided
                .')


class DerivativeGaussianPulse(SourceWaveletBase):

    @property
    def time_source(self):
        """bool, Indicates if wavelet is defined in time domain."""
        return True

    @property
    def frequency_source(self):
        """bool, Indicates if wavelet is defined in frequency domain."""
        return True

    @property
    def order(self):
        return 4

    @order.setter
    def order(self, n):
        pass

    def __init__(self, nu, **kwargs):
        DerivativeGaussianPulse.__init__(self, nu, order=self.order, **
            kwargs)

    def _evaluate_time(self, ts):
        return 1*DerivativeGaussianPulse._evaluate_time(self, ts)

    def _evaluate_frequency(self, nus):
        return 1*DerivativeGaussianPulse._evaluate_frequency(self, nus)

   def __init__(self, peak_frequency, order=0, threshold=1e-6,
        shift_deviations=6, t_shift=None):
      self.order = order
      self.peak_frequency = peak_frequency
      self.threshold = threshold
      self.shift_deviations = shift_deviations

      nu = peak_frequency

      self.sigma = 1/(math.pi*nu*_sqrt2)

      if t_shift is None:
         self.t_shift = self.shift_deviations*self.sigma
      else:
         self.t_shift = t_shift

      poly_coeffs = (order)*[0.0]+[1.0]
      self._hermite = np.polynomial.Hermite(poly_coeffs)

  def _evaluate_time(self, ts):

      ts_was_not_array, ts = _arrayify(ts)

      n = self.order

      x = (ts-self.t_shift)/(_sqrt2*self.sigma)
      c = (-1/_sqrt2)**n
      v = c*self._hermite(x)*np.exp(-(x**2))

      v[np.abs(v) < self.threshold] = 0.0

      return v[0] if ts_was_not_array else v

   def _evaluate_frequency(self, nus):
      nus_was_not_array, nus = _arrayify(nus)

      omegas = 2*np.pi*nus
      n = self.order

      shift = np.exp(-1j*2*np.pi*nus*self.t_shift)

      a = (-1)**n
      b = (1j*omegas)**n
      c = self.sigma**(n+1)
      d = math.sqrt(2*np.pi)
      v = d*a*b*c*np.exp(-0.5*(self.sigma**2) * omegas**2)*shift

      v[np.abs(v) < self.threshold] = 0.0

      return v[0] if nus_was_not_array else v


class RickerWavelet(DerivativeGaussianPulse):
   @property
   def order(self):
      return 2

   @order.setter
   def order(self, n):
      pass

   def __init__(self, nu, **kwargs):
      DerivativeGaussianPulse.__init__(self, nu, order=self.order, **
         kwargs)

   def _evaluate_time(self, ts):
      return -1*DerivativeGaussianPulse._evaluate_time(self, ts)

   def _evaluate_frequency(self, nus):
      return -1*DerivativeGaussianPulse._evaluate_frequency(self, nus)


class GaussianPulse(DerivativeGaussianPulse):
   @property
   def order(self):
      return 0

    @order.setter
    def order(self, n):
        pass

    def __init__(self, nu, **kwargs):
        DerivativeGaussianPulse.__init__(self, nu, order=self.order, **
            kwargs)


class WhiteNoiseSource(SourceWaveletBase):

    @property
    def time_source(self):
        return True

    @property
    def frequency_source(self):
        """bool, Indicates if wavelet is defined in frequency domain."""
        return True

    def __init__(self, seed=None, variance=1.0, **kwargs):

        self._f = dict()
        self._f_hat = dict()

        self.seed = seed
        if seed is not None:
            np.random.seed(seed)

        self.variance = variance

    def _evaluate_time(self, ts):

        ts_was_not_array, ts = _arrayify(ts)

        v = list()
        for t in ts:
            if t not in self._f:
                self._f[t] = self.variance*(np.random.randn())
            v.append(self._f[t])

        return v[0] if ts_was_not_array else np.array(v)

    def _evaluate_frequency(self, nus):
        nus_was_not_array, nus = _arrayify(nus)

        v = list()
        for nu in nus:
            if nu not in self._f_hat:
                self._f_hat[nu] = self.variance*(np.random.randn() + np.
                    random.randn()*1j)
            v.append(self._f_hat[nu])

        return v[0] if nus_was_not_array else np.array(v)





__all__ = ['generate_seismic_vector_data', '
    generate_shot_vector_data_time', 'generate_shot_vector_data_frequency
    ']
__docformat__ = "restructuredtext en"


def generate_seismic_vector_data(shots, solver, model, verbose=False,
    frequencies=None, **kwargs):

    if verbose:
        print('Generating vector data...')
        tt = time.time()

    for shot in shots:

        if solver.supports['equation_dynamics'] == "time":
            generate_shot_vector_data_time(shot, solver, model, verbose=
                verbose, **kwargs)
        elif solver.supports['equation_dynamics'] == "frequency":
            if frequencies is None:
                raise TypeError('A frequency solver is passed, but no
                    frequencies are given')
            generate_shot_vector_data_frequency(shot, solver, model,
                frequencies, verbose=verbose, **kwargs)
        else:
            raise TypeError("A time or frequency solver must be
                specified.")

    if verbose:
        data_tt = time.time() - tt
        print 'Vector Data generation: {0}s'.format(data_tt)
        print 'Vector Data generation: {0}s/shot'.format(data_tt/len(
            shots))
```

```python

def generate_shot_vector_data_time(shot, solver, model, wavefields=None, wavefields_padded=None, verbose=False, **kwargs):

   solver.model_parameters = model


    ts = solver.ts()
    shot.reset_time_series(ts)


    shot.dt = solver.dt
    shot.trange = solver.trange

    if solver.supports['equation_dynamics'] != "time":
        raise TypeError('Solver must be a time solver to generate vector data.')

    if(wavefields is not None):
        wavefields[:] = []
    if(wavefields_padded is not None):
        wavefields_padded[:] = []

    mesh = solver.mesh
    dt = solver.dt
    source = shot.sources


    solver_data = solver.SolverData()

    rhs_k   = np.zeros(mesh.shape(include_bc=True))
    rhs_kp1 = np.zeros(mesh.shape(include_bc=True))

    # k is the t index.  t = k*dt.
    for k in xrange(solver.nsteps):

        uk = solver_data.k.primary_wavefield



        uk_bulk = mesh.unpad_array(uk)



        shot.receivers.sample_data_from_array(uk_bulk, k, **kwargs)

        if(wavefields is not None):
            wavefields.append(uk_bulk.copy())
        if(wavefields_padded is not None):
            wavefields_padded.append(uk.copy())

        if(k == (solver.nsteps-1)): break

        if k == 0:
            rhs_k = mesh.pad_array(source.f(k*dt), out_array=rhs_k)
            rhs_kp1 = mesh.pad_array(source.f((k+1)*dt), out_array=
                rhs_kp1)
        else:

            rhs_k, rhs_kp1 = rhs_kp1, rhs_k
            rhs_kp1 = mesh.pad_array(source.f((k+1)*dt), out_array=
                rhs_kp1)

        solver.time_step(solver_data, rhs_k, rhs_kp1)
```

```python

        solver_data.advance()

def generate_shot_vector_data_frequency(shot, solver, model, frequencies
    , verbose=False, **kwargs):
        solver.model_parameters = model

        mesh = solver.mesh

        source = shot.sources

        if not np.iterable(frequencies):
            frequencies = [frequencies]

        solver_data = solver.SolverData()
        rhs = solver.WavefieldVector(mesh,dtype=solver.dtype)
        for nu in frequencies:
            rhs = solver.build_rhs(mesh.pad_array(source.f(nu=nu)),
                rhs_wavefieldvector=rhs)
            solver.solve(solver_data, rhs, nu)
            uhat = solver_data.k.primary_wavefield

            shot.receivers.sample_data_from_array(mesh.unpad_array(uhat)
                , nu=nu)


if __name__ == '__main__':
    pmlx = PML(0.1, 100)
    pmlz = PML(0.1, 100)

    x_config = (0.1, 1.0, pmlx, pmlx)
    z_config = (0.1, 0.8, pmlz, pmlz)

    d = RectangularDomain(x_config, z_config)

    m = CartesianMesh(d, 91, 71)

    C, C0, m, d = horizontal_reflector(m)

    Nshots = 1
    shots = []

    xmin = d.x.lbound
    xmax = d.x.rbound
    nx   = m.x.n
    zmin = d.z.lbound
    zmax = d.z.rbound

    source_list_p = []
    for i in xrange(Nshots):
        source_list_p.append(PointSource(m, (xmax*(i+1.0)/(Nshots+1.0),
            0.25), DipoleSecondDerivativeRickerWavelet(10.0), intensity =
             (5)))
    source_list_f = []
    for j in xrange(Nshots):
        source_list_f.append(PointSource(m, (xmax*(j+1.1)/(Nshots+1.0),
            0.26), SecondDerivativeRickerWavelet(10.0), intensity = (5)))
source_set = SourceSet(m,source_list_p+source_list_f)
zpos = (1./9.)*zmax
    xpos = np.linspace(xmin, xmax, nx)
    receivers = ReceiverSet(m, [PointReceiver(m, (x, zpos)) for x in
        xpos])
    shot = Shot(source_set, receivers)
    shots.append(shot)
    trange = (0.0,3.0)

    solver = ConstantDensityAcousticTimeScalar_2D_cpp(m,
                                               spatial_accuracy_order=6,
                                               trange=trange,
                                               kernel_implementation='cpp')

    tt = time.time()
    wavefields = []
    true_model = solver.ModelParameters(m,{'C': C})
    initial_model = solver.ModelParameters(m,{'C': C0})

generate_seismic_vector_data(shots, solver, true_model, verbose=False,
 wavefields=wavefields)

Dx = np.spdiags(ones(n+1,1)*[-1/h 1/h],[0,1],n,n+1)
d1 = Dx(n(1),h(1))
d2 = Dx(n(2),h(2))
Grad = [np.kron(spsp.eye(n(2)+1),d1),np. kron(d2,spsp.eye(n(1)+1))]
L = np.transpose(Grad)*Grad
```

```

__all__ = ['TemporalModelingVDPointForceSourceHorizontal']

__docformat__ = "restructuredtext en"

class TemporalModelingVDPointForceSourceHorizontal(object):

    @property
    def solver_type(self): return "time"
```

```python
    @property
    def modeling_type(self): return "time"

    def __init__(self, solver):

        if self.solver_type == solver.supports['equation_dynamics']:
            self.solver = solver
        else:
            raise TypeError("Argument 'solver' type {1} does not match
                modeling solver type {0}.".format(self.solver_type,
                solver.supports['equation_dynamics']))

    def _setup_forward_rhs(self, rhs_array, data):
        return self.solver.mesh.pad_array(data, out_array=rhs_array)

    def forward_model_vd(self, shot, m0, imaging_period,
        return_parameters=[]):
        solver = self.solver
        solver.model_parameters = m0

        mesh = solver.mesh

        d = solver.domain
        dt = solver.dt
        nsteps = solver.nsteps
        source = shot.sources


        if 'wavefield' in return_parameters:
            us = list()




        if 'simvdata' in return_parameters:
            simvdata1 = np.gradient(np.zeros((solver.nsteps, shot.
                receivers.receiver_count)))
            simvdata1 = np.array(simvdata1).squeeze()
            simvdata = simvdata1[1,:,:]
            simvdata = np.array(simvdata).squeeze()
            simvdata = np.transpose(simvdata)
            simvdata = np.array(simvdata).squeeze()
        if 'dWaveOp' in return_parameters:
            dWaveOp = list()


        solver_data = solver.SolverData()

        rhs_k   = np.zeros(mesh.shape(include_bc=True))
        rhs_kp1 = np.zeros(mesh.shape(include_bc=True))

        for k in xrange(nsteps):

            uk = solver_data.k.primary_wavefield
            uk_bulk = mesh.unpad_array(uk)

            if 'wavefield' in return_parameters:
                us.append(uk_bulk.copy())
            if 'simvdata' in return_parameters:
                shot.receivers.sample_data_from_array(uk_bulk, k, data=
                    simvdata)

            if k == 0:
                rhs_k = self._setup_forward_rhs(rhs_k, source.f(k*dt))
                rhs_kp1 = self._setup_forward_rhs(rhs_kp1, source.f((k+1)*dt))
            else:

                rhs_k, rhs_kp1 = rhs_kp1, rhs_k
            rhs_kp1 = self._setup_forward_rhs(rhs_kp1, source.f((k+1)*dt))


            solver.time_step(solver_data, rhs_k, rhs_kp1)


            if 'dWaveOp' in return_parameters:
                if k%imaging_period == 0: #Save every 'imaging_period' number of steps
                    dWaveOp.append(solver.compute_dWaveOp('time', solver_data))

                      if(k == (nsteps-1)): break



            solver_data.advance()

        retval = dict()

        if 'wavefield' in return_parameters:
            retval['wavefield'] = us
        if 'dWaveOp' in return_parameters:
                retval['dWaveOp'] = dWaveOp

        if 'simvdata' in return_parameters:
            retval['simvdata'] = simvdata

        return retval

    def migrate_shot(self, shot, m0,
                              operand_simvdata, imaging_period,
                                 operand_dWaveOpAdj=None, operand_model=
                                 None,
                              dWaveOp=None,
                              adjointfield=None, dWaveOpAdj=None):
        if dWaveOp is None:
            retval = self.forward_model_vd(shot, m0, imaging_period,
                return_parameters=['dWaveOp'])
            dWaveOp = retval['dWaveOp']

        rp = ['imaging_condition']
        if adjointfield is not None:
            rp.append('adjointfield')
        if dWaveOpAdj is not None:
            rp.append('dWaveOpAdj')

        rv = self.adjoint_model(shot, m0, operand_simvdata,
            imaging_period, operand_dWaveOpAdj, operand_model,
            return_parameters=rp, dWaveOp=dWaveOp)

        if adjointfield is not None:
            adjointfield[:] = rv['adjointfield'][:]
        if dWaveOpAdj is not None:
                dWaveOpAdj[:] = rv['dWaveOpAdj'][:]


        ic = rv['imaging_condition']


        return ic.without_padding()

    def _setup_adjoint_rhs(self, rhs_array, shot, k, operand_simvdata,
        operand_model, operand_dWaveOpAdj):

        rhs_array = self.solver.mesh.pad_array(shot.receivers.
            extend_data_to_array(k, data=operand_simvdata), out_array=
            rhs_array)


        if (operand_dWaveOpAdj is not None) and (operand_model is not
            None):
            rhs_array += operand_model*operand_dWaveOpAdj[k]

        return rhs_array

    def adjoint_model(self, shot, m0, operand_simvdata, imaging_period,
        operand_dWaveOpAdj=None, operand_model=None, return_parameters
        =[], dWaveOp=None):

        solver = self.solver
        solver.model_parameters = m0

        mesh = solver.mesh

        d = solver.domain
        dt = solver.dt
        nsteps = solver.nsteps
        source = shot.sources

        if 'adjointfield' in return_parameters:
            qs = list()
            vs = list()


        if 'dWaveOpAdj' in return_parameters:
            dWaveOpAdj = list()


        if dWaveOp is not None:
            ic = solver.model_parameters.perturbation()
            do_ic = True
        elif 'imaging_condition' in return_parameters:
            raise ValueError('To compute imaging condition, forward
                component must be specified.')
        else:
            do_ic = False


        solver_data = solver.SolverData()

        rhs_k   = np.zeros(mesh.shape(include_bc=True))
        rhs_km1 = np.zeros(mesh.shape(include_bc=True))

        if operand_model is not None:
            operand_model = operand_model.with_padding()


        for k in xrange(nsteps-1, -1, -1):

            vk = solver_data.k.primary_wavefield
            vk_bulk = mesh.unpad_array(vk)


            if 'adjointfield' in return_parameters:
                vs.append(vk_bulk.copy())


            if do_ic:
                if k%imaging_period == 0:
                    entry = k/imaging_period
                    ic += vk*dWaveOp[entry]

            if k == nsteps-1:
                rhs_k   = self._setup_adjoint_rhs( rhs_k,   shot, k, operand_simvdata, operand_model, operand_dWaveOpAdj)
                rhs_km1 = self._setup_adjoint_rhs( rhs_km1, shot, k-1, operand_simvdata, operand_model, operand_dWaveOpAdj)
            else:
                rhs_k, rhs_km1 = rhs_km1, rhs_k
            rhs_km1 = self._setup_adjoint_rhs( rhs_km1, shot, k-1, operand_simvdata, operand_model, operand_dWaveOpAdj)

            solver.time_step(solver_data, rhs_k, rhs_km1)


            if 'dWaveOpAdj' in return_parameters:
                if k%imaging_period == 0: #Save every 'imaging_period' number of steps
                    dWaveOpAdj.append(solver.compute_dWaveOp('time', solver_data))

            if(k == 0): break


            solver_data.advance()

        if do_ic:
            ic *= (-1*dt)
            ic *= imaging_period
            ic = ic.without_padding()

        retval = dict()

        if 'adjointfield' in return_parameters:

            qs = list(vs)
            qs.reverse()
            retval['adjointfield'] = qs
        if 'dWaveOpAdj' in return_parameters:
            dWaveOpAdj.reverse()
            retval['dWaveOpAdj'] = dWaveOpAdj

        if do_ic:
            retval['imaging_condition'] = ic

        return retval

```

```
    def linear_forward_model_vd(self, shot, m0, m1, return_parameters
        =[], dWaveOp0=None):

        solver = self.solver
        solver.model_parameters = m0

        mesh = solver.mesh

        d = solver.domain
        dt = solver.dt
        nsteps = solver.nsteps
        source = shot.sources

        m1_padded = m1.with_padding()

        if 'wavefield1' in return_parameters:
            us = list()
        if 'simvdata' in return_parameters:

            simvdata1 = np.gradient(np.zeros((solver.nsteps, shot.
                receivers.receiver_count)))
            simvdata1 = np.array(simvdata1).squeeze()
            simvdata = simvdata1[1,:,:]
            simvdata = np.array(simvdata).squeeze()
            simvdata = np.transpose(simvdata)
            simvdata = np.array(simvdata).squeeze()


        if 'dWaveOp0' in return_parameters:
            dWaveOp0ret = list()

        if 'dWaveOp1' in return_parameters:
            dWaveOp1 = list()


        solver_data = solver.SolverData()

        if dWaveOp0 is None:
            solver_data_u0 = solver.SolverData()


            rhs_u0_k   = np.zeros(mesh.shape(include_bc=True))
            rhs_u0_kp1 = np.zeros(mesh.shape(include_bc=True))
            rhs_u0_k   = self._setup_forward_rhs(rhs_u0_k,   source.f(0*
                dt))
            rhs_u0_kp1 = self._setup_forward_rhs(rhs_u0_kp1, source.f(1*
                dt))


            solver.time_step(solver_data_u0, rhs_u0_k, rhs_u0_kp1)


            dWaveOp0_k = solver.compute_dWaveOp('time', solver_data_u0)
            dWaveOp0_kp1 = dWaveOp0_k.copy()

            solver_data_u0.advance()

            rhs_u0_kp1, rhs_u0_kp2 = rhs_u0_k, rhs_u0_kp1

        else:
            solver_data_u0 = None

        for k in xrange(nsteps):
            uk = solver_data.k.primary_wavefield
            uk_bulk = mesh.unpad_array(uk)

            if 'wavefield1' in return_parameters:
                us.append(uk_bulk.copy())

            if 'simvdata' in return_parameters:
                shot.receivers.sample_data_from_array(uk_bulk, k, data=
                    simvdata)


            if dWaveOp0 is None:

                rhs_u0_kp1, rhs_u0_kp2 = rhs_u0_kp2, rhs_u0_kp1
                rhs_u0_kp2 = self._setup_forward_rhs(rhs_u0_kp2, source.
                    f((k+2)*dt))
                solver.time_step(solver_data_u0, rhs_u0_kp1, rhs_u0_kp2)
                dWaveOp0_k, dWaveOp0_kp1 = dWaveOp0_kp1, dWaveOp0_k
                dWaveOp0_kp1 = solver.compute_dWaveOp('time',
                    solver_data_u0)

                solver_data_u0.advance()
            else:
                dWaveOp0_k = dWaveOp0[k]
                dWaveOp0_kp1 = dWaveOp0[k+1] if k < (nsteps-1) else
                    dWaveOp0[k] # in case not enough dWaveOp0's are
                    provided, repeat the last one

            if 'dWaveOp0' in return_parameters:
                dWaveOp0ret.append(dWaveOp0_k)
```

```python

            if k == 0:
                rhs_k    = m1_padded*(-1*dWaveOp0_k)
                rhs_kp1 = m1_padded*(-1*dWaveOp0_kp1)
            else:
                rhs_k, rhs_kp1 = rhs_kp1, m1_padded*(-1*dWaveOp0_kp1)

            solver.time_step(solver_data, rhs_k, rhs_kp1)

            if 'dWaveOp1' in return_parameters:
                dWaveOp1.append(solver.compute_dWaveOp('time', solver_data))
            if(k == (nsteps-1)): break
            solver_data.advance()

        retval = dict()

        if 'wavefield1' in return_parameters:
            retval['wavefield1'] = us
        if 'dWaveOp0' in return_parameters:
            retval['dWaveOp0'] = dWaveOp0ret
        if 'dWaveOp1' in return_parameters:
            retval['dWaveOp1'] = dWaveOp1
        if 'simvdata' in return_parameters:
            retval['simvdata'] = simvdata

        return retval
```

B.3 Inverse Problem and Optimization

Here we put one of our inverse modelling codes for horizontal reflector model. Having forward problem and derivatives we can approach vector-acoustic FWI.

B.4 Inversion Python Code for Horizontal Reflector

```python

from my_extensions.vector_data_modeling import *
import copy

__all__ = ['TemporalLeastSquaresHorizontalVDFWI']

__docformat__ = "restructuredtext en"

class TemporalLeastSquaresHorizontalVDFWI(ObjectiveFunctionBase):

    def __init__(self, solver, parallel_wrap_shot=ParallelWrapShotNull()
        , imaging_period = 1):

        self.solver = solver
        self.modeling_tools =
            TemporalModelingVDPointForceSourceHorizontal(solver)

        self.parallel_wrap_shot = parallel_wrap_shot

        self.imaging_period = int(imaging_period)

    def _residual(self, shot, m0, dWaveOp=None):

        rp = ['simvdata']
        if dWaveOp is not None:
            rp.append('dWaveOp')

        retval = self.modeling_tools.forward_model_vd(shot, m0, self.
            imaging_period, return_parameters=rp)




        VD_retval = np.gradient(retval['simvdata'])
        VD_retval = np.array(VD_retval).squeeze()
        VD = VD_retval[1,:,:]
        VD = np.array(VD).squeeze()
        VD = np.transpose(VD)
        VD = np.array(VD).squeeze()

        resid = shot.receivers.interpolate_data(self.solver.ts()) - VD
        if dWaveOp is not None:
            dWaveOp[:]  = retval['dWaveOp'][:]

        return resid

    def evaluate(self, shots, m0, **kwargs):

        r_norm2 = 0
        for shot in shots:
            r = self._residual(shot, m0)
            r_norm2 += np.linalg.norm(r)**2

        if self.parallel_wrap_shot.use_parallel:
            new_r_norm2 = np.array(0.0)
            self.parallel_wrap_shot.comm.Allreduce(np.array(r_norm2),
                new_r_norm2)
            r_norm2 = new_r_norm2[()] # goofy way to access 0-D array
                element

        return 0.5*r_norm2*self.solver.dt

    def _gradient_helper(self, shot, m0, ignore_minus=False,
        ret_pseudo_hess_diag_comp = False, **kwargs):

        dWaveOp=[]
        r = self._residual(shot, m0, dWaveOp=dWaveOp, **kwargs)


        g = self.modeling_tools.migrate_shot(shot, m0, r, self.
            imaging_period, dWaveOp=dWaveOp)

        if not ignore_minus:
            g = -1*g

        if ret_pseudo_hess_diag_comp:
            return g, r, self._pseudo_hessian_diagonal_component_shot(
                dWaveOp)
        else:
            return g, r

    def _pseudo_hessian_diagonal_component_shot(self, dWaveOp):
        mesh = self.solver.mesh

        import time
        tt = time.time()
        pseudo_hessian_diag_contrib = np.zeros(mesh.unpad_array(dWaveOp
            [0], copy=True).shape)
        for i in xrange(len(dWaveOp)):
            unpadded_dWaveOp_i = mesh.unpad_array(dWaveOp[i])
            pseudo_hessian_diag_contrib += unpadded_dWaveOp_i*
                unpadded_dWaveOp_i

        pseudo_hessian_diag_contrib *= self.imaging_period #Compensate
            for doing fewer summations at higher imaging_period

        print "Time elapsed when computing pseudo hessian diagonal
            contribution shot: %e"%(time.time() - tt)

        return pseudo_hessian_diag_contrib

    def compute_gradient(self, shots, m0, aux_info={}, **kwargs):

        grad = m0.perturbation()
        r_norm2 = 0.0
        pseudo_h_diag = np.zeros(m0.asarray().shape)
        for shot in shots:
            if ('pseudo_hess_diag' in aux_info) and aux_info['
                pseudo_hess_diag'][0]:
                g, r, h = self._gradient_helper(shot, m0, ignore_minus=
                    True, ret_pseudo_hess_diag_comp = True, **kwargs)
                pseudo_h_diag += h
            else:
                g, r = self._gradient_helper(shot, m0, ignore_minus=True
                    , **kwargs)

            grad -= g # handle the minus 1 in the definition of the
                gradient of this objective
            r_norm2 += np.linalg.norm(r)**2


        if self.parallel_wrap_shot.use_parallel:

            new_r_norm2 = np.array(0.0)
            self.parallel_wrap_shot.comm.Allreduce(np.array(r_norm2),
                new_r_norm2)
            r_norm2 = new_r_norm2[()] # goofy way to access 0-D array
                element

            ngrad = np.zeros_like(grad.asarray())
            self.parallel_wrap_shot.comm.Allreduce(grad.asarray(), ngrad
                )
            grad=m0.perturbation(data=ngrad)

            if ('pseudo_hess_diag' in aux_info) and aux_info['
                pseudo_hess_diag'][0]:
                pseudo_h_diag_temp = np.zeros(pseudo_h_diag.shape)
                self.parallel_wrap_shot.comm.Allreduce(pseudo_h_diag,
                    pseudo_h_diag_temp)
                pseudo_h_diag = pseudo_h_diag_temp


        r_norm2 *= self.solver.dt
        pseudo_h_diag *= self.solver.dt


        if ('residual_norm' in aux_info) and aux_info['residual_norm'
            ][0]:
            aux_info['residual_norm'] = (True, np.sqrt(r_norm2))
        if ('objective_value' in aux_info) and aux_info['objective_value
            '][0]:
            aux_info['objective_value'] = (True, 0.5*r_norm2)
        if ('pseudo_hess_diag' in aux_info) and aux_info['
            pseudo_hess_diag'][0]:
            aux_info['pseudo_hess_diag'] = (True, pseudo_h_diag)

        return grad

    def apply_hessian(self, shots, m0, m1, hessian_mode='approximate',
        levenberg_mu=0.0, *args, **kwargs):

        modes = ['approximate', 'full', 'levenberg']
        if hessian_mode not in modes:
            raise ValueError("Invalid Hessian mode.  Valid options for
                applying hessian are {0}".format(modes))

        result = m0.perturbation()

        if hessian_mode in ['approximate', 'levenberg']:
            for shot in shots:

                retval = self.modeling_tools.forward_model_vd(shot, m0,
                    return_parameters=['dWaveOp'])
                dWaveOp0 = retval['dWaveOp']

```

```
                    linear_retval = self.modeling_tools.
                        linear_forward_model_vd(shot, m0, m1,
                        return_parameters=['simvdata'], dWaveOp0=dWaveOp0)

                    d1 = linear_retval['simvdata']

                    ##### vector-data again

                    VD_retval1 = np.gradient(d1)
                    VD_retval1 = np.array(VD_retval1).squeeze()
                    VD1 = VD_retval1[1,:,:]
                    VD1 = np.array(VD1).squeeze()
                    VD1 = np.transpose(VD1)
                    VD1 = np.array(VD1).squeeze()
                    result += self.modeling_tools.migrate_shot(shot, m0, VD1
                        , dWaveOp=dWaveOp0)

            elif hessian_mode == 'full':
                for shot in shots:

                    dWaveOp0 = list()
                    r0 = self._residual(shot, m0, dWaveOp=dWaveOp0, **kwargs
                        )

                    linear_retval = self.modeling_tools.
                        linear_forward_model_vd(shot, m0, m1,
                        return_parameters=['simvdata', 'dWaveOp1'], dWaveOp0=
                        dWaveOp0)
                    d1 = linear_retval['simvdata']
                    dWaveOp1 = linear_retval['dWaveOp1']

```

```python
                VD_retval1 = np.gradient(d1)
                VD_retval1 = np.array(VD_retval1).squeeze()
                VD1 = VD_retval1[1,:,:]
                VD1 = np.array(VD1).squeeze()
                VD1 = np.transpose(VD1)
                VD1 = np.array(VD1).squeeze()


                dWaveOpAdj1=[]
                res1 = self.modeling_tools.migrate_shot( shot, m0, r0, dWaveOp=dWaveOp1, dWaveOpAdj=dWaveOpAdj1)
                result += res1

                res2 = self.modeling_tools.migrate_shot(shot, m0, VD1, operand_dWaveOpAdj=dWaveOpAdj1, operand_model=m1, dWaveOp=dWaveOp0)
                result += res2

        if self.parallel_wrap_shot.use_parallel:

            nresult = np.zeros_like(result.asarray())
            self.parallel_wrap_shot.comm.Allreduce(result.asarray(), nresult)
            result = m0.perturbation(data=nresult)
        if hessian_mode == 'levenberg':
            result += levenberg_mu*m1

        return result

```

```python
from collections import deque
```

```

__all__=['LBFGSMODIF']

__docformat__ = "restructuredtext en"

class LBFGSMODIF(OptimizationBase):

    def __init__(self, objective_0, objective_1 = None, memory_length=
        None, reset_on_new_inner_loop_call=True, geom_fac = 0.6,
        geom_fac_up = 0.7, scale_step = False, *args, **kwargs):
        if objective_1 == None:
            objective_1 = objective_0

        OptimizationBase.__init__(self, objective_0, objective_1,
            geom_fac = geom_fac, geom_fac_up = geom_fac_up, *args, **
            kwargs)
        self.prev_alpha = None
        self.prev_model = None
        self.memory_length=memory_length
        self.reset_on_new_inner_loop_call = reset_on_new_inner_loop_call
        self.scale_step = scale_step

        self._reset_memory()

    def _reset_memory(self):
        self.memory = deque([], maxlen=self.memory_length)
        self._reset_line_search = True
        self.prev_model = None

    def inner_loop(self, *args, **kwargs):

        if self.reset_on_new_inner_loop_call:
            self._reset_memory()

        OptimizationBase.inner_loop(self, *args, **kwargs)

    def _select_step(self, shot_0, shot_1, beta, beta_scale,
       current_objective_value, gradient, iteration, objective_arguments
       , **kwargs):
        mem = self.memory

        q = copy.deepcopy(gradient)

        x_k = copy.deepcopy(self.base_model)


        if len(mem) > 0:
            mem[-1][2] += gradient # y
            mem[-1][1]  = x_k - self.prev_model #Subtraction will result
                 a model perturbation, which is linear.
            mem[-1][0]  = 1./mem[-1][2].inner_product(mem[-1][1]) # rho
            gamma = mem[-1][1].inner_product(mem[-1][2]) / mem[-1][2].
                inner_product(mem[-1][2])
        else:
            gamma = 1.0

        alphas = []

        for rho, s, y in reversed(mem):
            alpha_ = rho * s.inner_product(q)
            t= alpha_ * y
            q -= t
            alphas.append(alpha_)

        alphas.reverse()

        r = gamma * q

        for alpha_, m in zip(alphas, mem):
            rho, s, y = m
            beta_ = rho*y.inner_product(r)
            r += (alpha_-beta_)*s


        direction = -1.0*r

        alpha0_kwargs = {'reset' : False}
        if self._reset_line_search:
            alpha0_kwargs = {'reset' : True}
            self._reset_line_search = False

        self.unscaled_suggested_step = direction
        alpha_ = self.select_alpha(shot_0, shot_1, beta, beta_scale,
            gradient, direction, objective_arguments,
                                   current_objective_value=
                                       current_objective_value,
                                   alpha0_kwargs=alpha0_kwargs, **kwargs)

        self._print('  alpha {0}'.format(alpha_))
        self.store_history('alpha', iteration, alpha_)

        step = alpha_ * direction


        self.prev_model = x_k
        self.memory.append([None,None,copy.deepcopy(-1*gradient)])

        return step


    def _compute_alpha0(self, phi0, grad0, reset=False, *args, **kwargs)
        :
        if reset:
            self.did_grad_descent = True
            return phi0 / (grad0.norm()*np.prod(self.solver.mesh.deltas)
                )**2
        else:
            if self.scale_step and not self.did_grad_descent:
                mem = self.memory
                last_accepted_step = mem[-1][1]
                last_accepted_step_len = np.sqrt(np.linalg.norm(
                    last_accepted_step.p_0.data)**2 + np.linalg.norm(
                    last_accepted_step.p_1.data)**2)

                geom_fac_up = kwargs['upscale_factor']
                desired_new_step_len = last_accepted_step_len /
                    geom_fac_up

                current_new_step_len = np.sqrt(np.linalg.norm(self.
                    unscaled_suggested_step.p_0.data)**2 + np.linalg.norm
                    (self.unscaled_suggested_step.p_1.data)**2)


                ret_val = desired_new_step_len/current_new_step_len

                if ret_val > 1.0:

                    ret_val = 1.0

            else:
                ret_val = 1.0

            self.did_grad_descent = False
            return ret_val
```

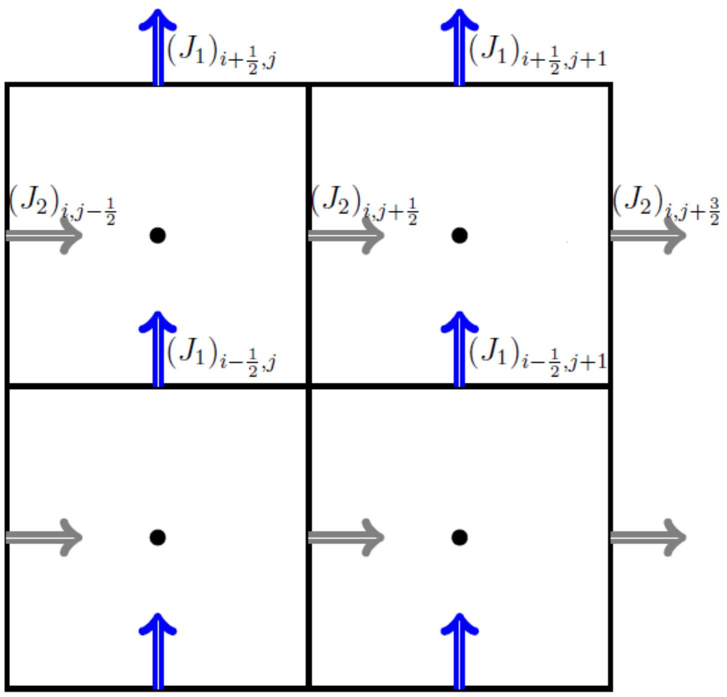
B.5 Discretization of Regularization

Here we explain the discretization of our quadratic regularization.

If we consider the regularization of the form

$$\begin{aligned}\mathcal{R} &= \frac{1}{2}\int_\Omega \alpha_0 m^2 + \alpha_1(x) m_x^2 + \alpha_2(x) m_y^2 \; dv = \\ &= \frac{1}{2}\int_\Omega \alpha_0 m^2 + (m_x \;\; m_y)\begin{pmatrix}\alpha_1 & \\ & \alpha_2\end{pmatrix}\begin{pmatrix} m_x \\ m_y\end{pmatrix} \; dv,\end{aligned} \tag{B.2}$$

where $\alpha_i(x)$ $i = 0, 1, 2$ are positive coefficient functions.



$(J_1)_{i+\frac{1}{2},j}$
$(J_1)_{i+\frac{1}{2},j+1}$
$(J_2)_{i,j-\frac{1}{2}}$
$(J_2)_{i,j+\frac{1}{2}}$
$(J_2)_{i,j+\frac{3}{2}}$
$(J_1)_{i-\frac{1}{2},j}$
$(J_1)_{i-\frac{1}{2},j+1}$

Figure B.1: Model discretization in a 2D staggered grid, explaining finite difference method by letting $J_1 = m_x$ and $J_2 = m_y$.

We can assume that m is discretized in cell centers as shown in Fig. B.1. Let

$J_1 = m_x$ and $J_2 = m_y$. Using finite difference method we have

$$(J_1)_{i+\frac{1}{2},j} = \frac{1}{h}(m_{i+1,j} - m_{i,j}) + \mathcal{O}(h^2),$$

$$(J_2)_{i,j+\frac{1}{2}} = \frac{1}{h}(m_{i,j+1} - m_{i,j}) + \mathcal{O}(h^2).$$

This formulation leads to a staggered grid, i.e. J_1, J_2 and m are discretized in different locations. Using a combination of the trapezoidal and midpoint method we can now discretize $\mathcal{R}(m)$. Considering i and j cells, we have

$$\int_{\Omega_{i,j}} m^2 \; dv = h^2 m_{i,j}^2 + \mathcal{O}(h^2).$$

Similarly, for approximation of the derivatives we have

$$\int_{\Omega_{i,j}} m_x^2 \; dv = \frac{1}{2}((m_{i+1,j} - m_{i,j})^2 + (m_{i,j} - m_{i-1,j})^2) + \mathcal{O}(h^2),$$

$$\int_{\Omega_{i,j}} m_y^2 \; dv = \frac{1}{2}((m_{i,j+1} - m_{i,j})^2 + (m_{i,j} - m_{i,j-1})^2) + \mathcal{O}(h^2).$$

If we sum over all cells we obtain a second order approximation to the integral. We can our discretization in matrix form. In order to discretize gradient of m, we need to use our gradient discretization in previous sections i. e.,

$$D = \frac{1}{h}\begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}.$$

Then, using the Kronecker product we can approximate the gradient by the matrix

$$\nabla_h = \begin{bmatrix} I \otimes D \\ D \otimes I \end{bmatrix}.$$

Here ∇_h : cell centers $\Rightarrow$ cell faces. Now we build a matrix that approximates the averaging process. We can do it by a combination of 1D matrices and Kronecker products. Therefore, in 1D we have

$$A = \frac{1}{2}\begin{bmatrix} 2 & & & \\ 1 & 1 & & \\ & \ddots & \ddots & \\ & & & 2 \end{bmatrix}.$$

In 2D we write

$$A_v = \begin{bmatrix} I \otimes A & A \otimes I \end{bmatrix},$$

and we have A_v : cell faces $\Rightarrow$ cell centers. By having all of these operators now we can approximate the integral as

$$\mathcal{R} = m^\top \mathrm{diag}(v)\ m + v^\top\ A_v((\nabla_h m) \odot (\nabla_h m)) = m^\top\ \nabla_h^\top \mathrm{diag}(A_v^\top v)\ \nabla_h\ m.$$

Where $\odot$ is Hadamard product.