ETH zürich

ETH Library

Full Waveform Inversion for Medical Ultrasound Tomography in Julia on multi-xPUs

Master Thesis

Author(s):
Aloisi, Giacomo

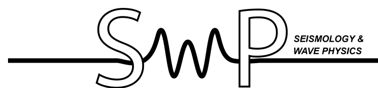
Publication date:
2023-10-30

Permanent link:
https://doi.org/10.3929/ethz-b-000668605

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Full Waveform Inversion for Medical Ultrasound Tomography in Julia on multi-xPUs

Master's Thesis in Computational Science and Engineering
Department of Mathematics
ETH Zürich

Giacomo Aloisi
galoisi@student.ethz.ch

Conducted under the Seismology and Wave Physics Group
Institute of Geophysics
Department of Earth Sciences
ETH Zürich

Supervisors:
Dr. Andrea Zunino
Prof. Dr. Andreas Fichtner

October 30, 2023

Dedicated...

to my family, which supported me through hard times and gave me the amazing opportunity of studying abroad to follow my dreams.

to my friends, old and new, for your presence and endurance of my crazy self, as well as sharing some unforgettable moments together.

to my fellow student colleagues, with whom I have spend countless hours solving exercises and studying together to pass tough exams.

to my university, for inspiring me and pushing myself to do better every day.

to my supervisors, Andrea and Andreas, for supporting me through scientific work and for sharing with me just a tiny bit of your knowledge.

to my colleagues in the SWP group, for your help and useful tips.

to the Julia community and the people that have introduced me to it, for your great contributions in computational software development and your dedication.

to everyone who supported me, for without you this work would not have been possible.

Science is what we understand well
enough to explain to a computer.
Art is everything else we do.

Foreword to the book "A=B"
DONALD E. KNUTH

Abstract

This Master's Thesis aims to develop efficient yet user-friendly tools to perform seismic tomography using the Full Waveform Inversion (FWI) approach, as derived from the field of Earth sciences, and apply them in the realm of ultrasound medical tomography. Since its introduction in geosciences a few years ago, FWI has provided spectacular images of the Earth's subsurface at different scales, revealing, with unprecedented details, the internal structure of our planet. Recently, FWI has also started to be used to reconstruct high-resolution medical images of soft tissues from ultrasound data. However, due to its high computational cost and complexity, it has yet to see extensive use in real-world applications.

This work aims to fill the gap between theory and practice by providing efficient, easy-to-use, and scalable finite-difference-based solvers for acoustic FWI written in the high-level Julia programming language. This allows also non-expert users to perform numerical experiments to test different setups and algorithms and to address real applications with ultrasound data. Our solvers are device-agnostic and simulations can be distributed on multiple devices (multi-xPUs), providing the user with a range of parallelization options that can fit different problems.

The correctness of the solvers is checked using rigorous tests and synthetic inversions, where we show the potential and pitfalls of the method. Finally, as an application, a case study inversion with ultrasound data gathered in a medical imaging setting is performed, to assess the feasibility in real-world scenarios. A set of benchmarks testing the memory throughput, since the algorithms under study are memory-bound, show that our solvers achieve a very high percentage of peak memory throughput (up to 90%) on modern GPUs and good weak scaling on distributed systems.

We conclude that using modern advances in software and hardware provided by the scientific computing community, the computational challenges of FWI can be addressed to make it a feasible method for ultrasound medical tomography.

Contents

Abstract **v**

1 Introduction **1**
- 1.1 Seismic tomography and full waveform inversion 2
- 1.2 Medical ultrasound tomography 2
- 1.3 The two-language problem: Julia as a possible solution 3
- 1.4 Motivation and aims 3
- 1.5 Outline 4

2 Theory **5**
- 2.1 Forward and inverse problems 5
- 2.2 Methods for solving inverse problems 7
  - 2.2.1 Probabilistic methods 7
  - 2.2.2 Deterministic methods 9
- 2.3 Adjoint methods 14
  - 2.3.1 Continuous adjoint method 15
  - 2.3.2 Discrete adjoint method 19
  - 2.3.3 Main takeaways from the adjoint method 22
- 2.4 Case study: acoustic wave equation 23
  - 2.4.1 Wave equation 23
  - 2.4.2 Initial and boundary conditions 25
  - 2.4.3 Acoustic wave equation 30
  - 2.4.4 Derivation of adjoint equations and sensitivity kernels using the continuous adjoint method 32
- 2.5 Summary 37

3 Numerical Methods **39**
- 3.1 Finite differences discretization of the acoustic wave equation . . 39
  - 3.1.1 Lowest-order finite difference stencils for acoustic wave equation 40
  - 3.1.2 C-PML boundary conditions implementation 48
  - 3.1.3 How to model point sources and source scaling 50
- 3.2 Technical details and implementation 52
  - 3.2.1 Checkpointing for storage of the forward wave field 52
  - 3.2.2 Device-agnostic (xPU) solvers 54
  - 3.2.3 From single xPU to multi-xPUs 56

4 Inversion results **59**
4.1 Adjoint and gradient checking . . . 59
4.1.1 Hockey stick plots . . . 60
4.1.2 Finite differences gradient comparison in 2D . . . 61
4.2 Synthetic inversions . . . 65
4.2.1 Noiseless synthetic inversions . . . 65
4.2.2 Noisy synthetic inversions . . . 70
4.3 Real data inversions . . . 75
4.4 Summary . . . 78

5 Benchmarks **79**
5.1 Benchmark metrics, setup, and statistical analysis of run time measurements . . . 79
5.2 Single node xPU kernel microbenchmarks . . . 84
5.3 Single node GPU forward solver macrobenchmarks . . . 87
5.4 Multi-node GPU weak scaling benchmarks . . . 88

6 Conclusions **91**
6.1 Retrospective thoughts . . . 91
6.2 Future work . . . 92

Chapter 1
Introduction

Space: the final frontier.
These are the voyages of the starship Enterprise.
Its five-year mission: to explore strange new worlds,
to seek out new life and new civilizations,
to boldly go where no man has gone before.

Star Trek The Original Series, by Gene Roddenberry

Humans have always been attracted, in one way or the other, to the unknown. Maybe it is just because we are, as a species, curious by nature. Was it not for our incredible curiosity you would not be able to read these words printed on a piece of paper or displayed by a monitor.

A strange feeling of emptiness and incompleteness pervades our senses whenever faced with a situation that eludes our understanding. We must, or we feel like we must, close this gap by exploring, searching, and testing the environment and our surroundings, in the never-ending pursuit of knowledge.

Let us perform a thought experiment: imagine being faced with a closed box containing an unknown object. Our first reaction is to try and open it such that we can see what lies inside. But what if, for some reason, we cannot open the box, because it is locked and we do not possess the key or maybe because the object contained inside of it would be ruined by the process of opening it. We then must find a way of knowing what is inside of the box without actually opening it. We could start by measuring the dimensions of the box, weighting it or even shaking it (if this does process does not ruin the object within of course). In other words, we need to gather indirect measurements from the box by interacting with it and use the gathered observations to infer some properties of the object for which we do not have direct access.

The process of computing, from a set of observations, the causal factors that produced them is called an *inverse problem*. Inverse problems are very fascinating because they represent the core value of curiosity, meaning the search for the unknown.

An interesting class of inverse problems is the one related to *tomography*, which is the process of imaging slices of an object by gathering measurements from any kind of penetrating waves that interact with the object. Tomography is used to solve different kinds of imaging problems in different scientific areas, ranging from atomic to planetary scale.

To better introduce the context and motivation of this thesis, in the next sections, we will talk about seismic tomography and how the imaging techniques

used in this field can be extended to related tomographic problems such as medical ultrasound tomography. We will then talk about the Julia programming language and how it is set to solve the two-language problem in the scientific computing field. Finally, we will give an outline of the contents of this thesis by briefly introducing each chapter.

1.1 Seismic tomography and full waveform inversion

Seismic tomography [1] utilizes data recorded at seismic stations on the surface of the Earth to infer the structure and material properties of the subsurface by solving a tomographic inverse problem. Many techniques have been developed over the years to tackle this challenging problem, the main categories being ray tomography[2] and, more recently, full waveform inversion (FWI) [3]. In most of this thesis, we will use as reference for the concepts regarding FWI in the context of seismic tomography the book 'Full Seismic Waveform Modelling and Inversion' [4] and for the concepts regarding inverse problems the 'Lecture Notes on Inverse Theory' [5] both by Andreas Fichtner. The term 'full waveform inversion' comes from the fact that this method uses the complete recorded seismograms, meaning not only the phase information or the arrival times of the different waves but also their amplitudes. In practice, this is not always the case: for example, surface waves are commonly ignored in exploration geophysics applications, and amplitudes are often disregarded in global tomographic reconstructions.

Ray-based inversion utilizing arrival times of seismic waves can nowadays be routinely used thanks to its small requirements in terms of computational resources, which makes it a very fast method. On the other hand, the resolution and quality of resulting images are substantially limited in comparison to FWI. FWI, in contrast, can provide much higher resolution images by exploiting most of the information contained in the recorded seismograms instead of only the first arrival time. These capabilities, though, come at the cost of being computationally much more expensive and with a more complex workflow.

1.2 Medical ultrasound tomography

Medical ultrasound tomography (or ultrasound computed tomography) is a type of tomography utilizing ultrasound waves for imaging and it is mostly used for soft tissue medical imaging. The object, submerged in water, is exposed to ultrasound waves transmitted by ultrasound transducers in the direction of the object and received with other or the same transducers. The data regarding the modulated ultrasound waves collected by the receiver transducers is then used to perform an image reconstruction of the interior parts of the object.

In recent years, the techniques used for imaging the solid Earth are increasingly employed also in the field of medical ultrasound tomography [6], since, apart from a drastic change of scale and frequency, the underlying physics are the same.

This allows for a transfer of knowledge across the two fields and permits us to apply seismic methods directly to help improve the results of medical imaging.

The work conducted in this thesis is motivated by the recent advances in applying FWI methods for medical ultrasound tomography. In particular, FWI methods coming from seismic tomography have been recently used to produce high-resolution images for transcranial ultrasound computed tomography [7] and breast imaging for cancer detection [8].

1.3 The two-language problem: Julia as a possible solution

Computational scientists and engineers that develop and maintain numerical code bases for scientific computations have been dealing for years with the so-called *two-language problem*: one high-level language (e.g. Python [9] or MATLAB [10]) for prototyping and visualization, and one low-level language (e.g. C [11] or FORTRAN [12]) for implementing efficient and fast production-level code. This means doubling the work by learning two programming languages and maintaining two different code bases, which may or may not interact with each other.

Julia [13] is a general-purpose, high-level, modern, interactive, and high-performance programming language, specifically designed for scientific computing. Julia aims at bridging the gap between high-level and low-level languages by being both easy-to-use by non-developers and very fast where high performance is needed. The language is based on Just-In-Time (JIT) compilation, meaning that code is compiled only when needed. This allows interactivity and interpreter-like behavior without sacrificing the performance of optimized and compiled code.

In recent years, Julia has seen growing success in the scientific community, especially in the field of computational physics. For this reason, many hardware companies such as NVIDIA and AMD started to aid the development of Julia packages for high-performance computing that could enable developers to write code for their specific architectures. Consequently, users started to develop specialized packages for high-performance computing written purely in Julia. The development of the solvers for FWI in this thesis was mostly possible because of recently developed packages for parallel stencil computations on CPUs and GPUs, namely `ParallelStencil.jl` and `ImplicitGlobalGrid.jl` [14]. These two packages are used to write device-agnostic multi-xPUs code (i.e. that can be run on both multiple CPUs and multiple GPUs), thus having to maintain and develop a single version of the solvers that can run on different device architectures.

1.4 Motivation and aims

This thesis aims at developing efficient yet user-friendly tools to perform acoustic FWI with a focus on medical ultrasound imaging, although the implemented

solvers can be used for a variety of different applications involving acoustic wave propagation.

The main motivation for this work is the current lack of availability of open-source software packages combining easiness of use, good performance, and scalability to address the amount of computation required by FWI methods. Our goal is to provide tools that can be used by non-experts in high-performance computing to tackle acoustic FWI for both educational and scientific research purposes, enabling the domain scientists with an open-source alternative that is efficient and scalable on modern massively-parallel architectures and clusters but can also run on a laptop if needed.

1.5 Outline

In this section, we outline the structure of this thesis and its contents by briefly introducing each chapter.

In chapter 2 we dive deep into the theoretical concepts of inverse problems, focusing our attention on deterministic acoustic full waveform inversion methods. We perform a case study on the acoustic wave equation and derive expressions for sensitivity kernels using the continuous adjoint method for both constant and variable density formulations.

In chapter 3 we describe the discretization method used to implement numerical solvers for the acoustic wave equation. The finite differences method (FD) is introduced and update schemes are described for forward and adjoint solvers, as well as expressions for the computation of sensitivity kernels with respect to model parameters. We also go over some technical details of the implementation, like checkpointing and xPU computing.

We perform numerical gradient checking and synthetic inversions using our solvers in chapter 4 to test their implementation. Finally, a case study is conducted to perform a real data inversion in the context of medical ultrasound tomography.

We show that our solvers are efficient and scalable in chapter 5 where numerous benchmarks are performed to assess their performance on a range of different devices, from consumer GPUs to professional ones and distributed high-performance computing clusters.

We conclude the thesis in chapter 6 with a retrospective and some final remarks on the work done, as well as outlining future work and problems that still need to be addressed.

Chapter 2

Theory

Don't drink and derive!

Unknown but obviously experienced author.

In this chapter, we will explore some theory-related concepts regarding this thesis. Although the focus and the goal of this thesis are mostly on implementation and numerical experiments, we will devote a fair amount of it to understanding the physics behind the problem of ultrasound full waveform inversion for medical imaging and talk about inverse problem theory in general, as well as diving into issues of more mathematical nature.

2.1 Forward and inverse problems

In this section, we will introduce the concept of forward and inverse problems, and how the two differ from each other but are related at the same time. We will not give a comprehensive study of inverse theory and limit ourselves to the scope of solving an inverse problem in the acoustic FWI setting.

Forward problem

A forward problem is usually given in the form

$$d = G(m), \tag{2.1}$$

where m are model parameters on which an operator G is applied to compute some observable quantity d. The forward problem consists of, given a suitable representation of m and G, to compute the observable d. This can be done analytically in the case that to operator G can be expressed as an explicit (closed-form) function of m, but it is often the case that such a representation is not possible or only given for a very simple problem. In most real-world scenarios, especially when dealing with problems related to physics, G is given as a differential equation and needs to be solved numerically.

An example of a forward problem is computing the trajectory of an asteroid approaching a planet with a certain initial velocity. In this case, some of the model parameters m would be the mass of the planet and the current distance and velocity of the asteroid, the operator G would be a combination of some Newton's laws for motion and universal gravitation, while the observable d would be the position of the asteroid in time.

A few things related to practical purposes should be noted. First of all, the observable d can only be computed (or measured) *discretely*, meaning we need to deal with a discrete number of observable quantities $\boldsymbol{d}$. In the case these observables come from measurements, we have a vector of observed values $\boldsymbol{d}^{\text{obs}}$ that come, for example, from some experiment or direct observation of a phenomenon in the real world. For the same reason, the model has to be discretized and thus the operator is now acting discretely on the model parameters. For this reason, the forward problem can be rewritten as

$$\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m}, \boldsymbol{c}), \tag{2.2}$$

where $\boldsymbol{c}$ is a vector of control parameters that influence the amount and type of observables $\boldsymbol{d}$ and are chosen accordingly to the problem to be solved. We will omit the dependency of the control parameters $\boldsymbol{c}$ for convenience, but it is important to keep in mind that they can have a huge influence on the amount of information that is gathered by the observables.

Another important thing to note is that various types of errors can be encountered: for example, while measuring observables $\boldsymbol{d}^{\text{obs}}$ or when approximating $\boldsymbol{G}$ with a numerical solver. That means that even if the true model $\boldsymbol{m}^{\text{true}}$ is given precisely, it is impossible to obtain a computed observable $\boldsymbol{d}$ that perfectly matches the observed data $\boldsymbol{d}^{\text{obs}}$.

Inverse problem

The inverse problem, as suggested in the name, deals with the recovery of an estimated model given the observed data generated by it

$$\boldsymbol{m}^{\text{est}} = \boldsymbol{G}^{-1}(\boldsymbol{d}^{\text{obs}}). \tag{2.3}$$

The task of solving an inverse problem is more problematic than one would think. First of all, a representation for the inverse operator $\boldsymbol{G}^{-1}$ usually does not even exist. Furthermore, there might be different models explaining the observed data equally well and, as explained above, the data might contain errors that would lead to a mistaken model estimation [15]. All of these issues and others which we will not discuss, make the task of finding a model $\boldsymbol{m}^{\text{est}}$ close to the real solution $\boldsymbol{m}^{\text{true}}$ that generated the observed data $\boldsymbol{d}^{\text{obs}}$ extremely difficult.

Returning to the example of the orbiting asteroid, an inverse problem might be finding the position and mass of the planed given the trajectory of the asteroid measured at some times. We can imagine that the trajectory of the asteroid might be the same for different combinations of position and mass of the planet. A lighter planet closer to the asteroid might perturb the trajectory of the asteroid equally as well as a heavier planet located further away.

The main difference between a forward and inverse problem is the uniqueness of the solution. We know that we can compute the trajectory of the planet given some sufficient model parameters, but it is very hard to determine how much data

we would need to estimate correctly the position and mass of the planet. Perhaps we would need not one, but multiple different asteroid trajectories. Or it might as well be that no matter how many we are given, we still would not be able to say precisely where and how heavy the planet is. This intrinsic characteristic of the inverse problem makes its treatment sometimes very obscure and subject to interpretation and subjectivity.

But why do we even care about inverse problems? The answer is simple: many tasks in physics deal with the study of processes and properties in a medium for which direct access is not possible. This is the case for the problem treated in this thesis, i.e. we are not allowed to take a sample of the tissue we want to study. For this reason, we are forced to observe some other quantities (in our case, acoustic pressure waves) by 'probing' the tissue and, hopefully, measuring some kind of 'response' from it.

In the next section, we will focus on some classes of methods used to solve inverse problems and, in particular, on deterministic general descent methods.

2.2 Methods for solving inverse problems

We can divide methods for solving inverse problems into two macro-categories: *probabilistic* methods and *deterministic* methods. Both methods are different but related to each other and are the result of two different philosophies originating from different perspectives on what can be regarded as a solution to an inverse problem.

It is not the scope of this thesis to make a detailed analysis of such methods as we will mostly focus on a specific deterministic optimization-based method for solving our inverse problem of interest. Nevertheless, it is useful to gain some insight into different methods and understand their advantages and disadvantages to make an informed choice on what is better suited for the problem at hand.

2.2.1 Probabilistic methods

The philosophy of probabilistic methods follows the Bayesian view of finding the solution to the inverse problem. Instead of focusing on the 'optimal' model that reproduces the observed data as best as possible, probabilistic methods aim to construct a probability density function (p.d.f.), called the *posterior* density, which represents the 'probability' that a certain model has, in fact, 'produced' the observed data. If we take the *maximum a posteriori model*, i.e. the model which has the maximum probability of being the 'true model' according to the resulting p.d.f., we can get a single 'solution' to the inverse problem (which can be useful for practical purposes), but we are not, in theory, limited to this when using probabilistic methods. In fact, one can recover multiple equally likely models and estimate their uncertainties as well, which makes probabilistic methods a very powerful class of methods for solving inverse problems.

More formally, following from Bayes' theorem for probability densities, the probability density of the model parameters $\boldsymbol{m}$ given the observed data $\boldsymbol{d}^{\text{obs}}$ is

$$\rho(\boldsymbol{m}|\boldsymbol{d}^{\text{obs}}) = \frac{\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m})\rho(\boldsymbol{m})}{\rho(\boldsymbol{d}^{\text{obs}})}, \tag{2.4}$$

where the other terms are described as follows:

- $\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m})$ is called the *prior in data space* (also named *likelihood function*) which encodes the prior knowledge about the data given a specific realization of model parameters.
- $\rho(\boldsymbol{m})$ is called the *prior in model space* which encodes the prior knowledge about model parameters.
- $\rho(\boldsymbol{d}^{\text{obs}})$ is called the *evidence* which encodes how well the data fits the assumptions giving all possible model parameters' realizations.

Prior in data space. The most important term in the right hand side of eq. (2.4) is $\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m})$. In simpler words, it describes how 'likely' our data matches a specific model. It is usually specified as some sort of 'difference' between the observed data $\boldsymbol{d}^{\text{obs}}$ and the data computed by solving the forward problem $\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m})$ on a specific realization of model parameters $\boldsymbol{m}$. The choice of the likelihood function depends on the type of errors in the observed data that we take into consideration. One of the most common choices is to consider normally distributed errors, hence:

$$\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m}) = c\; e^{-\frac{1}{2}(\boldsymbol{d}-\boldsymbol{d}^{\text{obs}})\boldsymbol{C}_{\text{noise}}^{-1}(\boldsymbol{d}-\boldsymbol{d}^{\text{obs}})}, \tag{2.5}$$

where $\boldsymbol{C}_{\text{noise}}^{-1}$ is the *covariance matrix* describing the covariance of the noise in the observed data, and c is a normalization constant. Returning the the example of inverting for the mass and position of a planet given the trajectory of an asteroid approaching it, we might approximately know the magnitude of the instrumental noise affecting the device recording the asteroid's trajectory data and consider that when constructing the covariance matrix.

Prior in model space. The term $\rho(\boldsymbol{m})$ is used to define some kind of prior information we have on the model parameters. Returning to the example as before, we might as well consider that the mass of the planet cannot be arbitrarily big, otherwise it would be a star which is out of the scope of our assumption (all of this knowledge is, in fact, encoded into the prior information that we have on the model). So we penalize some model parameters by constructing a suitable prior in model space $\rho(\boldsymbol{m})$ that effectively 'scales' the prior in data space $\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m})$. Say that we want to restrict the possible masses m of the planet to the range $[a, b] \subset \mathbb{R}$. Then the prior in model space we would use is going to be

$$\rho(m) = \begin{cases} \frac{1}{b-a} & \text{if } m \in [a, b] \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

Evidence. The last term is the evidence $\rho(\boldsymbol{d}^{\text{obs}})$ which is a term that scales the posterior probability density by a constant factor. It is often ignored in practice because it only scales the posterior by a fixed amount, hence it is not needed by most methods. Nevertheless, it is still a meaningful quantity that can be used, for example, to measure how well the hypotheses we have made (e.g. model discretization, forward modeling simplifications, etc...) match our measurements.

Posterior in model space. In the context of a probabilistic method, the posterior $\rho(\boldsymbol{m}|\boldsymbol{d}^{\text{obs}})$ is the solution to the inverse problem. It contains all the information needed to assess which models are more likely than others and their uncertainties. But how do we obtain this probability density? For a specific realization of model parameters $\boldsymbol{m}$ we need to compute the likelihood function (which involves solving a forward problem) and the prior in model space. Their multiplication gives the desired value of the posterior for the chosen model. We need to repeat the above procedure for all candidate models for which we want to know the posterior. This process of picking different models and assessing their posterior is called *sampling*.

Sampling the model space may be challenging due to its high dimensionality and usually requires a huge computational effort because, for each sample, we need to solve at least one forward model and we may need a very large amount of samples to accurately recover the posterior.

Sampling methods. The most simple sampling method is *grid sampling*, i.e. choosing the models and computing their posterior density by trying different combinations of uniformly spaced model parameters. This method becomes easily unfeasible in high-dimensional model spaces, so it is almost always never used apart for small 'toy' problems.

Importance sampling methods are a common choice to sample the model space more 'cleverly' by trying to focus the sampling in regions of the model space with higher relevance, hence making the required number of samples for accurate sampling smaller overall. There are many different importance sampling methods, some of them more complicated than others and some requiring more information or computations than others.

2.2.2 Deterministic methods

Deterministic methods, in contrast to probabilistic ones, will not give as the solution of the inverse problem a probability density of all possible model parameter choices. Instead, they focus on finding a single 'best' choice for model parameters minimizing some sort of *misfit* functional $\chi[\boldsymbol{d}; \boldsymbol{d}^{\text{obs}}]$, which, loosely speaking, measures the 'distance' between the observed data $\boldsymbol{d}^{\text{obs}}$ and the synthetic data $\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m})$ computed by solving the forward problem on some model parameters $\boldsymbol{m}$. This approach essentially involves finding the solution of the minimization

problem

$$\begin{aligned} \underset{\boldsymbol{m}}{\text{minimize}} \quad & \chi[\boldsymbol{d}; \boldsymbol{d}^{\text{obs}}] \\ \text{subject to} \quad & \boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m}). \end{aligned} \tag{2.7}$$

To task of solving the problem in (2.7) may be formidable for a variety of reasons. First of all, we have no guarantees that a unique minimizer exists, so there might be multiple models minimizing the misfit functional with the same minimum value. We call one of the solutions of problem (2.7) an *optimal* model $\hat{\boldsymbol{m}}$.

Now, say that our observed data was obtained by solving the forward problem with the true model $\boldsymbol{m}^{\text{true}}$. For the same reason, we have no guarantees that a solution $\hat{\boldsymbol{m}}$ to eq. (2.7) is even close to the true model $\boldsymbol{m}^{\text{true}}$, because we base our optimization problem on minimizing the 'closeness' of $\boldsymbol{d}^{\text{obs}}$ to the synthetic data generated by $\hat{\boldsymbol{m}}$ and not of $\boldsymbol{m}^{\text{true}}$ to $\hat{\boldsymbol{m}}$ itself.

General descent methods

Let us now focus on some practical aspects of finding a solution to the problem (2.7). Finding it analytically is (almost always) out of the question, so we have to rely on some iterative optimization-based method. Most of these methods are designed so that, given an initial model $\boldsymbol{m}_0$, a sequence of models $\{\boldsymbol{m}_n\}_{n\in\mathbb{N}}$ is constructed such that

$$\lim_{n\to\infty} \boldsymbol{m}_n = \hat{\boldsymbol{m}}. \tag{2.8}$$

We have to note that the convergence of the sequence $\{\boldsymbol{m}_n\}_{n\in\mathbb{N}}$ to $\hat{\boldsymbol{m}}$ is usually only guaranteed, for every choice of the initial model $\boldsymbol{m}_0$, if the objective function to minimize is *convex* with respect to the model parameters. Since the misfit functional χ is usually not convex, this means that a suitable choice for the initial model $\boldsymbol{m}_0$ is crucial in order to find a global optimum and not get stuck in some local optima.

Leaving the problem of convexity aside for now, we still need to give a clear procedure on how to find the sequence $\{\boldsymbol{m}_n\}_{n\in\mathbb{N}}$. This is a task for general descent methods that need information on the gradients (or higher order derivatives) of the misfit functional with respect to model parameters. The model update procedure to construct the sequence $\{\boldsymbol{m}_n\}_{n\in\mathbb{N}}$ can be summarized in the following way:

1. Choose an initial model $\boldsymbol{m}_0$ and set $i = 0$,
2. Compute the misfit $\chi(\boldsymbol{m}_i)$ and a the descent direction $\boldsymbol{h}_i$ associated to $\boldsymbol{m}_i$,
3. Update the model with $\boldsymbol{m}_{i+1} = \boldsymbol{m}_i + \gamma_i \boldsymbol{h}_i$ with a suitable step length γ_i such that $\chi(\boldsymbol{m}_{i+1}) < \chi(\boldsymbol{m}_i)$,
4. Set $i \to i+1$ and go back to step 2 if $\chi(\boldsymbol{m}_{i+1}) > \epsilon$ with ϵ a convergence constant that depends on the noise of the data.

The most crucial and computationally expensive step in the procedure is step 2, where the direction $\boldsymbol{h}_i$ needs to be computed. Depending on the specific method, this direction usually involves the computation of the gradients $\nabla_{\boldsymbol{m}}\chi(\boldsymbol{m}_i)$ or even higher order derivatives like the Hessian matrix.

The most simple method is the so-called *steepest descent* method where the descent direction $\boldsymbol{h}_i = -\nabla_{\boldsymbol{m}}\chi(\boldsymbol{m}_i)$ is chosen as the negative of the gradient. The method used in the context of this thesis is called L-BFGS [16](Limited memory BFGS) which is a quasi-Newton method approximating the current step Hessian matrix using information on a limited amount of previous iterations steps instead of storing the whole Hessian matrix approximation like for the original BFGS method.

We will not go into the details on how $\boldsymbol{h}_i$ is computed or step 3 is performed since it is out of the scope of this thesis. Nevertheless, the choice of a suitable descent method is severely tied to the specific inversion problem, the choice of the misfit functional, and the amount of computational resources available since some methods require more misfit and gradients evaluations than others to converge. Usually, the choice is made by confronting different methods directly on the problem instance and picking the one that seems more successful in minimizing the misfit functional.

Regarding computational costs of gradient descent methods versus general probabilistic methods, we can say that the former need far less computations of the forward operator than the latter in order to 'converge'[1], given that the computation of gradients is done efficiently (which is what we will focus on section 2.3).

Choice of misfit functional

The choice of a misfit functional for our deterministic method is dictated by the forward operator, the noise on the observed data, and the characteristics of the model we would like to estimate. A common choice is the squared L_2-norm of the difference between the synthetic data and observed data

$$\chi_{L_2}[\boldsymbol{G}(\boldsymbol{m}); \boldsymbol{d}^{\text{obs}}] := \frac{1}{2}\|\boldsymbol{G}(\boldsymbol{m}) - \boldsymbol{d}^{\text{obs}}\|_2^2, \tag{2.9}$$

where the norm $\|\cdot\|_2$ is defined depending on the norm definition in the data space. In the continuous case, this is an integral in space and time, while in the discrete case, it is an inner vector product. The factor of one-half in the definition is purely a convention and its usefulness will be revealed later on in the case study at section 2.4. Other similar definitions come from generalizing the concept of the norm induced by an inner product. In the discrete case, we

[1] Converge is not really the proper wording choice when talking about probabilistic methods. In this context, 'convergence' can be substituted with 'good enough' sampling of the model space.

have that an inner product between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be expressed by a positive-definite matrix $\boldsymbol{A}$ such that

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle_{\boldsymbol{A}} = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}, \tag{2.10}$$

and using the definition of the norm induced by the inner product we have

$$\|\boldsymbol{x}\|_{\boldsymbol{A}} \coloneqq \langle \boldsymbol{x}, \boldsymbol{x} \rangle_{\boldsymbol{A}} = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}. \tag{2.11}$$

If the matrix $\boldsymbol{A}$ is the identity matrix $\boldsymbol{I}$, the inner product is just the classic dot product between vectors and the norm $\|\cdot\|_{\boldsymbol{I}}$ is just the L_2-norm for the vector space. This leads to the definition of a misfit using the inner product defined by an arbitrary matrix. One of the most common choices is the matrix $\boldsymbol{C}_{\text{noise}}^{-1}$ which is the inverse of the covariance matrix containing the noise in the observed data. Using this matrix we can define the misfit

$$\chi_{\boldsymbol{C}_{\text{noise}}^{-1}}[\boldsymbol{G}(\boldsymbol{m}); \boldsymbol{d}^{\text{obs}}] \coloneqq \frac{1}{2} \|\boldsymbol{G}(\boldsymbol{m}) - \boldsymbol{d}^{\text{obs}}\|^2_{\boldsymbol{C}_{\text{noise}}^{-1}}. \tag{2.12}$$

In practice, we do not know the matrix $\boldsymbol{C}_{\text{noise}}^{-1}$ precisely, but a fair approximation can be constructed by analyzing the observed data. A link between the misfit defined in eq. (2.12) and the prior in data space $\rho(\boldsymbol{d}^{\text{obs}}|\boldsymbol{m})$ for the probabilistic case can be established by considering the data to be affected by Gaussian noise with a covariance matrix equal to $\boldsymbol{C}_{\text{noise}}$. In this case, the prior in data space would just be the exponential of $-\chi_{\boldsymbol{C}_{\text{noise}}^{-1}}$.

Regularization

One of the most common approaches to deal with the problem of non-convexity of the misfit functional is to add a *regularization* term $\mathcal{R}[\boldsymbol{m}; \boldsymbol{m}^{\text{prior}}]$ to the misfit, involving some prior knowledge of the model $\boldsymbol{m}^{\text{prior}}$, much of in the spirit of choosing the model prior in the probabilistic approach[2]. The misfit is then redefined as

$$\chi[\boldsymbol{d}, \boldsymbol{m}; \boldsymbol{d}^{\text{obs}}, \boldsymbol{m}^{\text{prior}}] = \chi[\boldsymbol{d}; \boldsymbol{d}^{\text{obs}}] + \mathcal{R}[\boldsymbol{m}; \boldsymbol{m}^{\text{prior}}], \tag{2.13}$$

where we fully explicit the dependence of the misfit on both the model and data. We will omit the dependence on the constants $\boldsymbol{d}^{\text{obs}}$ and $\boldsymbol{m}^{\text{prior}}$ to slim the notation when they are clear from the context. If we replace $\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m})$ using the forward operator, we get

$$\chi[\boldsymbol{G}(\boldsymbol{m}), \boldsymbol{m}] = \chi[\boldsymbol{G}(\boldsymbol{m})] + \mathcal{R}[\boldsymbol{m}], \tag{2.14}$$

in which now is clear that the misfit depends on the forward operator $\boldsymbol{G}$ as well as a specific choice of the model parameters $\boldsymbol{m}$.

[2] Although with some philosophical differences since the prior in model space $\rho(\boldsymbol{m})$ is a probability density chosen by hypothesis, while the regularization term $\mathcal{R}[\boldsymbol{m}; \boldsymbol{m}^{\text{prior}}]$ is usually chosen arbitrarily.

Similar reasoning can be applied to the definition of the regularization term $\mathcal{R}(\boldsymbol{m})$ by defining it as a suitably chosen norm multiplied by a scaling factor α, called *regularization parameter*, which controls the strength of the regularization term with respect to the data misfit term. A list of well-known regularization terms is given:

- $\|\boldsymbol{m}\|^2$ is the zeroth order Tikhonov regularization,
- $\|\operatorname{grad} \boldsymbol{m}\|^2$ is the first order Tikhonov regularization (higher orders can be derived similarly),
- $\|\operatorname{grad} \boldsymbol{m}\|_1$ is the total variation regularization (defined with the L_1-norm),

Once again, the norm $\|\cdot\|$ can be defined as the standard L_2-norm or a general norm induced by a vector product. An example of a misfit used in practice could be combining eq. (2.12) with a zeroth order Tikhonov regularization applied to the difference between the model and a prior model

$$\chi[\boldsymbol{G}(\boldsymbol{m}), \boldsymbol{m}] = \frac{1}{2}\|\boldsymbol{G}(\boldsymbol{m}) - \boldsymbol{d}^{\text{obs}}\|^2_{\boldsymbol{C}^{-1}_{\text{noise}}} + \frac{\alpha}{2}\|\boldsymbol{m} - \boldsymbol{m}^{\text{prior}}\|^2_{\boldsymbol{C}^{-1}_{\text{prior}}}, \tag{2.15}$$

where $\boldsymbol{C}^{-1}_{\text{prior}}$ is the inverse of the covariance matrix in model space. Similarly to the prior in data space, the prior in model space $\rho(\boldsymbol{m})$ is linked to the definition of eq. (2.15) in the case we consider the *a priori* most likely model $\boldsymbol{m}^{\text{prior}}$ to be affected by Gaussian noise with covariance matrix $\boldsymbol{C}_{\text{prior}}$.

Although there exist other misfit and regularization functionals (see [4]), for the context of this thesis we will limit ourselves to the aforementioned cases as they are well suited for the problem at hand.

Link between probabilistic and deterministic. As we have quickly mentioned above, the choice of the misfit functional in eq. (2.13) can be linked to the case of having Gaussian priors in model and data space from the probabilistic point of view. It turns out that, in this case, if the forward operator $\boldsymbol{G}$ is linear, the posterior in model space too is a Gaussian probability density. In this particular case, solving the minimization problem (2.7) is equivalent of finding the *maximum likelihood model* for the probabilistic inversion, i.e.

$$\hat{\boldsymbol{m}} = \underset{\boldsymbol{m}}{\operatorname{argmax}} \frac{\rho(\boldsymbol{m}|\boldsymbol{d}^{\text{obs}})}{\rho_h(\boldsymbol{m})}, \tag{2.16}$$

where $\rho_h(\boldsymbol{m})$ expresses the homogeneous probability density in the model space[3]. This means that, under the above assumptions, the two methods are practically equivalent, if not for the fact that we have access to the whole posterior probability distribution from the probabilistic point of view, while we only have the reconstructed most likely model $\hat{\boldsymbol{m}}$ from the deterministic point of view.

[3] This term is used to scale the posterior probability density to take into consideration possible changes of coordinate system.

2.3 Adjoint methods

In the previous section, we have seen the importance of efficiently computing derivatives with respect to model parameters of some misfit functional χ. First-order (and sometimes second-order) gradients are at the core of optimization-based methods for solving inverse problems and are also required by some probabilistic methods that rely on these gradients to sample the model space more effectively.

The difficulty in computing gradients of a general misfit functional is that dependencies with the model parameters $\boldsymbol{m}$ are usually not direct. Instead, the relation between model parameters and the misfit functional is given by some physical quantities $\boldsymbol{u}(\boldsymbol{m})$ that depend on the model $\boldsymbol{m}$ and are used to compute the misfit $\chi(\boldsymbol{u}(\boldsymbol{m}))$. These physical quantities $\boldsymbol{u}(\boldsymbol{m})$ are usually observable quantities we can measure using sensors or instruments. For example, in the case of elastic waves propagating in a 3D medium, $\boldsymbol{u}$ would represent the displacement field which in 3D has three components $\boldsymbol{u} = [u_x, u_y, u_z]^T$.

To compute these physical quantities we solve the forward problem $\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m})$ where $\boldsymbol{G}$ is used to 'extract' the synthetic data $\boldsymbol{d}$ from those physical quantities. In fact, we can reformulate the forward problem as

$$\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m}) = \boldsymbol{D}(\boldsymbol{u}(\boldsymbol{m})), \tag{2.17}$$

where $\boldsymbol{D}$ is an operator acting on the physical quantities $\boldsymbol{u}$, and $\boldsymbol{u}$ is the solution to the physical relationship

$$\boldsymbol{L}[\boldsymbol{u}, \boldsymbol{m}] = \boldsymbol{f}(\boldsymbol{m}), \tag{2.18}$$

where $\boldsymbol{L}$ is an operator and $\boldsymbol{f}$ represents external forces which may depend on model parameters. For our purposes, eq. (2.18) will represent some IBVP, hence solving a PDE with specified initial and boundary conditions. The operator $\boldsymbol{D}$ just selects some components of $\boldsymbol{u}$ which match the observed data $\boldsymbol{d}^{\text{obs}}$ to produce synthetic data $\boldsymbol{d}$ such that the two are 'comparable' to each other. That comparison is done via the misfit functional $\chi(\boldsymbol{u})$, which we recall as some kind of 'difference' measure between synthetic and observed data.

To compute the gradient of the misfit functional with respect to a single model parameter m_i, we could simply use a second-order finite difference approximation

$$\frac{\partial \chi}{\partial m_i} \approx \frac{\chi(\boldsymbol{u}(\boldsymbol{m} + \Delta m_i \boldsymbol{e_i})) - \chi(\boldsymbol{u}(\boldsymbol{m} - \Delta m_i \boldsymbol{e_i}))}{2\Delta m_i}, \tag{2.19}$$

where Δm_i is a perturbation of the i-th model parameter and $\boldsymbol{e_i}$ is the unit vector in the i-th direction. Following this method, to compute a derivative in a single direction we would need to evaluate the misfit functional $2n$ times (where n is the number of model parameters) which means solving the forward problem as many times. This becomes soon unfeasible for big values of n or high-cost misfit evaluations.

To efficiently compute $\partial\mathcal{X}/\partial\boldsymbol{m}$ we need *adjoint methods*. In adjoint methods the connection between $\boldsymbol{u}$ and $\boldsymbol{m}$ is captured by the so-called *adjoint field*. The adjoint field is computed through solving the adjoint forward problem which has almost the same cost as solving the original forward problem. After that, the gradient of the misfit can be recovered by 'correlating' the adjoint field and some $\boldsymbol{u}$-related quantity. Essentially, the cost of computing $\partial\mathcal{X}/\partial\boldsymbol{m}$ becomes roughly the same as solving two forward problems[4].

In the following sections, we will discuss two types of adjoint methods: *continuous* and *discrete* adjoint methods. While related to each other, they follow two different philosophies, namely *first optimize then discretize* and *first discretize then optimize* respectively. It is useful to note that the two methods are usually equivalent, although understanding the differences in the case they are not equivalent is essential because the discretization choice that naturally occurs when solving inverse problems in practice can lead to substantial errors if not taken into consideration.

For the next sections regarding the continuous and discrete adjoint method, we will mostly follow [5] for the definitions and theoretical concepts.

2.3.1 Continuous adjoint method

The continuous adjoint method's point of view is that of considering the continuous version of the forward problem $\boldsymbol{d} = \boldsymbol{G}(\boldsymbol{m})$ where $\boldsymbol{m}(\boldsymbol{x})$ is a continuous vector field of model parameters. This method has the advantage of being independent of the model discretization and it is mainly used when needing continuous derivatives of the misfit functional $\mathcal{X}$, usually in the form of sensitivity kernels, which we will introduce later on in this section.

Since the misfit in this case is a functional, hence a function of functions, we need to generalize the definition of the derivative of a function to the derivative of a functional. For this purpose, we introduce the *Frechét derivative* or *functional derivative* of a functional $f(\boldsymbol{u})$ at $\boldsymbol{u}$ in the direction $\delta\boldsymbol{u}$ as

$$\delta f(\boldsymbol{u};\delta\boldsymbol{u}) = \lim_{\varepsilon\to 0}\frac{1}{\varepsilon}(f(\boldsymbol{u}+\varepsilon\delta\boldsymbol{u}) - f(\boldsymbol{u})), \tag{2.20}$$

where the direction $\delta\boldsymbol{u}$ describes a perturbation of the vector-valued function $\boldsymbol{u}$. To be precise, this definition requires the *uniform convergence* of the limit, which is usually possible for the misfit functionals we are interested in. Note that this definition is similar to the usual definition of directional derivative for regular functions. The definition of the functional derivative shares many common properties with the usual derivative, such as linearity, product and chain rule, which we will all use in this section to derive the continuous adjoint method. We will sometimes use the short notation $\delta f(\delta\boldsymbol{u}) = \delta f(\boldsymbol{u};\delta\boldsymbol{u})$ when we consider f to be a functional that depends only on the function $\boldsymbol{u}$. We will also use the notation $\delta_{\boldsymbol{u}} f$ to denote the 'functional gradient' of f, which is a vector of functional derivatives

[4]With some caveats that are elaborated more in detail in chapter 3.

for all ‘unitary’ directions $\delta\boldsymbol{u}$ (this means effectively changing just a component of the function $\boldsymbol{u}$).

We will now consider misfit functionals that can be written in the form

$$\begin{aligned}\chi[\boldsymbol{u}(\boldsymbol{m}),\boldsymbol{m}] &= \int_0^T \int_\Omega \chi_u(\boldsymbol{u}(\boldsymbol{m}))\; d\boldsymbol{x}\; dt + \int_\Omega \chi_m(\boldsymbol{m})\; d\boldsymbol{x} \\ &= \langle \chi_u(\boldsymbol{u})\rangle + \langle \chi_m(\boldsymbol{m})\rangle_\Omega,\end{aligned} \tag{2.21}$$

where $\chi_u(\boldsymbol{u}(\boldsymbol{m}))$ and $\chi_m(\boldsymbol{m})$ are two functions (not functionals). The notation $\langle\cdot\rangle_T, \langle\cdot\rangle_\Omega$ is used for the integrals in time and space respectively, and $\langle\cdot\rangle = \langle\langle\cdot\rangle_\Omega\rangle_T$ is used for double integration in time and space. The definitions of misfit functionals that we introduced in section 2.2.2 dealt with the discretized observed data $\boldsymbol{d}^{obs}$ and the discretized model $\boldsymbol{m}$, so they need to be slightly adapted to the continuous case. This usually involves defining $\chi_u(\boldsymbol{u}(\boldsymbol{m}))$ and $\chi_m(\boldsymbol{m})$ to include delta functions picking only the values of $\boldsymbol{u}$ and $\boldsymbol{m}$ in time and space where the measurements are performed. Notice that the role of the integrals in eq. (2.21) is the same as the norm in the definitions of section 2.2.2. In fact, integrals usually pop up when generalizing norms in infinite-dimensional spaces.

Now, we are interested in the functional derivative of the misfit functional at $\boldsymbol{m}$ in a direction $\delta\boldsymbol{m}$, so by applying the chain rule we get

$$\begin{aligned}\delta\chi[\boldsymbol{u}(\boldsymbol{m}),\boldsymbol{m};\delta\boldsymbol{m}] &= \delta\chi(\boldsymbol{u};\delta\boldsymbol{u}) + \delta\chi(\boldsymbol{m};\delta\boldsymbol{m}) \\ &= \langle \delta\chi_u(\boldsymbol{u};\delta\boldsymbol{u})\rangle + \langle \delta\chi_m(\boldsymbol{m};\delta\boldsymbol{m})\rangle_\Omega,\end{aligned} \tag{2.22}$$

where the perturbation $\delta\boldsymbol{u} = \delta\boldsymbol{u}(\boldsymbol{m};\delta\boldsymbol{m})$ is the functional derivative of $\boldsymbol{u}$ in the direction $\delta\boldsymbol{m}$. The difficulty in computing $\delta\boldsymbol{u}$ (which requires solving a forward problem for each direction $\delta\boldsymbol{m}$) is what prohibits us from using eq. (2.22) to efficiently compute gradients, so we need to find a way to remove the dependency from $\delta\boldsymbol{u}$ in eq. (2.22). To do this we can use the physical relationship if eq. (2.18) and define the term $\boldsymbol{g}$ as

$$\boldsymbol{g}[\boldsymbol{u},\boldsymbol{m}] := \boldsymbol{L}[\boldsymbol{u},\boldsymbol{m}] - \boldsymbol{f}(\boldsymbol{m}) = \boldsymbol{0}, \tag{2.23}$$

which vanishes. If we take the functional derivative of $\boldsymbol{g}$ in the direction of $\delta\boldsymbol{m}$ we obtain, by the chain rule,

$$\delta\boldsymbol{g}(\boldsymbol{u}(\boldsymbol{m}),\boldsymbol{m};\delta\boldsymbol{m}) = \delta\boldsymbol{L}(\delta\boldsymbol{u}) + \delta\boldsymbol{g}(\delta\boldsymbol{m}) = 0. \tag{2.24}$$

We can now take the integral in both space and time, multiply by an arbitrary test function $\boldsymbol{\lambda}$ and add the expression (which still vanishes) to eq. (2.22) and get

$$\begin{aligned}\delta\chi[\boldsymbol{u}(\boldsymbol{m}),\boldsymbol{m};\delta\boldsymbol{m}] = &\langle \delta\chi_u(\delta\boldsymbol{u})\rangle + \langle \boldsymbol{\lambda}\cdot\delta\boldsymbol{L}(\delta\boldsymbol{u})\rangle \\ &+ \langle \boldsymbol{\lambda}\cdot\delta\boldsymbol{g}(\delta\boldsymbol{m})\rangle \\ &+ \langle \delta\chi_m(\delta\boldsymbol{m})\rangle_\Omega.\end{aligned} \tag{2.25}$$

Since $\chi_u(\delta\boldsymbol{u})$ is a function, the functional derivative is the equivalent of the directional derivative, such that

$$\delta\chi_u(\delta\boldsymbol{u}) = \delta\boldsymbol{u} \cdot \delta_{\boldsymbol{u}}\chi_u. \tag{2.26}$$

We now need to find a way to isolate $\delta\boldsymbol{u}$ from the expression $\boldsymbol{\lambda} \cdot \delta\boldsymbol{L}(\delta\boldsymbol{u})$. To do this, we introduce the Jacobian operator $\boldsymbol{J}(\delta\boldsymbol{u}) = \delta\boldsymbol{L}(\delta\boldsymbol{u})$ and its adjoint operator $\boldsymbol{J}^\dagger(\boldsymbol{\lambda})$ implicitly defined by the following relation

$$\langle\boldsymbol{\lambda} \cdot \boldsymbol{J}(\delta\boldsymbol{u})\rangle = \langle\delta\boldsymbol{u} \cdot \boldsymbol{J}^\dagger(\boldsymbol{\lambda})\rangle, \tag{2.27}$$

which should hold for every $\delta\boldsymbol{u}$ and every $\boldsymbol{\lambda}$. Using eq. (2.26) and eq. (2.27) we can rewrite eq. (2.25) as

$$\begin{aligned} \delta\chi[\boldsymbol{u}(\boldsymbol{m}), \boldsymbol{m}; \delta\boldsymbol{m}] = &\langle\delta\boldsymbol{u} \cdot (\delta_{\boldsymbol{u}}\chi_u + \boldsymbol{J}^\dagger(\boldsymbol{\lambda}))\rangle \\ &+ \langle\boldsymbol{\lambda} \cdot \delta\boldsymbol{g}(\delta\boldsymbol{m})\rangle \\ &+ \langle\delta\chi_m(\delta\boldsymbol{m})\rangle_\Omega. \end{aligned} \tag{2.28}$$

We now choose the arbitrary function $\boldsymbol{\lambda}$ to satisfy the following equation, which we will call the *adjoint equation*

$$\boxed{\boldsymbol{J}^\dagger(\boldsymbol{\lambda}) = -\delta_{\boldsymbol{u}}\chi_u}\,, \tag{2.29}$$

so we can remove the first term on the right-hand side of eq. (2.28) and obtain an expression for the functional derivative of the misfit functional

$$\boxed{\delta\chi[\boldsymbol{u}(\boldsymbol{m}), \boldsymbol{m}; \delta\boldsymbol{m}] = \langle\boldsymbol{\lambda} \cdot \delta\boldsymbol{g}(\delta\boldsymbol{m})\rangle + \langle\delta\chi_m(\delta\boldsymbol{m})\rangle_\Omega} \tag{2.30}$$

that can be directly computed for a specific model perturbation $\delta\boldsymbol{m}$.

In the special case that the operator $\boldsymbol{L}$ is linear with respect to the observables $\boldsymbol{u}$, we have that $\boldsymbol{J} = \boldsymbol{L}$ and by invoking the adjoint operator $\boldsymbol{L}^\dagger$ of $\boldsymbol{L}$, eq. (2.29) can be simplified to

$$\boldsymbol{L}^\dagger(\boldsymbol{\lambda}) = -\delta_{\boldsymbol{u}}\chi_u. \tag{2.31}$$

A few observations should be made regarding the continuous adjoint method. First of all, its abstractness, mainly due to the introduction of functional derivatives, can be quite challenging especially if the reader has no experience in basic functional analysis. In the derivation above, we have been quite loose with the definitions and the steps we applied may not be very rigorous mathematically speaking. Fortunately, once we use the continuous adjoint method to derive adjoint equations and expressions for first-order derivatives in practice, i.e. by choosing a specific physical operator $\boldsymbol{L}$ and a misfit functional χ, most of the functional derivatives can be recast as common derivatives, making the task much more manageable.

The method, in its general form, does not provide the steps necessary to find the adjoint operator $\boldsymbol{J}^\dagger$ explicitly. In practice, this is usually done by manipulating the expression $\langle\boldsymbol{\lambda} \cdot \delta\boldsymbol{L}(\delta\boldsymbol{u})\rangle$, usually by means of integration by parts or

similar techniques. This can be challenging depending on the properties of the operator $\boldsymbol{L}$ and care must be taken to correctly recover the adjoint equation. Fortunately, for the context of this thesis, the operator $\boldsymbol{L}$ describes a linear PDE with respect to the observables $\boldsymbol{u}$, so the simplification of eq. (2.31) can be used.

Sensitivity kernels The expression for the functional derivative of the misfit functional of eq. (2.30) gives use a scalar value for a determined perturbation of the model parameters $\delta\boldsymbol{m}$. What we usually need in order to compute spatial gradients are *sensitivity kernel* $\boldsymbol{K}(\boldsymbol{x})$, also called in this context *Frechét kernel*. These are defined as the volumetric densities of the Frechét derivative as follows

$$\boldsymbol{K}(\boldsymbol{x}) = \frac{d}{dV}\delta_{\boldsymbol{m}}\chi = \langle \boldsymbol{\lambda} \cdot \delta_{\boldsymbol{m}}\boldsymbol{g} \rangle_T + \delta_{\boldsymbol{m}}\chi_m \tag{2.32}$$

so we can rewrite the functional derivative $\delta\chi(\boldsymbol{u}, \boldsymbol{m}; \delta\boldsymbol{m})$ as the integral in space of the sensitivity kernel

$$\delta\chi(\boldsymbol{u}, \boldsymbol{m}; \delta\boldsymbol{m}) = \langle \boldsymbol{K}\delta\boldsymbol{m} \rangle_\Omega. \tag{2.33}$$

The sensitivity kernel is a vector field with the same number of components as the model parameters' vector field and describes how much the misfit functional is affected by a change in model parameters at the position $\boldsymbol{x}$ in space. The sensitivity kernels are especially useful when we have to compute gradients with respect to model parameters that have been discretized. Suppose that we have discretized the model parameters using a set of basis functions b_j with some parameters a_{ij} that have been determined such that

$$m_i(\boldsymbol{x}) = \sum_{j=1}^{N} a_{ij} b_j(\boldsymbol{x}), \forall i \in \{1, \ldots, M\}, \tag{2.34}$$

where N is the number of discretization points, M is the number of continuously defined model parameters (i.e. the components of the vector field $\boldsymbol{m}$) and $m_i(\boldsymbol{x})$ is the value of the i-th model parameter at position $\boldsymbol{x}$. With this discretization in place, we can compute the derivatives of the misfit functional with respect to the parameter a_{ij} in the following way

$$\frac{\partial\chi}{\partial a_{ij}} = \langle K_i \; b_j \rangle_\Omega, \tag{2.35}$$

where $K_i(\boldsymbol{x})$ is the i-th component of the vector field $\boldsymbol{K}(\boldsymbol{x})$. This can be interpreted as the projection of the sensitivity kernel onto the basis function. Using the above expression, we can easily compute gradients with respect to model parameters (or more precisely to the parameters a_{ij} that, together with the basis function, represent the discretized model parameters space) for the sake of using general descent methods as described in section 2.2.2.

We have to note that finding an analytic expression for the sensitivity kernel is usually impossible because the adjoint field $\boldsymbol{\lambda}$ is only given implicitly as the solution to the adjoin equation. This means that, in practice, one needs to solve eq. (2.29) *numerically*, meaning discretization is needed.

Summary. To summarize, the continuous adjoint method gives an expression for the sensitivity kernel $\boldsymbol{K}$ which can then be used to compute gradients with respect to model parameters after choosing a proper discretization. This involves the following steps:

1. find an expression for the adjoint operator $\boldsymbol{J}^\dagger$ using eq. (2.27),
2. find an expression for the gradient $\delta_{\boldsymbol{u}} \chi_u$,
3. solve eq. (2.29) (or eq. (2.31) in the case that $\boldsymbol{L}$ is linear with respect to the observables quantities $\boldsymbol{u}$) to find the adjoint vector $\boldsymbol{\lambda}$,
4. find an expression for the gradients $\delta_{\boldsymbol{m}} \boldsymbol{g}$ and $\delta_{\boldsymbol{m}} \chi_m$,
5. integrate in time to find an expression for the sensitivity kernel $\boldsymbol{K}$ using eq. (2.32),
6. compute gradients of the misfit functional with respect to the discretized model parameters by integrating in space the sensitivity kernel with the basis functions using eq. (2.35).

Steps 1, 2, and 4 can be done analytically using the continuous definitions of $\boldsymbol{L}, \boldsymbol{g}$ and χ, while steps 3, and 5 usually need to be done numerically. Step 6 is the only part where we need to use the discretization. We can see that the continuous adjoint method does not restrict ourselves to a particular discretization *a priori*, but rather develops all the tools and expression continuously and only requires a proper discretization at the end.

2.3.2 Discrete adjoint method

Although we could be done talking about the adjoint method after describing the continuous adjoint method, there are settings in which the discretization must be done *a priori* and have a certain impact on how gradients of the misfit functional are computed. Following the principle of 'first discretize then optimize', we will introduce the discrete adjoint method, which requires the discretization of the model parameters and the forward operator to be done before obtaining expressions for the gradients. Let us recall the physical relationship of eq. (2.18)

$$\boldsymbol{L}[\boldsymbol{u}, \boldsymbol{m}] = \boldsymbol{f}(\boldsymbol{m}) \tag{2.36}$$

where, in this context, $\boldsymbol{L}$ is a *discrete* differential operator acting on both the observables and the model parameters, while $\boldsymbol{f}$ is some discrete external force term that may depend on the model parameters.

The idea behind the adjoint method is to use eq. (2.36) to simplify the computations of the gradients of the misfit function[5] with respect to model parameters,

[5] In the discrete case, the misfit is not a functional anymore, but just a function.

similarly to what we have seen for the continuous adjoint method

$$\frac{d\chi}{d\boldsymbol{m}} = \frac{\partial \chi}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{d\boldsymbol{m}} + \frac{\partial \chi}{\partial \boldsymbol{m}}, \tag{2.37}$$

where we have used the total derivatives $d/d\boldsymbol{m}$ to express the gradients in the general case that the misfit function is in the form $\chi[\boldsymbol{u}(\boldsymbol{m}), \boldsymbol{m}]$[6]. The difficulty in eq. (2.37) is the presence of the term $d\boldsymbol{u}/d\boldsymbol{m}$ which is hard to compute given that the dependency of the observables $\boldsymbol{u}$ on the model parameters $\boldsymbol{m}$ is only given implicitly through eq. (2.36).

We will use the Lagrangian formulation and eq. (2.36) to simplify eq. (2.37) and get explicit expressions for the gradients. We use the same term $\boldsymbol{g}$ as defined in eq. (2.23) and invoke the implicit function theorem (see [17]) which allows us to recast $\boldsymbol{u} = \boldsymbol{u}(\boldsymbol{m})$ as in the definition of the misfit function. From now on we will omit the dependencies on $\boldsymbol{m}$ and $\boldsymbol{u}$ for the sake of conciseness, unless they are needed.

We define the *Lagrangian* $\mathcal{L}$ function as

$$\mathcal{L} \coloneqq \chi + \boldsymbol{\lambda}^T \boldsymbol{g}, \tag{2.38}$$

where $\boldsymbol{\lambda}$ is a arbitrary vector. We observe that the Lagrangian is just the misfit to which we added a term equal to zero, since $\boldsymbol{g} = \boldsymbol{0}$ by definition. It results that $\mathcal{L} = \chi$ for every $\boldsymbol{u}, \boldsymbol{m}$ satisfying $\boldsymbol{g} = \boldsymbol{0}$, so their gradients are also equal.

We can now take the derivative of the Lagrangian with respect to a single model parameter m_i which is the i-th component of $\boldsymbol{m}$ and we get

$$\frac{d\mathcal{L}}{dm_i} = \frac{d\chi}{dm_i} + \frac{d\boldsymbol{\lambda}^T}{dm_i}\underbrace{\boldsymbol{g}}_{=\boldsymbol{0}} + \boldsymbol{\lambda}^T \frac{d\boldsymbol{g}}{dm_i} \tag{2.39}$$

$$= \frac{\partial \chi}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{dm_i} + \frac{\partial \chi}{\partial m_i} + \boldsymbol{\lambda}^T \left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\frac{d\boldsymbol{u}}{dm_i} + \frac{\partial \boldsymbol{g}}{\partial m_i}\right], \tag{2.40}$$

where in the first equality we have used the product rule of differentiation and in the second equality we have also used the chain rule for total derivatives. The term $\partial\boldsymbol{g}/\partial\boldsymbol{u}$ is also called the Jacobian $\boldsymbol{J}_{\boldsymbol{g},\boldsymbol{u}}$ which is a matrix containing the partial derivatives of all components of $\boldsymbol{g}$ with respect to all components of $\boldsymbol{u}$. We can now continue the derivation by grouping the terms containing $d\boldsymbol{u}/dm_i$ in the following way

$$\frac{d\mathcal{L}}{dm_i} = \left[\frac{\partial \chi}{\partial \boldsymbol{u}} + \boldsymbol{\lambda}^T \boldsymbol{J}_{\boldsymbol{g},\boldsymbol{u}}\right]\frac{d\boldsymbol{u}}{dm_i} + \frac{\partial \chi}{\partial m_i} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial m_i}. \tag{2.41}$$

[6] This is not precisely the case we have seen in section 2.2.2. To be more precise we should have used the notation $\chi[\boldsymbol{d}, \boldsymbol{m}]$ and the relation of eq. (2.17) to express $\boldsymbol{d}$ as a function of $\boldsymbol{u}$. In practice, this is rarely a problem since the operator $\boldsymbol{D}$ usually just picks some of the values in $\boldsymbol{u}$ so that $\partial\boldsymbol{d}/\partial\boldsymbol{u} = \boldsymbol{D}$ where $\boldsymbol{D}$ is a (0,1)-matrix.

We choose the arbitrary vector $\boldsymbol{\lambda}$ cleverly such that the term inside the square brackets vanishes. We call the vector $\boldsymbol{\lambda}$ the *adjoint vector* and the equation resulting from this choice the *adjoint equation* such that

$$\boxed{\boldsymbol{J}_{\boldsymbol{g},\boldsymbol{u}}^T \boldsymbol{\lambda} = -\left(\frac{\partial \chi}{\partial \boldsymbol{u}}\right)^T} . \tag{2.42}$$

The inverse of the Jacobian $(\boldsymbol{J}_{\boldsymbol{g},\boldsymbol{u}}^T)^{-1}$ must exist for $\boldsymbol{\lambda}$ to be a unique solution to the linear system eq. (2.42). This condition is in the hypothesis of the previously invoked implicit function theorem.

We end up with an expression for the gradients of the misfit function of the form

$$\boxed{\frac{d\chi}{dm_i} = \frac{\partial \chi}{\partial m_i} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial m_i}} . \tag{2.43}$$

In the special case that the differential operator $\boldsymbol{L}$ is linear with respect to the observables $\boldsymbol{u}$, $\boldsymbol{g}$ can be written as $\boldsymbol{g} = \boldsymbol{L}(\boldsymbol{m})\boldsymbol{u} - \boldsymbol{f}$, where $\boldsymbol{L}$ is now a matrix taking the role of the differential operator[7]. This means that the Jacobian is just the matrix $\boldsymbol{L}$ and so eq. (2.42) becomes

$$\boldsymbol{L}^T \boldsymbol{\lambda} = -\left(\frac{\partial \chi}{\partial \boldsymbol{u}}\right)^T . \tag{2.44}$$

A few things should be noted regarding this method. First of all, in order to find compute eq. (2.43) we first need to solve the adjoint equation eq. (2.42) for $\boldsymbol{\lambda}$. This means that we need to be able to compute the partial derivatives of the misfit function with respect to the observables $\boldsymbol{u}$. This can be arbitrarily complex depending on the definition of the misfit function. Fortunately, the misfit functions introduced in section 2.2.2 are all easy to differentiate with respect to $\boldsymbol{d}$ and subsequently easy to differentiate with respect to $\boldsymbol{u}$ by substituting $\boldsymbol{d} = \boldsymbol{D}(\boldsymbol{u})$ and considering $\boldsymbol{d}$ a function of $\boldsymbol{u}$ so that, by the chain rule, we have:

$$\frac{\partial \chi}{\partial \boldsymbol{u}} = \frac{\partial \chi}{\partial \boldsymbol{d}} \frac{\partial \boldsymbol{d}}{\partial \boldsymbol{u}} \tag{2.45}$$

and the term $\partial \boldsymbol{d}/\partial \boldsymbol{u}$ is usually easy to compute (see footnote 6).

After having found a solution to eq. (2.42), we need to compute the partial derivatives of $\boldsymbol{g}$ with respect to the model parameters, which are once again the Jacobians of $\boldsymbol{g}$ and $\boldsymbol{f}$ with respect to the model parameters, that is

$$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{m}} = \boldsymbol{J}_{\boldsymbol{g},\boldsymbol{m}} = \boldsymbol{J}_{\boldsymbol{L},\boldsymbol{m}} - \boldsymbol{J}_{\boldsymbol{f},\boldsymbol{m}} \tag{2.46}$$

[7] Note that the matrix $\boldsymbol{L}$ still depends on the model parameters $\boldsymbol{m}$ in a (possibly) non-linear way.

and rewriting eq. (2.43) for the gradient with respect to all model parameters of the misfit function we get

$$\boxed{\frac{d\chi}{d\boldsymbol{m}} = \frac{\partial \chi}{\partial \boldsymbol{m}} + \boldsymbol{\lambda}^T \left(\boldsymbol{J}_{\boldsymbol{L},\boldsymbol{m}} - \boldsymbol{J}_{\boldsymbol{f},\boldsymbol{m}} \right)} \,. \tag{2.47}$$

In the case that the force term $\boldsymbol{f}$ is independent of $\boldsymbol{m}$ we have

$$\frac{d\chi}{d\boldsymbol{m}} = \frac{\partial \chi}{\partial \boldsymbol{m}} + \boldsymbol{\lambda}^T \boldsymbol{J}_{\boldsymbol{L},\boldsymbol{m}}. \tag{2.48}$$

Computation of the Jacobians $\boldsymbol{J}_{\boldsymbol{L},\boldsymbol{m}}$ and $\boldsymbol{J}_{\boldsymbol{f},\boldsymbol{m}}$ can almost always be done analytically, even if the dependencies on $\boldsymbol{m}$ are non-linear, although one must be careful to correctly differentiate the discrete versions of the operator $\boldsymbol{L}$ and the force term $\boldsymbol{f}$. This gets particularly complicated when interpolations of material properties are performed for the discretization of, for example, the differential operator.

Another thing to consider is the computation of the partial derivatives of the misfit function with respect to the model parameters, but this is easy for the same reasons as the partial derivatives with respect to the observables.

Summary. To summarize, the discrete adjoint method lets us compute gradients of the misfit function with respect to model parameters by applying the following steps:

1. Solve the forward equation given by eq. (2.36) for $\boldsymbol{u}$,
2. Compute the matrix $\boldsymbol{J}^T_{\boldsymbol{g},\boldsymbol{u}}$ (or just the transpose of the forward operator $\boldsymbol{L}^T$ if linear) and the partial derivatives $\partial\chi/\partial\boldsymbol{u}$ using the solution $\boldsymbol{u}$ from step 1,
3. Solve the adjoint equation given by eq. (2.42) for the adjoint vector $\boldsymbol{\lambda}$,
4. Compute the gradients of the misfit function by computing the partial derivatives $\partial\chi/\partial\boldsymbol{m}$, the Jacobians $\boldsymbol{J}_{\boldsymbol{L},\boldsymbol{m}}$, $\boldsymbol{J}_{\boldsymbol{f},\boldsymbol{m}}$ and putting all together with the adjoint solution vector $\boldsymbol{\lambda}$ from step 3 using eq. (2.47).

The two most expensive steps are steps 1 and 3 in which we basically solve a forward problem for each one. Steps 2 and 4 are relatively less expensive, in particular, eq. (2.47) can be computed on the fly while performing step 3.

2.3.3 Main takeaways from the adjoint method

Although some technical issues must be overcome to store $\boldsymbol{u}$ fully (which is usually not needed when solving the forward problem), the adjoint method essentially lets us compute the gradients of the misfit functional needed by first-order general descent methods at the cost solving two forward problems instead of $2n$ problems

using the naive finite differences approach (reduced to $n + 1$ by using a less accurate first-order finite difference approximation).

The two approaches for the adjoint method, namely continuous and discrete, are essentially equivalent in the case that the operator $\boldsymbol{L}$ is linear in $\boldsymbol{u}$, but may lead to different results for other cases. Fortunately, for the context of this thesis the operators used will all be linear, so we can either use the continuous or the discrete adjoint method.

In the next sections, we will explore the (acoustic) wave equation and use the continuous adjoint method to derive expressions for the sensitivity kernels. Later in chapter 3 we will use those expressions combined with a finite difference discretization to compute gradients with respect to model parameters that are needed to optimize the misfit and solve the deterministic inverse problem using general decent methods.

2.4 Case study: acoustic wave equation

In this section, we will introduce the wave equation and, in particular, the acoustic wave equation. This equation models the propagation of (acoustic) waves in a medium. Depending on the parametrization of the equation and the problem dimensionality, the medium can assume different physical counterparts: for example, the one-dimensional wave equation is generally used to model a string with both endpoints attached and stimulated with a pinch or bent from the relaxed position causing, after its release, oscillations producing waves.

For the sake of this thesis, we will mostly concentrate on the acoustic wave equation, modeling acoustic pressure waves in a fluid. This equation will be used to model the problem of ultrasound wave propagation with the specific application of medical imaging. However, the same approach can be used to model different types of problems related to acoustics or simplifications of elastic wave propagation models that lead to an acoustic-like behavior of the medium.

2.4.1 Wave equation

We start by introducing the two-way scalar wave equation, which is a second-order hyperbolic PDE involving an unknown displacement field $u = u(\boldsymbol{x}, t)\colon \Omega \times [0, T] \to \mathbb{R}$ and a known spatially varying wave propagation speed field $c = c(\boldsymbol{x})\colon \Omega \to \mathbb{R}$ (m/s)

$$\frac{\partial^2 u}{\partial t^2} = c^2 \, \nabla^2 \, u, \tag{2.49}$$

where $\Omega \subset \mathbb{R}^n$ is the spatial domain of interest and $[0, T]$ is the temporal domain of interest (s), i.e. we are interested in the values of the displacement field u inside Ω from time 0 to time T.

The wave equation is perhaps the most simple second-order hyperbolic PDE and has been a subject of study for mathematicians like d'Alambert, Euler, and

many others. We can interpret eq. (2.49) with the following statement: the displacement's acceleration is proportional to the divergence of the displacement's rate of change in each direction. In more simple terms, the way the displacement gets 'pushed around' is proportional to how 'spiky' the displacement is. The constant in the proportion is the propagation speed squared c^2, so if c doubles the acceleration of the displacement field will get four times bigger.

Note that the acceleration of the displacement field is related to the speed of propagation via the PDE but it has a completely different physical meaning, which is described in the following example. Consider a membrane stretched over one of the two open sides of a drum. The displacement of the membrane from the 'relaxed' state will be modeled as the displacement field u. The membrane's wave propagation speed c will depend on the membrane's thickness, tension, and material type. This is the speed at which transverse vibration waves (i.e. waves where the particles are displaced perpendicular to the direction of wave propagation) will travel on the membrane. The acceleration of the displacement field is, instead, the acceleration of the membrane's particles which move 'up and down' with respect to the 'relaxed' position.

Equation (2.75) is called the *two-way* wave equation because it can be used to describe standing waves (i.e. waves with a peak amplitude profile that does not change) where the displacement field can be computed as the superposition of two waves traveling in opposite directions. There is also a simpler, first-order wave equation, called *one-way* wave equation, which is used in the case that only the propagation of a single wave in a predefined direction is of interest. Note that will not consider this type of wave equation in our case study.

Some questions that arise spontaneously need to be addressed:

1. do solutions to the wave equation exist, meaning is there a function u such that eq. (2.49) is satisfied?
2. can the wave equation be solved analytically, and if it can be solved, is the solution unique?
3. if it can be solved analytically for some particular cases, which are these cases?
4. if it cannot be solved in the general case, how much information regarding the solution can be obtained without actually retrieving it?

The answers to all of these questions will come more thoroughly in the following sections. We will give a brief summary of the answers right away, but first, we need to understand what solving a partial differential equation even means. In general, solutions to PDEs are *not* unique, meaning multiple different functions can be found satisfying a partial differential equation. There is usually a *class of functions* containing all solutions to a specific PDE. If we go back to the physical implication of this fact, we soon conclude that it does not make sense. We should have a unique solution for a given physical problem right? Well, we

need to understand that PDEs do not model a physical problem in its entirety. They merely model the *interactions* between the physical quantities that appear in the equations, but this is not enough to provide a unique solution, for it can only say so much if we have not given enough context to begin with. Going back to the example of the drum head membrane oscillating, we have assumed in our imagination that it was stretched over and attached uniformly to the sides of the drum. But we could have attached it only partially or even not at all! This information we tacitly assumed is nowhere to be found in the PDE, so the solution cannot distinguish between the infinitely many cases arising from the physical model setup.

2.4.2 Initial and boundary conditions

In this section, we will introduce the concept of initial and boundary conditions as well as give some common ones for which eq. (2.49) can be solved (or at least a unique solution exists). We will also give some insights into how these conditions appear in differential equations theory and provide some physical intuition for those regarding the wave equation.

Initial conditions and initial value problems

From the theory of ordinary differential equations (ODEs), we know that a second-order ODE requires two initial conditions to be satisfied for it to have a unique solution. Intuitively, this comes from the fact that we need to perform two integral steps, each one 'spawning' an integration constant that can be chosen arbitrarily if we do not impose such initial conditions.

For example, consider the following second-order ODE

$$y'' = c, \tag{2.50}$$

where $c \in \mathbb{R}$ is a real constant and $y = y(x) \colon \mathbb{R} \to \mathbb{R}$ is a real-valued function. We can solve for y by double integration

$$y' = \int c \, dx = cx + c_1, \tag{2.51}$$

$$y = \int y' dx = \frac{c}{2}x^2 + c_1 x + c_2, \tag{2.52}$$

where $c_1, c_2 \in \mathbb{R}$ are two arbitrary constant. The resulting function $y(x)$ is a parabola with two yet-to-be-determined parameters. Let us impose two more conditions on y and y', namely

$$y(x_0) = y_0, y'(x_0) = y'_0 \tag{2.53}$$

where $x_0, y_0, y'_0 \in \mathbb{R}$ are fixed real numbers. We will call these *initial conditions*

of the ODE[8]. We can now determine the two constants c_1, c_2 by solving a system of equations

$$y(x_0) = \frac{c}{2}x_0^2 + c_1x_0 + c_2 = y_0, \tag{2.54}$$

$$y'(x_0) = cx_0 + c_1 = y_0', \tag{2.55}$$

which gives

$$c_1 = y_0' - cx_0, \tag{2.56}$$

$$c_2 = y_0 + \frac{c}{2}x_0^2 - x_0y_0' \tag{2.57}$$

and so the final solution of the original ODE is

$$y(x) = \frac{c}{2}x^2 + (y_0' - cx_0)x + (y_0 + \frac{c}{2}x_0^2 - x_0y_0'). \tag{2.58}$$

One can indeed substitute the values for $y(x_0), y'(x_0)$ and recover the initial conditions.

Returning to the scalar wave equation: it is a second-order PDE with respect to time derivatives, so two initial conditions are to be given

$$u(\boldsymbol{x}, t_0) = u_0(\boldsymbol{x}), \forall \boldsymbol{x} \in \Omega, \tag{2.59}$$

$$\frac{\partial u}{\partial t}(\boldsymbol{x}, t_0) = u_0'(\boldsymbol{x}), \forall \boldsymbol{x} \in \Omega, \tag{2.60}$$

where $t_0 = 0$ since we chose the displacement field u to be defined from 0 to T in time. In the PDE case, the initial conditions are a bit more complicated because u_0, u_0' are functions defined on the domain Ω instead of constants. From a practical point of view, these initial conditions are equivalent to imposing the initial 'shape' and 'velocity' of the displacement field at the initial time. A special case of initial conditions is *homogeneous* initial conditions, meaning that the functions u_0, u_0' are actually constants, in the simplest case they are zero everywhere in the domain. An ODE or PDE equipped with initial conditions forms a *initial value problem* (IVP). Note that not all IVPs have a unique solution. There are theorems (see [18]) specifying conditions for which a general IVP has a unique solution, however, we will not cover these since they are out of the scope of this thesis. Just note that specifying the wrong initial conditions for a particular ODE may lead to the non-uniqueness of the solution, making an IVP ill-posed.

Returning to the example of the drum head membrane oscillating, imposing initial conditions means specifying the initial position of the membrane and its initial speed. One possibly simple scenario is when the membrane gets pushed (or pulled) at some points and then released. In this case, the initial shape of the membrane will not be in a relaxed (i.e. constant to zero) state, depending on the position and strength of the force applied, and the initial velocity of the membrane will be zero.

[8] The name 'initial conditions' comes from the literature on ODEs where the function represents the evolution over time of a variable $y = y(t)$ and these conditions where usually given for $t = 0$.

Boundary conditions and boundary value problems

Other types of conditions that can be imposed on ODEs are the so-called *boundary conditions* (BDCs). These conditions are usually way more complex to handle than initial conditions and the theory behind the existence and uniqueness of solutions in this case is more complicated as well. In PDEs theory, these conditions are crucial to specify the behavior of the PDEs at the boundary of the domain Ω, hence the name of the conditions.

An ODE example for boundary conditions can be done by considering the following second-order ODE

$$y'' + 4y = 0, \tag{2.61}$$

where $y = y(x)$ is again a real valued function. The general solution of the above equation is (without derivation)

$$y(x) = c_1 \cos(2x) + c_2 \sin(2x), \tag{2.62}$$

which can be easily checked by differentiation and substitution back to eq. (2.61). Again we see that the general solution has two integration constants that make this solution not unique. If we impose the following conditions on y

$$y(0) = y_0, \tag{2.63}$$

$$y(\pi/4) = y_1, \tag{2.64}$$

where $y_0, y_1 \in \mathbb{R}$ are fixed real numbers, and we substitute back to the general solution eq. (2.62) we get the unique solution

$$y(x) = y_0 \cos(2x) + y_1 \sin(2x). \tag{2.65}$$

Notice the main difference between initial and boundary conditions, namely the fact that we have specified the values of the function y (or more generally its derivatives) at two *different points* (i.e. $x = 0$ and $x = \pi/4$) while for initial conditions we specified the values for y (and y') at the *same point* x_0. This difference, while subtle, is important and makes *boundary value problems* (BVPs) (i.e. the problem of solving an ODE/PDE with equipped boundary conditions) much harder than IVPs.

Returning to the scalar wave equation, we need to impose some type of boundary conditions on the displacement field u. For the purpose of this thesis, we will introduce two types of boundary conditions useful to solve the wave equation: homogeneous Dirichlet BDCs (also known as *free surface* conditions in the context of the wave equation) and C-PML BDCs (a special class of *absorbing* boundary conditions).

Homogeneous Dirichlet BDCs. The (homogeneous) Dirichlet BDCs are one of the most simple boundary conditions for PDEs. They impose the value of the

solution u at the boundary of the domain of interest Ω. For the wave equation, this would be

$$u(\boldsymbol{x}, t) = u_0(\boldsymbol{x}), \forall \boldsymbol{x} \in \partial\Omega, \forall t \in [0, T], \tag{2.66}$$

where $\partial\Omega \subset \Omega$ is the boundary of the domain Ω and $u_0(\boldsymbol{x})\colon \partial\Omega \to \mathbb{R}$ is a real valued function defined on the boundary. Notice that the boundary condition needs to be satisfied *for all* times $t \in [0, T]$. In the case that the function $u_0(\boldsymbol{x})$ is constant on the whole boundary (for all practical purposes $u_0 \equiv 0$ is a standard choice), the boundary conditions are called *homogeneous* Dirichlet BDCs. Back to the example of the drum head membrane oscillating, these BDCs have the physical meaning of 'holding' the membrane in place at the edge of the drum head. The consequence of this choice will affect the temporal evolution of the oscillations in the membrane. In particular, waves that arrive at the boundary will be reflected back to the direction they came from.

Absorbing boundary conditions (C-PML BDCs) This physical behavior of homogeneous Dirichlet BDCs is useful for simulating a drum head membrane, but can become detrimental for some other applications, specifically the ones we are interested in this thesis: simulating acoustic wave propagation for full-waveform inversions in the context of ultrasound medical imaging. If we simulate waves reflected from the domain's boundary we would be making a mistake because, in reality, this is not what happens as waves continue to propagate outside of the domain without necessarily being reflected back. We are only interested in the behavior of waves propagating *inside* the domain and interacting with the model that we want to eventually reconstruct by performing the inversion. An easy fix to this problem would be to make the domain Ω bigger such that no reflected waves from the boundary will come back to the original region of interest before the end time T. This is unfortunately not feasible from a computational point of view, because increasing the size of the domain increases the computational cost of the simulation.

To overcome this problem, we need some sort of alternative BDCs that will 'dampen' the waves reaching the boundary. In particular, we need a BDC of the class of *absorbing* boundary conditions. These BDCs are designed so waves incident to the boundary decay quickly and do not 'bounce back' into the inner part of the domain.

Various absorbing BDCs have been used in the literature, the most famous ones being Gaussian tapering [19] and Perfectly Matched Layers (PML) [20]. Gaussian tapering works by applying a so-called *sponge layer* to the boundary of the domain in which the amplitude of the waves is gradually reduced as it moves away from the computational domain. It is widely used because of its simplicity, especially in the context of finite difference solvers. Although Gaussian tapering works fine for most applications it has some issues, mainly the fact that we may need a large sponge layer when using higher-order methods. Otherwise, Gaussian tapering may produce reflected waves from the boundary between the

inner domain and the sponge layer. PML, on the other hand, introduced the PML region (similarly to the sponge layer in Gaussian tapering) where the wave equation is *perfectly matched*, effectively nullifying any reflections at the boundary of the inner domain and the PML region. We chose a special variant of the PML BDCs, namely *Convolutional* PML (or C-PML) [21] which has the advantage of being more effective in absorbing waves at grazing angles, meaning waves that are not perfectly incident to the domain's boundary. We will not derive the PML theory from scratch as it is out of the scope of this thesis (see [21, 22] for more details), but rather show how the equations are modified in the PML region and, in chapter 3, give more details on how to effectively implement CPML BDCs in practice.

The main idea of PML BDCs is to introduce a complex coordinate stretching parameter to the wave equation in the frequency domain and apply a recursive convolution algorithm to convert the equation back to the time domain. This effectively results in changing the definition of the spatial derivatives in the PML region as such

$$\frac{\partial u}{\partial \tilde{i}} = \frac{\partial u}{\partial i} + \psi_i, \forall \boldsymbol{x} \in \tilde{\partial}\Omega_i, \tag{2.67}$$

where i is used as Einstein notation to denote the various dimensions (e.g. $i \in \{x, y, z\}$ in 3D) and $\tilde{\partial}_i\Omega$ denotes an extension of $\partial\Omega$ in the i dimension which is the PML region. The variable ψ_i is an auxiliary variable (also called PML memory variable) that controls the dampening of the wave in the PML region and needs to be computed using a recursive convolution algorithm which can be expressed by the following recursive equation

$$\psi_i^n = b_i \psi_i^{n-1} + a_i \left(\frac{\partial u}{\partial i}\right)^n \tag{2.68}$$

where n is an index that represents the current time step, not a power exponent. The choice of the parameters a_i and b_i is what determines the strength of the dampening and depends on multiple factors, which are detailed in chapter 3.

Since we have a second-order PDE with respect to space, we need to have two PML memory variables per dimension, hence the modified second-order partial derivative in space becomes

$$\frac{\partial^2 u}{\partial \tilde{i}^2} = \frac{\partial}{\partial \tilde{i}}\left(\frac{\partial u}{\partial \tilde{i}}\right) = \frac{\partial^2 u}{\partial i^2} + \frac{\partial \psi_i}{\partial i} + \xi_i, \forall \boldsymbol{x} \in \tilde{\partial}\Omega_i, \tag{2.69}$$

where ξ_i is another PML memory variable with evolution

$$\xi_i^n = b_i \xi_i^{n-1} + a_i \left[\left(\frac{\partial^2 u}{\partial i^2}\right)^n + \left(\frac{\partial \psi_i}{\partial i}\right)^n\right]. \tag{2.70}$$

As last remark on absorbing boundary conditions is the fact that, even though we have defined boundary conditions on the 'extended' boundary region (i.e. the PML region in the above equations), we still need to define the behaviour of the

displacement field at the boundary $\partial\Omega$. Fortunately, since absorbing BDCs are designed to make the displacement field vanish as it reaches the boundary, we can combine them with homogeneous Dirichlet BDCs.

Initial-boundary value problem

Since the wave equation is a second-order PDE in both time and space, we need to combine both initial conditions and boundary conditions to obtain a *initial-boundary value problem* (IBVP). Again, choosing fitting initial and boundary conditions is essential to ensure the existence and uniqueness of the solution.

As an example, we give a complete IBVP for the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \qquad , \forall \boldsymbol{x} \in \overline{\Omega}, \forall t \in [0, T], \tag{2.71}$$

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\nabla^2 u + \left(\frac{\partial \psi_i}{\partial i} \right) + \xi_i \right) \qquad , \forall \boldsymbol{x} \in \tilde{\partial}_i \Omega, \forall t \in [0, T], \tag{2.72}$$

$$u = 0 \qquad , \forall \boldsymbol{x} \in \partial\Omega, \forall t \in [0, T], \tag{2.73}$$

$$u = u_0, \frac{\partial u}{\partial t} = u_0' \qquad , \forall \boldsymbol{x} \in \Omega, t = 0, \tag{2.74}$$

where eq. (2.72) are the CPML BDCs (for every dimension), eq. (2.73) are homogeneous Dirichlet BDCs and eq. (2.74) are arbitrary initial conditions.

2.4.3 Acoustic wave equation

In this section, we will introduce the acoustic wave equation which is a specialization of the two-way scalar wave equation in the case of acoustic waves propagating in a heterogeneous medium. In its most simple form, it is given as

$$\frac{1}{c^2} \frac{\partial^2 p}{\partial t^2} = \nabla^2 p + s, \tag{2.75}$$

where p is the acoustic pressure field (Pa) taking the role of the displacement field u in eq. (2.49) and $s = s(\boldsymbol{x}, t) \colon \mathbb{R}^n \times [0, T] \to \mathbb{R}$ (Pa/m^2) is the source field. The acoustic pressure field p actually represents the difference from equilibrium pressure $p = P - P_0$, where P is the pressure of the fluid and P_0, is some reference equilibrium pressure.

There are two main differences from eq. (2.49): the propagation speed c is moved to the left-hand side and a new term is added to the right-hand side, namely the source field. The source field term s is related to an external 'force' acting on the domain of interest that perturbs the pressure field at a location in space and time. We can think of this term from a physical point of view as the 'sound' generated by something (or someone). Source terms are essential for modeling acoustic sources like ultrasonic transducers.

Equation (2.75) is accurate for fluids with homogeneous density, hence $\operatorname{grad}\rho = \mathbf{0}$ where $\rho = \rho(\boldsymbol{x})\colon \mathbb{R}^n \to \mathbb{R}$ (kg/m^3) is the medium's density field. In the case of spatially varying density, the acoustic wave equation becomes

$$\frac{1}{\kappa}\frac{\partial^2 p}{\partial t^2} = \operatorname{div}\left(\frac{1}{\rho}\operatorname{grad} p\right) + \frac{1}{\rho}s, \tag{2.76}$$

where $\kappa = \kappa(\boldsymbol{x})\colon \mathbb{R}^n \to \mathbb{R}$ (Pa) is the medium's bulk modulus. Bulk modulus and density are related to the propagation speed by Newton-Laplace equation

$$c = \sqrt{\frac{\kappa}{\rho}}, \tag{2.77}$$

meaning we can recover another version of eq. (2.76) by substituting κ with ρc^2

$$\frac{1}{\rho c^2}\frac{\partial^2 p}{\partial t^2} = \operatorname{div}\left(\frac{1}{\rho}\operatorname{grad} p\right) + \frac{1}{\rho}s. \tag{2.78}$$

There are two differences between eq. (2.75) and eq. (2.78) that should be noted. In the first place, the presence of the density term ρ. Secondly, the Laplacian operator ∇^2 has been split into its components (namely the divergence and the gradient) to leave space for the density term to 'squeeze in' between the two. Obviously, in the case of constant density $\operatorname{grad}\rho = \mathbf{0}$, eq. (2.78) reduces to eq. (2.75).

Equation (2.78) can also be written in the so-called *pressure-velocity* formulation which introduces the particle velocity vector field $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{x}, t)\colon \mathbb{R}^n \times [0, T] \to \mathbb{R}^n$ (m/s) and it is a system of first-order PDEs as follows

$$\rho\frac{\partial \boldsymbol{v}}{\partial t} = -\operatorname{grad} p \tag{2.79a}$$

$$\frac{1}{\rho c^2}\frac{\partial p}{\partial t} = -\operatorname{div}\boldsymbol{v} + f \tag{2.79b}$$

where $f = f(\boldsymbol{x}, t)\colon \mathbb{R}^n \times [0, T] \to \mathbb{R}$ (1/s) is a different source field. The original second-order scalar PDE of eq. (2.78) can be recovered from eq. (2.79) by partial derivation in time of eq. (2.79b) and substitution on its right-hand side with eq. (2.79a).

The two source fields s and f are related as follows

$$s(\boldsymbol{x}, t) = \rho\frac{\partial f}{\partial t}(\boldsymbol{x}, t) \Leftrightarrow f(\boldsymbol{x}, t) = \frac{1}{\rho}\int^t s(\boldsymbol{x}, \tau)d\tau \tag{2.80}$$

and express different ways of injecting an acoustic source field into the equation.

2.4.4 Derivation of adjoint equations and sensitivity kernels using the continuous adjoint method

In this section, we will derive the continuous adjoint equations and expressions for the sensitivity kernels with respect to the model parameters. In the case of the acoustic wave equation with constant density, our only model parameter will be the wave propagation speed c, which can be interpreted as the speed of sound. In the case of variable density, we also need to account for changes in the density ρ which will be another model parameter for which to compute the sensitivity kernel. We will use the continuous adjoint method detailed in section 2.3.1. For this purpose, we will consider a general misfit functional χ which can be expressed as the sum of two integrals as in eq. (2.21).

Constant density acoustic wave equation

We will use $\boldsymbol{L} = L, \boldsymbol{g} = g, \boldsymbol{f} = f$ since we have a scalar equation with observables $\boldsymbol{u} = p$ and model parameters $\boldsymbol{m} = c$. We will also omit dependencies for the sake of conciseness when not needed.

We start by writing the acoustic wave equation with constant density eq. (2.75) in the form of eq. (2.18).

$$L(p; c) = f \Leftrightarrow \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} - \nabla^2 p = s, \tag{2.81}$$

and complement this PDE with initial and boundary conditions to form an IBVP. For simplicity, we choose homogeneous zero initial conditions and homogeneous Dirichlet boundary conditions (see section 2.4.2).

We note that the right-hand side term does not depend on p nor m and that the differential operator L is linear in p. Since this is the case, the functional derivative of the operator L with respect to p in the direction δp is just $L(\delta p)$. The same reasoning can be applied to the derivative of the adjoint operator $L^\dagger$ with respect to p. Thus the implicit definition of the adjoint operator in eq. (2.27) can be rewritten as

$$\langle \lambda L(\delta p)\rangle = \langle \delta p L^\dagger(\lambda)\rangle, \tag{2.82}$$

where $\boldsymbol{\lambda} = \lambda$ is also a scalar field in this context. In order to find an explicit expression for $L^\dagger$ we start by expanding the left-hand side of eq. (2.82)

$$\langle \lambda L(\delta p)\rangle = \left\langle \lambda \left(\frac{1}{c^2}\frac{\partial^2 \delta p}{\partial t^2} - \nabla^2 \delta p\right)\right\rangle = \left\langle \lambda \left(\frac{1}{c^2}\frac{\partial^2 \delta p}{\partial t^2}\right)\right\rangle - \left\langle \lambda \left(\nabla^2 \delta p\right)\right\rangle \tag{2.83}$$

and we focus on the two double integrals separately.

First, we integrate twice by parts the first double integral of eq. (2.83) with respect to time, such that

$$\left\langle \lambda \left(\frac{1}{c^2}\frac{\partial^2 \delta p}{\partial t^2}\right)\right\rangle = \left\langle \lambda \frac{1}{c^2}\frac{\partial \delta p}{\partial t} - \delta p \frac{1}{c^2}\frac{\partial \lambda}{\partial t}\right\rangle_\Omega \Bigg|_{t=0}^{t=T} + \left\langle \delta p \left(\frac{1}{c^2}\frac{\partial^2 \lambda}{\partial t^2}\right)\right\rangle. \tag{2.84}$$

We want the boundary values resulting from the integration by parts to vanish. For this to happen we need to provide *final conditions* on λ, i.e. the equivalent of initial conditions for p but for final time $t = T$, such that

$$\lambda = \frac{\partial \lambda}{\partial t} = 0, \forall \boldsymbol{x} \in \Omega, t = T. \tag{2.85}$$

Using these final conditions in combination with the provided initial conditions for p (which also apply for the perturbation direction δp), the boundary values indeed vanish and we get the relation

$$\left\langle \lambda \left(\frac{1}{c^2} \frac{\partial^2 \delta p}{\partial t^2} \right) \right\rangle = \left\langle \delta p \left(\frac{1}{c^2} \frac{\partial^2 \lambda}{\partial t^2} \right) \right\rangle . \tag{2.86}$$

Similarly, we integrate twice by parts the second double integral of eq. (2.83) with respect to space, such that

$$\left\langle \lambda \left(\nabla^2 \delta p \right) \right\rangle = \left\langle \oint_{\partial\Omega} \left(\lambda \operatorname{grad} \delta p - \delta p \operatorname{grad} \lambda \right) \cdot \boldsymbol{n} \, dS \right\rangle_T + \left\langle \delta p \left(\nabla^2 \lambda \right) \right\rangle . \tag{2.87}$$

Again, we want the surface integral resulting from the integration by parts to vanish. Similarly, we define the boundary conditions for λ to be the following homogeneous Dirichlet BDCs

$$\lambda = 0, \forall \boldsymbol{x} \in \partial\Omega, \forall t \in [0, T]. \tag{2.88}$$

Combining these with BDCs for p (which also apply for the perturbation direction δp), the surface integral indeed vanishes and we get the relation

$$\left\langle \lambda \left(\nabla^2 \delta p \right) \right\rangle = \left\langle \delta p \left(\nabla^2 \lambda \right) \right\rangle . \tag{2.89}$$

Combining the two relations eq. (2.86) and eq. (2.89) we can rewrite the implicit definition of the adjoint operator eq. (2.82) as

$$\left\langle \lambda \underbrace{\left(\frac{1}{c^2} \frac{\partial^2 \delta p}{\partial t^2} - \nabla^2 \delta p \right)}_{L(\delta p)} \right\rangle = \left\langle \delta p \underbrace{\left(\frac{1}{c^2} \frac{\partial^2 \lambda}{\partial t^2} - \nabla^2 \lambda \right)}_{L^\dagger(\lambda)} \right\rangle . \tag{2.90}$$

We have finally found an explicit expression for the adjoint operator $L^\dagger$, which is the following

$$\boxed{L^\dagger(\lambda, c) = \frac{1}{c^2} \frac{\partial^2 \lambda}{\partial t^2} - \nabla^2 \lambda} \, . \tag{2.91}$$

We note that the operator L and $L^\dagger$ are the same, so the differential operator L is called *self-adjoint*. This is very useful in practice (as we will see in chapter 3) because it makes it possible to solve the adjoint equation using the same numerical method we used to solve the forward equation.

There remains only one major difference in the adjoint equation, namely the fact that we have final conditions instead of initial conditions for the field λ. In practice this means that we have to solve the adjoint equation *backward* in time. We will see in chapter 3 how this is done, but we need not worry for now.

Now we need to retrieve the expression for the sensitivity kernel K with respect to the model parameter c. To do this, we need to compute the functional derivative of $g = L - s$ with respect to c in the direction δc, that is

$$\delta g(\delta c) = \delta L(\delta c) - \underbrace{\delta s(\delta c)}_{=0} = -\frac{2}{c^3}\frac{\partial^2 p}{\partial t^2}\delta c \tag{2.92}$$

and we recover the functional derivative of the misfit functional using eq. (2.30) such that

$$\delta \mathcal{X}(p, c; \delta c) = -\left\langle \lambda \frac{2}{c^3}\frac{\partial^2 p}{\partial t^2}\delta c \right\rangle + \langle \delta_c \mathcal{X}_c \rangle_\Omega, \tag{2.93}$$

where $\mathcal{X}_c$ depends on the misfit definition. The time-independent sensitivity kernel $K(\boldsymbol{x})$ is easily retrieved using eq. (2.32) as

$$\boxed{K(\boldsymbol{x}) = -\left\langle \lambda \frac{2}{c^3}\frac{\partial^2 p}{\partial t^2} \right\rangle_T + \delta_c \mathcal{X}_c}\,. \tag{2.94}$$

Variable density acoustic wave equation

For this derivation, we will use a slightly more general form of the acoustic wave equation, given by

$$m_0 \frac{\partial^2 \phi}{\partial t^2} - \operatorname{div}(m_1 \operatorname{grad} \phi) = \tilde{s}, \tag{2.95}$$

where ϕ is a general scalar wavefield, m_0, m_1 are two scalar fields parameters and $\tilde{s}$ is a source term. We rewrite eq. (2.95) into a system of first-order PDEs by introducing the auxiliary vector field $\boldsymbol{\nu}$ as follows

$$\frac{1}{m_1}\frac{\partial \boldsymbol{\nu}}{\partial t} + \operatorname{grad} \phi = \boldsymbol{0}, \tag{2.96a}$$

$$m_0 \frac{\partial \phi}{\partial t} + \operatorname{div} \boldsymbol{\nu} = \tilde{f}, \tag{2.96b}$$

where $\tilde{f}$ is another source term. It is clear that we can recover eq. (2.95) from eq. (2.96) in a similar way we can recover eq. (2.78) from eq. (2.79). The source term $\tilde{s}$ is easily recovered by partial differentiation in time of the other source term $\tilde{f}$.

We will use eq. (2.96) as the physical relationship $\boldsymbol{L}[\boldsymbol{u}, \boldsymbol{m}] = \boldsymbol{f}$, where

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{\nu} \\ \phi \end{bmatrix}, \boldsymbol{m} = \begin{bmatrix} m_0 \\ m_1 \end{bmatrix}, \boldsymbol{L}(\boldsymbol{m}) = \begin{bmatrix} \frac{1}{m_1}\frac{\partial}{\partial t} & \operatorname{grad} \\ \operatorname{div} & m_0 \frac{\partial}{\partial t} \end{bmatrix}, \boldsymbol{f} = \begin{bmatrix} \boldsymbol{0} \\ \tilde{f} \end{bmatrix}. \tag{2.97}$$

From now on we will express the differential operator $\boldsymbol{L}$ as a matrix since it is linear in $\boldsymbol{u}$ and omit its dependence on $\boldsymbol{m}$. The physical relationship can be written as a system

$$\boldsymbol{L}\boldsymbol{u} = \boldsymbol{f}. \tag{2.98}$$

We also need to provide appropriate initial and boundary conditions to form an IBVP. Similarly, as in the constant density case, we choose homogeneous zero initial conditions for ϕ and $\boldsymbol{\nu}$ and homogeneous Dirichlet BDCs for ϕ (see section 2.4.2).

Now we can start the derivation of the adjoint operator $\boldsymbol{L}^\dagger$ in a similar way as we did in the constant density case by using the linearity of the differential operator and the fact that the right-hand side of eq. (2.98) is independent by both $\boldsymbol{u}$ and $\boldsymbol{m}$. We recall the implicit definition of the adjoint operator in eq. (2.27) that in this case can be rewritten as

$$\langle \boldsymbol{\lambda} \cdot \boldsymbol{L}\delta\boldsymbol{u} \rangle = \langle \delta\boldsymbol{u} \cdot \boldsymbol{L}^\dagger \boldsymbol{\lambda} \rangle, \tag{2.99}$$

where $\boldsymbol{\lambda}$ is the adjoint vector field that can be split in two components $\boldsymbol{\lambda} = [\boldsymbol{\lambda}_1, \lambda_2]^T$. We expand the left-hand side of eq. (2.99) and divide the product between $\boldsymbol{\lambda}$

$$\langle \boldsymbol{\lambda} \cdot \boldsymbol{L}\delta\boldsymbol{u} \rangle = \left\langle \boldsymbol{\lambda_1} \cdot \left(\frac{1}{m_1} \frac{\partial \delta\boldsymbol{\nu}}{\partial t} + \operatorname{grad} \delta\phi \right) \right\rangle + \left\langle \lambda_2 \left(m_0 \frac{\partial \delta\phi}{\partial t} + \operatorname{div} \delta\boldsymbol{\nu} \right) \right\rangle, \tag{2.100}$$

where $\delta\boldsymbol{u} = [\delta\boldsymbol{\nu}, \delta\phi]^T$ is the perturbation determining the direction of the functional derivative with respect to $\boldsymbol{u}$.

We consider the two terms on the right-hand side of eq. (2.100) separately and further separate each one in two parts, thus obtaining the four integrals that we can integrate by parts, similarly to what we did in the constant density case

$$\left\langle \frac{1}{m_1} \boldsymbol{\lambda}_1 \cdot \frac{\partial \delta\boldsymbol{\nu}}{\partial t} \right\rangle = \left\langle \frac{1}{m_1} \boldsymbol{\lambda}_1 \cdot \delta\boldsymbol{\nu} \right\rangle_\Omega \Bigg|_{t=0}^{t=T} - \left\langle \frac{1}{m_1} \delta\boldsymbol{\nu} \cdot \frac{\partial \boldsymbol{\lambda}_1}{\partial t} \right\rangle, \tag{2.101a}$$

$$\left\langle m_0 \lambda_2 \frac{\partial \delta\phi}{\partial t} \right\rangle = \langle m_0 \lambda_2 \delta\phi \rangle_\Omega \Bigg|_{t=0}^{t=T} - \left\langle m_0 \delta\phi \frac{\partial \lambda_2}{\partial t} \right\rangle, \tag{2.101b}$$

$$\langle \boldsymbol{\lambda}_1 \cdot \operatorname{grad} \delta\phi \rangle = \left\langle \oint_{\partial\Omega} \delta\phi (\boldsymbol{\lambda}_1 \cdot \boldsymbol{n}) dS \right\rangle_T - \langle \delta\phi \operatorname{div} \boldsymbol{\lambda}_1 \rangle, \tag{2.101c}$$

$$\langle \lambda_2 \operatorname{div} \delta\boldsymbol{\nu} \rangle = \left\langle \oint_{\partial\Omega} \lambda_2 (\delta\boldsymbol{\nu} \cdot \boldsymbol{n}) dS \right\rangle_T - \langle \delta\boldsymbol{\nu} \cdot \operatorname{grad} \lambda_2 \rangle, \tag{2.101d}$$

and again providing homogeneous Dirichlet BDCs to λ_2 and homogeneous zero final condition to $\boldsymbol{\lambda}_1$ and λ_2, the boundary terms and surface integrals spawning from integration by parts vanish.

Summing up all the terms on the left-hand sides and right-hand sides of

eq. (2.101) we rewrite the adjoint relationship

$$\left\langle \boldsymbol{\lambda_1} \cdot \left(\frac{1}{m_1} \frac{\partial \delta \boldsymbol{\nu}}{\partial t} + \operatorname{grad} \delta\phi \right) \right\rangle + \left\langle \lambda_2 \left(m_0 \frac{\partial \delta\phi}{\partial t} + \operatorname{div} \delta\boldsymbol{\nu} \right) \right\rangle =$$
$$\left\langle \delta\boldsymbol{\nu} \cdot \left(-\frac{1}{m_1} \frac{\partial \boldsymbol{\lambda}_1}{\partial t} - \operatorname{grad} \lambda_2 \right) \right\rangle + \left\langle \delta\phi \left(-m_0 \frac{\partial \lambda_2}{\partial t} - \operatorname{div} \boldsymbol{\lambda}_1 \right) \right\rangle, \quad (2.102)$$

from which we can extract the explicit definition of the adjoint operator $\boldsymbol{L}^\dagger$ (in matrix form)

$$\boxed{\boldsymbol{L}^\dagger = \begin{bmatrix} -\frac{1}{m_1}\frac{\partial}{\partial t} & -\operatorname{grad} \\ -\operatorname{div} & -m_0 \frac{\partial}{\partial t} \end{bmatrix}}. \quad (2.103)$$

The obtained adjoint operator is equal to the forward operator $\boldsymbol{L}$ but negated. This is the result of the single integration by parts, while in the constant density case we performed two successive integrations by parts.

We now retrieve the expressions for the sensitivity kernels $\boldsymbol{K} = [K_{m_0}, K_{m_1}]^T$. To do this, we need to compute two derivatives of $\boldsymbol{g} = \boldsymbol{L}\boldsymbol{u} - \boldsymbol{f}$, one with respect to m_0 and the other with respect to m_1

$$\delta_{m_0}\boldsymbol{g} = \begin{bmatrix} \boldsymbol{0} \\ \frac{\partial\phi}{\partial t} \end{bmatrix}, \delta_{m_1}\boldsymbol{g} = \begin{bmatrix} -\frac{1}{m_1^2}\frac{\partial\boldsymbol{\nu}}{\partial t} \\ 0 \end{bmatrix}, \quad (2.104)$$

Using eq. (2.32) we recover expressions for the sensitivity kernels

$$\boxed{\begin{aligned} K_{m_0}(\boldsymbol{x}) &= \langle \boldsymbol{\lambda} \cdot \delta_{m_0}\boldsymbol{g} \rangle_T + \delta_{m_0}\chi_m \\ &= \left\langle \lambda_2 \frac{\partial \phi}{\partial t} \right\rangle_T + \delta_{m_0}\chi_m \end{aligned}} \quad (2.105)$$

$$\boxed{\begin{aligned} K_{m_1}(\boldsymbol{x}) &= \langle \boldsymbol{\lambda} \cdot \delta_{m_1}\boldsymbol{g} \rangle + \delta_{m_1}\chi_m \\ &= \left\langle \boldsymbol{\lambda}_1 \cdot \left(-\frac{1}{m_1^2} \frac{\partial \boldsymbol{\nu}}{\partial t} \right) \right\rangle_T + \delta_{m_1}\chi_m \\ &= \left\langle \boldsymbol{\lambda}_1 \cdot \left(\frac{1}{m_1} \operatorname{grad} \phi \right) \right\rangle_T + \delta_{m_1}\chi_m \end{aligned}} \quad (2.106)$$

where for the last equality of eq. (2.106) we have used eq. (2.96a) from the physical relationship, so we can recast both sensitivity kernels on the scalar field ϕ only. The reason for this will become more apparent when discussing the discretization and implementation of gradients in chapter 3.

We can go one step further and consider the parametrization that leads to the pressure-velocity formulation of the acoustic wave equation with variable density (see eq. (2.79)), that is

$$m_0 = \frac{1}{\rho c^2}, m_1 = \frac{1}{\rho}, \boldsymbol{u} = \begin{bmatrix} \boldsymbol{v} \\ p \end{bmatrix}, \boldsymbol{f} = \begin{bmatrix} \boldsymbol{0} \\ f \end{bmatrix}. \quad (2.107)$$

Using this parametrization, we can express the sensitivity kernels with respect to the model parameters ρ and c. We compute the derivatives of $\boldsymbol{g}$ with respect to ρ and c by applying the chain rule

$$\begin{aligned}\delta_\rho \boldsymbol{g} &= \frac{\partial m_0}{\partial \rho}\delta_{m_0}\boldsymbol{g} + \frac{\partial m_1}{\partial \rho}\delta_{m_1}\boldsymbol{g} = -\frac{1}{\rho^2 c^2}\delta_{m_0}\boldsymbol{g} - \frac{1}{\rho^2}\delta_{m_1}\boldsymbol{g} \\ &= -\frac{1}{\rho^2}\begin{bmatrix}\frac{1}{m_1}\operatorname{grad}\phi \\ \frac{1}{c^2}\frac{\partial \phi}{\partial t}\end{bmatrix} = -\frac{1}{\rho^2}\begin{bmatrix}\rho\operatorname{grad}\phi \\ \frac{1}{c^2}\frac{\partial \phi}{\partial t}\end{bmatrix} = -\frac{1}{\rho}\begin{bmatrix}\operatorname{grad}\phi \\ \frac{1}{\rho c^2}\frac{\partial \phi}{\partial t}\end{bmatrix},\end{aligned} \tag{2.108a}$$

$$\delta_c \boldsymbol{g} = \frac{\partial m_0}{\partial c}\delta_{m_0}\boldsymbol{g} + \underbrace{\frac{\partial m_1}{\partial c}}_{=0}\delta_{m_1}\boldsymbol{g} = -\frac{2}{\rho c^3}\delta_{m_0}\boldsymbol{g} \tag{2.108b}$$

$$= -\frac{2}{\rho c^3}\begin{bmatrix}\mathbf{0} \\ \frac{\partial \phi}{\partial t}\end{bmatrix}, \tag{2.108c}$$

then using eq. (2.32) we recover expressions for the sensitivity kernels with respect to ρ and c

$$\boxed{\begin{aligned}K_\rho(\boldsymbol{x}) &= \langle \boldsymbol{\lambda} \cdot \delta_\rho \boldsymbol{g}\rangle_T + \delta_\rho \chi_m \\ &= \left\langle -\frac{1}{\rho}\boldsymbol{\lambda} \cdot \begin{bmatrix}\operatorname{grad}\phi \\ \frac{1}{\rho c^2}\frac{\partial \phi}{\partial t}\end{bmatrix}\right\rangle_T + \delta_\rho \chi_m\end{aligned}} \tag{2.109}$$

$$\boxed{\begin{aligned}K_c(\boldsymbol{x}) &= \langle \boldsymbol{\lambda} \cdot \delta_c \boldsymbol{g}\rangle + \delta_c \chi_m \\ &= \left\langle -\frac{2}{\rho c^3}\lambda_2 \frac{\partial \phi}{\partial t}\right\rangle_T + \delta_c \chi_m\end{aligned}} \tag{2.110}$$

which can be used to compute gradients with respect to those parameters based on the chosen discretization using eq. (2.35).

2.5 Summary

In this chapter, we have seen many theoretical aspects regarding the solution of inverse problems, in particular with respect to this thesis case study, namely acoustic full-waveform inversion. We have focused mostly on PDE-based forward models and deterministic methods like general descent methods that require first-order derivatives with respect to model parameters that are used to minimize some sort of misfit functional. However, these concepts are still quite abstract and present many technical difficulties when applying them in practice.

In the next chapter, we will explore how the yet-to-be-solved problems can be tackled and efficient solvers can be implemented for the forward and adjoint equations introduced in this chapter's case study section.

Chapter 3

Numerical Methods

Don't approximate pi as 3.0!

Wisdom from a mathematician.

In this chapter, we will deal with the following yet-to-be-solved problems: discretization of model parameters and physical quantities, implementation of solvers for both forward and adjoint equations, gradients computations with respect to model parameters and chosen discretization, and efficient parallelization, scalability and memory optimizations of implemented solvers (as well as many more technical details).

Unsurprisingly, there are many ways to tackle these problems. Our focus, for the context of this thesis, will be on *time-explicit finite-difference-based* (FD) methods on a *staggered and uniform grid*. Although more accurate methods to solve the wave equation exists [23], FD methods have been extensively used in the scientific community to solve wave propagation [3, 24], especially in the context of full waveform inversion [25]. Also, the choice of a uniform grid instead of non-uniform mesh-based grids for inversions in the context of ultrasound medical tomography is motivated by the fact that we usually do not know a-priori the discontinuities in the model we are reconstructing. However, in some specific applications where discontinuities from high contrasts in the model parameters are present, e.g. transcranial ultrasound tomography [7], mesh-based methods are to be preferred because they are more accurate near these discontinuities.

3.1 Finite differences discretization of the acoustic wave equation

In this section, we will introduce the main concepts of the finite-difference discretization we use for implementing our acoustic wave equation solvers.

We will start from the lowest-order finite-difference stencils in space and time and define the update formulas for the discretized pressure field. For the generalized acoustic wave equation with variable density, we introduce the staggered grid leap-frog scheme method. Using the same discretization, we give formulas for the computation of the adjoint fields as well as the sensitivity kernels in the case of an L2 misfit functional.

Then we quickly discuss extensions to higher-order stencils and their practical implementation.

We explain more in detail how to implement C-PML boundary conditions by giving formulas for the computation of the damping coefficients and evolution of the C-PML fields for both regular and staggered grids.

Finally, we talk about how point sources can be modeled and the required numerical corrections needed to ensure correct solutions to the acoustic wave equation with point sources.

3.1.1 Lowest-order finite difference stencils for acoustic wave equation

Constant density formulation

For the constant density formulation of the acoustic wave equation (see eq. (2.75)), we need to discretize the pressure field p and the source field s in space and time, as well as the wave speed c model parameter. For the discretization in space, the simplest way is to define a grid of points and consider the value of the fields at each point.

Let us consider, for simplicity, the case where our model has only one dimension, hence the domain of interest can be defined as an interval $\Omega = [0, L]$ where L (m) is the length of the domain. We define a grid of points such that all points are equally spaced (hence why the grid is called uniform). We call the distance between two points the *grid step size* Δx (m). Then the grid is defined by the position of its grid points $x_0, \dots, x_N$ where $N + 1$ is the number of grid points such that $N = L/\Delta x$. The grid points positions are defined as

$$x_i = i\Delta x, \forall i \in \{0, \dots, N\}. \tag{3.1}$$

Then we can define the spatially discrete values $c_0, \dots, c_N$ for the wave speed field $c\colon [0, L] \to \mathbb{R}$ as such

$$c_i^2 = c(x_i) = c(i\Delta x), \forall i \in \{0, \dots, N\}. \tag{3.2}$$

We perform the same discretization in space on the pressure field p, but we also need to have a time discretization. We follow the same approach: consider the time domain $[0, T]$ and an equally spaced grid in time such that all points are at distance Δt (s) which we call the *time step size*. The time-discrete grid points are defined as

$$t_\ell = \ell\Delta t, \forall \ell \in \{0, \dots, N_t\}, \tag{3.3}$$

where N_t is the number of time discrete points such that $N_t = T/\Delta t$. Hence, the fully discretized pressure field p can be represented by the values

$$p_i^\ell = p(x_i, t_\ell) = p(i\Delta x, \ell\Delta t), \forall i \in \{0, \dots, N\}, \forall \ell \in \{0, \dots, N_t\}. \tag{3.4}$$

Discretization of the source field s is done using the same procedure.

Now we need to discretize the second-order partial derivatives in time and space contained in eq. (2.75). The finite differences (FD) method gives a formula

for the second-order derivative of a function f that depends on x at a point x_i such that

$$\frac{\partial^2 f}{\partial x^2}(x_i) = \frac{f(x_i - \Delta x) - 2f(x_i) + f(x_i + \Delta x)}{\Delta x^2} + O(\Delta x^2), \tag{3.5}$$

where $O(\Delta x^2)$ is the approximation error of the FD scheme. If we neglect this error, we can use the first part of the right-hand side of eq. (3.5) as an approximated value for the second-order derivative. This FD scheme is called *second-order central finite difference* approximation of the second-order derivative. One must be careful not to confuse the order of *accuracy* of the scheme with the order of the derivative it approximates. It turns out that this scheme is the lowest-order accurate scheme for the approximation of the second-order derivative. The details on how these schemes are produced are beyond the scope of this thesis, but we will mention that FD schemes are derived based on the Taylor expansion of the function for which we want to approximate the derivative and then by solving a linear system of equations to find the coefficients of the scheme.

Since we now have both the discretization of the fields p and c as well as their partial derivatives, we replace all fields and their partial derivatives in eq. (2.75) with their discretizations, such that

$$\frac{1}{c_i^2}\frac{p_i^{\ell-1} - 2p_i^{\ell} + p_i^{\ell+1}}{\Delta t^2} = \frac{p_{i-1}^{\ell} - 2p_i^{\ell} + p_{i+1}^{\ell}}{\Delta x^2} + s_i^{\ell}, \tag{3.6}$$

which gives us an explicit update scheme for inner points of the pressure field at time step $\ell + 1$ given the pressure fields at time step ℓ and $\ell - 1$

$$\boxed{p_i^{\ell+1} = 2p_i^{\ell} - p_i^{\ell-1} + c_i^2 \Delta t^2 \left(\frac{p_{i-1}^{\ell} - 2p_i^{\ell} + p_{i+1}^{\ell}}{\Delta x^2} + s_i^{\ell} \right)}\,. \tag{3.7}$$

If we consider homogeneous Dirichlet BDCs and homogeneous zero initial conditions on p such that

$$p_0^{\ell} = p_N^{\ell} = 0, \forall \ell \in \{0, \dots, N_t\}, \tag{3.8}$$

$$p_i^{-1} = p_i^{0} = 0, \forall i \in \{0, \dots, N\}, \tag{3.9}$$

we can use eq. (3.7) to iteratively compute the inner points (i.e. $i \in \{1, \dots, N-1\}$) of the pressure field at time step $\ell+1$ using the pressure fields at time step ℓ and $\ell - 1$, effectively computing the pressure field at all time steps $\{1, \dots, N_t\}$. This update scheme in time is usually referred to as the *explicit Euler* update scheme and it is one of the most simple update schemes in time but it is very practical because it allows us to treat the scheme as a simple *stencil* computation which can be easily parallelized (more info on this later in section 3.2).

We can use the same discretization and update scheme to compute the adjoint field λ by evolving it *backward in time*

$$\boxed{\lambda_i^{\ell-1} = 2\lambda_i^{\ell} - \lambda_i^{\ell+1} + c_i^2 \Delta t^2 \left(\frac{\lambda_{i-1}^{\ell} - 2\lambda_i^{\ell} + \lambda_{i+1}^{\ell}}{\Delta x^2} + \hat{s}_i^{\ell} \right)}\,, \tag{3.10}$$

where $\hat{s}_i^\ell$ is the discretized *adjoint source field*. Recall that we impose the same boundary conditions on λ as on p as well as homogeneous zero final conditions for λ, which can be expressed as

$$\lambda_0^\ell = \lambda_N^\ell = 0, \forall \ell \in \{0, \ldots, N_t\}, \tag{3.11}$$

$$\lambda_i^{N_t+1} = \lambda_i^{N_t} = 0, \forall i \in \{0, \ldots, N\}, \tag{3.12}$$

We will consider a simple L2 misfit like eq. (2.9) where we have observed data for the pressure field only at certain grid points positions $x_{r_1}, x_{r_2}, \ldots, x_{r_n}$ where n is the number of receivers. Then the discrete adjoint source field $\hat{s}_i^\ell$ at grid point i and time step ℓ is defined as

$$\hat{s}_i^\ell = \sum_{j \in \{1,\ldots,n\} | r_j = i} -(p_i^\ell - \hat{p}_{r_j}^\ell), \tag{3.13}$$

where $\hat{p}_{r_j}^\ell$ is the observed discretize pressure at time step ℓ of receiver at position x_{r_j}. The expression $(p_i^\ell - \hat{p}_{r_j}^\ell)$ comes from the derivative of the misfit functional with respect to the pressure $\delta_p \chi_p$. The minus sign in the expression comes from the definition of the adjoint equation (see eq. (2.42)).

Once we have both computed the pressure field and the adjoint field for all time steps, we can recover the sensitivity kernel $K(x_i)$ in a grid point x_i using the derived expression (see eq. (2.94)) which, after discretization, can be expressed as

$$\boxed{K(x_i) = -\frac{2}{c_i^3} \sum_{\ell=1}^{N_t-1} \lambda_i^\ell \frac{p_i^{\ell-1} - 2p_i^\ell + p_i^{\ell+1}}{\Delta t^2}}, \tag{3.14}$$

where we ignore the first and last time step since the initial and final conditions make their contribution vanish.

Variable density formulation

For the variable density formulation, we consider the generalized acoustic wave equation written as a first-order system of PDEs (see eq. (2.96)).

Let us consider the slightly more complex case of a two-dimensional model, where we have a rectangular domain $\Omega = [0, L_x] \times [0, L_y]$ where L_x and L_y (m) are the sizes of the domain in the x-direction and the y-direction respectively. Since we have a first-order system of PDEs to discretize, we define the so-called *staggered* grid such that the pressure field $\phi = p$ and the model parameters m_0, m_1 are collocated at the regular grid points, while the velocities $\boldsymbol{\nu} = [v_x, v_y]$ are collocated in between regular grid points. This is needed because the FD formula for the first-order derivative of a function f at a point x_i is

$$\frac{\partial f}{\partial x}(x_i) = \frac{f(x_i + \frac{\Delta x}{2}) - f(x_i - \frac{\Delta x}{2})}{\Delta x} + O(\Delta x^2), \tag{3.15}$$

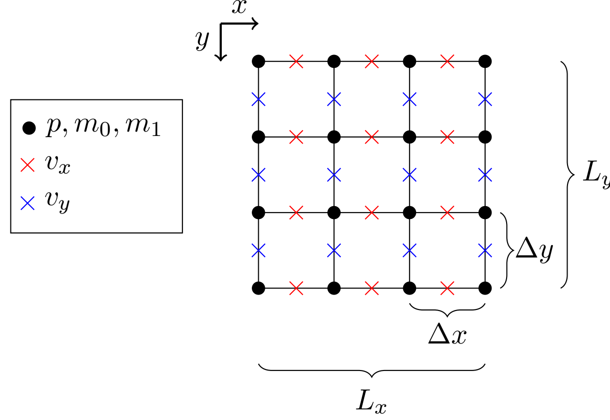


Figure 3.1: Staggered grid visualization for 2D pressure-velocity formulation.

hence we need the values of the function f in the middle of the grid points. In the 2D case, we will use the notation $f_{i+\frac{1}{2},j}$ or $f_{i,j+\frac{1}{2}}$ to denote values in between regular grid points. Figure 3.1 shows a visual representation of the staggered grid for the 2D pressure-velocity formulation where the fields have been staggered according to the spatial derivatives that are present in the system of equations.

Note that since the model parameter m_1 is collocated, by definition, at the regular grid points, we will need to *interpolate* it on the staggered points when computing the FD approximation of the first order derivatives. We will use a simple average interpolation of the model parameter m_1 such that

$$m_{1_{i+\frac{1}{2},j}} = \frac{m_{1_{i,j}} + m_{1_{i+1,j}}}{2}, \tag{3.16}$$

$$m_{1_{i,j+\frac{1}{2}}} = \frac{m_{1_{i,j}} + m_{1_{i,j+1}}}{2}. \tag{3.17}$$

Note that we now have two subscript indices: i and j which correspond to the x and y grid points directions respectively.

We also need to treat the discretization in time of the pressure p and the velocities v_x, v_y. We use a similar staggering technique where we collocate the velocities at the regular time steps $\ell \in \{0, \dots, N_t\}$, but we collocate the pressure in the middle of the time steps $\ell + \frac{1}{2}, \dots, N_t - \frac{1}{2}$, so the pressure field is now discretized as

$$p_{i,j}^{\ell+\frac{1}{2}} = p(i\Delta x, j\Delta y, (\ell + 1/2)\Delta t), \tag{3.18}$$

$\forall i \in \{0, \dots, N\}, \forall j \in \{0, \dots, M\}, \forall \ell \in \{0, \dots, N_t - 1\}$ where $M + 1$ is the number of grid points in the y-direction and Δy is the grid step size in the y-direction.

We have now all the tools to discretize the system in eq. (2.96) in the following

way

$$\frac{1}{m_{1_{i+\frac{1}{2},j}}}\frac{v_{x_{i+\frac{1}{2},j}}^{\ell+1}-v_{x_{i+\frac{1}{2},j}}^{\ell}}{\Delta t}+\frac{p_{i+1,j}^{\ell+\frac{1}{2}}-p_{i,j}^{\ell+\frac{1}{2}}}{\Delta x}=0 \tag{3.19}$$

$$\frac{1}{m_{1_{i,j+\frac{1}{2}}}}\frac{v_{y_{i,j+\frac{1}{2}}}^{\ell+1}-v_{y_{i,j+\frac{1}{2}}}^{\ell}}{\Delta t}+\frac{p_{i,j+1}^{\ell+\frac{1}{2}}-p_{i,j}^{\ell+\frac{1}{2}}}{\Delta y}=0 \tag{3.20}$$

$$m_{0_{i,j}}\frac{p_{i,j}^{\ell+\frac{1}{2}}-p_{i,j}^{\ell-\frac{1}{2}}}{\Delta t}+\left(\frac{v_{x_{i+\frac{1}{2},j}}^{\ell}-v_{x_{i-\frac{1}{2},j}}^{\ell}}{\Delta x}+\frac{v_{y_{i,j+\frac{1}{2}}}^{\ell}-v_{y_{i,j-\frac{1}{2}}}^{\ell}}{\Delta y}\right)=\tilde{f}_{i,j}^{\ell} \tag{3.21}$$

hence by rearranging the terms, we get the following explicit update scheme for pressure and velocities

$$\boxed{\begin{aligned} p_{i,j}^{\ell+\frac{1}{2}} &= p_{i,j}^{\ell-\frac{1}{2}}-\frac{1}{m_{0_{i,j}}}\Delta t\left(\frac{v_{x_{i+\frac{1}{2},j}}^{\ell}-v_{x_{i-\frac{1}{2},j}}^{\ell}}{\Delta x}+\frac{v_{y_{i,j+\frac{1}{2}}}^{\ell}-v_{y_{i,j-\frac{1}{2}}}^{\ell}}{\Delta y}\right)+\tilde{f}_{i,j}^{\ell} && \text{(3.22a)}\\ v_{x_{i+\frac{1}{2},j}}^{\ell+1} &= v_{x_{i+\frac{1}{2},j}}^{\ell}-m_{1_{i+\frac{1}{2},j}}\Delta t\frac{p_{i+1,j}^{\ell+\frac{1}{2}}-p_{i,j}^{\ell+\frac{1}{2}}}{\Delta x} && \text{(3.22b)}\\ v_{y_{i,j+\frac{1}{2}}}^{\ell+1} &= v_{y_{i,j+\frac{1}{2}}}^{\ell}-m_{1_{i,j+\frac{1}{2}}}\Delta t\frac{p_{i,j+1}^{\ell+\frac{1}{2}}-p_{i,j}^{\ell+\frac{1}{2}}}{\Delta y} && \text{(3.22c)}\end{aligned}}$$

We also need to specify homogeneous zero initial conditions on the pressure p and the velocities v_x, v_y as well as homogeneous Dirichlet BDCs for the pressure, hence

$$p_{0,j}^{\ell-\frac{1}{2}}=p_{N,j}^{\ell-\frac{1}{2}}=0,\forall j\in\{0,\dots,M\},\forall\ell\in\{0,\dots,N_t\}, \tag{3.23}$$

$$p_{i,0}^{\ell-\frac{1}{2}}=p_{i,M}^{\ell-\frac{1}{2}}=0,\forall i\in\{0,\dots,N\},\forall\ell\in\{0,\dots,N_t\}, \tag{3.24}$$

$$p_{i,j}^{-\frac{1}{2}}=0,\forall i\in\{0,\dots,N\},\forall j\in\{0,\dots,M\}, \tag{3.25}$$

$$v_{x_{i+\frac{1}{2},j}}^{0}=0,\forall i\in\{0,\dots,N-1\},\forall j\in\{0,\dots,M\}, \tag{3.26}$$

$$v_{y_{i,j+\frac{1}{2}}}^{0}=0,\forall i\in\{0,\dots,N\},\forall j\in\{0,\dots,M-1\}. \tag{3.27}$$

We follow a similar approach for the discretization of the adjoint equation where we keep the same discretization in space (i.e. collocating λ_2 in the regular grid points and $\boldsymbol{\lambda_1}=[\lambda_{1x},\lambda_{1y}]$ in the respective staggered points), while we change the time discretization by collocating $\lambda_{1x},\lambda_{1y}$ in the middle of the time steps. The reason for this choice will soon become apparent. The explicit update

scheme for the adjoint fields is given (without derivation) by

$$\lambda_{1x_{i+\frac{1}{2},j}}^{\ell-\frac{1}{2}} = \lambda_{1x_{i+\frac{1}{2},j}}^{\ell+\frac{1}{2}} - m_{1_{i+\frac{1}{2},j}} \Delta t \frac{\lambda_{2_{i+1,j}}^{\ell} - \lambda_{2_{i,j}}^{\ell}}{\Delta x} \tag{3.28}$$

$$\lambda_{1y_{i,j+\frac{1}{2}}}^{\ell-\frac{1}{2}} = \lambda_{1y_{i,j+\frac{1}{2}}}^{\ell+\frac{1}{2}} - m_{1_{i,j+\frac{1}{2}}} \Delta t \frac{\lambda_{2_{i,j+1}}^{\ell} - \lambda_{2_{i,j}}^{\ell}}{\Delta y} \tag{3.29}$$

$$\lambda_{2_{i,j}}^{\ell+1} = \lambda_{2_{i,j}}^{\ell} - \frac{1}{m_{0_{i,j}}} \Delta t \left(\frac{\lambda_{1x_{i+\frac{1}{2},j}}^{\ell+\frac{1}{2}} - \lambda_{1x_{i-\frac{1}{2},j}}^{\ell+\frac{1}{2}}}{\Delta x} + \frac{\lambda_{1y_{i,j+\frac{1}{2}}}^{\ell+\frac{1}{2}} - \lambda_{1y_{i,j-\frac{1}{2}}}^{\ell+\frac{1}{2}}}{\Delta y} \right) + \hat{f}_{i,j}^{\ell+\frac{1}{2}} \tag{3.30}$$

where $\hat{f}_{i,j}^{\ell+\frac{1}{2}}$ is the discretized adjoint source field. Again, we use the same boundary conditions as for the forward scheme and homogeneous final initial conditions for $\lambda_{1x}, \lambda_{1y}, \lambda_2$.

We will consider a simple L2 misfit like eq. (2.9) where we have observed data for the pressure field only at certain positions $(x_{rx_1}, y_{ry_1}), (x_{rx_2}, y_{ry_2}), \ldots, (x_{rx_n}, y_{ry_n})$ where n is the number of receivers. Then the discrete adjoint source field $\hat{f}_{i,j}^{\ell+\frac{1}{2}}$ at grid point i, j and time step $\ell + \frac{1}{2}$ is defined as

$$\hat{f}_{i,j}^{\ell+\frac{1}{2}} = \sum_{k \in \{1,\ldots,n\} | rx_k = i, ry_k = j} (p_{i,j}^{\ell+\frac{1}{2}} - \hat{p}_{rx_k,ry_k}^{\ell+\frac{1}{2}}), \tag{3.31}$$

where $\hat{p}_{rx_k,ry_k}^{\ell+\frac{1}{2}}$ is the observed discretize pressure at time step $\ell + \frac{1}{2}$ of receiver at position (x_{rx_k}, y_{ry_k}). The expression $(p_{i,j}^{\ell+\frac{1}{2}} - \hat{p}_{rx_k,ry_k}^{\ell+\frac{1}{2}})$ comes from the derivative of the misfit functional with respect to the pressure $\delta_p \chi_p$. Note that there is no minus sign like in the constant density case since the adjoint operator is equal to the forward operator but negated. Here we also see the reason for the choice of different discretization in time: since we previously computed the pressure in the middle of the time steps we can only inject the adjoint source field at these middle points which forces the above time discretization.

Once we have both computed the pressure field and the adjoint field for all time steps, we can recover the sensitivity kernels $K_{m_0}(x_i, y_j), K_{m_1}(x_i, y_j)$ in a grid point (x_i, y_j) using the derived expression (see eq. (2.105) and eq. (2.106)). The easiest one to recover is K_{m_0} which, after discretization, can be expressed as

$$K_{m_0}(x_i, y_j) = \sum_{\ell=0}^{N_t-1} \lambda_{2_{i,j}}^{\ell} \frac{p_{i,j}^{\ell-\frac{1}{2}} - p_{i,j}^{\ell+\frac{1}{2}}}{\Delta t}, \tag{3.32}$$

where we ignore the last time step since the final conditions make its contribution vanish. Notice that the discretization of the partial time derivative $\partial p / \partial t$ is taken

in the opposite direction since we are integrating backward in time (we did not notice this in the constant density case since the FD approximation for second-order derivatives is symmetric).

To recover K_{m_1} at the regular grid points we need to first compute its contributions in the staggered points and interpolate back to the regular grid points, so we have

$$K_{m_1}(x_{i+\frac{1}{2}}, y_j) = \frac{1}{m_{1_{i+\frac{1}{2},j}}} \sum_{\ell=0}^{N_t-1} \lambda_{1x_{i+\frac{1}{2},j}}^{\ell+\frac{1}{2}} \frac{p_{i+1,j}^{\ell+\frac{1}{2}} - p_{i,j}^{\ell+\frac{1}{2}}}{\Delta x}, \tag{3.33}$$

$$K_{m_1}(x_i, y_{j+\frac{1}{2}}) = \frac{1}{m_{1_{i,j+\frac{1}{2}}}} \sum_{\ell=0}^{N_t-1} \lambda_{1y_{i,j+\frac{1}{2}}}^{\ell+\frac{1}{2}} \frac{p_{i,j+1}^{\ell+\frac{1}{2}} - p_{i,j}^{\ell+\frac{1}{2}}}{\Delta y}, \tag{3.34}$$

and after using the same average interpolation as we used for the model parameter m_1 we get

$$K_{m_1}(x_i, y_i) = \frac{K_{m_1}(x_{i+\frac{1}{2}}, y_j) + K_{m_1}(x_{i-\frac{1}{2}}, y_j)}{2} + \frac{K_{m_1}(x_i, y_{j+\frac{1}{2}}) + K_{m_1}(x_i, y_{j-\frac{1}{2}})}{2}. \tag{3.35}$$

The last step of the discretization is to recover the sensitivity kernels with respect to wave speed and density K_ρ, K_c using the chain rule as in eq. (2.109) and eq. (2.110), such that

$$\boxed{\begin{aligned} K_\rho(x_i, y_j) &= -\frac{1}{\rho^2 c^2} K_{m_0}(x_i, y_j) - \frac{1}{\rho^2} K_{m_1}(x_i, y_j) \quad &(3.36a) \\ K_c(x_i, y_j) &= -\frac{2}{\rho c^3} K_{m_0}(x_i, y_j) \quad &(3.36b) \end{aligned}}$$

Numerical stability

The simple explicit Euler scheme is only *conditionally* stable, i.e. the pressure field does not suffer from numerical instability and does not *blow-up* only if a certain condition is met that relates the grid step size Δx, the time step size Δt and the maximum value of the wave speed $c_{\max} = \max(c_i), \forall i \in \{0, \dots, N\}$ (m/s). This condition is called the Courant-Friedrichs-Lewy (CFL) condition and it defines the *Courant number* C and a condition on it such that,

$$C = \frac{c_{\max} \Delta t}{\Delta h} \leq C_{\max}, \tag{3.37}$$

where $\Delta h = \Delta x$ is the grid step size (that is assumed to be the same in each dimension) and $C_{\max}$ depends on the accuracy order of the spatial FD approximation and the dimensionality of the problem. For second-order accurate FD approximations $C_{\max} = 1/\sqrt{D}$ in the D-dimensional case. For fourth-order accurate FD approximations using the staggered pressure-velocity scheme, an extra factor of 6/7 should be multiplied to $C_{\max}$.

Numerical dispersion

The scheme also suffers from *numerical dispersion*, meaning that the simulated waves show some undesired numerical artifacts similar to physical dispersion. To reduce this undesired effect, we need to consider the spatial discretization in relation to the minimum wavelength of the waves we want to model in our simulation. The minimum wavelength λ_{min} (m) is defined as

$$\lambda_{\text{min}} = \frac{c_{\text{min}}}{f_{\text{max}}}, \tag{3.38}$$

where $c_{\text{min}} = \min(c_i^2), \forall i \in \{0, \ldots, N\}$ (m/s) is the minimum value of the wave speed and f_{max} (Hz) is the maximum wave frequency we want to model. We can then define the number of *points per wavelength* $P_w = \lambda_{\text{min}}/\Delta x$ as the ratio between the minimum wavelength and the grid step size. Bigger values for P_w lead to better accurate simulations. We can get more points per wavelength by reducing the grid step size Δx or by increasing the minimum wavelength that we can model. A useful rule of thumb is to have at least 10 points per wavelength, although the error given by numerical dispersion depends also on the amount of time steps the simulation performs, so a more detailed analysis of numerical dispersion is needed for long simulations (see Igel [24] for more information).

Extension to higher-order stencils

It is often useful to consider spatial FD approximations which are more accurate than lowest-order ones, especially since they usually lead to less numerical dispersion. This means that we can define different FD approximations for first and second-order derivatives and exchange those definitions with the ones we have used in the previous section to obtain more accurate solutions. For example, a fourth-order accurate FD approximation of the first-order derivative of a function f at x_i is given by

$$\frac{\partial f}{\partial x}(x_i) = \frac{-f(x_{i+\frac{3}{2}}) + 27f(x_{i+\frac{1}{2}}) - 27f(x_{i-\frac{1}{2}}) + f(x_{i-\frac{3}{2}})}{24\Delta x} + O(\Delta x^4). \tag{3.39}$$

Higher-order FD stencils, however, require more function evaluations and more computations. Fortunately, as we will discuss in more detail in chapter 5, FD-based methods are memory-bound so having to perform more computations is generally not an issue. However, higher-order FD stencils do not come for free: we still need to perform more function evaluations which means reading more memory. Using the cache efficiently can expedite reads from memory, but it is not a trivial task since memory access becomes increasingly sparse as we go to higher-order stencils.

Also from a more practical point of view, higher order FDs require proper treatment of points on the boundaries since the stencils used are now wider and we may not have enough points to compute the derivatives on the boundaries.

The simplest solution is to compute the derivative approximations only on the positions for which we have sufficient points to compute it, hence effectively reducing the size of the domain by the extra width of the stencil used. This is what we have used in practice when implementing the fourth-order accurate FD solver for the variable density acoustic wave equation. Other approaches are based on using higher order stencils only for inner points and resort to lower order stencils as approaching the boundary. This means that the computations on the boundaries are less accurate, which can lead to errors propagating in the inner part of the domain as a result. More commonly, the so-called *fixture points* are added to the domain boundaries creating an extension of the boundaries such that a sufficient number of points is now available to compute the wider stencils.

3.1.2 C-PML boundary conditions implementation

We will now describe the implementation of absorbing C-PML boundary conditions in more detail. As introduced in section 2.4.2, C-PML BDCs are used to dampen the waves that reach the boundary of our domain of interest by introducing a memory variable per spatial derivative order, so in the case of the acoustic wave equation we will need two memory variables that we will name ψ and ξ.

Let us consider the 1D acoustic wave equation with constant density. We augment the second-order partial derivative in space with two more terms

$$\frac{\partial^2 p}{\partial \tilde{x}^2} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial \psi}{\partial x} + \xi, \tag{3.40}$$

for all points in the C-PML region. Since the discretization of the ψ and ξ memory variables needs to be chosen according to their spatial derivatives in the equation, we conclude that the ψ variable needs to be collocated in between the grid points (i.e. staggered) while the ξ variable needs to be collocated on the grid points.

The evolution of the memory variables is described by the following recursive equations

$$\psi^\ell = b\psi^{\ell-1} + a\left(\frac{\partial p}{\partial x}\right)^\ell, \tag{3.41}$$

$$\xi^\ell = b\xi^{\ell-1} + a\left[\left(\frac{\partial^2 p}{\partial x^2}\right)^\ell + \left(\frac{\partial \psi}{\partial x}\right)^\ell\right], \tag{3.42}$$

where ℓ indicates the time step at which the variables or the partial derivatives need to be taken and a, b are two coefficients that change based on the position in the C-PML region. The coefficients a, b are computed in the following way:

$$b(x) = \exp(-\Delta t(D(x) + \alpha(x))),$$
$$a(x) = D(x)\frac{b(x) - 1}{(D(x) + \alpha(x))},$$

and

$$D(x) = \frac{-(N+1)c_{\max}\log(R)}{2h} d^N(x),$$
$$\alpha(x) = \pi f_0(1 - d(x)),$$

where h (m) is the thickness of the C-PML boundary, N is a power coefficient, f_0 (hertz) is the dominating frequency of waves in the simulation, $c_{\max}$ (m/s) is the maximum wave speed in the model, R is a reflection coefficient and $d(x)$ is the normalized distance (between 0 and 1) from the interior part of the domain $\overline{\Omega}$ to the boundary of the domain. We picked some experimentally determined coefficients from reference studies of C-PML BDCs on the acoustic wave equation (see [21] and [22]). These are the following: power coefficient $N = 2$ and $R = [0.01, 0.001, 0.0001]$ for $[5, 10, 20]$ number of C-PML grid points respectively. Note that the coefficients can be computed once we know the following: the time step size Δt, the grid step size Δx and the number of C-PML grid points that determines the reflection coefficient R and the thickness of the C-PML region, as well as the dominating frequency of the waves in the simulations f_0 which can be chosen as the dominating frequency of the source time function s. This means that we need to recompute these coefficients when the dominating frequency changes, otherwise the waves will not be properly absorbed by the C-PML region.

The coefficients a, b also need to be discretized in order to be used in the FD stencils. Since they are both used in the recursive equations for ψ and ξ, they need to be discretized both in between the grid points and on the grid points. We use a similar staggered grid technique as we did for the generalized variable density acoustic wave equation, hence the FD stencil update schemes for the C-PML memory variables are as follows using the lowest-order FD approximations for first and second-order derivatives are as follows

$$\psi^{\ell}_{i+\frac{1}{2}} = b_{i+\frac{1}{2}}\psi^{\ell-1}_{i+\frac{1}{2}} + a_{i+\frac{1}{2}}\frac{p^{\ell}_{i+1} - p^{\ell}_i}{\Delta x}, \tag{3.43}$$

$$\xi^{\ell}_i = b_i\xi^{\ell-1}_i + a_i\left[\frac{p^{\ell}_{i-1} - 2p^{\ell}_i + p^{\ell}_{i+1}}{\Delta x^2} + \frac{\psi^{\ell}_{i+\frac{1}{2}} - \psi^{\ell}_{i-\frac{1}{2}}}{\Delta x}\right]. \tag{3.44}$$

while the modified update scheme for the pressure in the C-PML region becomes

$$p^{\ell+1}_i = 2p^{\ell}_i - p^{\ell-1}_i + c_i^2\Delta t^2\left(\frac{p^{\ell}_{i-1} - 2p^{\ell}_i + p^{\ell}_{i+1}}{\Delta x^2} + \frac{\psi^{\ell}_{i+\frac{1}{2}} - \psi^{\ell}_{i-\frac{1}{2}}}{\Delta x} + \xi^{\ell}_i + s^{\ell}_i\right). \tag{3.45}$$

We can use the same modified update scheme for the adjoint field λ in the C-PML region to solve the adjoint equation with C-PML boundary conditions.

Extension to higher dimensions is straightforward: a new set of C-PML memory variables is needed for each dimension, such that we will have, in the 2D case

for example, four memory variables $\psi_x, \psi_y, \xi_x, \xi_y$. Memory variables for the i-dimension are defined only on the boundaries incident to that dimension's direction, hence in the 2D case only the corners of the domain need C-PML memory variables updates in both dimensions.

As a last note, we mention the implementation for the variable density velocity-pressure formulation of the acoustic wave equation. In this case, only first-order spatial derivatives are present, so the definition of the C-PML memory variables is the same but ψ is going to be used on the pressure derivative, while ξ is going to be used on the velocity derivative, following the same concepts introduced above.

3.1.3 How to model point sources and source scaling

Modeling physical sources for the wave equation is generally a hard task since, in real-case scenarios, we do not know precisely the source time functions, i.e. the functions that model the source terms in the equation. Moreover, numerical approximations of these functions need to be carefully performed in order to produce correct wave amplitudes.

For the context of this thesis, we are mostly using *point sources*, meaning sources that are localized in an infinitesimally small point in space. These types of sources can be represented mathematically by δ-functions activated at some point $(\boldsymbol{x}_0, t_0)$ in space and time.

Recall the definition of δ-functions (taken by [24]) as

$$\delta(x) = \begin{cases} \infty & x = 0 \\ 0 & x \neq 0 \end{cases} \tag{3.46}$$

and such that

$$\int_{-\infty}^{\infty} \delta(x)dx = 1, \int_{-\infty}^{\infty} f(x)\delta(x)dx = f(0), \tag{3.47}$$

where $f(x)$ is a general function.

Physically, it is obviously impossible to have δ-functions as point sources, but it is nevertheless a reasonable approximation of a source located in the proximity of the modeled point source position.

Since we are modeling infinitesimally small point sources but we clearly do not have an infinitesimally small grid step size Δx after discretization, we need to deal with proper source time function rescaling based on the area of the element in which we are injecting the point source. This is commonly done my employing the so-called *boxcar function*, which is defined (in 1D) as

$$\delta_{\mathrm{bc}}(x) = \begin{cases} 1/\Delta x & |x| \leq \Delta x/2 \\ 0 & \text{otherwise} \end{cases}, \tag{3.48}$$

which fulfills the properties of a δ-function for $\Delta x \to 0$. In the n-dimensional case, we can generalize the definition of the boxcar function by introducing the

step size vector $\boldsymbol{\Delta} = [\Delta x_1, \ldots, \Delta x_n]^T$ and the generalized cube $C_n(\boldsymbol{x}_0, \boldsymbol{l})$ centered in $\boldsymbol{x}_0$ and with sides lengths equal to $\boldsymbol{l}$, which can be defined as a set of points using the following definition

$$C_n(\boldsymbol{x}_0, \boldsymbol{l}) = \{\boldsymbol{x} \in \mathbb{R}^n \mid |x_i| \leq x_{0_i} + l_i/2, \forall i \in \{1, \ldots, n\}\}. \tag{3.49}$$

The generalized boxcar function in n-dimensions is then defined as

$$\delta_{\mathrm{bc}}(\boldsymbol{x}) = \begin{cases} 1/V(C_n(\boldsymbol{0}, \boldsymbol{\Delta})) & \boldsymbol{x} \in C_n(\boldsymbol{0}, \boldsymbol{\Delta}) \\ 0 & \text{otherwise} \end{cases}, \tag{3.50}$$

where V is the volume function, which for the n-dimensional cube $C_n(\boldsymbol{x}_0, \boldsymbol{l})$ is just $V(C_n(\boldsymbol{x}_0, \boldsymbol{l})) = \prod_{i=1}^n l_i$.

In practice, we use the generalized boxcar function to rescale the source terms in the equations such that

$$s_{\mathrm{bc}}(\boldsymbol{x}, t) = s(\boldsymbol{x}, t)\delta_{\mathrm{bc}}(\boldsymbol{x}), \tag{3.51}$$

where $s_{\mathrm{bc}}(\boldsymbol{x}, t)$ is the rescaled source term that we use in the numerical simulations.

The procedure above ensures that the injected source terms behave like δ-functions (in the limit case of $\Delta x \to 0$) in the spatial domain, but we also have to make sure that the source time function behaves like a δ-function in the time domain. Unfortunately, we cannot directly use δ-functions in time since these functions have a white spectrum, i.e. they contain all frequencies and so they cannot be accurately modeled by the numerical discretization. Another function that has the same properties is the Gaussian function with standard deviation σ (s) which, in 1D, has the following form

$$\delta_{t_0,\sigma}(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{t^2}{\sigma^2}} \tag{3.52}$$

and fulfills the properties of a δ-function for $\sigma \to 0$. We can formulate the Gaussian function above with a different parametrization by substituting $\sigma = \frac{1}{\sqrt{2}\pi f_0}$ hence obtaining

$$\delta_{t_0,f_0}(t) = f_0\sqrt{\pi} e^{-\pi^2 f_0^2 (t-t_0)^2}, \tag{3.53}$$

where t_0 (s) is the *activation time* and f_0 (Hz) is the *dominating frequency.*

We can use $\delta_{t_0,f_0}(t)$ (or its derivatives in time) to model a source that is activated a time t_0 and has a dominating frequency of f_0. This function has the same properties of a δ-function $\delta(t - t_0)$ for $f_0 \to \infty$. The most common derivatives used are the first and the second derivatives, the latter also known as *Ricker wavelet.*

Depending on the dimensionality of the problem, one should choose different Gaussian derivatives depending on the shape of the resulting waveform one wants to obtain. For example, in the 1D case where we want the simulated waves to

have a Gaussian shape, we use the first derivative of the Gaussian since, in 1D, the resulting signal will be an integral of the source time function.

To test the correctness of numerical solutions for various problem dimensionalities, we can use analytical solutions in the form of *Green's functions* which can be computed for a single point source in the case of constant wave speed and density of the model. For more details on how to compute Green's functions in various dimensions for a generalized Gaussian-like source time function, we refer to Igel [24].

3.2 Technical details and implementation

In this section, we will discuss some technical details on the FD solvers we have implemented using the Julia [13] programming language. We developed a Julia package called `SeismicWaves.jl`[1] which is part of a larger set of packages (`HMCLab.jl`[2]) to perform forward and inverse calculations for geophysical problems focusing on the Hamiltonian Monte Carlo method. Although the main intended application for the package suite is that of geophysical problems, as explained in chapter 1, for the context of this thesis we have focused on the application for ultrasound medical tomography.

Most of the technicalities regarding efficient usage of computational resources are hardly ever discussed in detail in the literature, but are very important in practice since, without them, FWI approaches would take too much computation time and/or resources to be feasible.

3.2.1 Checkpointing for storage of the forward wave field

As we have seen in the previous sections, in order to compute sensitivity kernels for the material properties we want to reconstruct using FWI we need to perform 3 main steps: (1) solve a forward problem, (2) solve an adjoint problem, (3) integrate forward and adjoint wavefields in time to compute sensitivity kernels. Problems (1) and (2) are relatively easy to solve by using the FD methods as introduced in the previous sections.

Problem (3), however, has some technical intricacies because we need to integrate in time both forward and adjoint wavefields. Usually, when we solve the forward (or the adjoint) problem, we update the wave fields by keeping stored in memory only the most recent time step(s), hence performing a sort of *in-place* update on the wave field. This is fine when we are solving the two problems one after the other, but for computing the sensitivity kernels we somehow have to access both wave fields at the same time for the same time steps. Naively, this can be done by storing the forward wave field for all time steps while solving the forward problem and then computing the sensitivity kernels *on-the-fly* with

[1] `https://gitlab.com/JuliaGeoph/SeismicWaves.jl`
[2] `juliageoph.gitlab.io/HMCLab.jl`

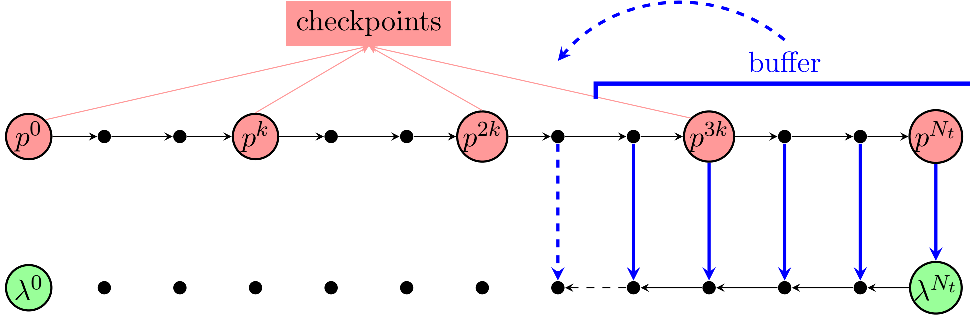


Figure 3.2: Linear and equally-spaced checkpointing scheme for the second order constant density acoustic wave equation solver.

the current time step adjoint wave field values. However, this approach quickly reaches unfeasible memory requirements for some realistic number of time steps used in practice and for simulations using 2D or 3D models. We could potentially store the wave field values not on the main memory of the device (which for the CPUs is the RAM), but on the disk, which would be feasible but doing this will greatly increase the computation time because of the I/O bottleneck of reading and writing to the disk.

What is usually done in practice is to save the state of the solver at some predetermined time steps, called *checkpoints*. We can then later use the information stored in the checkpoints to recompute the values of the forward wave field for a contiguous interval of time steps, which we will refer to as *buffer*, and use them when needed in the adjoint solver to compute on-the-fly the sensitivity kernels. Various checkpointing techniques can be used for this (see [26]) and we chose to implement the most simple form of checkpointing, namely *linear equally-spaced checkpointing*.

Figure 3.2 illustrates the checkpointing scheme for the second-order constant density acoustic wave equation solver. A *checkpointing frequency* of k time steps is set such that each checkpoint has the same distance, in number of time steps, from each other. When we compute the forward solver, we store the values of the pressure field (and, for simulations with C-PML BDCs the values of the memory variables) at the checkpoint time steps $0, k, 2k, \ldots$. Starting from the last checkpoint, which is always less than k time steps away from the last time step N_t, we save all the pressure fields up to the last time step in the buffer, which stores at most $k+2$ pressure fields. After the forward solver, we start the computation of the adjoint solver from the last time step and we use the stored pressure fields in the buffer for the integration in time of the sensitivity kernel. We continue until we run out of saved pressure fields in the buffer. At that point, we recompute all the pressure field values for the time steps between the second-last checkpoint and the last checkpoint time step starting a new partial forward solver by using the information stored at the second-last checkpoint. We shift

the buffer by repopulating it with the recomputed pressure fields and start using them in the time integration. We repeat this process until we reach the first time step and the integration in time for the sensitivity kernel is completed.

Using this technique, we have $\lceil N_t/k \rceil$ checkpoints and the length of the buffer is $k+2$ such that we need to store $\lceil N_t/k \rceil + k + 3$ pressure fields instead of the N_t we would have to store without checkpointing. If we choose $k = \lceil \sqrt{N_t} \rceil$, we find the optimal checkpoints-buffer ratio such that the number of wave fields we need to store is in the order of $O(\sqrt{N_t})$, or more precisely is $2\lceil \sqrt{N_t} \rceil + 2$.

Generalization of this checkpointing scheme to the variable density pressure-velocity formulation is straightforward: we now need to store in the checkpoints the pressure field as well as the velocity fields, making the number of fields to store dimension-dependent. In 2D, for example, we will have to store $3\lceil N_t/k \rceil$ fields for the checkpoints and only $k+1$ pressure fields for the buffer since we need first-order derivatives in time on the pressure field for the computation of sensitivity kernels.

3.2.2 Device-agnostic (xPU) solvers

The high computational cost of FWI-based inversions requires efficient and scalable usage of computational resources to make the method effective in practice. For this reason, most modern FWI implementations and code bases have some sort of parallelization features or can run on massively parallel architectures like GPUs (Graphics Processing Units), possibly also distributing the computations on multiple devices.

The explicit FD schemes introduced in this chapter have the nice feature of being easily parallelizable by using the famous *stencil* parallelization pattern. A stencil computation is, loosely speaking, a type of computation pattern where several time-step iterations on a given array are performed. During a time step, all elements in the array are updated using a fixed pattern that involves neighboring elements only. Stencils computations are usually *memory-bound* (more information on this in chapter 5) and GPGPUs (General-Purpose computations on Graphics Processing Units) approaches have been found to be most efficient [27].

Implementing and maintaining code for GPUs can be quite a challenge, especially for researchers who have no extensive skills in programming and computer science. Moreover, correctly using all the GPU capabilities and implementing optimized code with a high degree of efficiency is no easy task even for expert programmers.

Motivated by the choice of the Julia programming language, we based our implementation on the `ParallelStencil.jl`[3] package [14], which allows writing easy-to-read but efficient parallel stencil implementations that can run on both multi-threaded CPUs and GPUs. `ParallelStencil.jl` relies on the native kernel programming capabilities of `CUDA.jl` and `AMDGPU.jl` and on the Julia

[3] `https://github.com/omlins/ParallelStencil.jl`

`Base.Threads` for high-performance computations on both NVIDIA and AMD GPUs as well as CPUs. The package is particularly useful for non-computer scientists because it abstracts the different programming paradigms for CPUs and GPUs under a unified interface, resulting in implementing the code only once and then being able to run it seamlessly on both CPUs and GPUs, thus enabling true xPU computing. Moreover, `ParallelStencil.jl` automatically chooses the best way to launch kernel computations on CPUs and GPUs based on the device architecture and the problem dimensionality, which is very important to achieve high performance.

```julia
@parallel_indices (i, j) function update_p!(
    pold, pcur, pnew, c, dt, dx, dy
)
    # Pressure derivatives in space
    d2p_dx2 = (pcur[i+1, j] - 2.0 * pcur[i, j] + pcur[i-1, j]) / (dx^2)
    d2p_dy2 = (pcur[i, j+1] - 2.0 * pcur[i, j] + pcur[i, j-1]) / (dy^2)
    # Update pressure using explicit Euler scheme
    pnew[i, j] = 2.0 * pcur[i, j] - pold[i, j] + c[i, j]^2 * dt^2 *
    ↪  (d2p_dx2 + d2p_dy2)

    return nothing
end
```

Listing 1: Example of an xPU kernel function definition using `ParallelStencil.jl`'s `@parallel_indices` macro for 2D constant density acoustic pressure update scheme.

The usage of `ParallelStencil.jl` functionalities is best shown with an example. Listing 1 shows an example of a `ParallelStencil.jl` kernel function implemented using the `@parallel_indices` macro. This macro is used to define a kernel function that computes a single stencil update for an element of the stencil computation by defining the indices variables `(i, j)` in a way similar to a simple nested loop iterating over indices of the array. In this case, the kernel function update for the 2D constant density acoustic wave equation for the new time step pressure `pnew` at index `(i, j)` using the value of the current time step pressure `pcur` at the same index and its neighboring values to compute the derivatives using the second-order accurate FD approximation of the second order derivative. The values of the previous time step pressure `pold` and the wave speed `c` at index `(i, j)` are then used to perform the explicit Euler update scheme.

We note that the code is all written in basic Julia syntax and the kernel function only needs to be annotated with the macro `@parallel_indices` and the name of the indices to use in the computation. This is possible because of the Julia meta-programming capabilities which make `ParallelStencil.jl` generate

the appropriate code for CPUs and GPUs based on kernel function definition.

```
1 # Time loop
2 for it = 1:nt
3     # Update pressure
4     @parallel (2:(nx-1), 2:(ny-1)) update_p!(pold, pcur, pnew, c, dt,
      ↪  dx, dy)
5     # Inject sources
6     @parallel (1:nsrcs) inject_sources!(pnew, srctf, possrcs, it)
7     # Record receivers
8     @parallel (1:nrecs) record_receivers!(pnew, traces, posrecs, it)
9     # Swap refereces for next time step
10    pold, pcur, pnew = pcur, pnew, pold
11 end
```

Listing 2: Example of a time loop implementation for single-xPU 2D constant density acoustic pressure forward solver using using `ParallelStencil.jl`'s `@parallel` macro for kernel calls and the definition of the `update_p!` xPU kernel from listing 1. The functions `inject_sources!` and `record_receivers!` are similarly defined xPU kernels for source time function injection and recording of receivers (definitions not shown).

The kernel function can be called in a time loop update iteration using the `@parallel` macro as shown in listing 2. When called, the kernel needs to know the range of indices on which the stencil is applied. In this case, we only need to specify the range of inner points `(2:(nx-1), 2:(ny-1))` where `nx` and `ny` is the number of grid points in the x and y-direction respectively.

These examples showcase the ease of use of `ParallelStencil.jl` in the context of finite difference computations. In chapter 5 we show the results of benchmarks conducted to assess the efficiency of the solvers implemented using `ParallelStencil.jl` functionalities.

3.2.3 From single xPU to multi-xPUs

In the previous section, we have seen an example of `ParallelStencil.jl` and how it can be used to write xPU kernels on a single device. However, if we want to achieve very high-resolution model reconstructions using FWI-based methods, the problem becomes prohibitively large to be handled by a single device. This motivates the usage of the `ImplicitGlobalGrid.jl`[4] package which allows a seamless extension of `ParallelStencil.jl` to multi-xPU computing. The `ImplicitGlobalGrid.jl` package introduces the possibility of splitting the domain into different sub-domains and distributing them onto different devices

[4] https://github.com/eth-cscs/ImplicitGlobalGrid.jl

```
1 # Time loop
2 for it = 1:nt
3     # Hide communication block
4     @hide_communication b_width begin
5         # Update pressure
6         @parallel (2:(nx-1), 2:(ny-1)) update_p!(pold, pcur, pnew, c,
          ↪  dt, dx, dy)
7         # Inject sources
8         @parallel (1:nsrcs) inject_sources!(pnew, srctf, possrcs, it)
9         # Record receivers
10        @parallel (1:nrecs) record_receivers!(pnew, traces, posrecs, it)
11        # Exchange halo of new pressure with other processes/devices
12        update_halo!(pnew)
13    end
14    # Swap refereces for next time step
15    pold, pcur, pnew = pcur, pnew, pold
16 end
```

Listing 3: Example of a time loop implementation for multi-xPU 2D constant density acoustic forward solver using using `ImplicitGlobalGrid.jl`'s `@hide_communication` macro to hide computation behind communication and `update_halo!` function to perform the communication between neighbouring xPU devices.

using the capabilities of MPI-based communication offered by the `MPI.jl` package. Each device computes a single time loop iteration by performing a stencil computation on its sub-domain and then exchanges the boundary values of its sub-domain to each neighboring device. This is repeated for each time step, allowing a very simple but efficient parallelization on multiple devices.

The package `ImplicitGlobalGrid.jl` has a lot of interesting features like the automatic implicit creation of the global computational grid based on the number of devices the solver is run with and based on the network topology, which can be explicitly chosen by the user or automatically defined. Another important feature is the possibility of hiding communication behind computation by leveraging the fact that we can first compute the update on the boundary points and start the communication of those values while updating the rest of the inner points. This allows for maximal usage of computational resources and close to zero idle time due to communication. Moreover, `ImplicitGlobalGrid.jl` uses the highly optimized CUDA-aware or ROCm-aware MPI for GPGPUs to perform updates of the boundary values close to the hardware limit.

An example of the modified time loop for the 2D constant density acoustic forward solver using `ImplicitGlobalGrid.jl` and `ParallelStencil.jl` is shown in listing 3. We can see that only two modifications were added to the code, namely the `@hide_communication` macro and the `update_halo!` function. The

former specifies that the communication of boundary values should be hidden behind the computation of the inner values. The variable `b_width` is a tuple specifying the size of the boundary in number of grid points for each dimension for which computation should be done immediately and then communication of boundary values using `update_halo!` function can begin while updating the inner points.

Apart from a simple initialization procedure of the implicit global grid (not shown) and a different way of launching the solver using Julia and the MPI launcher, we see that the code is basically the same as before. We only need to take care of the correct initialization of the sub-domain sizes for each device, which can be done using other utility functions exposed by `ImplicitGlobalGrid.jl`.

In chapter 5 we show that good weak scaling on multiple GPUs can be achieved using `ImplicitGlobalGrid.jl` features in combination with `ParallelStencil.jl` for a relatively large problem size in 2D and 3D.

Chapter 4
Inversion results

In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment, and a very easy one, but people do not practise it much. In the every-day affairs of life it is more useful to reason forwards, and so the other comes to be neglected.
There are fifty who can reason synthetically for one who can reason analytically.
Let me see if I can make it clearer. Most people, if you describe a train of events to them, will tell you what the result would be. They can put those events together in their minds, and argue from them that something will come to pass.
There are few people, however, who, if you told them a result, would be able to evolve from their own inner consciousness what the steps were which led up to that result.
This power is what I mean when I talk of reasoning backwards, or analytically.

Sherlock Holmes, by Sir Arthur Conan Doyle

In this chapter, we will show the results of various synthetic inversion experiments using the implemented solvers for both constant and variable density acoustic wave equation formulations. We will also show a real inversion scenario example using the publicly available data set coming with the `pyruct` [28] Python package.

We start by checking the implementation of constant and variable density gradient solvers by computing hockey stick plots and comparing the gradients from the adjoint method with the expensive finite difference approximation gradients.

We then move on to synthetic inversions, i.e. using as observed data $\boldsymbol{d}^{\text{obs}}$ the seismograms computed by solving the forward problem using the same forward solver used to compute the gradients. We test both constant and variable density inversions in two different setups: ultrasound medical imaging and exploration seismic tomography. We also consider adding correlated noise to the 'observed' seismograms computed from the true model to mimic a noisy, real data scenario.

Finally, we show inversion results from real data gathered in an ultrasound medical imaging setup.

4.1 Adjoint and gradient checking

Before jumping into synthetic inversion, we need to check that the implementation of the adjoint solvers and the gradients computed from the sensitivity kernels are correct. There are multiple ways to test the correct implementation of adjoint solvers, the most common one being comparison with finite differences gradient approximations.

As introduced in section 2.3, we can compute gradients with respect to model parameters using a finite difference approximation like eq. (2.19), but this ap-

proach scales very poorly when increasing the number of model parameters. For 1D and relatively small 2D problems this approach is still feasible and can be used to compute gradients in a reasonable amount of time. These gradients can be then compared to the gradients computed with the adjoint method to test the correct implementation of the adjoint solvers.

4.1.1 Hockey stick plots

The first setup we used for gradient checking is a 1D model of length 1000 m initialized with a constant wave speed $c = c_0 = 1500$ m/s and constant density $\rho = \rho_0 = 1000$ kg/m^3. We then perturbed this model by adding a Gaussian perturbation to it of the form

$$g(a, x) = a \exp\left(\frac{-(x-500)^2}{500}\right). \tag{4.1}$$

The parameter a is chosen differently for the wave speed perturbation and the density perturbation. For the wave speed, we chose $a = -100$ while for the density we chose $a = 1000$. Applying only one perturbation at a time we get two perturbed models: one in which the wave speed is perturbed such that $c(x) = c_0 + g(-100, x)$ and the density is kept constant to ρ_0; the other in which the density is perturbed such that $\rho(x) = \rho_0 + g(1000, x)$ and the wave speed is kept constant to c_0.

In both models, we placed a source at $x = 400$ m and 10 receivers equally spaced between 300 and 600 m. The source time function used is the first derivative of a Gaussian with central frequency $f_0 = 50$ Hz and activation time $t_0 = 0.03$ s. The model is discretized using a uniform grid with 2001 grid points, hence the grid step size is $\Delta x = 0.5$ m. Both simulations with both models are run for a total time of $T = 0.3$ s with a time step size of $\Delta t = 3\text{e}{-4}$ s. The two described models were used to compute observed data $\boldsymbol{d}^{\text{obs}}$ while the constant velocity and density models were used to compute synthetic data $\boldsymbol{d}$. We used a simple L2 misfit with no regularization for this test.

We then computed gradients with respect to wave speed for the observed data computed from the model with perturbed wave speed and with respect to density for the observed data computed from the model with perturbed density using both the finite difference approach and the adjoint method approach. We used a first-order finite difference approximation for the gradients computed with the finite difference approach.

Figure 4.1 shows the so-called *hockey stick* plots for both perturbed models. These plots show the relative derivative error (i.e. the relative error between the adjoint method's gradients and the finite difference gradients) as a function of the finite difference increment Δm_i. The 'handle' of the stick (i.e. the right side of the plots) shows the finite difference approximation error, while the 'head' for the stick (i.e. the left side of the plots) shows the floating points precision

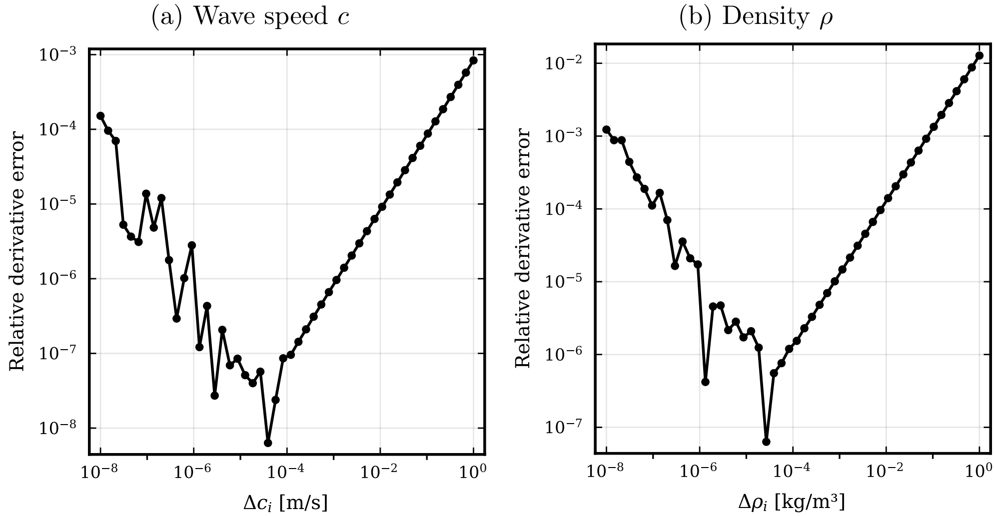


Figure 4.1: Hockey stick plots for 1D variable density solvers on two perturbed models. Panels 4.1a and 4.1b show the relative error for wave speed and density model parameters respectively located at position $x_i = 500$ m.

errors because the increment becomes so small that machine precision errors are predominant.

The expected behavior of the hockey sticks indicates that the adjoint solvers have a good chance of being implemented correctly, but it is not a 100% fault-proof test. To gather more insight into the correctness of our implementation we can compare visually the finite difference gradient approximation against the one computed with the adjoint method. This can be done in 1D using the same setup, but we were more interested in 2D setups because those are the ones that we will then use to perform inversions, so we need to check their implementation as well.

4.1.2 Finite differences gradient comparison in 2D

We move to a 2D setup much similar to what we used in the previous section, but now all the model perturbations are Gaussians in 2D. The 2D model size is now 1000×1000 m. Here we perturbed both wave speed and density models simultaneously, thus creating a single perturbed model. We again computed observed data using the perturbed model while a constant wave speed and density model at $c_0 = 1000$ m/s and $\rho_0 = 1500$ kg/m^3 is used to compute synthetic data.

Each of the 10 sources is activated separately while all 10 receivers are measuring seismograms for each source activation, thus creating 10 *shots* hence 10 different shot gathers. The source time function for each source is again the first derivative of a Gaussian with central frequency $f_0 = 10$ Hz in this case and activation time $t_0 = 0.4$ s. The model is discretized using a uniform grid with 201

grid points per spatial dimension (i.e. the grid has size 201×201 grid points), hence the grid step size is $\Delta x = \Delta y = 5$ m. Simulations are run for 1000 time steps with a time step size of $\Delta t = \Delta x/\sqrt{2}/1300 * 6/7 \approx 2.3\text{e}{-3}$ s. We used 4 C-PML layers of 20 grid points on every boundary side with a reflection coefficient $R = 1\text{e}{-4}$. We used a simple L2 misfit with no regularization for this test.

Figure 4.2 shows the perturbed model setup with sources-receivers configuration as well as gradients with respect to wave speed and density computed with the adjoint method. We note that the gradients with respect to wave speed are larger in absolute values compared to the density ones, even though the density perturbation is bigger compared to the wave speed one. This is because the L2 misfit is much more sensitive to wave speed perturbations than density ones. This behavior results from the chosen parametrization of eq. (2.95) with $m_0 = 1/(\rho c^2$ and $m_1 = 1/\rho$ where we can see that the impact of the density term is only through its gradient (or more precisely through the gradient of the inverse of the density). In fact, if the density is constant (i.e. its gradient is zero), it can be simplified in the equation. Since we have a rather smooth Gaussian perturbation on the density, its gradient will be rather small.

We also note that the gradients near the sources and receivers are quite large in absolute values compared to the ones in the middle where the perturbation is located. This is because sensitivity near the sources or receivers is higher. After all, perturbations in those regions will have a bigger effect on the L2 misfit that we are currently using for in this test.

By observing the zoomed-in panels fig. 4.2e and fig. 4.2f we can see the shape of the gradients (or sensitivity kernels) near the perturbation. The sign of the wave speed gradient is negative, meaning a positive perturbation is correctly detected. For the density gradient, we can see that it is both positive and negative, although it is mainly positive at the center of the density perturbation. The 'double banana' shape of the gradients is due to the perturbations but also to the positions of the sources and receivers, which in this example are at the top and the bottom of the domain respectively.

The comparison between adjoint gradients and finite difference approximation gradients is shown in fig. 4.3. The finite difference approximation of the gradients is computed by perturbing each model parameter by the increments $\Delta c = -1\text{e}{-3}$ m/s and $\Delta \rho = -1\text{e}{-3}$ kg/m^3 for wave speed and density respectively. In these figures, it is shown the base 10 logarithm of the relative error between adjoint and finite difference gradients.

We can see that the relative error is quite small in the inner region but gets bigger in the C-PML region. This is because the adjoint equations and sensitivity kernels we derived in section 2.4 are based on the assumption of homogeneous Dirichlet BDCs and not on C-PML BDCs. Since we generally do not want to update the model using the gradients in the C-PML region, we can ignore this error if we only use the gradients in the inner region when we update models using, for example, a general descent method.

If we look at the zoomed-in plots, we can see that the errors on the density

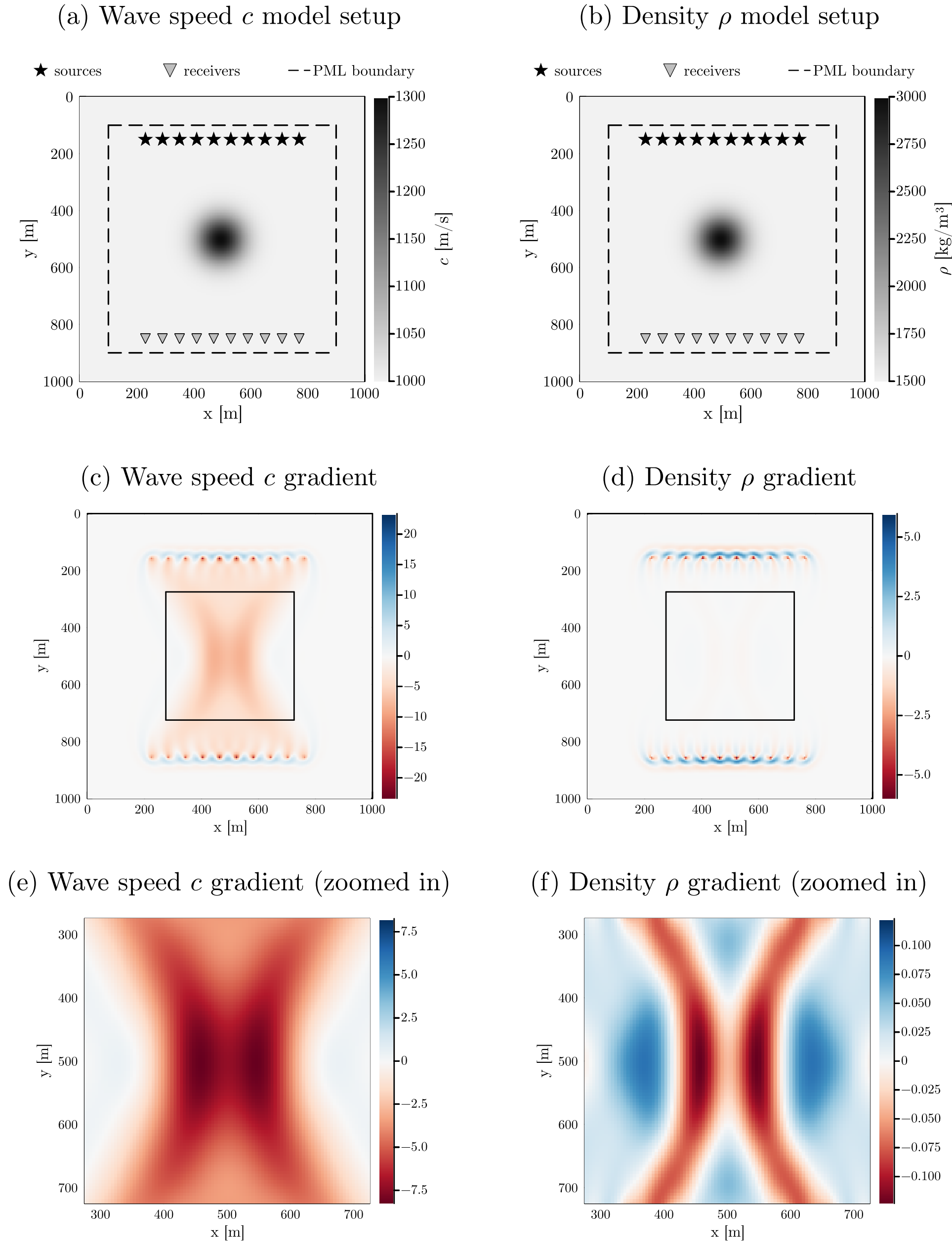


Figure 4.2: Adjoint gradients in 2D for gradient check setup. Panels 4.2a and 4.2b show the perturbed model setup for both wave speed and density as well as sources-receivers configuration. Panels 4.2c and 4.2d show the gradients computed with the adjoint methods. Panels 4.2e and 4.2f show a zoom-in of the gradients which correspond to the square black box of panels 4.2c and 4.2d respectively. Unit of measure for the gradients are s/m and m^3/kg for wave speed and density gradients respectively.

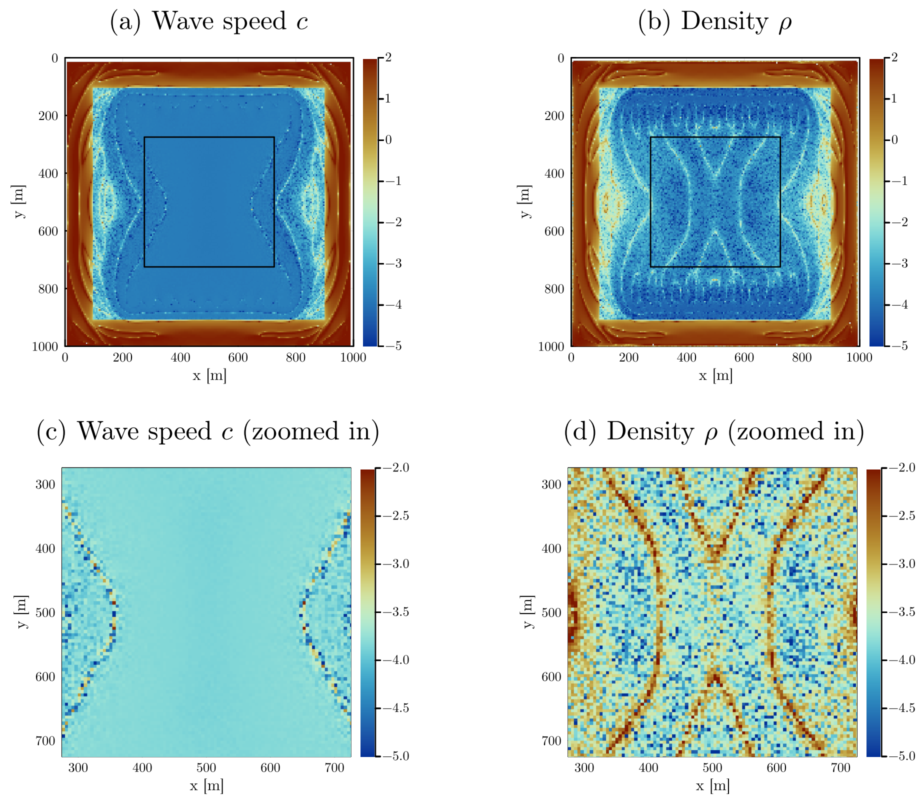


Figure 4.3: Relative error between adjoint and finite difference gradients. Panels 4.3a and 4.3b show the log10 relative error between the adjoint and finite difference gradients for both wave speed and density. Panels 4.3c and 4.3d show a zoom-in of both gradients.

gradients are higher than the ones for wave speed and regions with a sharper discontinuity in the density gradient have higher errors than the gradients in smoother regions. This is probably due to the interpolation needed to compute m_1 in between grid points and the interpolation needed to compute the sensitivity kernel with respect to density back onto the grid points.

We can conclude that, apart from the C-PML region, the adjoint gradients match quite accurately the finite difference approximations, hence the implementation seems to be correct. On a side note, we mention that it took 22 hours on a single GPU to compute all finite difference gradients for both wave speed and density, while with the adjoint method, all gradients were computed in just a couple of seconds.

4.2 Synthetic inversions

We now move to synthetic inversions, where the observed data is computed by solving the forward problem on the true model that needs to be reconstructed. This is a so-called *inverse crime* because the same solver that is used to generate the observed data is also used to perform the inversion. Although this procedure can lead to optimistic results, it is a common approach to benchmark the accuracy of a synthetic inversion before moving to real data inversions.

We will consider noiseless and noisy synthetic inversions: in the first case the generated data is used as is, while in the second case noise is added to the observed data to mimic measurement uncertainties. It is important to add some noise to the data in order to test the robustness of the inversion procedure to noise that will for sure be present in real data inversions.

4.2.1 Noiseless synthetic inversions

In this section we will take a look at two synthetic inversions in a variable density setting similar to ultrasound medical imaging ones, meaning that we will use wave speed and densities close to the ones of fluids or soft tissues in the human body, frequencies greater than 20 kHz[1] and model sizes in the order of 10s of cm.

The first model we will try to reconstruct is a phantom[2] composed of three circles: two smaller circles with different perturbations inside a bigger one. The diameter of the big circle is 10 cm while the diameter of the two smaller ones is 3 cm. The center of the big circle is located at the middle of the model, which has a size of 20×20 cm, and has a wave speed of 1480 m/s and a density of 1000 kg/m^3. The two smaller circles' centers are located at the same x-coordinate $x = 10$ cm while $y = 7.5$ cm for the top small circle and $y = 12.5$ cm for the

[1] In reality the frequency range used for medical purposes is usually much higher (between 2 and 18 MHz). We will use lower frequencies because higher frequencies alone cannot resolve sharp discontinuities. More on this in the section on real data inversions.

[2] The term *phantom* is frequently used in the medical imaging context to indicate a body with known material properties that is used to test imaging equipment.

bottom small circle. The top small circle has a wave speed of 1460 m/s and a density of 1010 kg/m^3. The bottom small circle has a wave speed of 1550 m/s and a density of 1040 kg/m^3. The background model has a wave speed of 1500 m/s and a density of 980 kg/m^3.

We have placed 16 sources and 32 receivers around the phantom in an equally spaced circular configuration of radius 7.5 cm. Each of the 16 sources is activated separately while all 32 receivers are measuring seismograms for each source activation, thus creating 16 shots. The source time function for each source is the first derivative of a Gaussian with central frequency $f_0 = 50$ kHz and activation time $t_0 = 2.4\text{e}{-5}$ s.

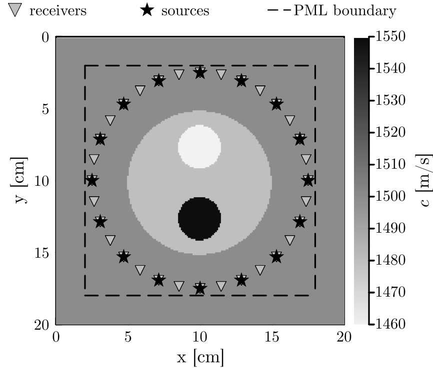
The model is discretized using a uniform grid with 200 grid points per spatial dimension, hence the grid step size is $\Delta x = \Delta y \approx 0.1$ cm. Simulations are run for 1500 time steps with a time step size of such that the Courant number of the forward simulation with the true model is 0.5. We used 4 C-PML layers of 20 grid points on every boundary side with a reflection coefficient $R = 1\text{e}{-4}$. We used a simple L2 misfit with no regularization for this inversion. We also applied gradient smoothing for each shot at the source location with a radius of 10 grid points. Figure 4.4a and fig. 4.4b show the true model setup as well as the sources-receivers configuration.

We performed 200 iterations of the L-BFGS algorithm filtering the gradients computed by the solver such that grid points inside the C-PML region would have zero gradient. The reconstructions for both wave speed and density at iteration 10 and 200 of L-BFGS are shown in fig. 4.4. We can see that the wave speed is reconstructed quickly while the density takes longer to be resolved. This is expected due to the same reasons we introduced in the previous section when we looked at the sensitivity kernels for both wave speed and density. The final reconstruction is quite similar to the true model, although discontinuities are not as sharp. We can also see that the sources-receivers configuration is somehow affecting the reconstruction by polluting the image with something that seems to resemble noise. In reality, this is due to the fact that, as we have seen previously, the gradient near the sources and receivers is generally larger (in absolute values) than the one where the perturbations are located.

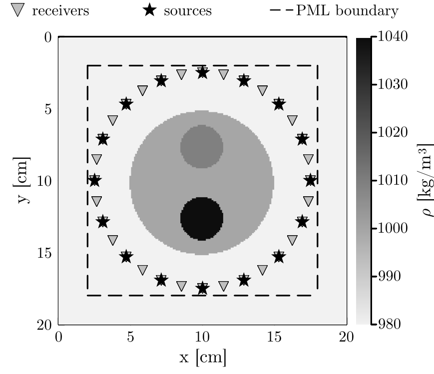
The second synthetic inversion we performed is almost identical to the previous one, except that the density of the true model is rotated 90 degrees counterclockwise with the center of rotation equal to the center of the model. This setup is a bit non-physical since real materials will hardly be able to match these model parameters. Nevertheless, it gives great insight into the coupling of wave speed and density gradients. Indeed, wave speed and density are related to each other via the Newton-Laplace equation eq. (2.77) so a perturbation on one will reflect on the other. This means that the two gradients are not independent from each other, so by testing this setup we try to see if this coupling is correctly modeled by our solvers.

Figure 4.5 shows the setup and reconstructed models for the rotated three circles phantom inversion. As in the previous inversion, the wave speed gets

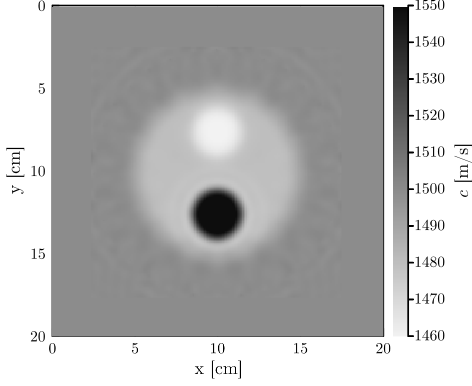
(a) Wave speed c true model + setup



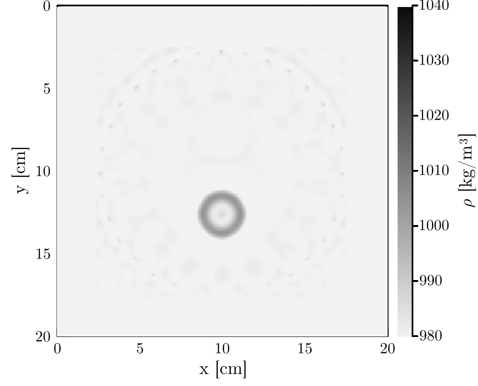
(b) Density ρ true model + setup



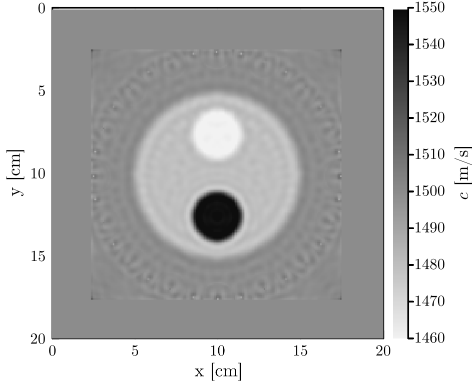
(c) Wave speed c model at iteration 10



(d) Density ρ model at iteration 10



(e) Wave speed c model at iteration 200



(f) Density ρ model at iteration 200

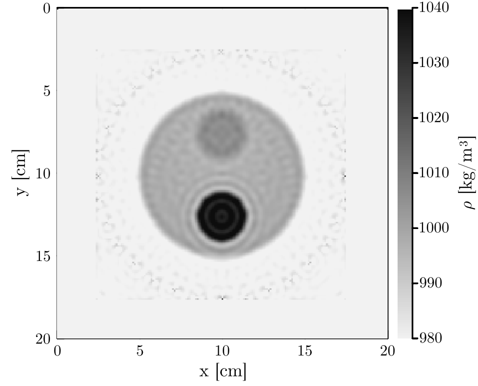
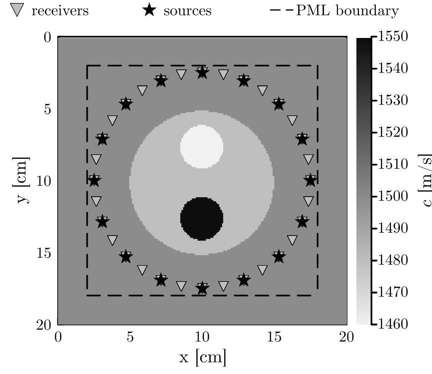
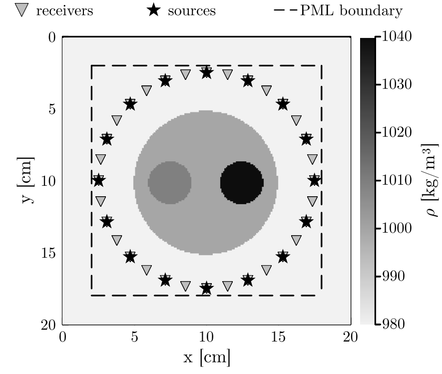


Figure 4.4: Three circles phantom inversion. Panels fig. 4.4a and fig. 4.4b show the true model setup as well as the sources-receivers configuration. Panels fig. 4.4c and fig. 4.4d and panels fig. 4.4e and fig. 4.4f show the reconstructed model parameters at iteration 10 and 200 of L-BFGS respectively.

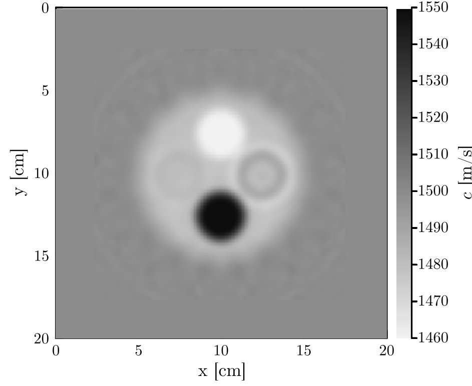
(a) Wave speed c true model + setup



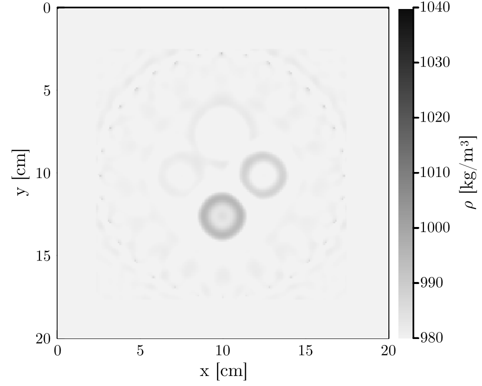
(b) Density ρ true model + setup



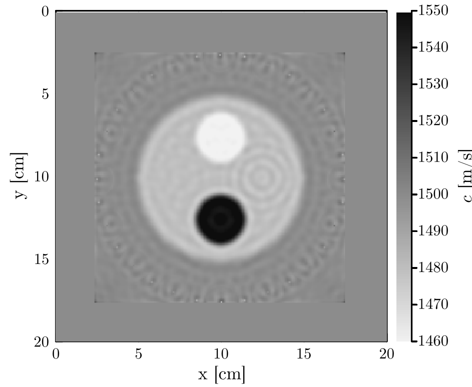
(c) Wave speed c model at iteration 10



(d) Density ρ model at iteration 10



(e) Wave speed c model at iteration 200



(f) Density ρ model at iteration 200

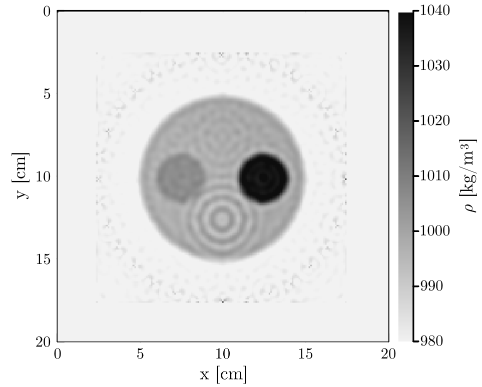
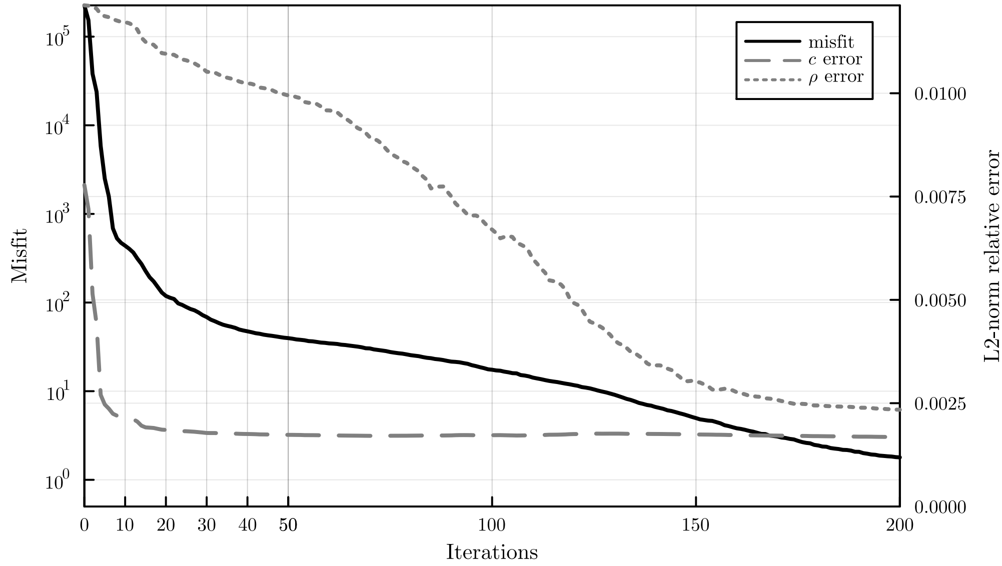


Figure 4.5: Rotated three circles phantom inversion. Panels fig. 4.5a and fig. 4.5b show the true model setup as well as the sources-receivers configuration. Panels fig. 4.5c and fig. 4.5d and panels fig. 4.5e and fig. 4.5f show the reconstructed model parameters at iteration 10 and 200 of L-BFGS respectively.

(a) Three circles phantom



(b) Rotated three circles phantom

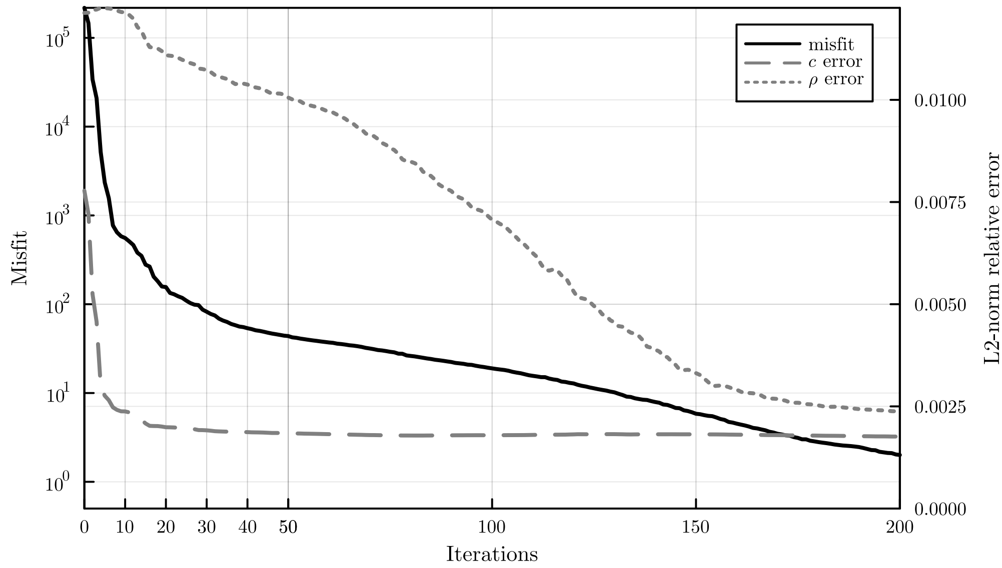


Figure 4.6: Misfit and relative error evolutions for circles phantom inversions. Panel 4.6a and 4.6b show the evolutions of the misfit functional and the relative error between the reconstructed model and the true model for the three circle phantom and its rotated variant. The misfit evolution y-axis is in log scale while the relative error y-axis is in linear scale.

resolved much faster than the density, although this time we can clearly see the effect of the coupling between wave speed and density. In the wave speed reconstruction at iteration 10 fig. 4.5c we see the circle with a higher density interfering, while in the density reconstruction fig. 4.5d we see the circle with a higher wave speed interfering. After the wave speed model is somewhat resolved both interferences start to even out, although never quite disappear as we can see in the reconstructions after 200 iterations in fig. 4.5e and especially in fig. 4.5f where we can see that the high wave speed circle is misinterpreted as a density perturbation.

We can get a better idea of what is happening if we observe the misfit evolution and the relative error $E_{r,i}$ of the reconstructed model at iteration i with respect to the true model computed as

$$E_{r,i} = \frac{\|\boldsymbol{m}_i^{\text{est}} - \boldsymbol{m}^{\text{true}}\|_2}{\|\boldsymbol{m}^{\text{true}}\|_2}, \tag{4.2}$$

where $\boldsymbol{m}^{\text{true}}$ is the true model, $\boldsymbol{m}_i^{\text{est}}$ is the reconstructed model at iteration i. This is shown for both inversions in fig. 4.6. We see that both the misfit and the relative error of the wave speed reconstruction rapidly decay in the first 10 iterations and flatten out, while the density error decays more slowly and picks up after roughly 100 iterations before flattening. This clearly shows that the density model is much harder to reconstruct than the wave speed model and also that a good wave speed reconstruction is crucial to correctly resolve density.

4.2.2 Noisy synthetic inversions

In this section, we will investigate the effect of noise added to the generated observed data to ensure the robustness of the inversion method. For this synthetic inversion, we will focus on a different model setup that is more similar to exploration seismic tomography, namely using the well-known SEG/EAGE Salt and Overthrust Models [29]. These models are commonly used in literature to benchmark inversion methods for exploration seismic tomography. We did this to show that our solvers can be used to invert models at different scales. Figure 4.8a shows the overthrust wave speed true model and the sources-receivers setup. Note that in this case, we assumed a constant density model, so we used the constant density solvers for this inversion. This is not true for real exploration seismic tomography settings, but we adopted it to facilitate a bit of the inversion and focus only on the problem of reconstructing a model using noisy data.

Regarding the noise, we injected *correlated noise* into our generated data using the FFT Moving Average (FFT-MA) method [30]. This means that the noise added to our data will not just be uncorrelated Gaussian noise, but instead, the noise will depend on the adjacent noise in time. The parameters of the correlated noise we injected are the following: Gaussian noise with mean 0 and standard deviation of 0.05, correlation length of 30 time steps. The effect of the noise in the data is shown in fig. 4.7, where we compare clean and noisy seismograms for

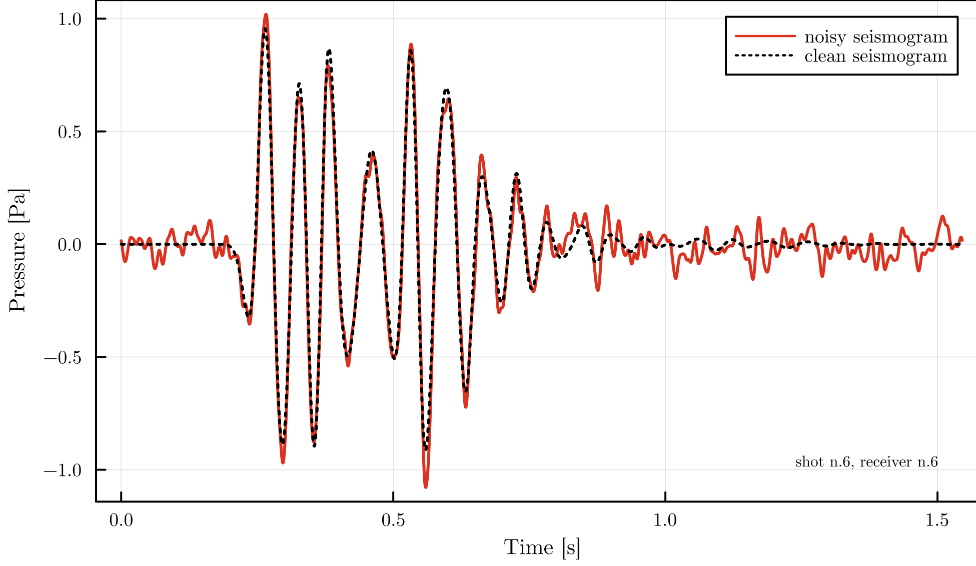


Figure 4.7: Comparison of clean and noisy seismograms for shot 6 receiver 6 of overthrust inversion.

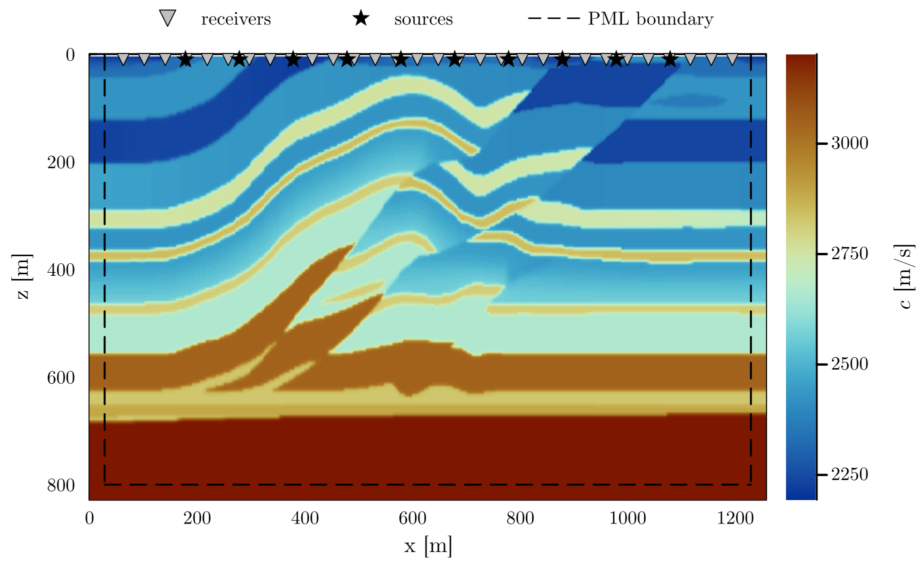
a specific source-receiver pair. We can see that this type of additive noise mostly 'corrupts' the parts of the signal with lower amplitude, but also some effect is seen in the higher amplitude parts with a small amplitude perturbation on the peaks.

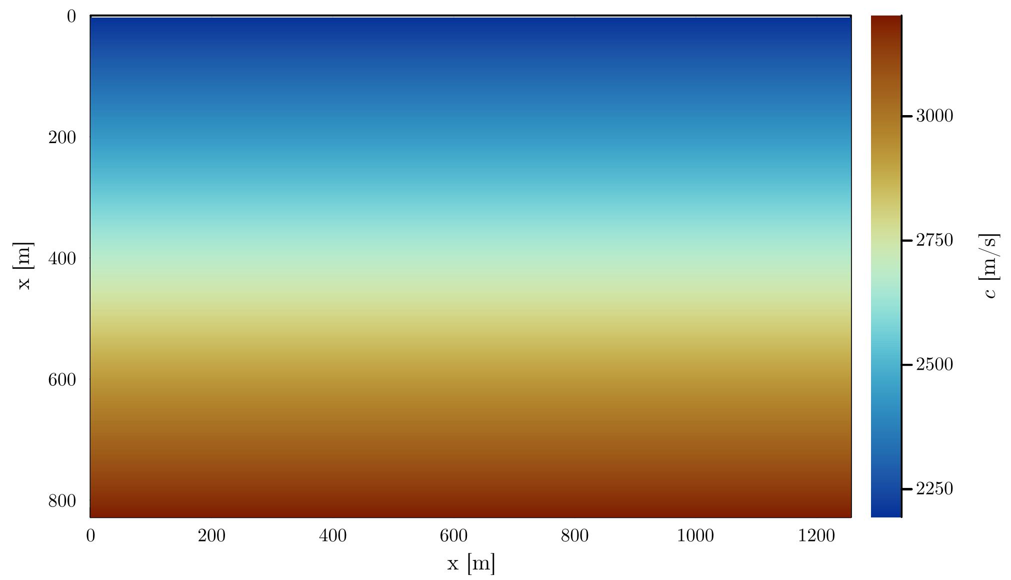
More details on the numerical setup are as follows. The model has a size of 1200 m in the x-direction and 800 m in the z-direction and has been discretized using 856×574 grid points, hence the resulting grid step sizes are $\Delta x \approx \Delta z \approx 1.4$ m. C-PML layers of 20 grid points and reflection coefficient $R = 1\mathrm{e}{-4}$ have been added to the model at the right and left boundaries as well as at the bottom boundary. For the top boundary, we used free surface BDCs (i.e. homogeneous Dirichlet BDCs in this case).

The initial model (fig. 4.8b) is set as a vertical gradient from top to bottom with minimum wave speed (at the top) of 2200 m/s and maximum wave speed (at the bottom) of 3200.0 m/s.

A total of 10 sources and 30 receivers are located at the top of the model at a depth of 10 m. The sources are equally spaced with a distance of 100 m from each other, while the receivers are also equally spaced but with a distance of 39 m from each other. The source time function used for each source is a Ricker function (second derivative of a Gaussian) with a central frequency of 12.0 Hz and scaled amplitude of 1000 Pa. Each of the 10 sources is activated separately while all 30 receivers are measuring seismograms for each source activation, thus creating 10 shots.

Simulations are run for 10000 time steps with a time step size of such that


(a) Wave speed c true model + setup
receivers
sources
PML boundary
0
200
400
600
800
z [m]
0
200
400
600
800
1000
1200
x [m]
3000
2750
2500
2250
c [m/s]


(b) Wave speed c initial model
0
200
400
600
800
x [m]
0
200
400
600
800
1000
1200
x [m]
3000
2750
2500
2250
c [m/s]

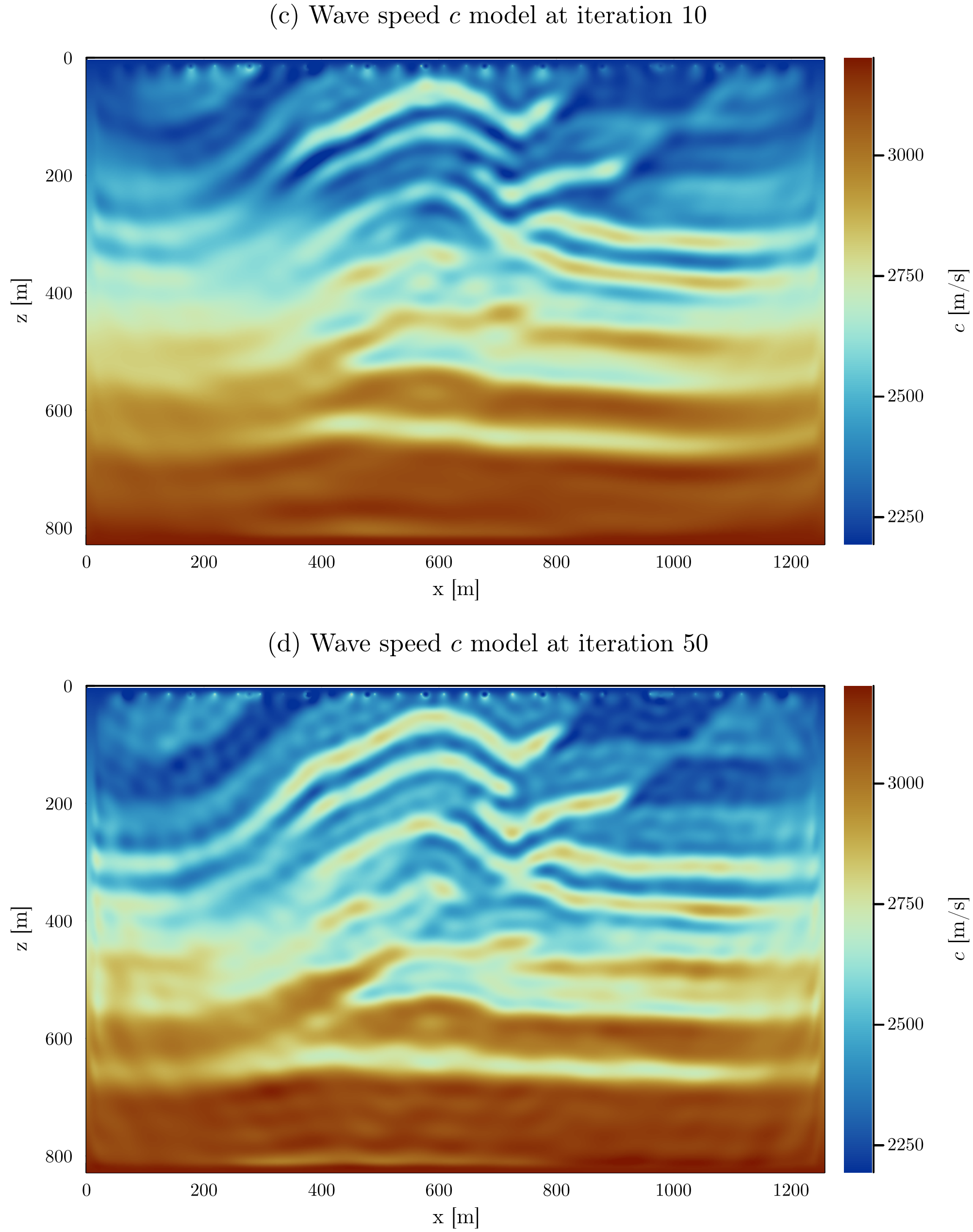


Figure 4.8: Noisy overthrust inversion. Panel fig. 4.8a shows the true model setup and sources-receivers configuration. Panel fig. 4.8b shows the initial model at iteration 0 we used for the inversion. Panels fig. 4.8c and fig. 4.8d show the reconstructed wave speed model at L-BFGS iteration 10 and 50 respectively.

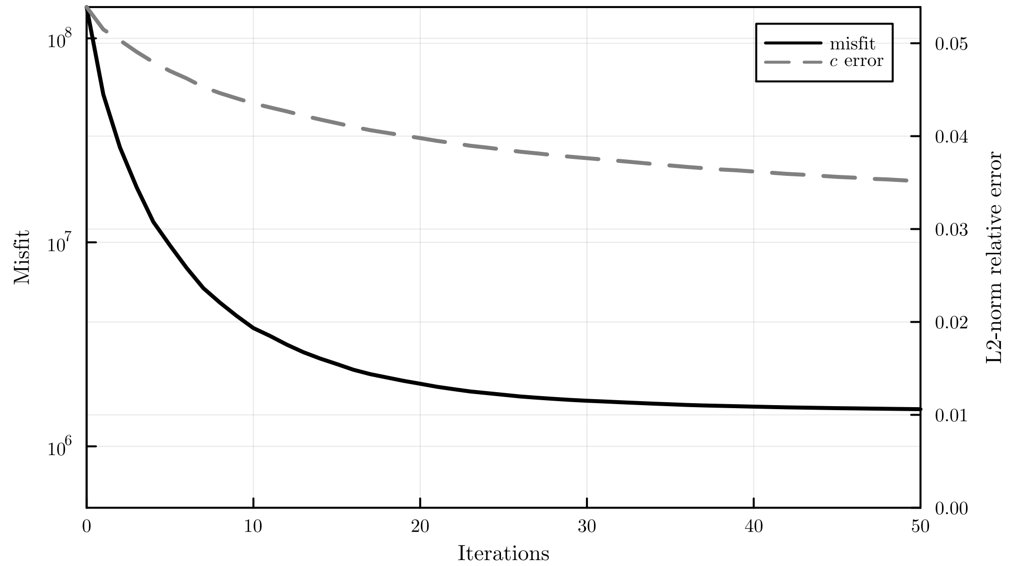


Figure 4.9: Misfit and relative error evolutions for noisy overthrust inversion.

the Courant number of the forward simulation with the true model is 0.5. We also applied gradient smoothing for each shot at the source location with a radius of 5 grid points. We performed 50 iterations of the L-BFGS algorithm using the whole gradient.

For the misfit, we used an L2 misfit like eq. (2.12) with a diagonal covariance matrix and the same standard deviation as we used for the correlated noise. We could have used a covariance matrix that perfectly modeled the correlated noise (in that case it would have not been diagonal anymore), but since we usually do not know exactly the amount of correlation in the noise in real case scenarios, we decided to be conservative about the correlated noise and only know its standard deviation correctly for the inversion.

Reconstructed models for iterations 10 and 50 are shown in fig. 4.8c and fig. 4.8d. We can see that most of the characteristics of the true model are resolved, at least for the upper half and center of the model. In the lower half and on the model sides the reconstruction is less accurate and smoothing of detailed structures is present. This is partially due to the noise but mostly due to the model geometry that forces the energy to be quickly dissipated to the boundary. In fact, waves that propagate from the sources at the top will, for the most part, travel along the discontinuities in the upper half of the model and reach the boundary where they will be dissipated. The few reflections and waves that go through the upper half discontinuities will be the ones that permit the reconstruction of the lower part of the model.

In fig. 4.9 we see the misfit evolution and the relative error of the reconstruction with respect to the true model. In contrast to what we have seen for the

noiseless inversions, we note that smoother evolutions of misfit and relative error appear, which is a consequence of the noise in our data as well as the choice of a smoother initial model to start the inversion with.

4.3 Real data inversions

In this section, we will show a case study inversion performed on a real data set in the context of ultrasound medical imaging. We used the public data set coming with the `pyruct` [28] Python package that was developed by researchers from the University of Zürich and ETH to reconstruct images using Reflection Ultrasound Computed Tomography (RUCT) Delay And Sum (DAS) Beamforming [31, 32].

The data provided with the package contained various data sets with observed data measured using different source-receivers configurations and different phantom models. We picked the 'test' data set which was obtained by submerging a hexagonal key in water surrounded by ultrasound transducers which could act as sources or receivers at the same time. For detailed information about the data set, the measurement methods, and the configuration of the sources and receivers, we refer to the original paper by Lafci et al. [28]. Nevertheless, we will briefly mention some relevant information about the inversion setup and the post-processing of the data we needed to perform to get a reasonable reconstruction.

First of all, we have filtered the raw data set using a band-pass filter in the frequency range of 1 to 2 MHz. Since the central frequency of the transducers in the experiment was set to 5 MHz, we decided to use lower frequencies in our inversion because of various numerical (e.g. points per wavelength requirement) and physical (e.g. high contrast between water and hexagonal key speed of sound) reasons. We also had to up-sample the data going from a sampling frequency of 24 MHz to 48 MHz, still because of numerical reasons on the time step Δt requirement for accurate and stable numerical simulations. The time step Δt of the numerical simulations is fully determined by the sampling frequency of the data, such that the smallest possible Δt we can use is $\Delta t = 1/f_s$ with f_s (Hz) being the sampling frequency.

Secondly, it is important to explain the transducer geometry setup. The transducers were positioned around the phantom (i.e. the hexagonal key in our case) in a double semi-circle geometry (one semi-circle at the top and one semi-circle at the bottom) forming an almost complete circle around the phantom with a radius of 40 mm. The number of transducers in the array is 512 which can be activated independently, hence the data set consists of 512 shots each of them containing 512 recorded seismograms. We did not use all of the shots, but we picked a subset of 26 equally spaced shot sources and, for each shot, we only used the furthermost 150 receivers from the shot's source. This was done for two different reasons. First, using all 512 shots would have been computationally expensive as we need to compute a forward and adjoint simulation for each shot. Of course, reducing the number of shots used also reduces the amount of spatial

ray coverage for the phantom reconstruction. Secondly, we used only the furthest receivers since, by observing the shot gather, we noticed that those were the ones containing most of the information regarding the phantom perturbation. Also, we saw some noise contamination (probably due to electrical interference) in the shot gathers that prohibited the use of all the data set. For this reason, we had to apply windowing on the observed data to filter out all noise coming from the contaminated region of the shot gather. In practice, this resulted in a modified misfit functional where the integral in time is split into different windows such that the synthetic and observed data outside of the chosen windows is not considered in the misfit computation.

We also had to estimate the source time functions for the transducers. This is a somewhat complicated step that is needed when dealing with real data since we do not know the source time functions of each source. Fortunately, we can recover a somewhat good approximation of the source time function by looking at the first arrivals of receivers adequately close the the source. The procedure we adopted to retrieve the source time function of a single source is explained as follows. After up-sampling and frequency filtering of the data, we computed an estimate of the first arrival travel times for receivers close to the source (i.e. with a distance between 10 and 20 mm from the source) using the water speed of sound given in the data set, which is 1490 m/s in our case. This assumes that we know the activation time of the sources, which we computed manually by looking at the seismograms of the same transducer that acted as a source and receiver for that shot. Using the information of the estimated first arrivals, we picked a window of 200 samples centered at the estimated arrival time for each receiver and summed all the windowed seismograms for all receivers, effectively computing a stacked source time function. We took the average over all receivers and smoothed the tails using a hamming window. This is the resulting source time function we used for the source. Of course, we needed to also know the central frequency of this source time function which we computed by taking the frequency value associated with the maximum magnitude in the frequency spectrum of the signal. The resulting source time function central frequencies, using the band-pass filter with the cutoffs as above, were all roughly around 1.3 MHz.

Finally, we need to spend a few words on the misfit functional we used for the inversion. As usually is the case for real data inversions, regularization is needed in order to not fit the uncertainties in the observed data, as well as a reasonable choice for the covariance matrix of the data noise. We used a misfit similar to the one in eq. (2.15). We assumed Gaussian noise in the observed data with a standard deviation of 0.2 that was computed by looking at the seismograms at the times when no wave field should be present, e.g. at the relaxed state. For regularization we used a simple zeroth order Tikhonov regularization with a constant prior model $\boldsymbol{m}^{\text{prior}}$ for both the wave speed and the density equal to the initial water model (i.e. with $c = 1490$ m/s and $\rho = 1000$ kg/m^3), covariance matrix in model space equal to the identity matrix and a regularization coefficient $\alpha = 1\text{e}{-4}$. We tested different regularization coefficients and managed to get the

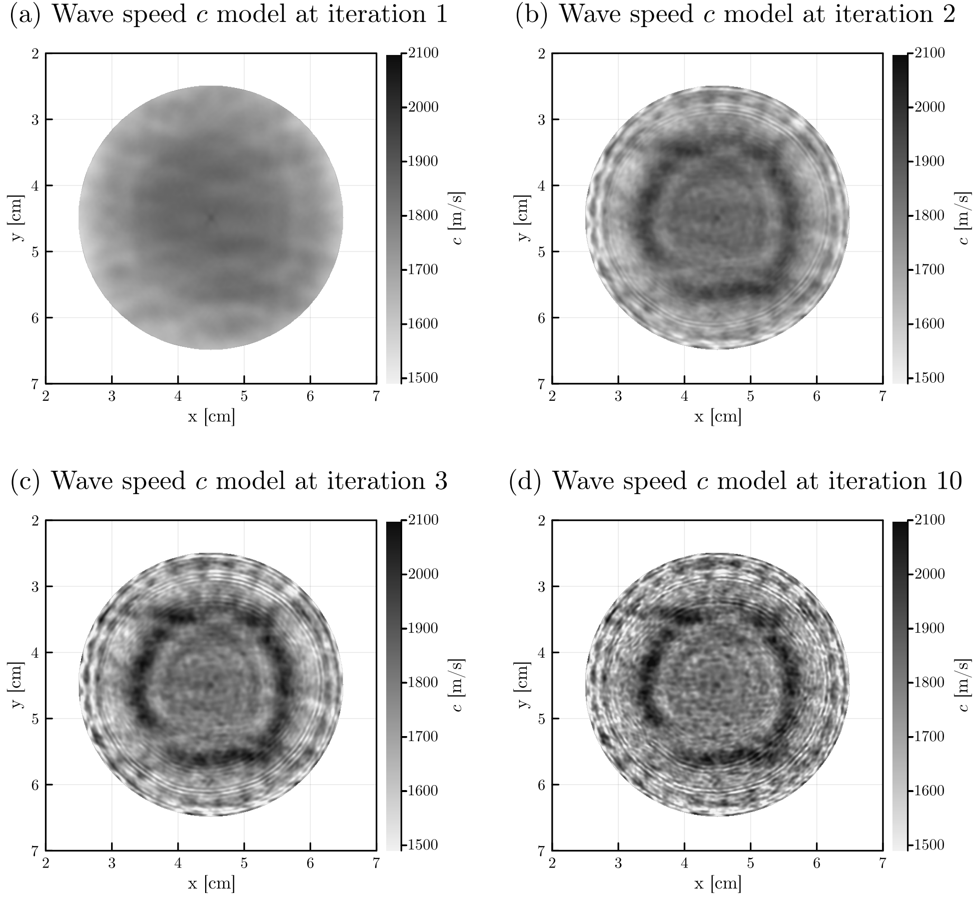


Figure 4.10: Hexagonal key inversion results. Panels fig. 4.10a, fig. 4.10b, fig. 4.10c and fig. 4.10d show the reconstructions for the wave speed at L-BFGS iterations 1, 2, 3 and 10.

best results with the one above.

We needed to translate all of this information into a numerical setup, so we chose to make a model of size 9×9 cm such that the center of the model would be equal to the center of the circular geometry setup of the transducer array. We discretized both wave speed and density model parameters using a uniform regular grid with a grid step size of $\Delta x = \Delta y = 8\mathrm{e}{-5}$ m, resulting in a grid with 1126×1126 grid points. C-PML layers of 20 grid points and reflection coefficient $R = 1\mathrm{e}{-5}$ on all boundaries have been used for the inversion. We ran the solver for 3800 time step iterations which was enough to cover all the useful samples of the observed data seismograms. We also applied gradient smoothing for each shot at the source location with a radius of 10 grid points and used only the gradients for model parameters with a distance less than 2 cm from the center of the model.

A total of 10 L-BFGS iterations have been performed for the inversion, which took around 15 minutes on a single NVIDIA A100 GPU. The reconstructed images are shown in fig. 4.10. We only show the wave speed reconstructions since the density ones just show noise. This is expected since, as we discussed in the previous section, we can only resolve density after wave speed gets resolved.

Looking at the reconstructed images, we can see the shape of the hexagonal key being reconstructed even at the first iteration. After a few iterations, the shape is clearer, but the inner part of the key is not correctly resolved. If we look closely, we can see that another smaller hexagonal shape is visible inside of the black outer hexagonal shape. We know that the size of the hexagonal key used as a phantom is 12 mm which is consistent with the size of the smaller key we see in the reconstruction. The wave speed inside the hexagonal key should be much higher since the speed of the sound in steel is in the order of 5000 m/s, but we highest wave speed we get is just 2100 m/s. Unfortunately, after 10 L-BFGS iterations we start to fit the noise and the image reconstruction does not improve.

It seems that using this frequency range of the data we can only resolve the hexagonal key shape but the size and specifically the wave speed magnitude are far from the true model. We think that because of the very high contrast (in the wave speed) at the boundary of the hex key, the inversion fails to accurately reconstruct the wave speed in the inner part of the phantom.

A better accurate boundary and estimate of the key size can possibly be done by better 'massaging' of the raw data set and, perhaps, using a lower frequency range to invert for. Unfortunately, this was not possible in our case since we observed that if we used lower frequencies (e.g. by using a lower-frequency band-pass filter) the noise in the data set started to dominate.

4.4 Summary

We have performed several experiments to check for the correctness of our solvers' implementation involving comparisons with finite difference gradient approximations in 1D and 2D. These comparisons showed a good match between the gradients computed with the adjoint method and the finite difference ones. This comparison was possible because of the low dimensionality of the models which let us compute the finite difference gradients in a reasonable amount of time.

Moving to synthetic inversions, we assessed the ability of our solvers to be used on inversions in constant and variable density cases as well as on different scales (ultrasound medical imaging and exploration seismic tomography). A noisy data inversion was performed to test the robustness of the inversion method to noise.

Finally, real data inversions showed the possibility for our solvers to be used in real ultrasound medical imaging settings, although with the need for data manipulation (frequency filtering, windowing, and source time function retrieval) and correct tuning of regularization parameters.

Chapter 5
Benchmarks

Report if the measurement values are deterministic.
For nondeterministic data, report confidence intervals of the measurement.

Rule 5 of 'Scientific Benchmarking of Parallel Computing Systems'
from Torsten Hoefler and Roberto Belli.

In this chapter, we will evaluate the efficiency and scalability of the solvers we have implemented. We will focus on the constant density kernels and solvers which are of easier interpretability because no interpolation nor staggering of fields is needed, but we note that similar results have been obtained for the variable density solvers as well.

We will first introduce the performance metrics used to assess the efficiency of our solvers as well as discuss the benchmarking setup in detail with a comprehensive description of the devices used and timing methods to ensure robust and reproducible benchmarks.

After this brief introduction, we will look at the results for both CPUs and GPUs benchmarks on the kernel functions, i.e. the functions that compute a single time step update using the explicit Euler scheme introduced in chapter 3. We will then move on to more complex benchmarks of the full forward solvers and talk about the overhead time of setting up the simulation. In the end, we will give a preview of the scalability of the proposed solvers by showing a weak scaling multi-GPU benchmark.

5.1 Benchmark metrics, setup, and statistical analysis of run time measurements

Choosing the right metrics to evaluate numerical code is a critical step to assess its efficiency and performance and it is a difficult task because the code usually involves different algorithms and numerical methods that interact differently with the hardware on which they are executed. Generally speaking, we can divide algorithms into two categories, namely *compute-bound* and *memory-bound* algorithms. Compute-bound algorithms have the property of doing much more computations than memory operations, hence their computation time is bounded by the speed at which the computations are performed. On the other hand, memory-bound algorithms perform a lot of memory operations and not so many computations, so their computation time is bounded by how fast memory operations are executed.

These definitions of compute-bound and memory-bound algorithms are a bit loose: what does it mean that an algorithm performs 'a lot of memory operations', 'not so many computations', or 'much more computations than memory operations'? To be more precise on the above definitions we need to introduce the concept of *operational intensity* of an algorithm. The operational intensity of an algorithm $I = W/Q$ (flop/byte) is defined as the number of (floating point) operations W (flop) performed by the algorithm divided by its transferred bytes Q (byte) from main memory to the execution unit. To be a bit more general, the operational intensity can be expressed as a function of the input size n such that $I(n) = W(n)/Q(n)$ where the number of operations and the transferred bytes are now also functions of the input size.

In practice, W and Q can be computed by counting the number of operations and the amount of memory transfer performed by the execution unit. This counting task can be hard if the algorithm's number of operations or the memory access pattern does not depend only on the size of the input but also on the input data itself. If an algorithm's memory access pattern does not depend on the input data but only on its size, then the algorithm is called *data-oblivious*. Data-oblivious algorithms are especially nice to deal with because W and Q can be usually computed analytically for a specific input size n. Fortunately, all numerical methods based on finite differences introduced in chapter 3 are data-oblivious, hence we can write analytical formulas for the number of operations and the memory transfer.

Returning to the computational intensity I, how can this 'ratio' be used to assess whether an algorithm is compute- or memory-bound? If the algorithm's operational intensity I is *bigger* than the ratio between the peak performance π (flop/s) and the memory bandwidth β (byte/s) of the execution unit, we say that the algorithm is compute-bound on the execution unit. On the other hand, if it is smaller then the algorithm is memory-bound on the execution unit. This means that an algorithm can be compute-bound on some execution units and memory-bound on others. Also, an algorithm can be compute-bound or memory-bound based on different input sizes n.

In the context of this thesis, we deal with finite difference methods which are inherently memory-bound on most architectures. In fact, more and more algorithms are destined to become memory-bound in the future because of the increasing gap between peak performance and memory bandwidth [33]. Table 5.1 shows the vendor peak performance π for double precision floating points (FP 64), memory bandwidth β, and their ratios π/β for two different state-of-the-art CPU and GPU. We can see that, in the optimal case, many floating point operations can be executed per transferred byte before reaching the compute-bound regime. In particular, the last row of table 5.1 shows the number of double precision floating points operations that can be executed *per operand*, e.g. per floating point. This number is way bigger than any reasonable number of floating points operation per operand we would ever need to reach the compute-bound regime, even for very high order spacial derivative stencils.

Device	AMD EPYC 7282	NVIDIA A100 40GB SXM
π (FP 64)	700 Gflop/s	9.7 Tflop/s
β	85.3 Gbyte/s	1.55 Tflop/s
π/β	$\approx$ 8.2 flop/byte	$\approx$ 6.2 flop/byte
$\pi/\beta *$ sizeof(double)	$\approx$ 65.5 flop	$\approx$ 50 flop

Table 5.1: Peak performance, memory bandwidth, and their ratios for an AMD EPYC 7282 CPU and an NVIDIA A100 40GB SXM GPU.

This reasoning motivates the adoption of a metric that can be used to evaluate the performance of memory-bound algorithms. In contrast to what we would use if we were in a compute-bound regime, namely the algorithm's performance (flop/s), we are interested in the algorithm's *memory throughput* (byte/s) instead, which measures the amount of memory transferred per unit of time. Monitoring the 'raw' memory throughput alone is, however, not an indication that our implementation performs well. Having a close-to-peak memory throughput just means that most of the device's memory bandwidth is used, but it does not necessarily mean that it is used *efficiently*. We need to find a way to express the *useful* memory transfers of our algorithm, meaning the ones that actually contribute to progress in the algorithm execution.

Effective memory access. One possible measure used to express the concept of useful memory transfers is the *effective memory access* A_{eff} (byte) of an algorithm. This metric represents the *strictly needed* memory transfers that an algorithm must perform for its execution. Since when we talk about memory transfers we usually mean transfers from main memory to the lowest-level cache (LLC), a few things should be noted regarding its computation. We must not consider memory transfers from LLC to the execution unit as those are much faster than transfers from main memory to LLC. This can be ensured by considering the start of the algorithm execution to be in a cold cache scenario, meaning that the cache is not yet populated or it has been invalidated by the previous algorithm's execution in the case of repeated execution.

For the context of this thesis, we consider the computation of A_{eff} for one iteration of a general iterative stencil-based finite difference solver, which is

$$A_{\text{eff}} = 2U + K, \tag{5.1}$$

where U (byte) represents all the fields that are updated, hence loaded and stored from memory, while K (byte) represents the fields that are only read from memory but not updated. The above assumption of cold cache at the start of each iteration for the FD solver is motivated if we assume that the problem

size is much bigger than the size of the cache and that usage of time-blocking techniques is not feasible or advantageous for real-world applications.

As an example, we can compute the effective memory access of an iteration of the 1D pressure update for the scalar second-order acoustic wave equation with constant density using the finite difference stencil introduced in chapter 3. The field that gets updated is the pressure field p which is loaded at the current time step and stored for the next time step, so $U = n * 8$ byte if n is the number of grid points in the discretization and we consider double precision floating points that have size of 8 bytes each. The fields that are only read are the pressure field p at the previous time step and the speed of propagation field c, so $K = 2n * 8$ byte. Putting all together, we obtain $A_{\text{eff}} = 2U + K = 4n * 8$ byte which of course depends on the number of grid points in the discretization. Similar results can be computed for the 2D and 3D cases, as well as in the case of different update schemes like the first-order variable density staggered scheme we use to solve the variable density wave equation. The contributions of the C-PML auxiliary fields can be added as well using the same principles, although it is negligible for very large numbers of grid points since the size of the C-PML region scales in the order of $O(n^{d-1})$ instead of $O(n^d)$ where d is the number of spatial dimensions.

Effective memory throughput. We can use the effective memory access to compute the *effective memory throughput* T_{eff} (byte/s) which is defined as

$$T_{\text{eff}} = \frac{A_{\text{eff}}}{t}, \tag{5.2}$$

where t (s) is the execution time of the algorithm. This metric measures the number of effective memory transfers per unit of time, hence it can be used as a measure for the performance of a memory-bound algorithm. The higher the effective memory throughput the higher the algorithm's performance.

Percentage of peak memory bandwidth. The effective memory throughput of an algorithm can then be compared to the memory bandwidth of the device β, to produce the relative metric called *percentage of peak memory bandwidth* P_{peak} (%) achieved by the algorithm which is defined as

$$P_{\text{peak}} = \frac{T_{\text{eff}}}{\beta} * 100. \tag{5.3}$$

In the above definition, β must be chosen carefully based on the device used to perform the benchmark. In most scenarios, using the theoretical maximum memory bandwidth of the device (usually specified by the vendor) is not realistic, since these values can only obtained by performing ad-hoc computations and are (plausibly) never obtained in real-world applications. The 'real' memory bandwidth of the device must almost always be computed using a benchmarking software like STREAM [33] (for CPUs) or GPU-STREAM [34] (for GPUs), which perform

various simple computations like memory copy or the famous triad $c = \alpha * a + b$ where a, b, c are vectors and α is a scalar value. This will give us a more realistic value for the memory bandwidth of the device which can be obtained in practice. We will distinguish the vendor's memory bandwidth of the device β from the benchmarked peak memory bandwidth by calling the latter one the *peak memory throughput* T_{peak} (byte/s) of the device. For all practical purposes, we can replace β with the benchmarked T_{peak} to compute the percentage of peak memory bandwidth P_{peak} which will then become

$$P_{\text{peak}} = \frac{T_{\text{eff}}}{T_{\text{peak}}} * 100. \tag{5.4}$$

Benchmarking setup

We benchmarked our finite difference solvers implementations on both multi-core CPUs and GPUs. For single node benchmarks, we had access to a workstation from the Seismology and Wave Physics group at ETH equipped with an AMD Ryzen Threadripper PRO 5995WX and an NVIDIA RTX 4070 GPU, and another workstation, courtesy of the Laboratory of Hydraulics, Hydrology and Glaciology at ETH, equipped with an AMD EPYC 7282 @ 2.8GHz CPU and a total of 8 NVIDIA Ampere A100 40GB SXM GPUs. For multi-node benchmarks, we had access to the CSCS Piz Daint XC50 compute nodes[1] which are equipped with Intel Xeon E5-2690 v3 @ 2.60GHz CPU and an NVIDIA Tesla P100 16GB GPU on each node.

We used Julia 1.8.3 for all the single-node benchmarks, while for multi-node benchmarks we used Julia 1.7.2 which was available on Piz Daint at the time of benchmarks. We used the following flags for the Julia runtime: `-O3 -check-bounds=no`[2]. These flags ensure a high degree of code optimizations, as well as disabling bounds checking to speed up indexed array access. For multiple time-step benchmarks, we disabled the Julia garbage collector (GC) for the duration of the time loop to not interfere with the measurement times. The CUDA version used for GPU benchmarks is 12.1 for all nodes except for the Piz Daint node where we had CUDA 11.0.2 available. Since we also wanted to test the efficiency of the C-PML region computations, we fixed the number of C-PML layers to 20 for all models used for benchmarking. This is usually a reasonable amount of layers used in real applications.

Statistical analysis of run times

To compute the effective memory throughput of an algorithm (or more specifically of a function in the implemented code) we need to measure run times. This procedure can be very complex for a variety of reasons. The most predominant

[1] `https://www.cscs.ch/computers/piz-daint`

[2] From Julia 1.9 onward, the `-check-bounds=no` flag is not supported and usage of `@inbounds` macro is preferred.

one is for sure the non-deterministic behaviour of the operating system scheduler which leads to different run times every time we execute the piece of code we are measuring. For this reason, statistical analysis of run times is crucial to ensure fair, robust, and reproducible measurements. We followed some of the guidelines from Hoefler and Belli [35] to analyze and report our measurements. In particular, we adopted two different strategies to measure run time for single-node and multi-node measurements.

For single node measurements (either multi-core CPU or single GPU), we wanted to make sure we could gather consistent runtime measurements for kernels, i.e. the computation of a single time step iteration. For this reason, we used the `BenchmarkTools` package to execute multiple runs of the kernel we wanted to benchmark. We decided to let the package decide how many runs would be needed to get a big enough set of run times for statistical analysis, although the main reasoning behind choosing how many runs to sample was to get as many of them to reach at least a few seconds of computation. This usually ensures that the device is properly warmed up.

After gathering a set of measurements, we need to apply some statistical checking procedures and summarize them. We computed the 95% non-parametric confidence interval for the median run time of the set and checked that it was within 5% of the median itself. This ensures that the median run time is a robust and reproducible indicator of the average run time, so we chose this value to summarize the set of measurements. This is the reason why we do not show confidence intervals on the reported mean run times or derived metrics such as effective memory throughput because they are guaranteed to be within 5% of the median run time.

For multi-node measurements, we could not benchmark a single time step iteration, because communication between nodes could be captured only for multiple time steps. For this reason, we chose a different approach to compute run times for our multi-xPUs implementations. We chose the best-performing model size from single xPU benchmarks and measured enough time steps to reach at least a few seconds of runtime while discarding the first few iterations to have warmed-up measurements only. We divide the measured runtime by the number of time steps and get a 'per time step' runtime. We could, in principle, have adopted a similar strategy as the single node measurements (i.e. multiple runs, statistical checking, etc), but we did not have enough node hours available on Piz Daint to get the samples needed, so we decided to stick with single run measurements.

5.2 Single node xPU kernel microbenchmarks

In this section, we will focus on the performance of the various kernel functions that compute one-time step iteration using the explicit Euler scheme introduced in chapter 3. This type of benchmark is usually referred to as *microbenchmark*

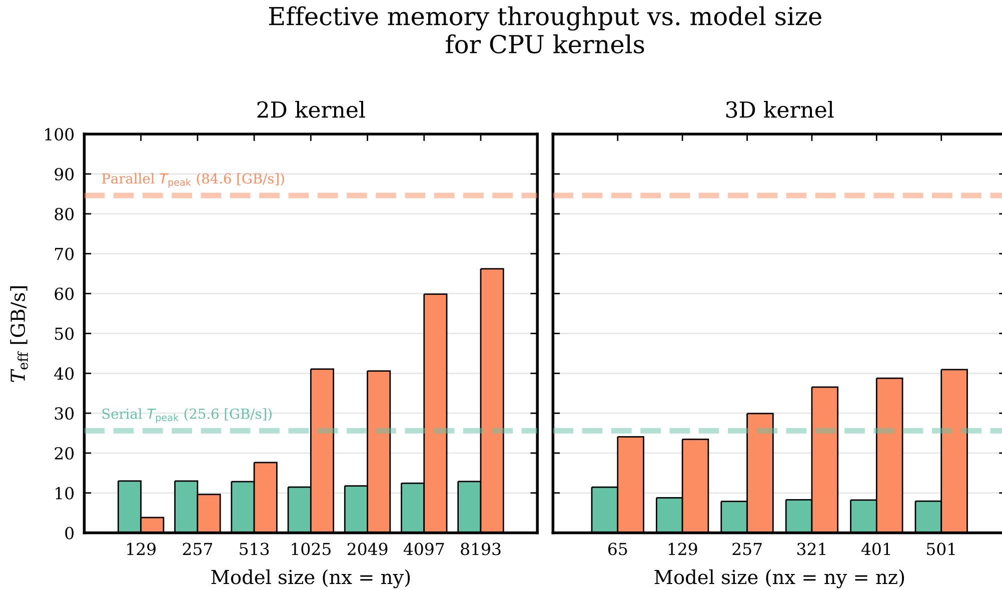


Figure 5.1: Effective memory throughput of CPU kernels for different model sizes for serial and multi-threaded parallel implementations measured on the AMD EPYC 7282 CPU. The dashed lines represent the serial and the parallel peak memory throughput of the AMD EPYC 7282 CPU benchmarked with STREAM. The parallel version was measured with 8 active threads, which was the number of active threads giving the best performance on the AMD EPYC 7282 CPU.

since we are benchmarking relatively low run-time functions (in the order of micro or milliseconds). In order to ensure a precise microbenchmark, many measurements need to be performed to compute a reasonable and stable median run time to then use for the assessment of the performance.

We reported benchmarks for both multi-core CPUs and GPUs implementations of 2D and 3D kernels. We do not have reported results for 1D kernels since they are of little relevant use in real applications. However, we noted that 1D kernels perform usually better than 2D kernels. This is most likely because of better cache usage since spatial derivatives are computed mostly using elements from the same cache line. In the 2D and 3D cases, this is not true since spatial derivatives in the second and third dimensions require elements on cache lines far away from the one of the element that gets updated.

In fig. 5.1 is shown the effective memory throughput of the serial and parallel CPU implementations versus the model size, with a reference to the serial and parallel peak memory throughput of the CPU used which, in this case, is the AMD EPYC 7282 CPU. This CPU has 16 cores, but in the figure, a run using only 8 cores is reported for the parallel version. This is because we observe the best run times for this specific number of active threads, which is a known effect of multi-threading applications that rarely benefit from using a number of active

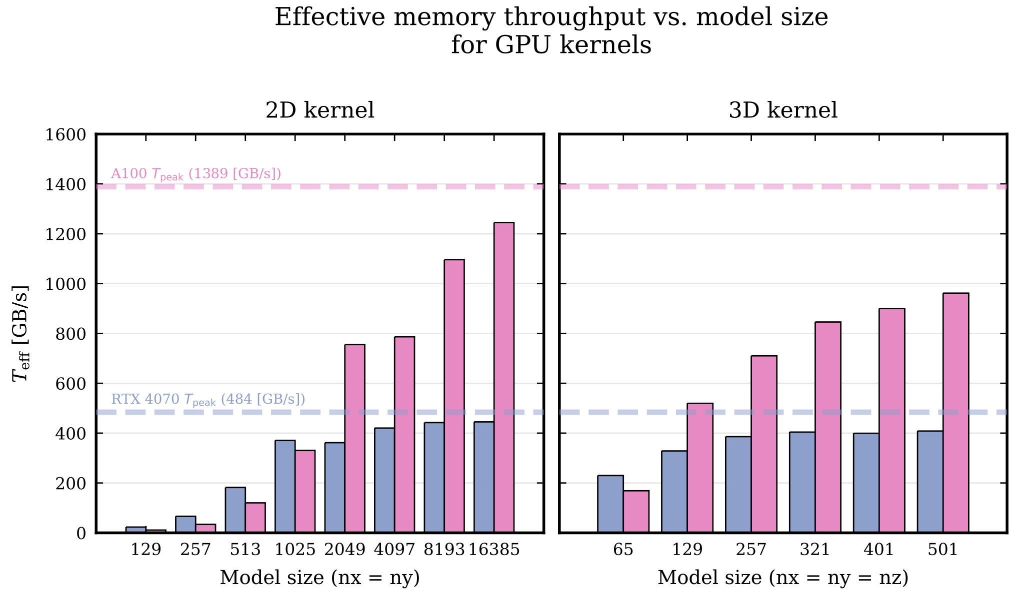


Figure 5.2: Effective memory throughput of single GPU kernels for different model sizes measured on both the NVIDIA RTX 4070 and the A100 GPUs. The dashed lines represent the peak memory throughput of both GPUs benchmarked with GPU-STREAM.

threads equal to the number of all available cores of the CPU. The serial run times have been measured by benchmarking a simple serial implementation not using ParallelStencil.jl, which performs better than the ParallelStencil.jl version using only one active thread.

We can see that the parallel versions start to surpass the serial version only for large enough model sizes for the 2D kernels, while in 3D kernels the parallel version always performs better. For the 2D kernels, we get quite close to peak performance (almost 80% for the largest model size benchmarked) in the parallel version, while for the 3D kernels, we are far away from the peak. This is again due to the memory access pattern that, in the 3D case, leads to more cache misses and increases the amount of memory transferred from the main memory to LLC. The simple serial version is quite far from the peak serial performance for both 2D and 3D kernels.

In fig. 5.2 a similar benchmark is shown but for single GPU kernels. We observe that the performance for small sizes in 2D is very low compared to peak performance. The behavior for both 2D and 3D GPU kernels as model size grows is similar to the parallel CPU kernel in fig. 5.1, although GPU kernels reach closer to to peak performance for certain sizes. Low performance on small model sizes is due to the fact that the overhead of launching the GPU kernel dominates the run time instead of the actual computation. The same can be said in the case of multi-threaded CPU kernels where spawning and synchronizing the threads has

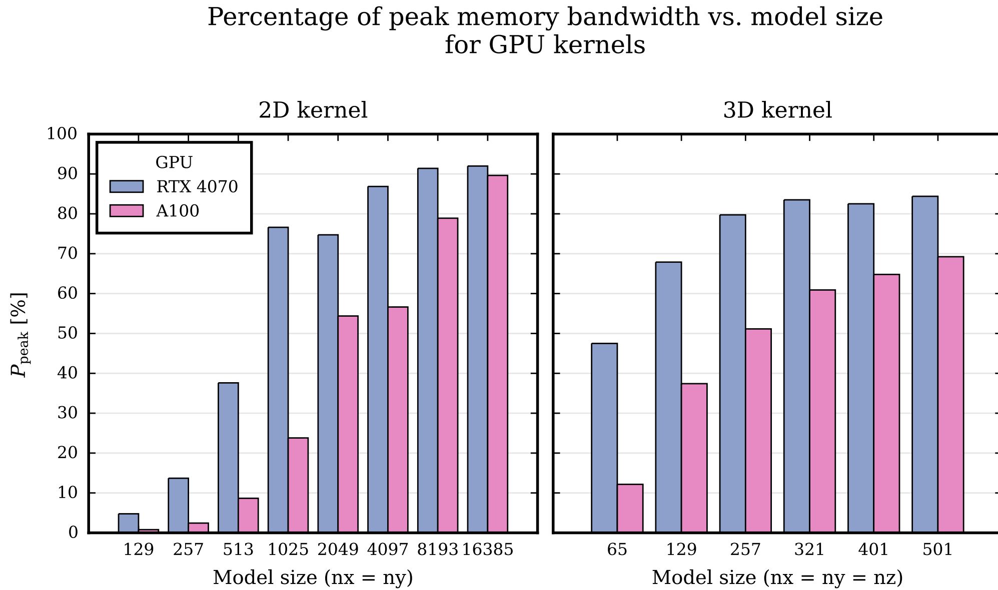


Figure 5.3: Percentage of peak memory bandwidth of single GPU kernels for different model sizes measured on both the NVIDIA RTX 4070 and the A100 GPUs.

a fixed cost that dominates the run time for small model sizes.

In fig. 5.3 we show the percentage of peak performance of the GPU kernels. We can see that we obtain close to peak performance, namely around 90% for the 2D kernel and around 85% for the 3D kernel on the RTX 4070 when considering large enough model sizes. The percentage of peak performance is a bit worse for the 3D kernel on the A100, although this is expected since the A100 has a higher peak performance which is harder to leverage if the cache is invalidated frequently like in the case of 3D kernels.

5.3 Single node GPU forward solver macrobenchmarks

In this section we will focus on benchmarking the single GPU forward solver in its entirety, hence performing what is usually called a *macrobenchmark*. In contrast to the microbenchmarks of the kernels we showed in the previous section, we are now interested in measuring the solver run time including parts that are not critical. In this way, we can get a measure of how much overhead is introduced to set up the simulation and perform the relevant checks. We will focus on the single GPU forward solver because arrays need to be allocated on the device and communications from CPU main memory (RAM) to GPU main memory (device memory) need to be performed before the start of the computation. This usually

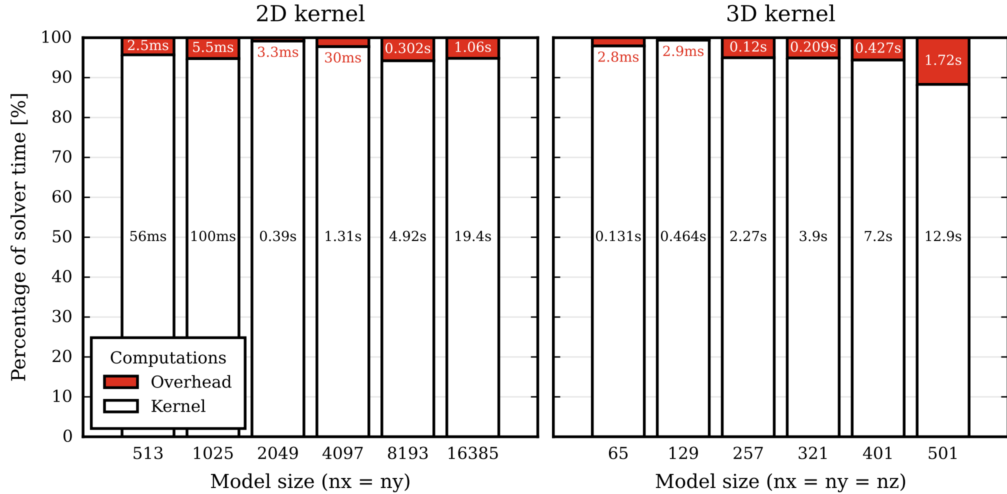


Figure 5.4: Percentage of solver run time (kernel time and overhead time) for GPU kernels on the NVIDIA RTX 4070 versus model size for 1000 time steps. Absolute run time values for both kernel and overhead time are shown in the figure with annotations on the bars.

leads to a higher overhead than the CPU implementations.

In fig. 5.4 we show the percentage of the forward solver run time (as well as absolute run time values) for both kernel and overhead times on the NVIDIA RTX 4070. The run times have been measured by running the solver for 1000 time step iterations after a proper warm-up to ensure accurate run times. Multiple measurements are performed and statistical checking is conducted as in the case of microbenchmarks, median run times are reported. We chose to run 1000 time step iterations because this is usually the number of time step iterations needed to complete a simulation with a realistic size.

We can see that the overhead times are very small compared to the time spent in the computation kernels: almost always below 10% of the run time is due to overhead. This is a good property to have which makes strong scaling theoretically possible since the kernel part of the code can be fully parallelized.

5.4 Multi-node GPU weak scaling benchmarks

In this last section of the benchmarks chapter, we will show the results for multi-node / multi-GPUs simulations on Piz Daint using MPI and ImplicitGlobalGrid.jl. In fig. 5.5 is shown the percentage of peak memory bandwidth versus the number of nodes / GPUs on Piz Daint for 2D and 3D multi-GPUs kernels performed on a weak scaling experiment. This means that the number of grid

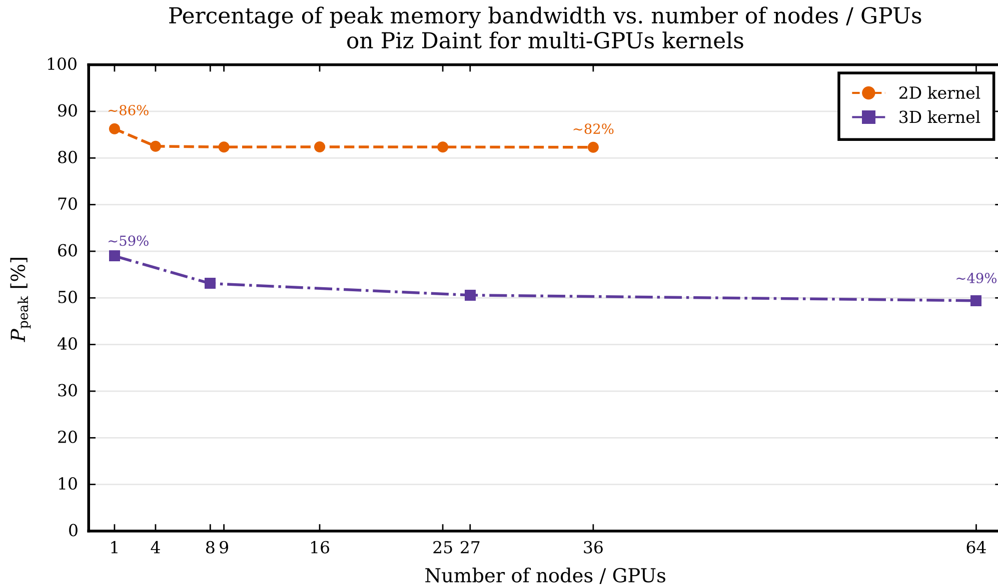


Figure 5.5: Percentage of peak memory bandwidth versus number of nodes / GPUs on Piz Daint for 2D and 3D multi-GPUs kernels performed on a weak scaling experiment where the number of grid points per node is fixed to 16385 in each spatial dimension.

points per node was kept constant (16385 in each spatial dimension), hence the total number of grid points grows linearly with the number of nodes used. The number of time step iterations performed in the experiment was calibrated such that at least 10 seconds of run time was measured for the single node GPU kernel. This ensures that the run time is big enough to be measured in a consistent and repeatable way. The peak memory bandwidth for multiple nodes was computed simply as the product between the single node peak memory bandwidth (which was benchmarked with GPU-STREAM for the P100 GPU as 559 Gbyte/s) and the number of nodes, meaning that the peak memory bandwidth is supposed to be the combined peak memory bandwidth of all the single nodes. This makes sense if we consider the communication costs to be negligible, which is the case because, as explained in chapter 3, we hide communication behind computations so that communications are performed at the same time as computations are. Note that the dashed lines in the figure are to be interpreted only as 'trends' for the percentage of peak memory bandwidth where data points are not available.

We note that the 2D multi-GPUs kernels scale quite well with only a loss of about 4% of peak memory bandwidth from 1 to 36 nodes. The 3D kernels scale a bit worse with roughly a 10% drop in peak memory bandwidth from 1 to 64 nodes. This weak scaling trend is quite good considering that we made very small changes in the code from single to multiple GPUs.

Chapter 6

Conclusions

In this thesis, we have explored various theoretical and practical aspects of acoustic FWI, to bridge the gap between theory and practice by providing the scientific community with the tools to, hopefully, iterate faster on the research in this field. We have shown, by developing the `SeismicWaves.jl` Julia package, that it is possible to write high-performance code using a high-level language, which is much more readable than traditional HPC code. This empowers the users with the possibility to understand and extend it, making the development of custom functionalities possible. We think this is very important for expediting research and letting more people know and use acoustic FWI in various applications. All of this was possible mostly because of the recent advances in scientific software and the development of the Julia packages `ParallelStencil.jl` and `ImplicitGlobalGrid.jl`, which have been the building blocks in the development of `SeismicWaves.jl`.

The gradient check and synthetic inversion results confirmed the correctness of the solvers. Real data inversions showcased the possibility of using our package for ultrasound medical tomography applications. However, they were not fully satisfactory and improvements are needed in this direction. The benchmarks showed a close-to-peak memory throughput performance of our solvers, at least for 2D applications, and a promising weak scaling for multi-xPUs.

Wrapping up this thesis, I would like to express some final thoughts and insights on the encountered difficulties and talk about future work regarding this project.

6.1 Retrospective thoughts

First of all, I would like to say that I am personally very happy with how this thesis developed. This work made me grow a lot as a person, for I am just a young student who is peeking for the first time into the vast world of research.

Many difficulties were encountered in the development of this thesis. First of all, the focus Andrea (my supervisor) and I had in mind when we discussed the project together shifted towards a more theoretical analysis of acoustic FWI in general, rather than focused on the context of ultrasound medical imaging. Nevertheless, we were able to perform a great number of synthetic inversions, most of them not showcased in this thesis for a lack of relevance or time, on both constant and variable density acoustic FWI, even in 3D dimensions which

is impressive in my opinion. We developed `SeismicWaves.jl` basically from scratch, using some code previously written for my semester project on a similar topic, but most of the work for the inversions and the adjoint solvers was done in more or less 6 months by the two of us. This meant that we also needed to set up a testing infrastructure and be sure that our code worked and produced the expected results. I have to say that this was probably the hardest part of this thesis work since we encountered some unexpected numerical issues and had to work our way through them.

Another difficult part was when I had to perform inversions using real ultrasound data. This was a completely new challenge for me and, as I was told by many in the group, a very time-consuming task. Fortunately, I had the help of my colleagues from the SWP group who guided me through it, and, even though the results are not very satisfactory, I will take them as a success since this was my very first time dealing with real data.

6.2 Future work

Regarding future work, we recently released `SeismicWaves.jl` as an open-source Julia package with the aim of extending it to elastic FWI. This will be a hard challenge since elastic wave propagation is generally much more involved with regard to the implementation.

We also look forward to using our package for probabilistic full waveform inversions using the Hamiltonian Monte Carlo method, much in the spirit of the suite of packages `HMCLab.jl` developed by the SWP group for which `SeismicWaves.jl` is a part of. Since the computational costs of performing probabilistic inversions are generally much higher than deterministic ones, having efficient and easily parallelizable solvers will be even more important for these applications.

Bibliography

[1] J. Tromp, “Seismic wavefield imaging of Earth’s interior across scales,” *Nature Reviews Earth & Environment*, vol. 1, no. 1, pp. 40–53, 2020. [Online]. Available: http://www.nature.com/articles/s43017-019-0003-8

[2] G. Nolet, *Seismic wave propagation and seismic tomography.* Dordrecht: Springer Netherlands, 1987, pp. 1–23. [Online]. Available: https://doi.org/10.1007/978-94-009-3899-1_1

[3] J. Virieux, “P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method,” *Geophysics*, vol. 51, pp. 889–901, 01 1984.

[4] A. Fichtner, *Full Seismic Waveform Modelling and Inversion*, ser. Advances in Geophysical and Environmental Mechanics and Mathematics. Springer, 2011. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15807-0

[5] ——, “Lecture notes on inverse theory,” 2021, posted on Cambridge Open Engage.

[6] R. G. Pratt, “Medical ultrasound tomography: lessons from exploration geophysics,” in *KIT Scientific Publishing*, 2017, p. 65.

[7] P. Marty, C. Boehm, and A. Fichtner, “Acoustoelastic full-waveform inversion for transcranial ultrasound computed tomography,” in *Medical Imaging*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:232357801

[8] F. Lucka, M. Pérez-Liva, B. E. Treeby, and B. T. Cox, “High resolution 3d ultrasonic breast imaging by time-domain full waveform inversion,” *Inverse Problems*, vol. 38, no. 2, p. 025008, dec 2021. [Online]. Available: https://dx.doi.org/10.1088/1361-6420/ac3b64

[9] P. S. Foundation, “Python language reference, , version 3.12.0,” 2023. [Online]. Available: https://www.python.org

[10] T. M. Inc., “Matlab version: 9.13.0 (r2022b),” Natick, Massachusetts, United States, 2022. [Online]. Available: https://www.mathworks.com

[11] B. W. Kernighan and D. M. Ritchie, *The C programming language, 1st edition.* Prentice Hall, 1978.

[12] T. Ellis, I. Philips, and T. Lahey, *Fortran 90 Programming.* Addison-Wesley Longman, Limited, 1997.

[13] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671

[14] S. Omlin and L. Räss, "High-performance xpu stencil computations in julia," 2022. [Online]. Available: https://arxiv.org/abs/2211.15634

[15] J. Hadamard, "Sur les problèmes aux dérivés partielles et leur signification physique," *Princeton University Bulletin*, vol. 13, pp. 49–52, 1902.

[16] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980. [Online]. Available: http://www.jstor.org/stable/2006193

[17] T. M. Apostol, *Mathematical analysis; 2nd ed.*, ser. Addison-Wesley series in mathematics. Reading, MA: Addison-Wesley, 1974. [Online]. Available: https://cds.cern.ch/record/105425

[18] W. E. Boyce and R. C. DiPrima, *Elementary differential equations and boundary value problems*, 3rd ed. New York: J. Wiley & Sons, 1977.

[19] C. Cerjan, D. Kosloff, R. Kosloff, and M. Reshef, "A nonreflecting boundary condition for discrete acoustic and elastic wave equations," *GEOPHYSICS*, vol. 50, no. 4, pp. 705–708, 1985. [Online]. Available: https://doi.org/10.1190/1.1441945

[20] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185–200, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999184711594

[21] D. Komatitsch and R. Martin, "An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation," *GEOPHYSICS*, vol. 72, no. 5, pp. SM155–SM167, 2007. [Online]. Available: http://library.seg.org/doi/10.1190/1.2757586

[22] D. Pasalic and R. McGarry, *Convolutional perfectly matched layer for isotropic and anisotropic acoustic wave equations.* Society of Exploration Geophysicists, 2010, pp. 2925–2929.

[23] M. Afanasiev, C. Boehm, M. van Driel, L. Krischer, M. Rietmann, D. A. May, M. G. Knepley, and A. Fichtner, "Modular and flexible spectral-element waveform modelling in two and three dimensions," *Geophysical Journal International*, vol. 216, no. 3, pp. 1675–1692, 2019.

[24] H. Igel, *Computational Seismology: A Practical Introduction*, 1st ed. Oxford University Press, 2016.

[25] C. Bunks, F. M. Saleck, S. Zaleski, and G. Chavent, "Multiscale seismic waveform inversion," *GEOPHYSICS*, vol. 60, no. 5, pp. 1457–1473, 1995.

[26] N. Kukreja, J. Hückelheim, M. Louboutin, P. Hovland, and G. Gorman, "Combining Checkpointing and Data Compression to Accelerate Adjoint-Based Optimization Problems," in *Euro-Par 2019: Parallel Processing*, ser. Lecture Notes in Computer Science, R. Yahyapour, Ed. Springer International Publishing, 2019, pp. 87–100.

[27] A. Schäfer and D. Fey, "High performance stencil code algorithms for gpgpus," *Procedia Computer Science*, vol. 4, pp. 2027–2036, 2011, proceedings of the International Conference on Computational Science, ICCS 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050911002791

[28] B. Lafci, J. Robin, X. L. Deán-Ben, and D. Razansky, "Expediting image acquisition in reflection ultrasound computed tomography," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 69, no. 10, pp. 2837–2848, 2022.

[29] F. Aminzadeh, J. Brac, and T. Kunz, *3-D Salt and Overthrust Models*. SEG/EAGE, 1997.

[30] M. Le Ravalec, B. Noetinger, and L. Hu, "The fft moving average (fft-ma) generator: An efficient numerical method for generating and conditioning gaussian simulations," *Mathematical Geology*, vol. 32, pp. 701–723, 08 2000.

[31] R. J. Mailloux, "Phased array theory and technology," *Proceedings of the IEEE*, vol. 70, no. 3, pp. 246–291, 1982.

[32] R. E. McKeighen and M. P. Buchin, "New techniques for dynamically variable electronic delays for real time ultrasonic imaging," in *1977 Ultrasonics Symposium*. IEEE, 1977, pp. 250–254.

[33] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[34] T. Deakin and S. McIntosh-Smith, "Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units," in *International Conference on Software Composition*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:60970807

[35] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance

results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807644

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Full Waveform Inversion for Medical Ultrasound Tomography in Julia on multi-xPUs

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Aloisi	Giacomo

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zurich, 30/10/2023	Aloisi Giacomo

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.